

Proceedings of the Eighth Annual National Conference on Ada Technology

March 5-8, 1990

Sponsored by
ANCOST, INC.

With Participation by
United States Army
United States Navy
United States Marine Corps
United States Air Force
Federal Aviation Administration
Strategic Defense Initiative Office

Co-Hosted by
Morehouse College
Georgia Institute of Technology
Tuskegee University

19980302 066

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

BMDOTIC
DTIC QUALITY INSPECTED 4

PLEASE RETURN TO:
SDI TECHNICAL INFORMATION CENTER

U32591

Accession Number: 3259
Title: Proceeding of the Eighth Annual
National Conference on Ada
Technology, March 5-8, 1990
Corporate Author or Publisher: US Army Communications-Electronics
Command
Fort Monmouth, NJ 07703
Report Prepared For: Strategic Defense Initiative
Organization
Publication Date: Mar 08, 1990
Pages: 669
Comments on Document: Inventory for TN
Descriptors, Keywords: ADA MIS SQL Life Cycle Environment
Software Development Computer
Program

PROCEEDINGS OF EIGHTH ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

Sponsored By:
ANCOST, INC.

With Participation By:
United States Army
United States Navy
United States Marine Corps
United States Air Force
Federal Aviation Administration
Strategic Defense Initiative Office

Co-Hosted By:

Morehouse College
Georgia Institute of Technology
Tuskegee University

Hyatt Regency-Atlanta, GA

March 5-8, 1990

Approved for Public Release: Distribution Unlimited

8th ANNUAL NATIONAL CONFERENCE ON ADA TECHNOLOGY

CONFERENCE COMMITTEE 1989-1990

MR. JAMES BARBOUR
Digital Equipment Corp.
Merrimack, NH 03054

MS. CHRISTINE BRAUN
Contel Technology Center
Fairfax, VA 22033

MR. MIGUEL A. CARRIO, JR.
Teledyne Brown Engineering
Fairfax, VA 22030

MS. LUWANA S. CLEVER
Florida Inst. of Technology
W. Melbourne, FL 32904

DR. MARY R. ELLIS
Hampton University
Hampton, VA 23668

MR. DONALD C. FUHR
Tuskegee University
Tuskegee, AL 36088

MS. JUDITH M. GILES
Intermetrics, Inc.
Cambridge, MA 02138

MS. DEE. M. GRAUMANN
General Dynamics, DSD
San Diego, CA 92138

DR. GEORGE C. HARRISON
Norfolk State University
Norfolk, VA 23504

MR. DAVID L. JOHNSON
GTE Government Systems
Rockville, MD 20850

DR. ARTHUR JONES
Morehouse College
Atlanta, GA 30314

DR. GENEVIEVE M. KNIGHT
Coppin State College
Baltimore, MD 21216

DR. RICHARD KUNTZ
Monmouth College
W. Long Branch, NJ 07764

DR. RONALD LEACH
Howard University
Washington, DC 20059

MS. SUSAN MARKEL
TRW
Fairfax, VA 22031

MS. CATHERINE PEAVY
Martin Marietta Information
& Communications Systems
Denver, CO 80201-1260

MR. RICHARD PEEBLES
Concurrent Computer Corp.
Tinton Falls, NJ 07724

DR. M. SUSAN RICHMAN
The Pennsylvania State
University at Harrisburg
Middletown, PA 17057

MR. JOHN W. ROBERTS
EDO Corp.
Chesapeake, VA 23320

MR. WALTER ROLLING
Ada Technology Group Inc.
Washington, DC 20001

MS. SUSAN ROSENBERG
Cadre Technologies, Inc.
Providence, RI 02903

MR. MICHAEL SAPENTER
Telos Federal Systems
Lawton, Ok 73501

MR. TERENCE P. STARR
General Electric Company
Philadelphia, PA 19101

MR. JAMES E. WALKER
Network Solutions
Herndon, VA 22070

MR. JESSE WILLIAMS
Cheyney University
Cheyney, PA 19319

CONFERENCE DIRECTOR

MARJORIE Y. RISINGER, CMP
Rosenberg & Risinger, Inc.
Culver City, CA 90230

ADVISORY MEMBERS

MR. LOUIS J. BONA
FAA Technical Center
Atlantic City Airport, NJ 08405

MR. DANIEL E. HOCKING
AIRMICS
Atlanta, GA 30332-0800

MAJOR DOUG SAMUELS
HQ, AFSC/PLRT
Andrews AFB, DC 20334-5000

LTC ROBERT SETTLE
Marine Corps., Tactical Systems
Support Activity
Camp Pendleton, CA 92055

ANTOINETTE STEWART
Department of the Navy
Washington, DC 20350

MS. KAY TREZZA
HQ, CECOM, CSE
Ft. Monmouth, NJ 07703-5000

MR. GEORGE WATTS
HQ. CECOM, CSE
Ft. Monmouth, NJ 07703-5000

PANELS AND TECHNICAL SESSIONS

Tuesday, March 6, 1990

9:00 AM	Opening Session	
10:00 AM	Panel I	Ada Policy, Practices & Initiatives
2:00 PM	Session 1	Education/Training
2:00 PM	Session 2	MIS Ada/SQL
2:00 PM	Session 3	Embedded Applications
2:00 PM	Session 4	Ada Language Issues
4:00 PM	Session 5	MIS Lessons Learned
4:00 PM	Session 6	Life Cycle Environments

Wednesday, March 7, 1990

8:30 AM	Panel II	Total Quality Management
8:30 AM	Panel III	Ada 9X
10:30 AM	Session 7	Student Presentations
10:30 AM	Session 8	Student Presentations
10:30 AM	Session 9	Student Presentations
1:30 PM	Session 10	Project Management
1:30 PM	Session 11	Software Development
1:30 PM	Session 12	Distributed/Parallel Processing
1:30 PM	Session 13	Life Cycle Envinroments
3:30 PM	Panel IV	Preparing Students for Industry
3:30 PM	Session 14	Reuse

Thursday, March 8, 1990

8:30 AM	Panel V	SEI Capabilities Assessment
8:30 AM	Session 15	Reuse
8:30 AM	Session 16	Metrics
10:30 AM	Session 17	System Development
10:30 AM	Session 18	Life Cycle Productivity
2:00 PM	Panel VI	Future Directions in Ada

PAPERS

The papers in this volume were printed directly from unedited reproducible copies prepared by the authors. Responsibility for contents rests upon the author and not the symposium committee or its members. After the symposium, all the publication rights of each paper are reserved by their authors, and requests for republication of a paper should be addressed to the appropriate author. Abstracting is permitted, and it would be appreciated if the symposium is credited when abstracts or papers are republished. Requests for individual copies of papers should be addressed to the authors.

PROCEEDINGS

EIGHTH NATIONAL CONFERENCE ON ADA TECHNOLOGIES

Bound—Available at Fort Monmouth

2nd Annual National Conference on Ada Technology Proceedings—1984 (Not Available)
3rd Annual National Conference on Ada Technology Proceedings—1985—$10.00
4th Annual National Conference on Ada Technology Proceedings—1986 (Not Available)
5th Annual National Conference on Ada Technology Proceedings—1987 (Not Available)
6th Annual National Conference on Ada Technology Proceedings—1988—$20.00
7th Annual National Conference on Ada Technology Proceedings—1989—$25.00
8th Annual National Conference on Ada Technology Proceedings—1990—$25.00

Extra Copies: 1-3 $25; next 4 $20; next 11 & above $15 each

Make check or bank draft payable in U.S. Dollars to ANCOST and forward requests to:

Annual National Conference on Ada Technology
U.S. Army Communications-Electronics Command
ATTN: AMSEL-RD-SE-CRM (Ms. Kay Trezza)
Fort Monmouth, New Jersey 07703-5000

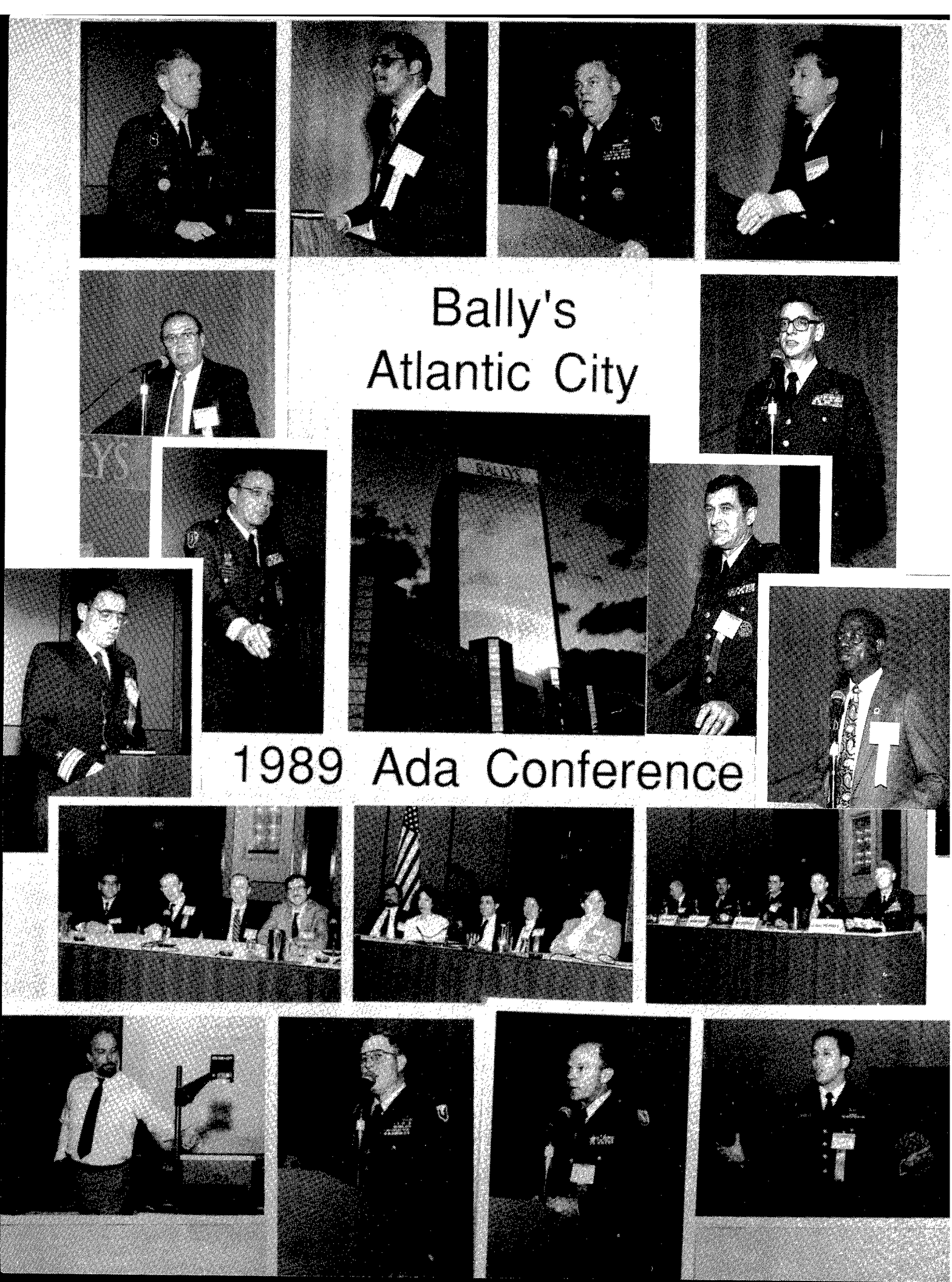
Telephone inquiries may be directed to Ms. Kay Trezza at (201) 532-1898

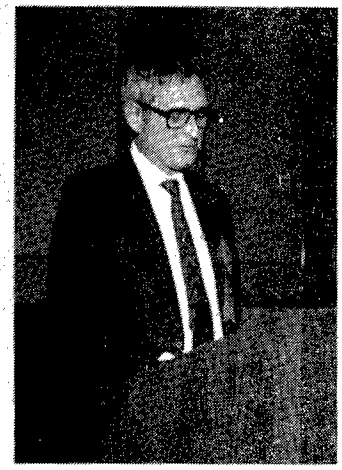
Photocopies—Available at Department of Commerce. Information on prices and shipping charges should be requested from:

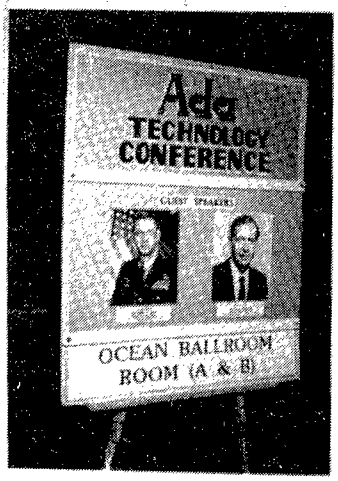
U.S. Department of Commerce
National Technical Information Service
Springfield, Virginia 22151
USA

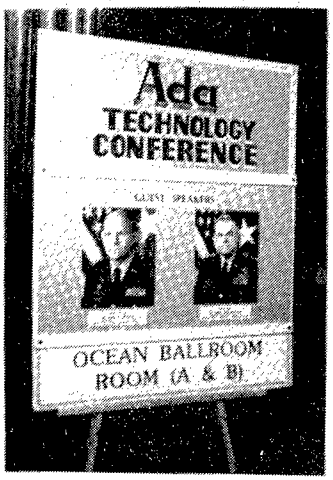
Include title, year, and AD number

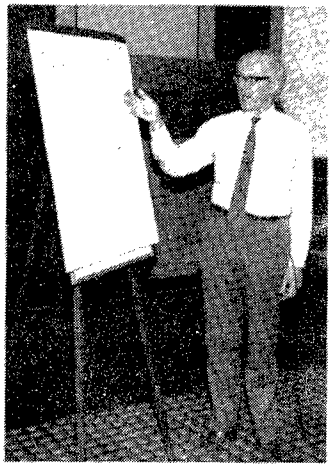
2nd Annual National Conference on Ada Technology—1984—AD A142403
3rd Annual National Conference on Ada Technology—1985—AD A164338
4th Annual National Conference on Ada Technology—1986—AD A167802
5th Annual National Conference on Ada Technology—1987—AD A178690
6th Annual National Conference on Ada Technology—1988—AD A190936


Bally's
Atlantic City
1989 Ada Conference







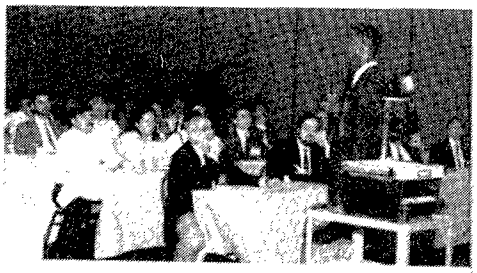




Ada Sessions and Panel Discussions









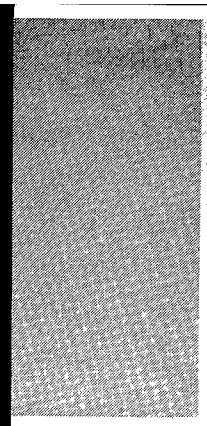






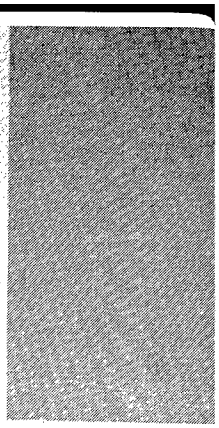












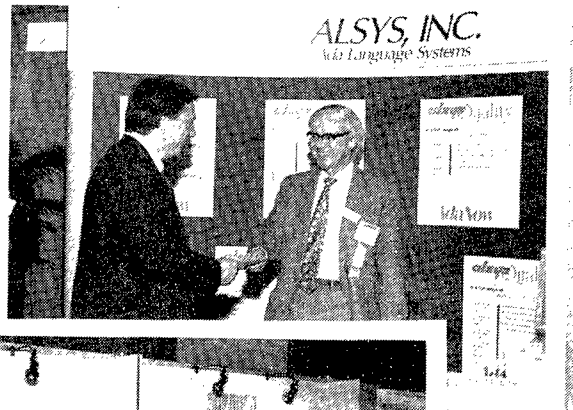


Ada Conference Registration and Exhibits



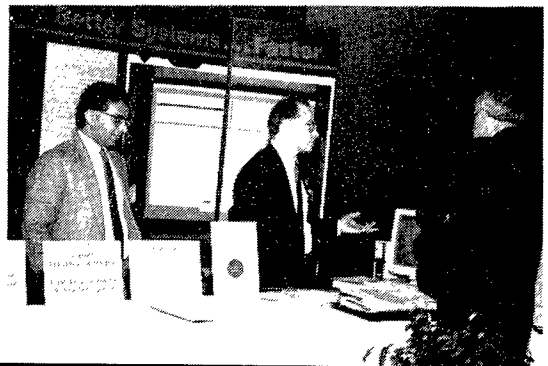


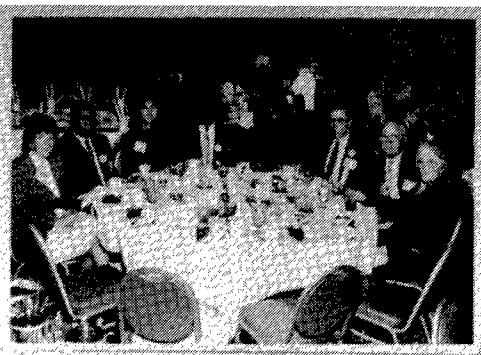


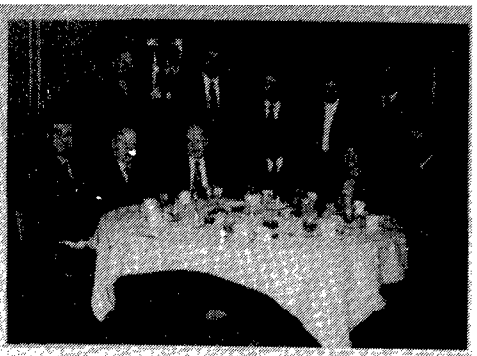




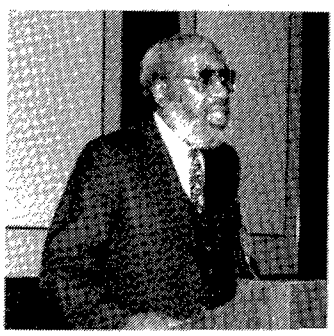






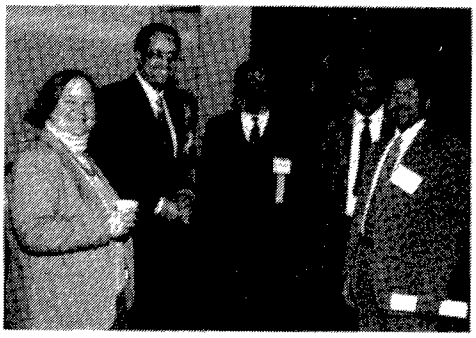






Ada Conference Highlights













Ada Conference Highlights

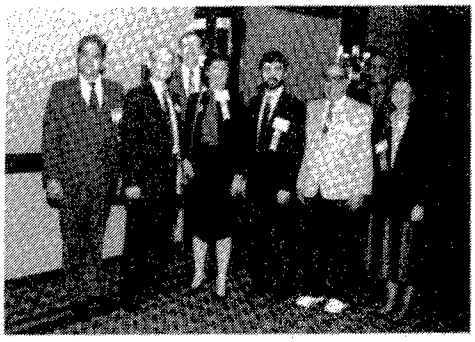
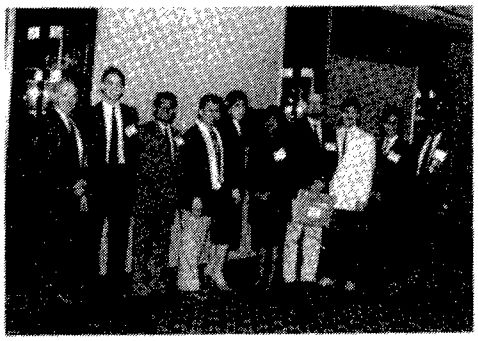




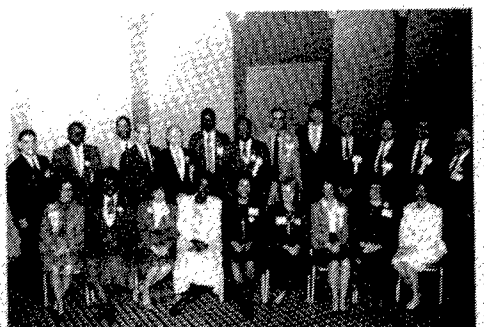


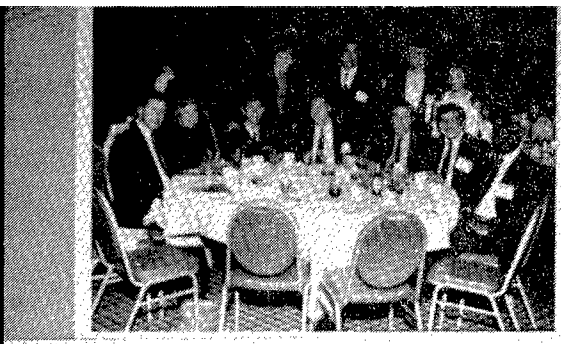




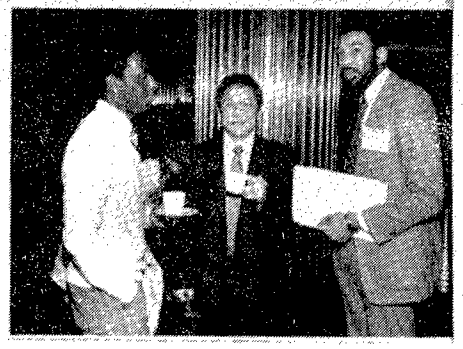










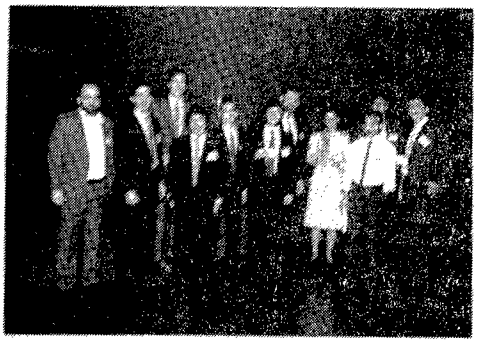


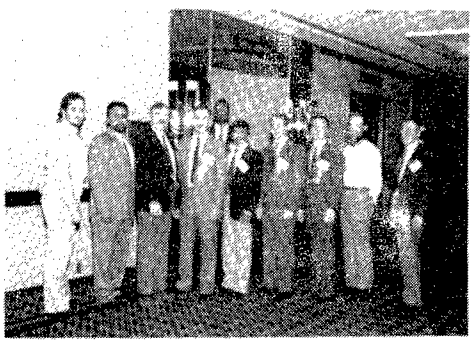




Ada Conference Highlights







7th ANNUAL CONFERENCE ON ADA TECHNOLOGY
Awards and Thank You's



Susan Richmond, of Penn State University at Harrisburg, received a congratulatory award from John Sintec, Acting Director of CECOM. Ms. Richmond chaired the ANCOST Committee in planning the 7th Annual National Conference on Ada Technology.



The 1989 Ada Technology Conference Planning Committee posed for this group picture after a most successful conference.

TABLE OF CONTENTS

TUESDAY, MARCH 6, 1990

Vendor Showcase Hours: 9:30 am-12:00 Noon
2:00 pm-7:00 pm—Ivy Hall A

OPENING SESSION—9:00 am-12:00 noon—Falcon/Condor
Keynote Speaker: LTG Jerome B. Hilmes, Director of Information Systems for Command, Control, Communications, and Computers, U.S. Army 1

Panel Discussion I: Ada Policy, Practices, and Initiatives
Moderator: Mr. John H. Sintic, Acting Director, CECOM Center for Software Engineering, U.S. Army CECOM, Fort Monmouth, NJ 2
Panelists:
MG Peter A. Kind, Program Executive Officer, Command and Control Systems, Fort Monmouth, NJ 3
Nina Cornett, Acting Director, Information Resources Management, Dept. of Navy 4
Brig. Gen Albert J. Edmonds, Assistant Chief of Staff, Systems for Command, Control, Communications, and Computers, HQ USAF 5
Dr. James A. Painter, SES, Special Assistant/Technical Advisor, Director, Command, Control, Communications, and Computers 6
Dr. John A. Scardina, Director of Automation Engineering, Federal Aviation Agency

LUNCH—12:00 noon-2:00 pm—Hanover Hall
Guest Speaker: Lt Gen George L. Monahan, Jr., USAF, Director of the SDIO 7

Session 1: 2:00 pm-5:30 pm—Phoenix
Education/Training
Chairperson: Ms. Luwana Clever, Florida Institute of Technology, Melbourne, FL
1.1 Introducing Ada as New Language in a Software Engineering Course—T. Korson, Clemson University, Clemson, SC 8
1.2 Teaching Software Engineering with Ada—Y.Y. Wu, ITT Research Institute, Annapolis, MD 13
1.3 Designing an Ada Tutorial Using Object Oriented Design C.D. Marable, Jr. and C. C. Belgrave, Hampton University, Hampton, VA 19
1.4 The Design of a PC-Based CAI Program in Ada—J.H. Heidema, Telos Federal Systems, Shrewsbury, NJ 21
1.5 A First Programming Exercise Using Ada Tasking: The Odometer—J.K. Joiner, USAFA/DFCS, US Air Force Academy, Colorado Springs, CO 25
1.6 A Student Project to Develop a Distributed Flight Simulator in Ada—R.F. Vidale, Boston University, Boston, MA 32

Session 2: 2:00 pm-3:30 pm—Falcon
MIS Ada/SQL
Chairperson: Mr. David Johnson, GTE Government Systems, Rockville, MD

2.1 Interfacing Ada and SQL: The Two Layer Approach—T. Wheeler and J. Richardson, US Army CECOM, CSE, Fort Monmouth, NJ; and A. Ferre, RADC/COEE, Griffiss AFB, NY 36
2.2 A Practical Ada/SQL Implementation—J.L. Schoenecker III, US Army Information Systems Software Development Center-Washington, Fort Belvoir, VA 49
2.3 Creating a Database Application Using the SQL Ada Module Extension (SAME) Method—T.D. Fenton, USAISSDCL, Fort Lee, VA 56

Session 3: 2:00 pm-5:30 pm—Condor
Embedded Applications
Chairperson: Mr. James Walker, Network Solutions, Herndon, VA

3.1 FED Successfully Solved the Ada, Software Engineering/DOD-STD-2167A Algorithm, J. Sodhi, T. Archer, and G. Daubenspeck, Telos Systems, Lawton, OK 63
3.3 Developing an In-House VHDL Capability—S. Adamus and K. Crossland, SAIC Comsystems, San Diego, CA 69
3.4 Ada in an Embedded Real-Time Environment—D. Schmidt, P.P. TEXEL & CO., Inc., Eatontown, NJ 76
3.5 On the Design of a Protected Interface to a C-2 System J.P. Fitzgibbon and R. Madsen, Martin Marietta Information & Communication Systems, Colorado Springs, CO 84

Session 4: 2:00 pm-3:30 pm—York/Stuart
Ada Language Issues
Chairperson: Ms. Christine Braun, Contel Technology Center, Chantilly, VA

4.1 Reusability and Verification of Generics in Ada—F. Lin and G. W. Ernst, Case Western Reserve University, Cleveland, OH 98
4.2 Engineering for Reuse Using Procedure Types—L. Latour and C. Meadow, University of Maine, Orono, ME 106
4.3 Disgorging Ada 9X Revision Requirements Issues from the Ada Issues and Commentaries—C.D. Ardoin, R.J. Knapper, J.L. Linn, and R. N. Meeson, Jr., Inst. for Defense Analyses, Alexandria, VA 118

Session 5: 4:00 pm-5:30 pm—Falcon
MIS Lessons Learned
Chairperson: Mr. Daivd Johnson, GTE Government Systems, Rockville, MD

5.1 Software Engineering, Software Development Methodologies—Emphasis on Lessons Learned—W.H. Pitts, US Army Information Systems Command, Software Development ..... Center, Fort Benjamin Harrison, Indianpolis, IN 129
5.2 An MIS Ada Practicum—K. Fussichen, Computer Sciences Corp., Indianapolis, IN 132
5.3 Commercial MIS Business Systems, Objected-Oriented Development & Ada—Can They Fit Together?—R. Steinberg, Andersen Consulting, Chicago, IL 142

Session 6: 4:00 pm-5:30 pm—York/Stuart
Life Cycle Environments I
Chairperson: Ms. Susan Rosenberg, Cadre Technologies, Inc., Providence, RI

6.1 Procedures for Computer-Aided Ada Software Engineering Tool Assessment—A. Goel, UNIXPROS Inc., Colts Neck, NJ; and J. LeBaron, US Army CECOM, Fort Monmouth, NJ 147
6.2 CASE Tools Supporting Ada—Reverse Engineering: State of the Practice—D.E. Mularz, M. C. Smith, and T.J. Smith, The MITRE Corp., McLean, VA 157
6.3 The Ada Language System/Navy Project—W. L. Wilder, Department of the Navy, Naval Sea Systems Command, Washington, D.C. 165

5:45pm-7:00 pm Vendor Reception—Ivy Hall A
7:30 pm-10:00 pm Commercial Ada Users Working Group—Phoenix

WEDNESDAY, MARCH 7, 1990

Vendor Showcase Hours: 10:00 am-7:00 pm—Ivy Hall

Panel Discussion II: 8:30 am-10:00 am—Falcon
Total Quality Management
Panel Chairperson: Susan D. Markel, TRW
Spotlight Speaker: MG Peter A. Kind, Program Executive Officer, Command and Control Systems, Fort Monmouth, NJ
Panel Members: Jack Dreyfus, VP and General Manager, TRW Systems Division; Gary Floss, VP Corporate Quality, Control Data Corporation; Robert L. Knickerbocker, Director of Technical Operations, Martin Marietta

Panel Discussion III: 8:30 am-10:00 am—Condor
Ada 9X
Panel Chairperson: Chris Anderson, Ada 9X Project Manager
Panel Members: Key Participants in the Ada 9X Process

Student Presentations (Sessions 7, 8, and 9)

Session 7: 10:30 am-12:00 noon—Falcon
Ada Packages
Chairperson: Dr. Richard Kuntz, Monmouth College, W. Long Branch, NJ

7.1 An Ada Graphics Package for Personal Computers—J.A. Fox and V.S. Dobbs, Wright State University, Dayton, OH 187
7.2 Problems with Ada Numeric Routines—P.D. Buck, S.L. Day, and D. Gonzalez, National University, San Diego, CA 195
7.3 Intelligent Ada Tutor—S. Palanisamy, Oklahoma State University, Stillwater, OK 205
7.4 Software Engineering Project Using Ada to Express Requirements, Specifications, and Design for a System to Solve the Quadratic Equation—T. Every, L. Lahti, A. Lipscomb, and W. Longshore, Embry Riddle Aeronautical University, Daytona Beach, FL

Session 8: 10:30 am-12:00 noon—Condor
Applications
Chairperson: Dr. Mary Ellis, Hampton University, Hampton, VA

8.1 Petri Net Modeling of Ada Programs for Deadlock Detection and Metrics—D. Divine and M. Fowles, National University, San Diego, CA 214
8.2 Problems in Porting and Maintaining Ada Software—S. Shen, Howard University, Washington, D.C. 222
8.3 Design of Integrity Kernels for Distributed Systems—S. Ramanna, and J.F. Peters, Kansas State University, Manhattan, KS 226
8.4 A Clean Solution to the Readers-Writers Problem without the COUNT Attribute—D. Levin, Unversity of Wisconsin-Parkside, Kenosha, WI; D. Nohl, Illinois Benedictine College, Lisle, IL; and T. Elrad, Illinois Institute of Technology, Chicago, IL 236
8.5 Use of Ada for Expressing Requirements Specifications for a System for the Automation of the Air Traffice Controller's Assistant's Assistant—L. Lahti, A. Lipscomb, T. Every, and W. Longshore, Embry-Riddle Aeronautical University, Daytona Beach, FL 241

Session 9: 10:30 am-12:00 noon—Phoenix
CASE
Chairperson: Mr. Jesse Williams, Cheyney University, Cheyney, PA

9.2 Automated Ada Versus Hand Written Ada Code—M.U. Devi, University of Mississippi, University, MS 251
9.3 Ada and the Software Engineering Life Cycle—L. Pope and Y.D. Couevas, University of Mississippi, University, MS 255
9.4 An Environment for Testing Ada Programs Implemented in Ada—O.R. Weyrich, Jr., Athens, GA 262
9.5 Reusability and Extensibility in Ada—F. Lin and A. Datta, Case Western Reserve University, Cleveland, OH 269
9.6 Some Ada Generic Packages for Tasks Communications C.Y. Hsieh, University of Oklahoma, Norman, OK 275

LUNCH BREAK: 12:00 noon-1:15 pm

Session 10: 1:30 pm-3:00 pm—Falcon
Project Management
Chairperson: Mr. Donald Fuhr, Tuskegee University, Tuskegee, AL

10.1 Preparing Non-Ada Personnel for Transitioning into the Ada World—S. Sutherlin, Telos Federal Systems, Lawton, OK 281
10.2 Managing Ada Software Development in the Real World S.J. Fee, Vitro Corp., Silver Spring, MD 283
10.3 Transition to AdaBIS—C.J. Hornung, Computer Sciences Corp., Indianapolis, IN 289

Session 11: 1:30 pm-5:00 pm—Condor
Software Development
Chairperson: Dr. George Harrison, Norfolk State University, Norfolk, VA

11.1 A Comparison Between Functional Decomposition and Object-Oriented Methodologies—A CASE Study—R. Fulbright, Fluor Daniel, Inc., Greenville, SC 299
11.2 An Ada Plasma Panel Controller—R.T. Mraz, USAFA/DFCS, US Air Force Academy, Colorado Springs, CO ..307
11.3 A Structured Stepwise Refinement Method for VDM—S.A. Conger, K. Kumar, E.R. McLean, V.K. Vaishnavi, M.D. Fraser, R.A. Gagliano, and G.S. Owen, Georgia State University, Atlanta, GA 311
11.4 Formal Design Methods for Dynamic Ada Architectures C.M. Bymes, The MITRE Corp., Bedford, MA 321
11.5 Ada Exceptions and Fault-Tolerance—D.M. Coleman and R.J. Leach, Howard University, Washington, D.C. 338
11.6 A Better Approach to Software Engineering—H.P. Woodward, TRW Federal Systems Group, Fairfax, VA ..343

Session 12: 1:30 pm-3:00 pm—Phoenix
Distributed/Parallel Processing
Chairperson: Dr. Ronald Leach, Howard University, Washington, D.C.

12.1 Increasing System Reliability through Multi-level Fault-Tolerance—D.J. Stautz, SofTech Inc., Colorado Springs, CO 349
12.2 Designing Parallel Data Types for Ada—P.B. Anderson, Planning Research Corp., McLean, VA; and E.K. Park, US Naval Academy, Annapolis, MD 358

12.3 The Portable Common Execution Environment (PCEE): An Approach to System Software for Reliable, Distributed Systems—C. Randall, GHG Corp., Houston, TX; D. Auty, SofTech, Houston, TX; A. Burns, University of Bradford, Bradford, England; C.W. McKay and P. Rogers, University of Houston-Clear Lake, Houston, TX 367

Session 13: 1:30 pm-5:00 pm—York/Stuart
Life Cycle Environments II
Chairperson: Mr. Brian Baker, Navy Department, Washington, D.C.

13.1 Ada Composite Benchmark for Intelligence/Electronic Warfare Systems—A. Goel, Colts Neck, NJ; and M.E. Bender US Army, CECOM, Fort Monmouth, NJ 383
13.2 Proof Method for Ada/TL Temporal Specifications—W. Hankley and J. Peters, Kansas State University, Manhattan, KS 392
13.3 Automatic Production of Mil/DoD/NASA-Std Documentation—Lee Jensen and T. S. Radi, Software Systems Design, Inc., Claremont, CA 399
13.4 The Intelligent Test Bed: A Tool for Software Development and Software Engineering Education—S.A. Conger, M.D. Fraser, R.A. Gagliano, K. Kumar, E.R. McLean and G.S. Owen, Georgia State University, Atlanta, GA 408
13.5 An Implementation of Testing and Debugging Tools for Concurrent Ada Programs—W.N. Taylor, IBM Corp., Research Triangle Park, NC; R.H. Carver, North Carolina State University, Raleigh, NC; and K.C. Tai, Software Engineering Program, National Science Foundation, Washington, D.C. 419

Session 14: 3:30 pm-5:00 pm—Essex
Reuse I
Chairperson: Mr. Daniel Hocking, AIRMICS, Atlanta, GA

14.1 Software Reuse: Managerial and Technical Guidelines J.W. Hooper, The University of Alabama in Huntsville, Huntsville, AL; and R.O. Chester, Martin Marietta Energy Systems, Inc., Oak Ridge, TN 424
14.2 The Economics of Software Reuse: Issues and Alternatives—S.L. Pfleeger and T.B. Bollinger, CONTEL Technology Center, Chantilly, VA 436
14.3 Economic Analysis of Software Reuse on Large Ada Projects—A Model Study—H.F. Joiner II, Telos Federal Systems, Shrewsbury, NJ 448

Panel Discussion IV: 3:30 pm-5:00 pm—Lancaster D
Preparing Students for Industry
Panel Chairperson: Miguel A. Carrio, Teledyne Brown Engineering
Panel Members: Dr. Phillip Caverly, Jersey City State College; Mr. Dave Johnson, GTE Government Systems; Dr. Susan Richman, Pennsylvania State University; Mr. Terence Starr, General Electric

Banquet: 7:00 pm-9:30 pm—Hanover Hall
Guest Speaker: MG Billy M. Thomas, Commanding General, US Army CECOM

THURSDAY, MARCH 8, 1990
Vendor Showcase Hours: 10:00 am-2:00 pm—Ivy Hall

Panel Discussion V: 8:30 am-10:00 am—Lancaster Hall
SEI Capabilities Assessment
Panel Chairperson: Ms. Catherine H. Peavy, Martin Marietta
Panel Members: Mr. Ken Dymond, Software Engineering Institute; Mr. George Mineah, Ford Aerospace; Ms. Toni Shetler, TRW; Mr. Conrad Czaplicki, Naval Air Development Center

Session 15: 8:30 am-12:00 noon—Essex A/B
Reuse II
Chairperson: Chirstine Braun, Contel Technology Center, Chantilly, VA

15.1 RAPID—Lessons Learned During Pilot Operation—T. Vogelsong and J. Rothrock, USAISSDCW, Fort Belvoir, VA 452
15.2 Domain Analsyis within the ISEC RAPID Center—E. Guerrieri, SofTech, Inc., Waltham, MA; and W. Vitaletti, SofTech, Inc., Alexandria, VA 460
15.3 Repository Support for a Reuse Process—B.J. Kitaoka, Science Applications International Corp. (SAIC), Clearwater, FL 471
15.4 Software Reclamation—J. Bailey and V. Basili, University of Maryland, College Park, MD 477
15.5 Literate Programmning for Reusability: A Queue Package Example—T.L. Pappas, Intermetrics, Inc., Warminster, PA 500
15.6 Should Data Abstraction be Violated to Enhance Software Reuse?—S. Muralidharan and B.W. Weide, Ohio State University, Columbus, OH 515

Session 16: 8:30 am-12:00 noon—French/Italian/Austrian Metrics
Chairperson: Mr. Daniel Hocking, AIRMICS, Atlanta, GA

16.1 Measurement of Ada Throughout the Software Development LifeCycle—S. Henry and K. Mayo, Virginia Tech., Blacksburg, VA; and B. Chappell, Software Technology, Inc., Alexandria, VA 525
16.2 Software Reliaiblity Prediction and Management—B. Papanicolaou, Sanders Associates, Nashua, NH 533
16.3 Effectiveness Measures for the Software Process—R. Guilfoyle, Monmouth College, W. Long Branch, NJ 537
16.4 Measuring Coupling of Ada Program Modules—R.J. Leach and K.S. Bagley, Howard University, Washington, D.C.; D.E. Butler, AT&T Bell Laboratories, Naperville, IL; I.L. Brown and P.W. Spann, Bell Communications Research, Piscataway, NJ 546
16.5 Analysis of Portability of AFATDS Concept Evaluation Source Code for Reuse on ATCCS Common Hardware Software—C.A. Burnham, P. Ho, and H.F. Joiner, II, Telos Systems, Shrewsbury, NJ 592
16.6 Experience Using an Automated Metrics Framework in the Review of Ada Source for AFATDS—J.D. Anderson and J.A. Perkins, Dynamics Research Corp., Andover, MA; S. Levine, PM FATDS, Fort Monmouth, NJ 597

Session 17: 10:30 am-12:00 noon—Lancaster A/B/C
System Development
Chairperson: Ms. Dee Graumann, General Dynamics, DSD, San Diego, CA

17.1 A Performance-Oriented Development Methodology for Distributed Real-Time Systems—A.J. Clough, The Charles Stark Draper Laboratory, Inc., Cambridge, MA; R.F. Vidale and T. J. Yuhas, Boston University, Boston, MA 613
17.2 A Development Interface for an Expert System Shell—J. E. Courte and V.S. Dobbs, Wright State University, Dayton, OH 623
17.3 The Automatic Translation of Lisp Applications into Ada H.G. Baker, Nimble Computer Corp. Encino, CA 633

Session 18: 10:30 am-12:00 noon—Lancaster D/E
Life Cycle Productivity
Chairperson: Dr. Arthur Jones, Morehouse College, Atlanta, GA

18.1 Portability: A Key Element in Life Cycle Productivity—T. Archer and S.A. Easson, Telos Systems, Lawton, OK ... 640
18.2 An Ada Runtime Supervisor Simulator—R.T. Lewis, Yuma Proving Ground, Yuma, AZ; D.A. Workman and S.D. Lang, University of Central Florida, Orlando, FL 643
18.3 The Resolution of Performance Problems in an Ada Communication System—T.L.C. Chen, E-Systems, Inc., St. Petersburg, FL; and R. Cooley, Sequent Computer Systems, Tampa, FL 651

LUNCH BREAK: 12:00 noon-1:45 pm

Panel Discussion VI: 2:00 pm-3:30 pm—Lancaster Hall
Looking to the Future with Ada
Panel Chairperson: Ms. Dee Graumann, General Dynamics; **Mr. James Barbour,** Digital Equipment Corporation
Panel Members: Lt Col Charles Lillie, SDIO; **Lt Col Robert Lyons,** JIAWG; **Dr. Jack Kramer,** STARS; **Mrs. Susan Voigt,** NASA/Langley; **Chris Anderson,** ADA 9X; **Bob Roe,** Boeing

KEYNOTE SPEAKER



LTG Jerome B. Hilmes
Director of Information Systems for
Command, Control, Communications, and
Computers, U.S. Army

Lieutenant General Jerome B. Hilmes was born in Carlyle, Illinois on 21 December 1935. Upon completion of studies at the United States Military Academy in 1959, he was commissioned a second lieutenant and awarded a Bachelor of Science degree. He also holds a MS and PhD degree from Iowa State University. His military education includes completion of the U.S. Army Engineer School, U.S. Army Command and General Staff College and the Naval War College. He is a registered professional engineer in the state of New York.

He has held a wide variety of command and staff positions culminating in his current assignment as Director of Information Systems for C4, Office Secretary of the Army. These include:

Commander of OTEA, a field operating agency of the Office of the Chief of Staff, Army. General Hilmes had responsibility for planning and conducting continuous comprehensive evaluation (C2E) and operational testing for all major Army systems.

Commander of the Southwestern Division, U.S. Army Corps of Engineers in Dallas (1985-88) and the Corps' North Central division in Chicago (1983-85). He was also Chairman of the Board of Engineers and a member of the Mississippi River Commission.

Commander, 7th Engineer Brigade and Ludwigsburg-Kornwestheim Military Community, Stuttgart, Germany (1978-80); Commander, 23rd Engineering Battalion, 3rd Armored Division, Hanau, Germany (1976); and Commander, Task Force Sierra, 18th Engineer Brigade, Vietnam (1970-71).

General Hilmes' major staff assignments include Deputy Assistant Chief of Engineers and Programs, Washington, DC (1981-83); Director of Facilities Engineering and Housing, Fort Bragg, NC (1980-81); and Assistant Deputy Chief of Staff Engineer Headquarters, U.S. Army, Europe (1976-78).

His publications include, "LHX", "Green Ribbon Panel Report, Mar 85", "Winter Reforger '79: Lessons Learned", "Stopping an Armored Attack", MICV-UTTAS", "Statistical Analysis of Under-reinforced Prestressed concrete Flexural Members" and "Seventh Army Expedient Bridge".

Awards and decorations which General Hilmes received include the Distinguished Service Medal, Legion of Merit (three awards), Bronze Star Medal (two awards), Meritorious Service Medal, Air Medal, Joint Service Commendation Medal, and Army Commendation Medal with V Device (two awards) and the Gallantry Cross with Silver Star.

He is married to the former Geri McDonough of Albany, NY. They have four sons: Bruce, Gary, Douglas and Andrew. Bruce and Gary are both in the U.S. Army.

OPENING PANEL

Ada POLICY, PRACTICES, AND INITIATIVES



Mr. John H. Sintic
Acting Director, CECOM center for
Software Engineering, U.S. Army
CECOM, Ft. Monmouth, NJ

John H. Sintic assumed his current position as Acting Director, CECOM Center for Software Engineering (CECOM CSE), on 1 April 1988. The CECOM Center for Software Engineering is the single CECOM focal point for providing software life cycle management, software engineering and software support to Mission Critical Defense Systems (MCDSs) used in strategic and tactical Battlefield Functional Areas (BFAs) supported by CECOM. The CECOM CSE is also the Army/Army Material Command focal point for Computer Resource Management (CRM), Advanced Software Technology (AST), Ada Technology, Joint/Army Interoperability Testing (JAIT), and software quality and productivity.

Mr. Sintic has been with the Center since December 1983. Prior to his present assignment, he served as Deputy Director of the CECOM CSE. He also served as Associate Director, Computer Resource Management and Software Engineering Support, CECOM CSE. Mr. Sintic has over 25 years of experience in the field of software and computer technology.

Before joining the Center, Mr. Sintic was Chief of the Engineering Division, Joint Interface Test Force/Joint Interoperability Tactical Command and Control Systems (JITF/JINTACCS) from 1978-1983. In this position, he directed engineers and computer scientist (military and civilian) in the research, development and engineering for Joint Service interoperability of Command, Control and Communications Systems. He served as project manager for the development of the Joint Interface Test Systems (JITS) - the world's largest distributed command and control interoperability test bed developed to eleven Joint Service/Agency test sites.

Mr. Sintic has a Bachelor of Science degree in Computer Science. He is involved in many civic functions and is currently serving as Vice President, Ocean Township Board of Education. He is also a member of the Monmouth College High Technology Advisory Board.

Mr. Sintic and his wife, Trudy, have four sons - John, Drake, Todd and Jimmy. They reside in Oakhurst, New Jersey.

OPENING PANEL

Ada POLICY, PRACTICES, AND INITIATIVES



MG Peter A. Kind
Program Executive Officer
Command and Control Systems
Ft. Monmouth, NJ

Major General Peter A. Kind is a native of Wisconsin. Upon completion of studies at the University of Wisconsin in 1961, he was commissioned a Second Lieutenant and awarded a Bachelor of Science degree in Economics. He also holds a Master of Business Administration from Harvard University. His military education includes the Basic Officer Course at the Signal School, the Communications Officer Course offered at the U.S. Marine Corps Amphibious Warfare School, the U.S. Army Command and General Staff College and the U.S. Army War College.

He was assigned to the 97th Signal Battalion (Army), 10th Special Forces Group (Airborne) in Germany and as Signal Advisor to the 21st Infantry Division (Air Assault) in Vietnam.

Following duty as Assistant Division Signal Officer of the 82d Airborne Division and as Executive Officer and S2/S3 (Intelligence/Operations and Training) for the 82d Signal Battalion, Fort Bragg, North Carolina, he served in the War Plans Division of the Office of the Deputy Chief of Staff for Operations and Plans, Headquarters, Department of the Army. He commanded the 1st Cavalry Division's 13th Signal Battalion, Fort Hood, Texas and studied at the Logistics Management Center's School of Management Science. General Kind then served as Chief of the Concepts and Studies Division, Directorate of Combat Developments at the Signal Center prior to Army War College attendance.

He then served as Comander of the 1st Signal Brigade with concurrent duty as the Assistant Chief of Staff, J6, U.S. Forces in Korea and G-6, Eighth U.S. Army; as Director of Combat Development, and as Deputy Commanding General and Assistant Commandant, U.S. Army Signal Center and School, Fort Gordon, Georgia. He served as Deputy Controller of the NATO Integrated Communications System Central Operating Authority, NATO's equivalent to the U.S. Defense Communications Systems, headquartered at SHAPE, Belgium, prior to his appointment as Program Executive Officer, Command and Control Systems.

General Kind has been awarded the Legion of Merit, the Bronze Star Medal (with two Oak Leaf Clusters), and the Meritorious Service Medal (with two Oak Leaf Clusters). He is also the recipient of the Air Medal with two device and the Army Commendation Medal, the Senior Parachutist Badge and the Army General Staff Identification Badge.

He is married to the former Sandra L. Hanson. They have a son, Lee.



NINA G CORNETT

ACTING DIRECTOR
INFORMATION RESOURCES MANAGEMENT
DEPARTMENT OF NAVY

Mrs. Cornett became the Associate Director for Department of Navy Information Resources Management in 1989. She is responsible for oversight of the two billion dollars Navy spends annually in the commercial computer and business software area. That Department of Navy for commercial embedded computer systems, dealing with Congress and other federal agencies, and assuring that the Department's resources in the area of Information Resources Management are well spent.

Prior to her appointment to the Senior Executive Service, she held a series of increasingly responsible positions in computer and information management for Navy. She has also worked for the Department of the Air Force in the computer field and for the Department of the Army as a budget analyst.

Mrs. Cornett has a Bachelor of Science degree in physics, with a minor in mathematics and public administration in addition to a Master's degree in business, with concentration in management information systems.

She has received a number of awards and citations, including military meritorious unit award for work at the Sacramento Air Logistics Center.

Mrs. Cornett is a published novelist. Her 1971 book, Alaska Summer, was published in hard cover by Avalon books and in paperback by Dell Publishing Company.

Mrs. Cornett was born in Davenport, Virginia and grew up there and in Southern West Virginia, where her parents, Mr. and Mrs. Walter Crabtree, still live. She is married to Dean Cornett and resides, with him, in Alexandria, Virginia.

OPENING PANEL

Ada POLICY, PRACTICES, AND INITIATIVES



Brig Gen Albert J. Edmonds
Assistant Chief of Staff, Systems for
Command, Control, Communications, and
Computers, HQ USAF

Brigadier Albert J. Edmonds is Assistant Chief of Staff, Systems for Command, Control, Communications and Computers, Headquarters U.S. Air Force, Washington D.C. He is responsible for establishing communications and computer policy for the entire Air Force.

General Edmonds was born on January 17, 1942, in Columbus, GA., where he graduated from Spencer High School in 1960. He received a Bachelor of Science degree in chemistry from Morris brown College in 1964 and a Master of Arts degree in counseling psychology from Hampton Institute in 1969. He graduated from Air War College as a distinguished graduate in 1980 and completed the National Security program for senior officials at Harvard University in 1987.

The General entered the Air Force in August 1964 and was commissioned upon graduation from Officer Training School, Lackland Air Force Base, Texas, in November 1964.

General Edmonds was assigned to Air Force Headquarters in May 1973. As an Action Officer in the Directorate of Command, Control and Communications, he was responsible for managing Air Communications Programs in the Continental United States, Alaska, Canada, South America Greenland and Iceland.

In June 1975 the General was assigned to the Defense Communications Agency and headed the Commercial Communications Policy Office. General Edmonds was assigned to Andersen Air Force Base, Guam, in 1977, as Director of Communications-Electronics for Strategic Air Command's 3rd Air Division and as Commander of the 27th Communications Squadron.

After completing Air War College in June 1980, he returned to Air Force headquarters as Chief of the Joint Matters Group, Directorate of Command, Control and Telecommunications, Office of the Deputy Chief of Staff, Plans and Operations. From June 1, 1983, to June 14, 1983, he served as Director of Plans and Programs for the Assistant Chief of Staff for Information Systems.

General Edmonds then was assigned to Headquarters Tactical Air Command, Langley Air Force Base, as Assistant Deputy Chief of Staff for Communications and Electronics, and Vice Commander, Tactical Communications Division. In January 1985 he became he became Deputy Chief of Staff for Communications-Computer Systems, Tactical Air Command Headquarters, and Commander, Tactical Communications Division, Air Force Communications Command, Langley. In July 1988 he became Director of Command and Control, Communications and Computer Systems Directorate, U.S. Central Command, MacDill Air Force Base, FL. He assumed his present duties in May 1989.

His military decorations and awards include the Defense Distinguished Service Medal, Legion of Merit, Meritorious Service Medal with two oak leaf clusters, and Air Force Commendation Medal with three oak leaf clusters.

The General was named in Outstanding Young Men of America in 1973. He is a member of Kappa Delta Pi Honor Society and is a life member of the Armed Forces Communications and Electronics Association.

General Edmonds is married to the former Jacquelyn Y. McDaniel of Biloxi, MS. They have three daughters: Gia, Sheri and Alicia.



Dr. James A. Painter, SES
Special Assistant/Technical
Advisor, Director, Command. Control
Communications, and Computers

Dr. Painter is currently Special Assistant/Technical Advisor to Director, Command, Control, Communications and computer (C4) Systems Division, Headquarters, U.S. Marine Corps.

Prior to holding this position, he was Technical Director, Wide World Military Command and Control System (WWMCCS) ADP Directorate, Defence Communications Agency.

Dr. Painter has a Bachelor of Science and a Master of Science degree in Mathematics from the University of Pittsburgh. He has a Ph.D. in Computer Science from Stanford University. He has been on the faculty at University of California, Berkeley and at the University of Maryland. Dr. Painter has been active and in the reserves of the U.S. Army from 1946-1977.

Dr. Painter is a member of the Institute of Electrical and Electronic Engineers, Association for Computing Machinery, Mathematical Association of America, American Association for the Advancement of Science, the American Institute of Aeronautics and Astronautics and the Armed Forces Communications and Electronics Association. He is also the author and coauthor of more than a dozen papers and books.

Dr. Painter is married with 3 children and 1 grandchild.

BANQUET SPEAKER



MG Billy M. Thomas
Commanding General
U.S. Army CECOM
Ft. Monmouth, NJ

Major General Billy M. Thomas was born in Crystal City, Texas on 14 August 1940. He grew up in Kileen, Texas. Upon completion of the Reserve Officer's Training Corps curriculum and the educational course of study at Texas Christian University in 1962, he was commissioned a second lieutenant in the U.S. Army and was awarded a Bachelor of Science degree in Secondary Education. General Thomas holds a Master of Science degree in Telecommunications Operations from George Washington University.

His military education includes completion of the Signal Officer Basic and Advanced Courses, the Army Command and General Staff College, and the Army War College.

In addition to General Thomas' many important command assignments in Germany, Thailand, and Vietnam, he has also held a variety of significant staff assignments prior to his assuming the position of Commanding General, U.S. Army Communications-Electronics Command, among them were: Special Assistant to the Dean, National Defense University; Deputy Commanding General, U.S. Army Signal Center and School; Army Staff as the Deputy Director; Combat Support Systems, Office of the Deputy Chief of Staff for Research, Development and Acquisition.

Awards and decorations which General Thomas has received include the Legion of Merit, Bronze Star Medal, the Meritorious Service Medal and the Army Commendation Medal. He is also authorized to wear the Parachutist's Badge.

He is married to the former Judith K. McConnell of Boise, Idaho. They have four children: Jon, Kim, Kirsten, and David.

LUNCHEON SPEAKER



Lt Gen George L. Monahan, Jr.
Director of the Strategic Defense
Initiative Organization, USAF
Washington, D.C.

Lieutenant General George L. Monahan, Jr., is the director of the Strategic Defense Initiative Organization.

His education includes a Bachelor of Science degree from the U.S. Military Academy at West Point and a Master of Science degree in Electric Engineering. He is also a graduate of both the Army Command and General Staff College and the Air War College.

Later he worked on the Gemini space vehicle program and also flew aircraft missions in support of the testing of space vehicles and missiles.

In Southeast Asia, he flew 122 combat missions, 75 of which were over North Vietnam. Then he was assigned to activities in the Pentagon responsible for developing plans for enhanced night combat capability.

Later he was assigned to the Aeronautical System Division for a subsonic cruise armed decoy program.

Then he contributed to the development of the lightweight fighter aircraft prototypes, becoming the first Chief of the F-16 European Systems Program Office in Europe.

Subsequently, he was the Assistant Deputy Chief of Staff for Systems at the Air Force Systems Command and then Systems Program Director for the F-16 Multi-National Fighter Program.

Then he was assigned as Director of Development and Production, Office of the Deputy Chief of Staff for Research, Development and Acquisition, Headquarters, U.S. Air Force; then later becoming Vice Commander of the Air Force Systems Command.

In July, 1987 General Monahan became the Principal Deputy Assistant Secretary of the Air Force for Acquisition. He assumed his present position on February 1, 1989.

Introducing Ada as a New Language in a Software Engineering Course

Timothy D. Korson
Department of Computer Science
Clemson University
Clemson, S.C. 29634-1906
(803) 656-5866, korson@hubcap.clemson.edu

Abstract

This paper outlines the problems of introducing Ada as a new language in a software engineering course. It then suggests a solution based on the use of Ada artifacts developed at the Software Engineering Institute.

Introduction

Ada has played an important role in promulgating many modern software engineering techniques. The explicit language support for data abstraction, information hiding, and generics, along with the general emphasis on reusability, maintainability, and portability within the Ada community have led to the close association of Ada with modern software engineering.

I have 10 textbooks on Ada sitting on my bookshelf. Every one of them has a section in the preface similar to the following [Somm89] "So this is not just a book about Ada, it is also a book about software engineering." The general idea is that the authors are using Ada as a vehicle for teaching software engineering.

In light of this, it has been suggested [Brooks87] that the largest contribution of Ada to the software community is **not** in any single language feature, nor in all of them together. There is nothing in the language that constrains a programmer to write good programs. In fact, many very bad programs have been written in Ada. Ada's greatest contribution is its association with modern software development techniques and the training in good design principles that usually accompanies training in the Ada language.

Most academic software engineering courses include a project as part of the course requirements. Because of the close association of Ada with software engineering, it is natural for software engineering instructors to want to use Ada as the implementation language for the project. With the increased availability of Ada compilers, and typical educational discounts offered by the vendors, this option is increasingly feasible.

The Problem

The root of the problem is that very few colleges and universities require Ada in any courses that are prerequisite to their software engineering course.

Thus the instructor who tries to introduce Ada as a new language in a soft-

ware engineering course ends up with too much material to cover. He must not only cover the expected classical software engineering content, but now must also cover the syntax and semantics of Ada, as well as the specifics of how to use Ada to achieve the goals of modern software engineering.

And he must do that in spite of the fact that:

1. Ada is a large enough language that an entire course could be devoted to merely learning the syntax and semantics of the language.
2. Another whole course could be devoted to portability, reusability, and object based design in Ada.
3. Software engineering is much more than design and implementation. A course in software engineering is also expected to cover analysis, specification, project planning, project management, testing, other design techniques, proofs of correctness, CASE tools, and quality assurance.

Instructors who have tried to accomplish all 3 of the above points in one course have often found that only point 1 was in fact accomplished.

Yet this is not acceptable for an academic course in software engineering. Catalog descriptions for such courses read much more like point 3 than point 1.

In fact, this problem is generalizable to other courses. A data structures course provides another good example. Most of the programming assignments in data structures could profitably be implemented with generic Ada packages. Here again, however, the course has so much content of its own that if the instructor tries to also lecture on Ada, there will be too much material to cover.

What then, does the instructor who wishes to use Ada, do when faced with students who have no prior knowledge about Ada?

In the next sections, the proposed solutions focus on introducing Ada as a new language in a software engineering course. The solutions are, however, generalizable to other courses.

The Typical Solution

This Solution avoids the problem of trying to cover to much material in the lecture, by simply requiring the students to learn most of Ada on their own. This is possible because even though that students may not have had exposure to Ada, they certainly will have had instruction and experience in some procedural language, most likely Pascal or Modula 2.

Thus there is a subset of Ada with which the students are essentially already familiar. To extend this subset the instructor dedicates 2-4 lectures to those features of Ada which do not have a counterpart in Pascal. The students are then left on their own to learn anything else they need to know about Ada in order to complete the class requirements. In addition, Ada is used whenever code is employed to illustrated some aspect of the course content.

In a software engineering course some time is spent on design issues such as data abstraction, information hiding, and reusability. These issues give natural rise to the discussion of Ada features such as limited types, the package construct, and generics.

In addition to the instruction outlined above, the project is required to be implemented in Ada.

If the instructor can resist the temptation to spend too much class time

on Ada syntax, this approach can work. The problems with it include the following:

1. Students may end up spending much more time coding and debugging the project than is warranted. Conceptually, a new language may be easy to learn, but there are always subtleties with a language and compiler that can cost a student hours of frustration when trying to push a newly learned language to its limits. This is especially true with a complex language like Ada.
2. Some students may not be able to cope with learning the details of Ada on their own in addition to the standard course materials. This may push the drop rate to higher than desired. Students who do succeed may complain that they signed up for a course in software engineering, not in Ada.
3. With so little instruction in a language with so many new constructs, the Ada code that the students do write may turn out to be quite poor.
4. Some software engineering material will have to be condensed or left out to leave room for the lectures on Ada.

In spite of these problems, Ada can be successfully introduced as a new language in a software engineering course.

The problems don't go away, but by being aware of them one can keep them from getting out of hand.

The next section describes how these problems can be minimized by preparing course materials and assignments that explicitly deal with the problems.

A New Approach

For the class project, require students to modify an existing Ada program that is reasonably well written and exceeds 5000 lines of code. Do this in at least three phases.

First give them the complete, working source code and documentation for the existing program and have them install, compile, and produce the executable and associated libraries. During this phase the students will learn the mechanics of how the environment works without getting bogged down in syntax errors. If the compiler does not provide a tool to do so, you should provide them with a valid compilation order.

Next have them make a change that is trivial to code, but that requires them to read much of the existing code and have a basic understanding about how the program works. It is my hypothesis that this kind of exercise is the most efficient method of getting students quickly up to speed in Ada. In my experience it also is the method most enjoyed by the students. In addition it helps prepare them for the rest of the project.

Finally have them do the major part of the project which should be a substantial enhancement or modification to the existing system (source code and documentation). I propose that one of the best and most efficient ways to teach students correct coding practices and good design techniques is to require them to work with source code that implements both.

Of course, when lecturing on design issues such as data abstraction, information hiding, and reusability the instructor will still want to discuss the corresponding Ada features. But given access to the appropriate reference manuals, all of the above assignments can be done

without a single lecture on Ada, so no class time need be lost.

Many of the other problems associated with the typical solution also disappear. It is easier to modify code written in a new language than it is to come up with entirely original code, so students are not as overwhelmed. Furthermore the code produced by the students will quite likely be better and employ more sophisticated Ada techniques. This possibility arises because of learning from and emulating the original source code.

The basic problem with this method is finding an Ada system around which to base the project.

This Ada system should exhibit several characteristics:

1. It should be of the correct size. My experience is that 5,000 - 10,000 lines of code is an appropriate range.
2. It should be reasonably well written.
3. The application domain must be easily understood by a wide range of students.
4. It should have an accompanying specification document, design document and user manual.

Such systems are not easy to find. One might try the Ada Software Repository [Conn87]. The problem there is that most of the systems have little or no documentation and are not always well written.

Fortunately, the Software Engineering Institute (SEI) has taken an interest in this problem.

Case Study

CPSC 472 is a software engineering course taught at Clemson University for upper level undergraduates. A few graduate students usually also register for the course under the number CPSC 672. Prior to the course, most students have had no exposure to Ada.

During the last year the Author has taught three sections of the course in which it was required that Ada be used for implementing a major project.

The solution that has proven successful for the author is to use existing materials [Engle89] packaged for this purpose by the Software Engineering Institute. These materials include source code written in Ada, some documentation, and a report containing suggested exercises around which a project can be based. The author has spent the last two summers as a visiting scientist at the SEI developing and packaging educational support materials and then using them in actual courses taught at Clemson.

I have had the most success in teaching the course when I have followed the plan outlined in the previous section. At the end of the courses I always ask for verbal feedback from the students as well as written evaluations. When asked to evaluate the environment in which the most learning occurs, all but 1 of the students have favored making a modification to a existing system rather than writing one from scratch. On those occasions when, in the software engineering course, I have asked students with no Ada experience to write Ada programs from scratch I have almost always encountered problems. As previously suggested, these assignments often result in excessive effort, student discouragement, and poor coding. Perhaps this is because I assign hard problems, but the point is that I can get away with assigning hard problems if I do it in the context of an existing system.

The material that I helped package at the SEI during the summer of 1988 [Engle89] has proved in practice to be

quite useful even though it does not meet all of the guidelines suggested in the previous section. It is of the appropriate size and problem domain, but it is not as well written as I would like, and the documentation is not complete. This material was produced by simply porting and packaging an existing Ada program taken from the Ada Software Repository along with documentation produced by the students in a software engineering class at Indiana-Purdue University and exercises developed at the SEI.

Conclusions

The use of an existing Ada system eases the introduction of Ada as a second language in a software engineering course. It not only frees the instructor from the need to devote lecture time to Ada, but it is a most effective means of quickly bringing students up to speed in Ada and then helping them to master the correct usage of advanced Ada features and design techniques.

My experience with the support materials that the SEI provides in this area has been positive. I would encourage other instructors to use their material.

Work is currently under way on the development of a model system, complete with documentation, that meets all of the guidelines suggested in this article. It should be available by December 1990.

Further information on SEI educational support materials is available from the author or Gary Ford, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213.

References

[Som87] Sommerville and Morrison, *Software Development with Ada*, Addison-Wesley, Reading, Mass., 1987

[Bro87] Brooks, "No Silver Bullet." IEEE Computer, April 1987

[Eng89] Engle, Ford, and Korson, "Software Maintenance Exercises for a Software Engineering Project Course." CMU/SEI-89-EM-1 Carnegie Mellon University, Feb. 1989

[Con87] Conn, *The Ada Software Repository and the Defense Data Network.*, Zoetrope, New York 1987

Tim Korson is an assistant professor of computer science at Clemson University in Clemson, South Carolina. He received his B.A. in French at Atlantic Union College, his MS. in Mathematics from Ohio University, and his Ph. D. in Computer Information Systems from Georgia State University. He has worked at the Software Engineering Institute as a Visiting Scientist for the past two summers. His current research interest is in object-oriented systems development.



TEACHING SOFTWARE ENGINEERING WITH ADA

Yen-Yi Wu

IIT Research Institute
185 Admiral Cochrane Drive
Annapolis, MD 21401

ABSTRACT

This paper describes how to incorporate the teaching of Ada into a software engineering course so that the students can gain an appreciation and understanding of the semantics of the Ada language and learn many of the Ada language constructs and the software engineering principles involved. Specifically, the advantages of using Ada in a software engineering course and an approach to teaching a 16-week semester course in software engineering using Ada are presented. This paper also discusses some of the Ada language features associated with software engineering.

INTRODUCTION

Although closely related, software engineering and the Ada language are usually taught separately in computer science departments. Students learning the Ada language for the first time often find themselves overwhelmed by the complexity of Ada's syntax and the semantics of the Ada language constructs. Because Ada was designed to be a software engineering language, any approach to the study of the Ada language without exploring the software engineering principles involved will result in disappointment. Although software engineering principles can be taught independent of any programming language, many software practitioners doubt that these software engineering principles can actually be implemented. To address this issue, we propose that software engineering be taught with a software engineering language, specifically Ada, so that students can successfully implement the software engineering principles that they learn.

Recently, teaching both software engineering and Ada in the same course has been addressed in the literature. For example, Booch [1] proposes teaching Ada with software engineering principles, and Wiener and Sincovec [2] provide an approach to teaching software engineering with a software engineering language.

This paper focuses on an approach to teaching software engineering with Ada. Our approach calls for teaching Ada language in a software engineering course. The major benefit realized from this approach is that the students will not only learn the techniques, tools, methods and procedures used in modern software engineering practices, but also learn the Ada language and how those software engineering principles can be implemented. We shall provide and discuss a course outline for a 16-week semester course on software engineering incorporating the teaching of Ada.

This paper begins with a generic view of software engineering, leading the readers to the subjects and issues usually addressed in a software engineering course. Next, an outline of a software engineering course incorporating the teaching of Ada is presented, and the contents of this course are briefly discussed. Then, major features of the Ada language that support and enforce software engineering principles are discussed. Finally, we evaluate the results of using this method and draw some conclusions.

A GENERIC VIEW OF SOFTWARE ENGINEERING

Many of the software systems developed today are among the most complicated human engineering products. Without properly applying the engineering techniques, software problems occur. Indeed the term "software crisis" has been mentioned frequently in the literature, referring to problems such as software not responding to users needs, software failures, difficulty of maintainance, software not being transportable. There is no single best way to solve these problems. However, by developing a methodology that includes techniques, tools, and procedures, we can establish an approach to software engineering that will alleviate most of these problems.

The major goal of software engineering is to enhance software reliability and software maintainability. According to ANSI/IEEE standards, software reliability is defined as (1) "the probability that software will not cause the failure of a system for a specified time under specified conditions" and (2) "the ability of a program to perform a required

function under stated conditions for a stated period of time". Likewise, software maintainability is the ease with which software products can be modified to correct faults, to improve performance, or to adapt the product to a changed environment.

To achieve these goals, applicable software engineering techniques, methods, tools, and procedures must be used in each phase of software development. In general, software development phases consist of a requirements analysis, specifications, design, coding, installation and testing, and maintenance. The requirements analysis and specification phases focus on what the software will do. How the software is to be implemented is the subject of design, coding, and testing phases. Finally, the software maintenance phase is concerned with three types of modifications: corrections, adaptations, and enhancements.

To a beginning student in computer science, the depth and breadth of software engineering principles may not be fully appreciated. Therefore, it is our position that a software engineering course should present the techniques, tools, methods, and procedures available in each of the above mentioned phases using a programming language such as Ada and emphasize how they can be used to enhance the software reliability and maintenance.

OUTLINE OF A SOFTWARE ENGINEERING COURSE INCORPORATING THE TEACHING OF ADA

The following course outline of a standard software engineering course that incorporates the teaching of Ada is presented for a 16-week semester with a format of forty one-hour lectures. It is assumed that the students have had no previous experience with Ada. The topics covered in the course are presented in the order given, and the number after each major topic indicates roughly the number of one-hour lectures needed to cover that topic.

Introduction to Ada (9)
- Basic syntax rules, Ada statements, types and objects, subprograms and packages
- Text input-output, exception handling, Separate compilation

Introduction to Software Engineering (3)
- Software engineering paradigm
- Software engineering overview
- Software engineering and Ada

Requirement Analysis and Specifications (3)
- Requirements analysis methods and tools
- The software requirements document
- Data flow diagram, structural chart

Software Design (6)
- Introduction to software design
- Design practices (general issues and methods)
- Structured analysis and design

Ada Generic Units and Tasks (3)
- Generic unit and portability
- Ada task and concurrent processing

Object-Oriented Design (2)

Real-time Design (2)

Programming and Coding Method (2)
- Languages and coding
- Language characteristics
- Choosing a language
- Good coding style

Software Testing (4)
- Unit testing
- Integration testing (top-down, bottom-up, depth-first, breadth-first)
- Testing techniques (white box testing, basic path testing, loop testing, black-box testing)

Software Quality Assurance (2)
- Software quality
- Formal technical reviews

Software Maintenance (3)
- Cost and category of maintenance
- Maintenance organization
- Software configuration management

DoD Standards 2167 (1)

DISCUSSION OF THE COURSE OUTLINE

The first nine lectures concentrate on the Ada language. The intent of these lectures is to provide the students with the basic syntax and features of the Ada language. We found that G. Bray and D. Pokrass [5] offer an excellent text book for exploring many of the software engineering issues involved in Ada language. We propose that the package of Text_IO be covered to include file-oriented input/output and text input/output and that the feature of separate compilation be covered so that programming assignments can be made to the students. Ada generic unit and task can be covered right before the object-oriented design and real-time design.

After the students are introduced to the basics of the Ada language, a discussion of fundamental software engineering principles follows. The materials for the software engineering topics listed in the outline can be drawn from Wiener and Sincovec [2] and Pressman [3, 4]. Other sources of information are available and can be used in the course.

In particular, we propose that the workshop material prepared by the Software Engineering Institute be used, where appropriate.

A major project should be assigned at the time the requirements analysis and specifications are covered. The project should focus on developing an application system starting from requirements and specifications through the design and coding, complete with a user's manual. That is, students are asked to write a requirements and specifications document, a design document, a user's manual, and then are required to code and test the system as time permits. The design document should use the object-oriented design approach and the implementation should occur using a modular software construction approach proposed in Wiener & Sincovec [2].

Finally, at the end of the course, DoD Standard 2167 should be discussed so that the students know about the standards and practices of the Department of Defense in software procurement. Additional information about the current status of Ada usage or validated Ada compliers can be found in the Ada Information Clearinghouse Newsletter.

ADA LANGUAGE FEATURES ASSOCIATED WITH SOFTWARE ENGINEERING

Since the essence of this course is to demonstrate that software engineering principles can be implemented using Ada language, it is essential that the Ada language features associated with the software engineering principles be discussed. We shall discuss these Ada features in the context of software reliability and software maintenance. Our goal is to let the students realize that these features do support and enforce software engineering principles. Examples that illustrate the use of these language features should be given whenever it is appropriate to do so.

Naming Objects and Subprograms

A general rule of good software engineering is to write readable programs. That is, using descriptive names for various program entities such as constants, types, variables, and subprograms improves readability and supports maintainability. The Ada language permits the programmers great flexibility in the length and choice of symbols for naming the program entities. Ada reserved words are also English-like words for ease of reading.

Control Statements

Certain kinds of statements, such as if-then and go-to statements can divert the natural top-to-bottom ordering of reading in English if they are used improperly. Ada provides a full set of structured control statements, including if statement, loop statement, and case statement, each with additional constructs such as "elseif", "exit", and "other" to ensure that the destination of the branch taken is clear.

For example, by using the "elseif" construct, additional Boolean conditions can be determined at the same level as the first condition following the if-clause, enabling the statement that follows the first condition that is evaluated, if true, to be executed next. This construct is easier to read than the nested if-then-else statements.

Another example is the use of "other" reserved word in a case statement, which ensures that all the possible values of the case selector are considered. Without this construct, branching may be arbitrary if not all the values of the case selector are specified.

Object Declaration and Type Checking

Unlike other high-order languages such as Basic or FORTRAN, where an object can be implicitly declared, Ada requires that every object to be used in a program be explicitly declared and have a type. A type in Ada is a set of values together with operations to be performed on these values. The declared object may have associated with it only the values and the operations permitted for the type. The Ada compiler enforces these restrictions to prevent certain widespread mistakes during compiler time.

Ada's strong typing also protects the program during program execution. Ada compiler generates error-checking code during run time so that assignments to various data objects do not violate these type and range constraints. They ensure internal program consistency and error-checking against erroneous data inputs.

Data Abstraction and User-Defined Types

Many programming languages do not provide a means to define new data abstractions beyond the predefined data and control structures. Data abstraction, a major technique for building complex structures, permits the user to define a type and specify the operations that are meaningful to it. Ada permits the programmer to create data types that are tailored to a particular application. In addition to the predefined types, such as integer, floating point, character, string, and Boolean, Ada provides enumeration, real, array, record, access, and other types. Furthermore, Ada permits the type representation to be hidden, or protected, from other parts of the program. This ensures the consistent use of the type and makes the process of changing the program less painful when the representation must be changed.

Information Hiding

The key to data abstraction is information hiding. Each program unit should hide the design decision or implementation detail from the rest of the program. Information hiding facilitates maintainability since there is no chance that hidden information can be corrupted by a program unit that is not entitled to that information. To facilitate information hiding, Ada provides two types of data abstractions, the data object abstraction and the data type abstraction.

Ada package offers a better means to hide information and representation. Packages are used to group logically related data and associated operations into a single abstraction. The user of a package can use only the declarations that appear in the package specification. The user is prohibited from using the underlining details in the package body. This technique of hiding objects in a package body and making visible only the well-defined operations to access the object is called data object abstraction. The advantage of hiding objects in a package is to ensure program reliability, to free the user of the data object from implementation detail, and to simplify maintenance and enhancement.

To accomplish data type abstraction, Ada provides a private type for the creation of new abstract data types with associated operations in a package. The users of the package are free to declare objects of this new type. However, the operations on these objects are hidden in the private part of the specification.

Subprogram and Reliability

A subprogram is a block of codes, separate from the main body of a program that usually performs a specific function. They are program units that can be called from other points in the program. Information exchange between the subprograms and the caller is accomplished by parameters. The rules governing the exchange of parameters and the scope of the subprogram can affect the reliability of the program.

In Ada, the subprograms are procedures and functions. The Ada subprogram rules permit the programmer to control the scope and visibility of the data objects. In particular, Ada permits subprograms to be nested to support the principle of locality (i.e., variables and other program entities should be declared textually close to where they are used). Since local declarations have a smaller scope, if the declaration is changed, the source text potentially affected by the change is small.

Another Ada subprogram rule that promotes reliability is the type-checking of the parameters, i.e., the type of the actual parameter must match the type of the corresponding formal parameter. Also the "in" parameter is a constant within the subprogram and may not be assigned a value or passed to another subprogram except as another "in" parameter. These rules ensure that the parameters conform with the intended use of the formal parameter.

Generic Program Unit and Software Reusability

There are obviously economic and technical benefits to reusing software components, especially those that describe a variety of data structures, such as stacks, lists, queues, and trees, and a variety of algorithms, such as those used for sorting or searching.

Ada provides generic program units to promote software reusability. Ada generic program units are templates used to create new program units by means of instantiating the generic formal parameters. An instantiation creates a new program unit by providing actual parameters for the corresponding generic formal parameter.

The generic program unit also permits common data structures and algorithms to be specified with more abstraction, hence making them more general. Using the library of generic program units, the programmer can avoid "reinventing the wheel" and concentrate on using the existing program units in the building of more useful systems.

Concurrent Processing

Concurrent processing offers an economical way to enable several processes to execute concurrently. It is an important feature in real-time processing and in multitask operating systems. To accomplish concurrent processing, non-concurrent language usually uses a technique to code the individual process, compile it, and place it into a system queue by the executive waiting to share the processor. This approach is implementation-dependent and therefore not portable. Moreover, no automatic checking is done to ensure that the parameters passed between the application process and the executive are correct. The advantage of Ada concurrent processing is that, since there is no need for the process to interface with the executive, Ada concurrent programs are portable.

Another issue involved in concurrent processing is the inter-process communication and synchronization. In the early days of computing, inter-process communication was

accomplished by reading and writing shared global variables. Later on, semaphore was used to avoid the problems associated with reading and writing shared global data. However, semaphores are low-level implementations that cannot be used in high-level programming languages.

By enabling the programmer to decompose a program into concurrent processing units known as tasks, Ada facilitates inter-task communication and synchronization with entry and accept constructs. A rendezvous occurs when a task has called another task entry and the called task has reached a corresponding accept statement. During the rendezvous, the calling task is suspended while the called task executes the accept statement. Furthermore, the parameters passed between the calling task and called task are checked to ensure that they match in number and type.

Exception Handling

A reliable program must be able to check the existence of errors and provide a graceful alternative if an error is detected. However, due to the conditions of the error, it is infeasible to check all the errors. Programs that rely on the underlying operating system or hardware to detect and recover from errors are difficult to use and maintain.

Traditionally, error-checking is handled by a go-to statement and use of status flags. The go-to statement is used to transfer control to a program section to handle the error. This approach tends to render the program difficult to read, often proliferating unrestricted go-to statements; it also makes returning control to the calling subprogram very difficult. Because the significance of an error is known only in the calling program, the most effective recovery can be made at this higher level, not at the level where the error was detected.

Another way error-checking is handled is through the use of the status flag. When a called program detects an error, it sets an error flag and returns the flag to the calling program. The disadvantage of this approach is that the calling program must check the error flag each time it calls.

Ada provides exception-handling, an alternative way to name an error condition and transfer control to handle that error condition. Exception-handling provides a convenient mechanism for the programmer to define what kind of error is to be checked, and, when detected, where to transfer control and when to handle the error. If the error is detected and not handled, the error condition always propagates to the calling program up to the main program.

The advantage of Ada exception-handling is that error-handling can be isolated in an exception block rather than left to interfere with the normal processing flow. This improves the program readability. The other advantage of exception-handling is that the executing program cannot ignore the error condition. If the error is not handled, it propagates to the calling program up to the main program. Thus, the error cannot remain undetected, thereby causing either another error or incorrect results.

Separate Compilation

Another important programming language feature is the ability to compile a program as physically separate modules, i.e., a portion of a program that can be submitted as a unit to the compiler. Ada's separate compilation enables a program to be submitted in units with full compiler checking. The full-compiler checking ensures that the referenced subprogram units exist on file in precompiled form and that the parameters passed in module interface are consistent in types and number.

To accomplish these, an Ada compiler must retain certain information about the modules it compiled. The compiler uses this information to perform the same consistency checks between separately compiled parts of a program as it performs when the program is written as one single compilation unit. This feature permits the early discovery of errors that might otherwise go undetected until the separately compiled parts of a large program are combined.

The Ada language also provides rules to define when a change to one compilation unit requires recompilation of another module. These rules prevent the compiler from making a program executable if the needed units are outdated. This automatic detection of errors that are caused by outdated compilation units greatly simplifies maintenance.

CONCLUDING REMARKS

Software engineering principles are concepts and goals; means to demonstrate how they can be implemented in a programming language are crucial. We found that using Ada as a programming language in teaching software engineering provides an ideal approach to address this situation. Many of the software engineering principles such as data abstraction and information hiding can be explained easily with examples using the Ada language constructs. Examples to illustrate these principles are therefore vital for the students to understand the principles and to gain an appreciation of Ada's underlying design philosophy. This is the main reason

for our approach to teaching software engineering with Ada.

Another obvious benefit of our approach is that students get a chance to learn the Ada language in addition to the software engineering. It is not expected that the students, after taking this course, will become experts in the Ada language. Since the basic syntax and software engineering principles involved in Ada are covered, it will give them a good starting point to pursue further study or practice in Ada programming. During this course, the students were assigned numerous homework assignments to write Ada programs that incorporate the package, generic unit and task. Students anticipating programming assignments found this portion of the course exciting and were pleased that they had a chance to program in Ada.

As a result of teaching software engineering in an academic institution, we found that emphasis should be placed on providing documentation for the software. Computer science students seldom have a chance to write documents for software requirements, design, testing, and a user's manual. We firmly believe that documentation is part of the software and, therefore, cannot be overlooked. Not providing documentation for the software is one of the major sources of problems in software maintenance. In this course, a project was assigned to develop a software system from requirements and design to implementation, and to write a user's manual. We found that this part of the course work is an integral part of the instruction for a software engineering course.

REFERENCES

[1] G. Booch, Software Engineering with Ada, Second Edition, Benjamin/Cummings Publishing Co., Inc., 1986.

[2] R. Wiener and R. Sincovec, Software Engineering with Modula-2 and Ada, John Wiley and Sons, Inc., 1984.

[3] R. S. Pressman, Software Engineering, A Beginner's Guide, McGraw-Hill, Inc., 1988.

[4] R. S. Pressman, Software Engineering, A Practitioner's Approach, Second Edition, McGraw-Hill, Inc., 1987.

[5] G. Bray and D. Pokrass, Understanding Ada, A Software Engineering Approach, John Wiley and Sons, 1985.



Dr. Yen-Yi Wu has been a member of the technical staff with IIT Research Institute since 1977. He has an M.S. degree in computer science and a Ph.D. in mathematics, both from Pennsylvania State University. During 1988-1989, he was appointed a visiting assistant professor of Computer Science at the U.S. Naval Academy. Comments and suggestions may be addressed to Dr. Wu at: IIT Research Institute, 185 Admiral Cochrane Dr., Annapolis, MD 21401.

DESIGNING AN ADA TUTORIAL USING OBJECT ORIENTED DESIGN

by
Charles D. Marable and Carla C. Belgrave

Department of Computer Science
Hampton University
Hampton, VA

ABSTRACT

The Ada tutorial system was developed under a grant from the Department of Defense. Its purpose is to aid students and teachers in any course material outside the classroom environment. Our purpose in presenting this paper is to explain how we went about developing the program and how we integrated object oriented design techniques in the design process.

INTRODUCTION

The original Ada tutorial was written by The Morehouse Software Group in Pascal. That version was translated to Ada by the Hampton University Software Group. This was the version that we had as a basis for our efforts. We analyzed the given code, as well as the requirements analysis and detailed design, in order to derive its highlights and deficiencies. We then reevaluated the needs of the tutorial package (i.e. a more priviledged instructor and student user) to come up with a new detailed design. From this point, we were able to perceive the necessary abstractions to implement an object oriented design methodology.

OBJECT ORIENTED DESIGN

The basics of object oriented design call for the following:

(i) Problem definition.
(ii) Mapping of real-world objects to software equivalents by:
- -identifying the objects and their attributes
- -definition of operations on these objects
- -establishment of interfaces or relations between objects and their operations.

(iii) Implementation through detailed design.

PROBLEM DEFINITION

The Ada tutorial system should be an interactive system for instructors, students, and others who are interested in learning the programming language Ada. A tutorial module should exist to allow anyone to read text, examples, and try exercises on the material. A student module should do these same functions, but as requested as per the instructor's class schedule. A student should also be able to have examinations on the subject material and his grade should be updated in the instructor's log. The instructor module should have all the capabilities of the tutorial but also be able to modify classes, students, grades, as well as get a hard copy of the examination and answers.

DESIGN PROCESS

(i) Identify the Objects and their Attributes

The tutorial module is characterized by the operations that it undergoes. It prompts the user through menus for input to the terminal. In this manner, a user should be able to read through chapters and sections, looking at the examples of the material, and do the corresponding exercises. The module student should be able to also retrieve a class schedule from the instructor's schedule file, if it exists, as well as take scheduled tests as requested. The most important attribute of the student, however, is his name, represented as a array of characters. He should also have a correspondant class number, an integer. The instructor module also requires a name for identification purposes. Also associated with an instructor should be his class numbers, integers. Those class numbers should be associated with the students in the classes. The operations that an instructor should be able to perform are numerous. Since the instructor is in a privileged

mode, there must exist some sort of security to protect from users with malicious intent. A password must be used, therefore naturally that password should be able to be changed. The instructor should be able to access student information including the modification of student names, classes, and grades. He should be able to change, create, or delete schedules for classes. The instructor should also be able to print out the hard copies of the testing material, and have access to the original tutorial.

The above definitions of our objects came about after several revisions of the detailed design according to the our specifications of the Ada tutorial package. The total design process took up approximately two weeks worth of time during which no coding was done, only modification and revision of the design.

Implementation

The logical representations of our design were easy to code because of the length of the design process. We separated each module into an Ada package. The original tutorial code was revised and made more modular. Extractions from the revised tutorial module were made usable by other modules through Ada packages. The tutorial module was called Student_menus.Ada along with the rest of the Student module. The instructor module was implemented within the package Instructor_menus.Ada. Other packages were involved to do the file manipulation, which was the underlying structure of all of the tutorial. A further look at figure 1 of this paper will show the structure of the Ada_tutorial package. The package Global declarations.Ada supplied us with the necessary declarations to implement the names of users and global types needed. Graphics.Ada was used for the output of text to the terminal. Student_functions.Ada supplied us with the actual reading, testing, examples needed for the tutorial package. Validation.Ada was used to do error checking of all data.

TESTING AND VALIDATION

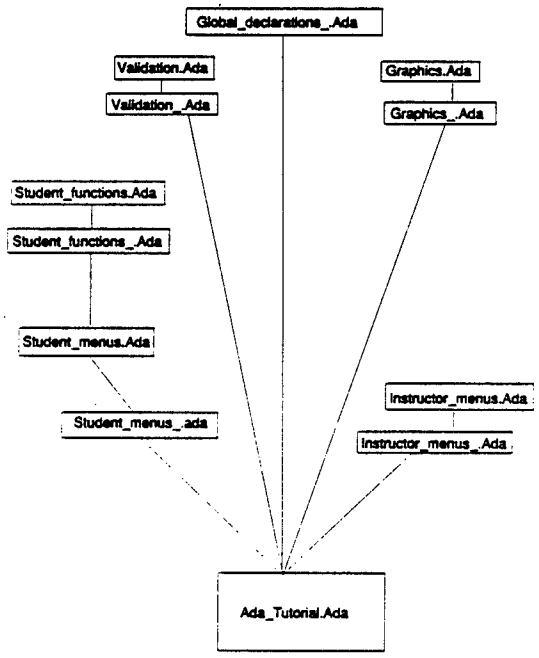
The testing phase of the software was composed of bad data formulation, illogical functionality of commands, visibility of modules, etc. Security validation was performed by using outside sources. Also, the package was tested by outside sources for user friendliness.

BIOGRAPHICAL SKETCHES

Carla C. Belgrave is currently enrolled at Hampton University and will be graduating in May of 1990. Her major is Computer Science with a minor in Mathematics. Her future plans include graduate school in Computer Science as well as a career in Software Design.

Charles Marable Jr. is also a senior Computer Science major at Hampton University. He plans to seek a graduate degree in Mathematics as well as a degree in Operations Systems or Software Engineering.

Dependency Diagram for The Ada Tutorial Package



(Note : Underscores denote package specifications, others are package bodies)

FIGURE 1.

THE DESIGN OF A PC-BASED CAI PROGRAM IN Ada

John H. Heidema

Telos Corporation, 55 N. Gilbert Street, Shrewsbury, NJ 07702

Abstract

This paper describes some Ada design strategies and implementation techniques that have produced a CAI tool called TEACHER for MS-DOS computers. The authoring language is briefly described, as well as the user-interface features of TEACHER. Several programming techniques are mentioned that have been shown to be effective for building window-oriented color user interfaces in Ada on MS-DOS systems.

INTRODUCTION

An immediate need arose within our corporation to produce a Computer Assisted Instruction (CAI) product for a military customer. The product needed to operate on MS-DOS systems with EGA color monitors, and we wished to build it in Ada if possible. This software needed to work immediately in one subject area, but it needed to be adaptable to full courses of instruction in a variety of subject areas. An ability to provide module tests was also required, and student performance needed to be monitored and recorded.

While this was basically a rapid-prototyping effort, our emphasis here is not on design methodologies. This paper concentrates on some techniques that we found effective for building CAI and user-interface software in Ada for MS-DOS computers.

Our design approach was to build something similar to a rule-based expert system shell in Ada. However, rather than following a set of rules, the Ada "shell" program would be driven by instructional text (and diagrams) written by an instructor knowledgeable in the subject area using a simple authoring language.

Our resulting program in operation presents the student with screens of text. A course menu and simple one-key commands permit the student to move back and forth through the text screens, much like moving through the pages of a book. But, unlike a book, the software also permits very interactive questions at any point, with individualized feedback based on the student's answers. And lessons can be followed by sets of quiz or exam questions. Test answers can be graded, and correct answers can be reviewed by the student. A simple database of student usage patterns, individual progress, and test results can be maintained.

THE USER INTERFACE

A typical example instructional screen is shown in Figure 1. A top window displays the date and time in the corners, with a title in the middle. A bottom window provides the appropriate keypress menu for the given screen. The middle window has room for 17 lines of topic text. (Given was a 25-line screen with 3-line windows at the top and bottom and with one blank line at each window border.) The 17 lines per screen proved to be plenty, though instructors initially tended to cram that space. Most effective and pleasing to student users were "information bites" of 7 to 15 lines, with ample borders on all sides.

We wanted the window frames shown and we also wanted color in our interface. We wanted to minimize assembly language and nonstandard Ada to achieve this. We knew that the nonASCII character codes (128..255) in MS-DOS systems could provide the window-framing graphics. But such "characters" cannot be printed in Ada without a range check error. Our eventually successful approach was to open the screen as a Sequential_IO file and write the byte-sized variables into this file at the proper positions to obtain the desired window-framing graphics. This approach, however, gives less portable Ada, since the file name needed for the screen tends to be compiler-specific and a way to declare one-byte variables also tends to depend on the compiler in use. A more portable way to get these frames in Ada is to "pragma SUPPRESS(RANGE_CHECK)" within the window-drawing routine. Then you should be able to Put the necessary nonASCII characters onto the screen.

We wanted text windows and cursor control. That was achieved by using the ANSI.SYS driver commands available on MS-DOS systems (see your MS-DOS manual). They become simple Text_IO.Put statements in Ada; e.g.,

```
Put (ASCII.ESC & "[2J"); -- clears the screen.
Put (ASCII.ESC & "[2;70H"); -- cursor to top right.
```

They will compile and run with any compiler. They will give the desired control on any MS-DOS system with the normal ANSI.SYS driver installed. These ANSI standard codes will also work on VAX and Sun terminal screens, and on HP terminals set to emulate a VT100.

Screen foreground and background colors are also obtainable using Put statements to control the MS-DOS ANSI.SYS driver. After testing user reactions to many different color combinations, we selected yellow frame boxes on a black background and yellow letters on a blue background for our screen colors.

User input was in the form of one-key commands or question responses. We achieved no-echo response to keypresses by calling a ROM BIOS routine. In one fashion or another, this is achievable using tools provided by each vendor of Ada compilers for MS-DOS systems. ANSI.SYS and this keypress control will also permit selecting a highlighted answer from a menu choice using the arrow keys, but that feature proved of little interest to our test users.

THE AUTHORING LANGUAGE

This language was intended to be easily usable by courseware developers who are not programmers, and it was intended to be relatively easy to parse. The author should be able to enter the text for screens of information or questions with a normal ASCII text editor. Since lines of text almost never start with a period, we thought that lines beginning with a period would be a good way to signal an authoring language command within the text file. (This technique is also used by several other well-known software tools.)

We decided to use clear English for authoring language commands, and also provide an alternative abbreviation. This approach led to commands (and their abbreviated alternatives) like:

.START TEXT	(.ST)
.END TEXT	(.ET)
.START QUESTION	(.SQ)
.END QUESTION	(.EQ)
.START MODULE TEST	(.SMT)
.END MODULE TEST	(.EMT)

There were also provisions for blank lines and comment lines.

Although we provided suggestions, we left format control within a screen up to the courseware developers (using their text editor tools). If less than 17 lines were specified for a screen, however, the software would fill the remaining lines with blanks.

While users have experienced a slow learning curve with some authoring languages, that has not been a problem in this case. The authoring language we so quickly designed has actually proved workable and fairly easy for a courseware developer to use.

THE CONVERSION/PARSER PROGRAM

We were aware that the sequential access required when using Ada's TEXT_IO to read the courseware developer's authoring language text file would be inadequate for some of our needs. Our user-interface shell program needed to move smoothly backward one text or question screen, and it would also need to move backward to the table-of-contents menu and forward to any component the student might select. We decided against storing all the text data in RAM, since that might cramp the space available to our object code or limit the size of the author's potential "textbook" of instructional material.

We therefore decided to write a conversion program that would make a Direct_IO file of text lines from the Sequential_IO authoring language textfile. As developed, our resulting CONVERT program is a syntax checker as well as a simple parser. It catches most syntax errors and ambiguities in an author's text and points to the line number where the problem has been found.

The CONVERT program creates a data file of 80-character records, one for each line of screen text plus some for control information useful to the user-interface program. The control information is in the form of short sequences at the beginning of lines that clearly could not be normal text, such as ";;;;" to signal the end of a text screen and "????" to signal the end of a question screen.

We recognized that using 80-character lines for each line of text and each control code was somewhat wasteful of disk space. (An authoring language text file gets converted to a Direct_IO text file of about double its size.) However, it makes the Direct_IO data file into a very useful tool for understanding and debugging, since a typical monitor scanning it also scrolls automatically every 80 characters.

The CONVERT program also creates a short file of integers as it scans through the authoring language text file. These integers point to the location in the Direct_IO text file where the table-of-contents menu is found and also to the beginning of each module selectable from this menu. No other location pointers

proved necessary for the user-interface shell program, since the starting points for previous screens and the location of the module tests can be found quickly by simple linear search for the appropriate control codes.

The logic of the CONVERT program is fairly simple. It either passes text lines straight over into a Direct_IO file or it interprets a command. The command interpreter is just a series of case-like elsif options, one for each authoring language command and none of them very complex. (It is a good example of code that is rather straightforward to a programmer but that is considered very complex by a typical cyclomatic-complexity tool.)

THE CAI USER-INTERFACE SHELL

Our CAI shell program has been called TEACHER (TELOS Easy Author for Computer-Hosted Educational Requirements). A set of instructional modules using TEACHER typically begins with some introductory screens (or "pages") followed by a one-screen Table of Contents, or master menu. The course modules follow, and the students may select any one of them (with the first module as the default).

While the flow of events within a TEACHER instructional module is determined by the courseware author, a typical module begins with some screens introducing the topic and ends with a module test. Students can be given the option to jump to the module test immediately if they are already familiar with the topic. Within the module, students progress through text screens (pages) of information much as with a book. They may also move backward one or several screens. Information screens are typically interspersed with interactive questions, where students receive positive feedback that depends on their particular answer choices.

The module test questions are like the other questions, except that students move through these questions without interactive responses to their answers. All TEACHER questions are in multiple-choice format, where any number of answer choices (two to nine) are possible. On completing the module test, students are told their scores (0% to 100%), and they are given the option to review the test to compare their answers with the correct ones.

Students may exit a session at any time by pressing the ESC key (or CTRL C). This produces an exit screen, from which they may go to the main menu, return to the lesson, or quit the session. On quitting, students have the option to mark their exit position to permit resuming at that point later.

Operationally, the TEACHER program simply prints text lines into the central window as it reads them from the Direct_IO file created earlier by the CONVERT program. (CONVERT also prevents authors from building screens of more than 17 text lines.) The end-of-screen control codes produced by CONVERT lead to various types of code-dependent processing by TEACHER. Thus, these control codes guide TEACHER to the right response to student answers; the control codes determine what prompts TEACHER will place at the bottom of each screen (the author's text screens may also contain instructions); and the end-of-screen control codes bring TEACHER into a wait-for-user-keypress mode. User keypresses are then processed to determine the subsequent direction of flow through the Direct_IO file of text lines.

A screen graphics package encapsulates the screen layout tools (see the user interface discussion earlier) that are used to produce the Exit and other special screens. We have successfully built screens that combine text with block graphics diagrams, by opening the screen as a sequential file and sending it certain MS-DOS graphic characters whose ASCII-type codes are greater than 127. However, this has so far only been done with TEACHER using a hard-coded Ada procedure stored in the screen graphics package and callable with an authoring language command.

DATA FILES AND TOOLS

We have mentioned the short file of integer pointers created by the CONVERT program that TEACHER uses to find the main menu and the beginning of each module within the Direct_IO file of text. The TEACHER program also produces or updates two other data files.

One data file stores records of student usage statistics by any appropriate student ID. Besides the number of times the student has used TEACHER and the last usage date, the record also stores the student's last exit point, which permits resuming the lesson later at that point.

The other data file stores module test score data for each student, including last test completion date, last test score, and number of times the student has taken the module test.

Simple program tools are also available (to instructors) that can read the usage data and the score data and print it out on request in tabular form for later inspection.

CONCLUDING OBSERVATIONS

We have shown that an approach like that employed with rule-based expert systems is an effective way to build a CAI shell program driven by authoring language text for MS-DOS systems in Ada. And we have implemented a convenient authoring language to control the TEACHER shell program.

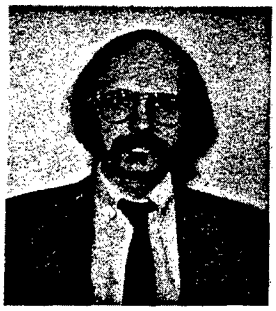
We used rapid prototyping of Ada implementation alternatives at several points to improve the efficiency and appearance of this CAI shell program. We found the ANSI.SYS screen control commands to be an effective set of tools for building an attractive, window - oriented user interface in Ada under MS-DOS. We were able to access the nonASCII MS-DOS graphics characters as necessary by opening the screen as a Sequential_IO file and writing the desired 8-bit values to it. We found no-echo responses to user keypresses to by very feasible in Ada on MS-DOS systems.

Our TEACHER CAI program runs on all MS-DOS systems. It works best on systems with EGA-level color graphics running at 8 MHz or better. No hard drive is required.

ACKNOWLEDGEMENTS

Mr. Kevin Derheimer contributed several ideas to the screen design and to the authoring language. Mr. David Hendrickson made several useful suggestions during his extensive work with the authoring language, the CONVERT program, and the TEACHER user interface. Many others offered useful user-interface suggestions as they tested various instructional modules.

ABOUT THE AUTHOR



John H. Heidema is an Ada training analyst and software engineer with TELOS Corporation. He has had active interests in Ada design and coding methods, programming language comparisons, artificial intelligence, and computer graphics. He has received a B.S. in Chemistry from Upsala College, and M.S. and a Ph.D. in Chemistry from the University of Chicago, and an M.S. in Computer Science from the George Washington University.

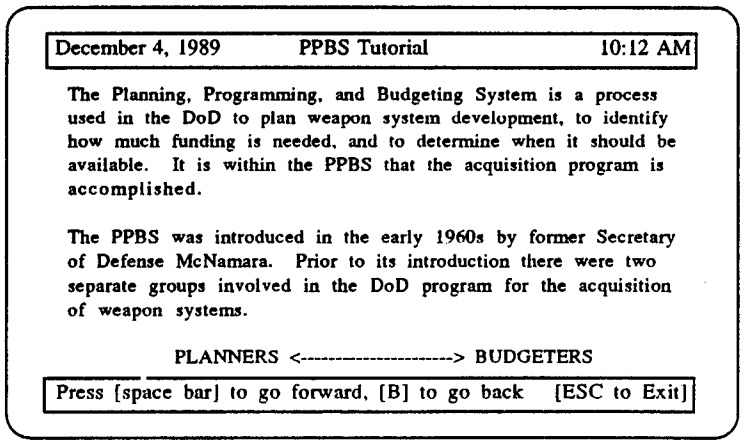


Figure 1.

A FIRST PROGRAMMING EXERCISE USING ADA TASKING: THE ODOMETER

Maj Jay Joiner, Dept of Computer Science

U. S. Air Force Academy, Colorado 80840

Abstract

This paper describes the specification, design, and implementation of a programming exercise that is intended to be a student's first exposure to concurrent programming in Ada. As such, it includes as many of the facets of Ada tasking as practical while remaining as concise as possible. After discussing the specification and design of the exercise, each major component of the implementation is analyzed with emphasis placed on the specific tasking constructs needed to implement the concurrency. Finally, the paper comments on techniques for validating the student exercises.

Introduction

The purpose of this paper is to describe a student exercise I developed for my real-time systems course at the U.S. Air Force Academy. My primary goal was to devise a fairly simple and clear programming exercise that could be used as a student's first exposure to concurrent Ada programming. The students in question had been exposed to sequential programming in Ada and concurrent programming in Pascal in previous courses, but this exercise was to be their first exposure to concurrent programming in Ada. I wanted to select an exercise that would illustrate most of the capabilities of the tasking model without involving too much coding. Furthermore, I wanted to choose a familiar system that wouldn't require an exceptionally complicated specification, and whose behavior could be comprehended without too much difficulty. With those criteria in mind, I came upon the notion of modelling the rotating counter wheel mechanism commonly used in automobile odometers. This system is certainly familiar to today's college students, and it involves the aspects of concurrent programming supported by the Ada tasking model I was hoping to include, namely multiple independent components, synchronization, and data transfer. This paper describes the specification, design, and implementation of this exercise. I will use a bottom-up approach in describing the implementation of the exercise, beginning with the screen I/O package, followed by the task specification and task body, and finishing with the main procedure body. A bottom-up exposition was chosen because code fragments included in the discussion of later units will reference routines from the earlier units. Finally, I will comment on validating the student exercises.

Specification

As we all know, the individual counter wheels in a typical odometer are independent of one another except for a synchronizing pin that trips the next wheel over when the preceding wheel's count reaches nine. In this exercise, the counter wheels are modeled by individual tasks. The least significant wheel task "rotates" freely, while the rotations of the more significant wheels are synchronized by rendezvous with the adjacent less significant wheels. Specifically, a less significant wheel "trips" its adjacent, more significant, wheel by a rendezvous when its count reaches nine. In order to incorporate data transfer between tasks as well as simple synchronization, all wheel task objects are declared from the same task type. As a result, each wheel task has to be told its position in the odometer, rather than have the information hard coded in the task body, through an initial rendezvous involving data transfer. Further enriching the design of the system is a requirement that the odometer accept a reset signal to set the count back to zero. Given this specification, the exercise turns out to be very rich in

terms of task interaction and permits quite a sophisticated design.

Design

The first step in the design is to define the terminology used to describe the system. The odometer consists of five *wheels* that display the decimal digits zero through nine as they rotate through a complete revolution. The wheels are arranged horizontally on the screen so the individual digits from each wheel combine to make up a five-digit decimal number. The wheels are identified by their position in this row. The rightmost wheel is the least significant digit in the number and is named wheel 0 (zero). The remaining wheels are numbered sequentially as you move to the left, so the leftmost wheel is the most significant digit in the number and is named wheel 4. This numbering scheme meshes nicely with the decision to store the task objects representing the wheels in an array indexed from zero to four.

The least significant wheel, wheel 0, "drives" the rest of the system. Once set in motion, this wheel rotates freely, not waiting on any external signals before rotating to the next digit. Every time wheel 0 reaches a count of nine, it must send a signal the next more significant wheel, wheel 1, to rotate to its next position. Then, wheel 0 can continue to freely rotate through another revolution. When wheel 1 reaches a count of nine, it signals wheel 2 to advance, and so on. Of course, wheel 4 should not try to signal any other wheel.

Wheel 0 is not totally independent, however. It must respond to two events from the external environment: a *reset* signal and a *kill* signal. Upon recognizing the reset signal, wheel 0 must cease rotating, reset its count to zero, signal wheel 1 to reset, and then wait until wheel 1 acknowledges the reset before continuing. It is crucial that wheel 0 wait for wheel 1 to respond before continuing, in order to ensure the odometer completely resets to zero before wheel 0 takes off again. As was the case with the advance signal, wheel 1 will signal wheel 2 to reset, then wait for an acknowledgment, etc. Also, as before, wheel 4 should not try to send the reset signal to any other wheel.

Upon receiving the kill signal, each wheel displays an asterisk where its count was and exits. I will now describe the individual program units which comprise the system, beginning with the screen I/O package.

The Screen I/O Package

One way to get students motivated is to have them do a programming exercise that produces an interesting display. Therefore, I wanted the program display for the odometer to look like a real one with the digits counting off as they increment. But, I wanted the students to concentrate on the tasking problem, not the I/O, so I provided a direct screen I/O package they could use in their programs. Fortunately, this is not too difficult with Ada's packaging machinery and program library concept. I provided the routines in library format along with the package specification so the students could simply include them in their programs.

Another Ada feature that came in handy for the screen I/O package was the *image* attribute that converts integer values to their corresponding character string images for output. It meant I only had to provide a routine to put character strings on the screen at specific X-Y coordinates, instead of both string output and integer output routines. I also provided a procedure to erase an entire line on the screen. This procedure could be called prior to displaying a new message to erase the old message on that line. The specification of the screen I/O package is:

```
package SCREEN_IO is
    procedure ERASE_LINE (LINE : NATURAL);
        -- Erases the entire line, where
        -- 0 <= LINE <= 24.
    procedure PUT (X, Y : NATURAL;
                   ITEM : STRING);
        -- Puts a character string on the
        -- screen at the specified X-Y
        -- position, where 0 <= X <= 79
        -- and 0 <= Y <= 24.
end SCREEN_IO;
```

An interesting aspect of implementing the screen I/O package arises because the screen is a shared resource. The two routines should only be called one at a time, and they should not be interrupted in the middle of a screen update. In a multitasking environment, more than one task could be calling the procedures in a package at the same time because the package mechanism by itself provides no means of synchronizing calls to its

routines. Before Ada, the way to protect a shared resource like this from multiple updates was to surround every such call with a lock/unlock pair. Of course this is unreliable since the programmer must remember to include these statements around every call. Fortunately, Ada provides a much safer solution by using a task in the library package to handle the synchronization. The visible part of the package specification includes normal procedure interfaces, but there is a hidden task in the package body with entries corresponding to those procedures. Calls to the procedures are translated into calls to the hidden task entries in the body of the package. In that way, the calls on the procedures are synchronized by the semantics of the select statement in the task body. The only drawback is that since the task is hidden in the body of the package, the client of the package is not aware it is actually dealing with a task and that procedure calls may be delayed.[1]

Including a task in the screen I/O package introduces another complication. It needs to be written in an infinite loop so it can continue to accept calls as long as they are needed. The problem is how to exit the infinite loop so the task can complete. Because this situation is so common, Ada includes a special select alternative, called the *terminate* alternative, to handle it. A task delayed on a select statement with an open terminate alternative will complete when its master is completed, and all other tasks dependant on that master are terminated or similarly delayed on their own terminate alternatives. The terminate alternative provides a clean and elegant way to write server tasks that will complete without resorting to awkward and unnecessary entry calls.[2] Let's now turn our attention to the individual wheel task type.

The Task Type Specification

From the specification for the odometer, we can list the required behavior of each wheel according to its position. Then by combining these behaviors, we can come up with a design for a single task type that can fulfill the needs for every wheel in the odometer. The first requirement is a mechanism for the main procedure to identify the individual wheel tasks. A suitable approach is to define an IDENTIFY entry with an *in* parameter representing the index of the task in the array of tasks comprising the odometer. We can also use this opportunity to pass in a screen location for each task to use for its output operations.

Next, every wheel, except wheel 0, is required to wait until it is explicitly advanced by the preceding wheel. This can be accomplished by an ADVANCE entry with no parameters. Similarly, the requirement that each wheel respond to reset and kill signals can be accommodated by RESET and KILL entries with no parameters. Thus, the task specification looks like the following:

```
task type COUNTER_WHEEL is
    entry IDENTIFY (NUMBER,
                    COLUMN : NATURAL);
    entry ADVANCE;
    entry RESET;
    entry KILL;
end COUNTER_WHEEL;
```

The Task Body

In writing the body of the task that will serve as the common code for every wheel in the odometer, we have to remember to account for the different behaviors of the wheels based upon their positions. For example, we can't have wheel 0 blocking on the ADVANCE entry, nor can we have wheel 4 calling the ADVANCE entry of a nonexistent wheel 5. We also need to provide each task with state variables representing its position in the odometer, its screen position for output, and its count value.

One of the first things we have to concern ourselves with is how tasks begin executing in Ada because the proper operation of the odometer depends upon the tasks knowing who they are when they begin. Unlike other concurrent languages that require an explicit statement to activate a task, Ada tasks are activated implicitly, immediately after the elaboration of the declarative part in which the task object was declared. Another way of saying this is that a task will begin executing when it first comes into scope. We therefore must cause each task to block immediately so it can receive its identification number from the main procedure. We can do this by beginning with an accept statement for the IDENTIFY entry. The main procedure can then call these entries in the order required to get the odometer started

properly. In the body of that accept statement, the two parameters will be copied into the state variables representing the task's identification number and screen position. Upon being advised of its screen position, the task should next display its initial count value. The initial code might look like this:

```
task body COUNTER_WHEEL is
    ID    : NATURAL;
    X     : NATURAL;
    Y     : NATURAL := 12;
    COUNT : NATURAL :=  0;
begin
    accept IDENTIFY (NUMBER,
                     COLUMN : NATURAL) do
        ID := NUMBER;
        X  := COLUMN;
    end IDENTIFY;
    PUT (X, Y, NATURAL'IMAGE (COUNT));
    -- The rest of the task body.
end COUNTER_WHEEL;
```

The main processing in the wheel task takes place in an infinite loop, which is often the case in real-time software. Recall that each wheel will be modeled by an instance of the same task type. Since the behaviors of the wheels are different according to their position in the counter mechanism, this must be accounted for in the select statement in the task body. By making use of guarded select alternatives, we can achieve the desired behaviors based upon the identification number. Remember that the wheel objects are stored in an array indexed by zero to four, where wheel 0 is the least significant, and wheel 4 is the most significant. Since the least significant wheel in the odometer spins freely and doesn't wait to be "tripped" by any other wheel, we don't want wheel 0 to block on the ADVANCE entry. We can achieve this by guarding the ADVANCE alternative in the select statement with "WHEN ID /= WHEEL'FIRST =>". Notice that when dealing with array objects in Ada, it is convenient to use the attributes associated with the index range instead of hard coding the index values. This practice makes the program more modifiable in case a different array index is desired.

Every wheel in the system should accept calls to their RESET and KILL entries, so these alternatives will be unguarded. Finally, in keeping with its unfettered behavior, only wheel 0 should not block if no entry calls are pending. All other wheel tasks must block, or the synchronization would be lost. This will be handled by using a guarded delay alternative with a duration value of zero seconds in the select statement. This allows wheel 0 to continue without delaying if there are no pending calls on its RESET or KILL entries. The resulting structure is known as the *selective wait* statement.[3] The skeleton of the select statement looks like the following:

```
select
    when ID /= WHEEL'FIRST =>
    accept ADVANCE do
        -- Advance processing
    end ADVANCE;
or
    accept RESET do
        -- Reset processing
    end RESET;
or
    accept KILL do
        -- Kill processing
    end KILL;
    exit;
or
    when ID = WHEEL'FIRST =>
    delay 0.0;
end select;
```

Notice the *exit* statement associated with the KILL entry. It allows the task to complete by exiting the infinite loop after the rendezvous completes. This is an alternative to the terminate statement mentioned in conjunction with the screen I/O package, and is the preferred method in this case because the tasks in the odometer are under the direct control of another task that can call the KILL entries. In the case of the screen I/O package, there was no program unit in charge of killing the task in that package. Let's now look at the processing required for each of the entry calls.

In the body of the ADVANCE entry, all wheel tasks, except the last one, need to conditionally call the ADVANCE entry of the next higher numbered wheel task if their own count is at nine, otherwise they simply advance their count by one. This can be accomplished with the following code:

```
accept ADVANCE do
    if COUNT = 9 then
        if ID /= WHEEL'LAST then
            WHEEL(ID + 1).ADVANCE;
        end if;
        COUNT := 0;
    else
        COUNT := COUNT + 1;
    end if;
end ADVANCE;
```

Notice we compute an index to the next higher wheel task in order to call the appropriate entry. This convenience is a direct result of Ada making a task a data type, thus allowing us to declare structured variables of the task type.

The really interesting problem is implementing the reset processing. As was hinted at earlier, we must ensure all wheel tasks get the signal to reset before wheel 0 is allowed to continue. The way to do this is with a multi-way synchronization.[4] Normally, a rendezvous is a synchronizing event only between the two tasks directly involved. The other tasks in the system are allowed to continue unaffected by it. In order to accomplish a multi-way synchronization, we make entry calls to the other tasks while still in the rendezvous with the first task. Remember that a rendezvous is not over, and the calling task is not allowed to proceed, until the called task reaches the *end* statement of the accept body. All we need to do is have every task, except the last one, rendezvous with the next higher task inside the same accept statement. As was done with the processing of the ADVANCE entry, we make use of array indexing to simplify the coding. The following code does what we desire:

```
accept RESET do
    if ID /= WHEEL'LAST then
        WHEEL(ID + 1).RESET;
    end if;
    COUNT := 0;
    if ID = WHEEL'FIRST then
        PUT (X, Y, NATURAL'IMAGE (COUNT));
        delay 1.0;
    end if;
end RESET;
```

The reason for the *if* statement following the assignment to COUNT is the way the program display is handled. Also, the delay statement for wheel 0 allows the display to pause so the observer can witness the resetting of the odometer before wheel 0 takes off again.

The processing required for the kill entry consists of replacing the count display with an asterisk indicating the task is terminated. The following short code fragment does the trick:

```
accept KILL do
    PUT (X, Y, " *");
end KILL;
```

This completes the discussion of the select statement in the task body. I will now describe the remainder of the processing in the task body's main loop, which is primarily concerned with advancing the count for wheel 0 and then displaying the updated counter values for every task.

The select statement in the task body synchronizes the advancement of wheels 1 through 4. Wheel 0 will drop straight through it in most cases, so we must now include the statements to advance its count, and conditionally call the ADVANCE entry for wheel 1 when its count is nine.

Finally, we need to display the updated counter value for each task. In the select statement, the count value will either increment in the body of the ADVANCE entry or will be reset to zero in the body of the RESET entry. Recall that we displayed each task's initial count value immediately after leaving the IDENTIFY entry and before entering the main loop, so we now have to display the new updated count value after coming out of the select statement. A simple call to the PUT procedure takes care of displaying the new value. The final processing in the main loop of the task body is:

```
loop
    -- Select statement
    if ID = WHEEL'FIRST then
        if COUNT = 9 then
            WHEEL(ID + 1).ADVANCE;
            COUNT := 0;
        else
            COUNT := COUNT + 1;
        end if;
    end if;
    PUT (X, Y, NATURAL'IMAGE (COUNT));
end loop;
```

This concludes the discussion of the task body. I will now describe the main procedure body.

The Main Procedure

The main procedure serves to drive the rest of the program in that it initializes the odometer, then delays for a period of time letting the other tasks execute. After delaying an initial time period, the main procedure sends the reset signal to the odometer and delays an additional period of time. Finally, it sends the kill signal to the odometer, ending the program. In this manner, the main procedure exercises each specific behavior of the odometer.

The first duty of the main procedure is to initialize the wheel tasks with their identification numbers and screen locations for output. Strictly speaking, the five wheel tasks could be initialized in any order without jeopardizing the concurrency, but it turns out the tasks should be initialized in reverse order for appearance sake. By initializing in reverse order, wheels 4 through 1 will display zeroes on the screen and enter their main loops awaiting the first call to the ADVANCE entry. Then, wheel 0 will start counting and call the ADVANCE entry of wheel 1 when its count reaches nine.

After initializing the wheel tasks, the main procedure executes a delay statement, taking itself out of the ready queue and allowing the wheel tasks to run by themselves. After the initial delay, the main procedure calls the reset entry for wheel 0, since the reset operation must begin there. Then, the main procedure delays an additional time period. Finally, the main procedure must terminate each wheel task by calling its kill entry. As was the case with the initialization calls, the kill entries could be called in any order. However, an in-order sequence allows us to take advantage of the handy *range* attribute for arrays. Following is an example of the main procedure body:

```
begin
    WHEEL(4).IDENTIFY (NUMBER => 4,
                       COLUMN => 35);
    WHEEL(3).IDENTIFY (NUMBER => 3,
                       COLUMN => 37);
    WHEEL(2).IDENTIFY (NUMBER => 2,
                       COLUMN => 39);
    WHEEL(1).IDENTIFY (NUMBER => 1,
                       COLUMN => 41);
    WHEEL(0).IDENTIFY (NUMBER => 0,
                       COLUMN => 43);
    delay 10.0;
    WHEEL(0).RESET;
    delay 10.0;
    for I in WHEEL'RANGE loop
        WHEEL(I).KILL;
    end loop;
end MAIN;
```

Packaging

Except for the stand-alone screen I/O package, the odometer program is coded in a single procedure. The task type and the array of five task objects will be declared in the declarative part of the procedure. The task body can either be included in the declarative part also or be made a subunit of the procedure body with a *separate* clause to unclutter the source text. The declarative part of the ODOMETER procedure might look like the following:

```
with SCREEN_IO;
use SCREEN_IO;
procedure ODOMETER is
    task type COUNTER_WHEEL is
        -- The entry declarations
    end COUNTER_WHEEL;
    WHEEL : array (0 .. 4) of
        COUNTER_WHEEL;
    task body COUNTER_WHEEL is separate;
begin
    -- Sequence of Statements
end ODOMETER;
```

Validating the Exercise

A primary concern when designing a graded academic assignment is to make it easy to score the work. In the dynamic environment of a multitasking software program, this goal is made even more difficult; as a result, I would now like to comment on the techniques used for designing a multitasking programming exercise to make it easier to validate.

First, try to design the exercise so the program's output display will provide sufficient feedback as to the proper operation of the program. This serves two purposes. First, it makes the program easier to grade. Second, I believe students are more motivated by a program which produces an interesting display rather than one that simply produces long tables of values. Doing this for a multitasking program absolutely requires a direct screen I/O facility. Given a direct screen I/O facility, you must then allocate different areas of the screen for each task to put its messages. Otherwise, messages will scroll off the screen, making it difficult observe the output over time. I recommend the instructor provide this design information for the students to standardize the output displays. Otherwise, you will have as many variations of screen layouts as you have students doing the project.

In the early phases of program debugging, have the tasks display simple status messages on the screen, even including a simple indication whenever they get scheduled on the CPU. Using this technique can quickly show you if a task is not being scheduled properly, or perhaps if another task is hogging the

CPU. Keeping these guidelines in mind will make your multitasking programs easier to debug as well as making them easier to grade.

By design, the odometer exercise is fairly easy to validate by observing the output of the program. By visual inspection you can determine if the synchronization is functioning properly. When I gave this assignment to my class of ten students, every one was able to correctly handle the synchronization requirements. In fact, most students wrote their task bodies using similar constructs to the ones I had used. This was easy to determine just by watching how high the odometer counted during the time it ran. One notable exception was an individual who misused the flexibility of the select statement to program a busy wait situation. His odometer reached a count an order of magnitude lower than everyone else's, graphically illustrating how much CPU time was wasted in those tasks not able to proceed.

Conclusion

This paper has described the specification, design, and implementation of a multitasking programming exercise for Ada. The exercise was designed to require the use of nearly all the tasking constructs in Ada, and still be manageable for undergraduate computer science students. Additionally, I talked about some guidelines for designing multitasking program assignments to make them both interesting to students and easy to grade.

Notes

[1]Alan Burns, Concurrent Programming in Ada (Cambridge: Cambridge University Press, 1985) 112-17.

[2]Grady Booch, Software Engineering with Ada (Menlo Park: Benjamin Cummings, 1983) 249-50.

[3]Theodore Elbert, Embedded Programming in Ada (New York: Van Nostrand Reinhold, 1986) 405-20.

[4]Burns 72-3.



Major Jay K. Joiner is an Assistant Professor of Computer Science at the U.S. Air Force Academy in Colorado Springs, Colorado. He has taught courses in Programming Languages, Computer Architecture and Real-Time Systems. He received his B.S. in 1977 from the Air Force Academy, and his M.S. in Information and Computer Science in 1987 from the Georgia Institute of Technology. Maj Joiner is a member of the ACM and is the Air Force Representative on the ANSI Joint Pascal Committee.

Author's mailing address:

Maj Jay K. Joiner
Dept of Computer Science
HQ USAFA/DFCS
USAF Academy, CO 80840

A STUDENT PROJECT TO DEVELOP A DISTRIBUTED FLIGHT SIMULATOR IN Ada

R.F. Vidale

Boston University, Boston, Massachusetts 02215

Abstract

This paper describes a student project conducted in the spring of 1989. The goal was to develop a flight simulator in Ada, distributed among four networked workstations. The project was part of a capstone course in the software engineering Master's program and made use of the new Embedded Systems Laboratory at Boston University. The fourteen students in the class were divided into four teams, each responsible for one of four technical areas: modeling and simulation, graphic displays, interfacing, and performance. At the end, a distributed simulation of a lunar lander spacecraft was working, and a prototype simulation of a business jet was running on a single workstation.

Introduction

The flight simulator project was part of a capstone course in the Master's program in software engineering at Boston University. A unique feature of this program is its emphasis on the specification and design of real-time embedded system software. This emphasis is put to the test in a capstone course, SC714 - The Computer as a System Component. The class is assigned a project to build software that meets hard-deadline timing constraints.

The reasons for choosing a flight simulator are as follows: A flight simulator has many of the distinguishing characteristics of a real-time embedded system. The simulator must update the vehicle's state and generate screen displays within a strict periodic schedule. Strict acyclic timing requirements are imposed by the inputs of the pilot and a flight instructor. In addition, a flight simulator provided the motivation of being able to "fly" it when completed, and it could be implemented with the hardware in the new Embedded Systems Lab.

At the start of the semester, the new Embedded Systems Lab had just become operational. It included nineteen networked workstations, two file/compute servers, five terminals, and six assorted printers. The fourteen students in the class held Bachelor's degrees in diverse fields: one in biochemistry, two in computer engineering, three in computer science, three in electrical engineering, three in mathematics, and two in physics. The first class in SC714 possessed the equipment and raw talent, but no experience with the lab or the application.

Approach

At the outset, we required answers to four basic questions: (1) how to digitally simulate vehicle flight dynamics; (2) how to interface the workstations; (3) how to produce CRT displays of control and instrument panels and scenery; (4) whether we could meet the hard-deadline timing requirements. In addition, the class had to gain working knowledge the Ada language. Our approach was to cover the Ada language and numerical methods on a class-wide basis, in a conventional lecture, lab exercise, homework assignment format. The balance of our approach was to use divide-and-conquor and phased-development strategies to build the required knowledge base. The class was divided into four teams, one to address each of the four major technology problems: A modeling and simulation team, a graphics team, an interfacing team, and an Ada performance issues team. The flight simulator was developed in two major phases: a lunar lander simulation, and a business jet simulation. The lunar lander was developed first to gain experience with a

relatively simple dynamic model before moving on to the more complicated aerodynamic equations.

Modeling and Simulation Team.

This team had the responsibility for writing differential equations for the vehicle flight dynamics, transforming them to discrete-time difference equations, and writing the Ada code to implement the difference equations. The implementation had to provide for asynchronous control inputs from the pilot. The team dubbed itself the "Crack Modeling and Simulation Team" (CMAST). In the first phase of the project, CMAST developed two versions of the lunar lander simulation. The initial version included the following features:

- Variable mass model
- Vertical motion only
- Main rocket engine only
- Lunar surface view, point display
- Fuel and altitude indicators

The final version added:

- Horizontal and rotational motion
- Attitude thrusters
- Two-dimensional image of the lander
- Attitude and velocity indicators

In the second phase, CMAST developed two versions of the jet aircraft. The initial version verified that the aerodynamic forces were being correctly computed in highly constrained flight (constant pitch angle). The final version included the following features:

- Motion in the aircraft's plane of symmetry
- Engine thrust control
- Elevator angle control
- Cockpit view of landing strip and horizon
- Altitude, air speed, and vertical rate of descent indicators

Graphics Team.

The graphics team was responsible for generating displays of the instrument panels and scenes for the lunar lander and jet aircraft. The graphics team adopted the motto "We show it like it really is." For the lunar lander phase, the team produced a scene of the lander viewed from the lunar surface, and a panel displaying data on fuel, throttle, horizontal and vertical velocity, pitch, and engine state. For the jet aircraft phase, the team produced a cockpit view of the airstrip and horizon, and an instrument panel showing airspeed, vertical speed, altitude and horizon (non-functional).

The graphics team solved two major problems: First, they developed the transformations required to project the airfield's outline and the horizon onto the pilot's viewing plane. The image was drawn using the graphics library provided with the system. Secondly, the team worked out the sequence of graphics commands to draw the instrument panels.

Interface Team.

The ("*crack*") interface team was responsible for for three major software packages, the instructor interface package for the lunar lander simulation, the pilot input package for the lunar lander, and the communication package for sending data between workstations.

The instructor interface package allows the flight instructor to control the simulation by a set of commands to set up initial parameters that define the state of the vehicle at time zero, terminate the setup, start the simulation, stop the simulation, and request a printout of the state of the lunar lander at touchdown. This last command provides the instructor with an asynchronous input that the simulation has to handle while it is the updating the vehicle's state. The instructor enters commands through a screen display driven by the instructor's keystrokes. The keystrokes, in turn, drive a finite-state machine which initiates actions according to the state it is in.

The pilot interface package reads pilot inputs that control the motion of the lunar lander. These include main engine thrust (off, on) main engine throttle (0 to 100%), and attitude thrusters (off, left, and right). The forward and backward movements of a mouse are used to

control the engine throttle. The middle key of the mouse is used to toggle the main engine on and off. Holding down the left mouse key turns on the left thruster; the right key turns on the right thruster.

The communication package defines a message type that is "withed" by all inter-process communication packages: A simulation sender package to send the current state of the vehicle, simulation receiver a package to receive the current state of the vehicle, a package to send an instructor message, and a package to receive an instructor message. The simulation sender package is used by the Ada program that performs the vehicle state update. The simulation receiver package is used by the Ada program that generates the lunar scene, the Ada program that generates the instrument displays, and the Ada procedure that produces the landing report. The send and receive instructor message packages provide for communication from the Ada program handling the instructor's inputs to the Ada program performing the state update.

Performance Team.

The performance team dubbed themselves the "Statistically Unachievable Performance Evaluation of Requirements (SUPER)" team. The team's initial responsibilities were to benchmark Ada constructs and programs provided by the other teams and to develop a scheduling mechanism for achieving the hard deadline requirement that the vehicle's state and displays be updated every 30 milliseconds. As the semester progressed, the SUPER team added the tasks of configuration management, system integration, system testing, and debugging. The burden of these tasks was increased by the strategy of pursuing the technical areas of modeling and simulation, graphic displays, interfacing, and performance in parallel. The achievement of a working distributed simulation was due in large part to the SUPER team's efforts.

The Lunar Lander Simulator

The distributed lunar lander flight simulator provides for interaction of a flight instructor and a pilot with the real-time simulation. This is portrayed by the simulator's context diagram, Figure 1. The context diagram defines the boundary between the environment and the simulator software, which handles interactions with the instructor and the pilot. The solid arrows in Figure 1 represent data flows; single-headed arrows represent data that is only available at discrete points in time, such as initial conditions provided by the instructor; double-headed arrows represent data that is continuously available, such as engine throttle control. Shaded arrows represent events that occur at discrete points in time and carry no data other than the occurrence of an even, such as a request for a landing report from the instructor. The context diagram follows the conventions of Ward-Mellor requirements analysis [WARD85] in which technology independence is preserved, e.g., the output data flow "instrument display," though in reality discrete-time updates every 30 milliseconds, is perceived by the pilot as continuous in time.

Figure 2 shows how the software was allocated to the four workstations that ran the simulator. A local area network carried the signals between the workstations.

The Jet Aircraft Simulator

The business jet flight simulator was the initial objective of the project. At the end a prototype simulator was running on a single workstation, not in real time. Using windows, a cockpit scene of the airfield and an instrument panel were provided. The only control available to the pilot was elevator angle, which controlled the aircraft's pitch. Control inputs were entered through the workstation keyboard. The groundwork had been laid, at least, for a distributed jet aircraft simulator.

Lessons Learned

At the end, it was evident that we had bitten off more than we could chew in one semester. It was also evident that partitioning the class by technical issues allowed rapid parallel development of the technology needed for the simulator, but that not enough attention was paid to project management functions, which became painfully evident as the teams tried to integrate

the results of their research. In retrospect, it would have proved more efficient at the end of the project to allocate more effort to project management functions at the beginning to enforce coding standards, interfacing standards, and configuration management throughout the project. Another possible approach in this situation would have been to treat the project as strictly a feasibility study and not attempt to integrate the individual team results into a working simulator the first time around. I believe, however, that a feasibility study by itself would not have motivated the class to accomplish all that it did. The principal lesson learned from this project is that even in a small-scale feasibility study the project management function cannot be neglected.

Acknowledgements

The author wishes to acknowledge the generosity of Digital Equipment Corporation, which established the Embedded Systems Laboratory with its gift of hardware and software. Also to be commended are the ground-breaking, diligent efforts of the first class in SC 714 to use this facility: Doug Brown, Lee Chan, Glenn Coleman, Hollis Dezieck, Charles Ellison, Gentry Gardner, John Herman, George Keereweer, Imran Mirza, Dan Sandini, John Schmeling, Chris Vaughan, Mary Ann Wassenberg, and Tim Yuhas. Special thanks go to Genry Gardner who brought the laboratory to operational status over the Christmas holidays and served as Lab Manager during the spring semester.

Reference

[WARD85] Ward, P.T. and S.J. Mellor, Structured Development for Real-Time Systems, Englewood Cliffs, NJ: Prentice-Hall, 1985.

Author

Richard F. Vidale has been a member of the faculty at Boston University since 1964, serving as Chairman of the Department of Electrical, Computer, and Systems Engineering from 1971 to 1981. He has been involved in applications of structured programming and software engineering since 1977. Dr. Vidale first began teaching software engineering and Ada in 1982. Since then, his research has focused on Ada design methodologies for embedded computer applications. Dr. Vidale has consulted in the areas of software engineering and Ada at C.T. Main, Inc., GTE, US Navy, MITRE Corporation, Data General Corporation, C.S. Draper Laboratory, Kollsman, and The Analytic Sciences Corporation (TASC).

Context Diagram for Lunar Lander Simulator

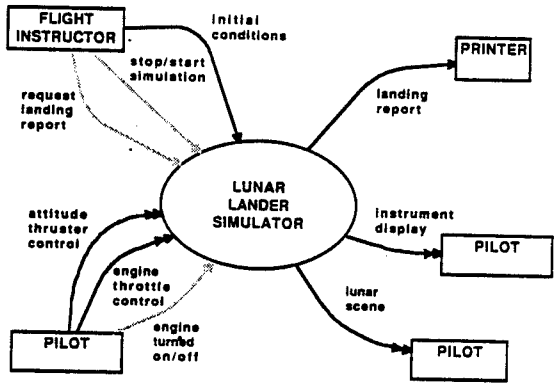


Figure 1

Allocation Diagram for Lunar Lander Simulator

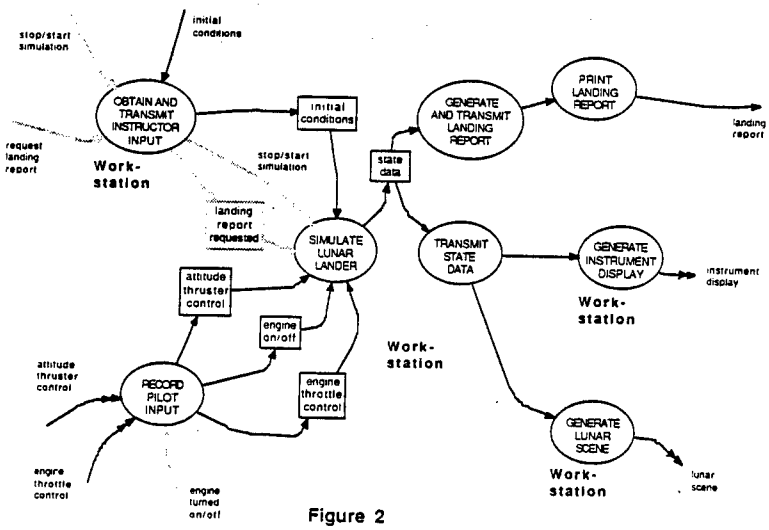


Figure 2

Ada and SQL : The Two Layer Approach

Judith D. Richardson
US Army CECOM
Center for Software Engineering
Advanced Software Technology
Ft. Monmouth, NJ

Dr. Thomas J. Wheeler
US Army CECOM
Center for Software Engineering
Software Engineering Research Laboratory
Monmouth College, NJ

Arturo B. Ferrer
US Air Force Systems Command
Rome Air Development Center
Command and Control Directorate
Griffiss AFB, NY

Abstract : The use of relational databases within Ada programs is an example of multiparadigm programming. The problems of using multiple paradigms center on resolving the impedance mismatch (or the differences in the data models and thought patterns of each paradigm) and how to reflect the differences across the language boundary. One technique is to build an interface between the two paradigms. The interface should strive to resolve the mismatches while providing access to those language features valuable to the problem solution. This means that semantic issues as well as syntactic issues must be addressed. Features of each language that contribute to the mismatch need to be identified and examined in order to develop a solution.

This paper describes a two layer interface architecture based upon the idea of abstract interfaces. The upper layer resolves semantic issues while the lower layer is concerned with syntactic issues. The lower layer hides the implementation details of the actual binding. This paper discusses the characteristics contributing to the impedance mismatch between Ada and Structured Query Language (SQL). It develops the ideas behind the architecture, outlines the design method for developing an interface, and illustrates the technique through an example.

INTRODUCTION

The move toward the use of standards has increased in recent years. The standardization on the use of Ada as a programming language and SQL for relational databases implies a need for the development of a standard interface between the two. However, there are incompatibilities in the paradigms of the two languages. Ada is a procedural programming language which takes advantage of strong typing, data abstraction, and complex data structures. SQL, on the other hand, is a query language for relational databases which uses simple data types and flat record structures (or tuples) providing a collection of persistent data. SQL is designed to allow efficient processing of declarative queries on large databases, something that cannot be readily done in Ada. Objects in Ada are accessed by naming the variables while SQL entities are accessed by describing properties of tuples. Ada programs also take advantage of software engineering techniques such as information hiding and modularity, which do not map easily onto SQL. Each language has strengths which should be made available to programmers. A program using both Ada and SQL is an instance of multiple paradigm programming. A layered interface architecture is one approach to resolve the mismatch which is inherent in programs using two different paradigms. A layered interface attempts to reconcile the differences and at the same time provide access to the features of both languages or environments. In the case of Ada and SQL, this approach provides an Ada program with access to the SQL querying mechanism in order to retrieve and modify persistent information in a database while hiding the syntax and details of SQL.

The Ada Joint Program Office (AJPO) recognized the need for a standard method of interfacing Ada and SQL and held a workshop to look into approaches [Wheeler 87]. The issues raised at the workshop centered on two areas: the architecture for providing access to database facilities and the binding method for transferring information across the boundary between the two languages. The consensus of the workshop participants was that a layered interface architecture was the "right" approach. Of the methods of binding that were presented at the workshop, the AJPO decided to pursue the module language method. This approach uses the module language specification in the ANSI SQL Standard [ANSI 86]. To establish guidelines for this binding, the AJPO established the SQL Ada Module Extension-Design Committee (SAME-DC) chaired by the SEI [Graham 89].

The work discussed in this paper grew from involvement in the above efforts. We were interested in exploring the needs of non-traditional database application domains as they relate to the use of Ada and SQL. In particular we were interested in addressing the needs of command and control systems. These systems tend to be large, database oriented programs that provide complex functions to the user. Also, the objects in these application programs are usually more structurally complex than what is found in traditional database applications. The approach of command and control system developers is to use object-oriented development strategies. We wished to form an interface solution that would support systems that use complex objects in an Ada application, but which currently need to store and maintain them in a relational database.

This paper will examine the general problems in interfacing Ada and SQL. It will compare the characteristics of existing data models and discuss the needs of non-traditional database applications written in Ada. It will examine the structure clash inherent in systems written in Ada using an SQL database. This paper will then describe a two layer interface architecture based on the idea of abstract interfaces and outline the design method for developing such an interface. The architecture will be illustrated through an example.

CONSIDERATIONS

Multiple Paradigms :

A paradigm is a model of a particular thought pattern used to solve a class of problems. A paradigm supplies a systematic, cohesive approach to looking at a problem and thereby assists the solver to think more clearly about the possible solutions. The trade-off is that a paradigm which helps solve one problem may not be flexible enough to solve a wide range of problems [Zave 89]. As a result, large complex systems need to make use of multiple paradigms. Multiparadigm systems are made up of subsystems, each utilizing a different paradigm suited to its particular needs. This allows the overall system to take advantage of the strengths offered by the separate paradigms [Hailpern 86]. The major problem associated with this approach is the mechanism for transferring information between the different paradigms and getting it into a meaningful representation. One approach to solving this problem is to build an interface layer which provides a mapping at the conceptual level of both syntax and semantics. In the case of Ada and SQL this means that systems written in Ada which need the ability to store and retrieve persistent data may desire the ability to utilize a relational database through SQL. To build an interface, the differences between the two languages must be examined.

Data Models :

A data model is a mental model used to abstract the real world into a form that can be used to solve some problem. It contains a representation of real world objects and a mechanism for representing relationships between them as well as a set of operations for their manipulation. Each data model gives a "flavor" to the solutions derived from its use. It is through the use of a data model representation that approaches to problem solving are supported.

The relational model was first proposed by Codd [Codd 70]. In the model the abstract representation of data is a collection of tables, which are treated as mathematical relations, made up of named columns called attributes. Relations can represent entities or relationships. Relationships are established by using shared attributes between relations. This results in the possibility of many-to-many relationships which were not supported by the network or hierarchical models. The relational model was seen as an alternative to the existing network and hierarchical models which required some knowledge of the physical organization within the database. In both the network and hierarchical models entities are records and relationships are the connecting links between the records. The fields of the record are the attributes of an entity. The structure of a network model is a graph while a hierarchical model is a tree. To access the value of a tuple, the path to the tuple must be known. Queries are processed by walking the graph or tree (navigational queries) and can be very slow. In the relational model values are obtained by describing the attributes of the tuple, and not by the path to its location. This was seen as an advantage because the user is not aware of any changes to the internal representation which could result from changes in the pattern of database usage or growth of the database.

The entity-relationship (ER) model was proposed by Chen to form a more generalized data model [Chen 72]. He felt that the network model provided a natural view of data but lacked data independence, while the relational model contained a high degree of data independence at the loss of semantic information. The ER model was an attempt to form a unified view of data. In the ER model an entity is a thing which can be distinctly identified. A relationship is an association among entities. The information about entities is kept separate from the information about relationships. This feature helps establish functional dependencies and leads to the capturing of semantic information. The ER model takes a fundamental view of the data and so is a useful tool in developing the conceptual model of a database. Although it is most often used as a first step in developing a relational database, it can be used to form the initial analysis for other types of databases.

Although there is not a formal definition for the object-oriented data model, there are certain features that are to be expected in any object-oriented database. In an object-oriented database, entities are objects of abstract data types. This means they need not be atomic, as in the relational model, but may be complex structures. Associated with an object are classes (types) and messages (operations). Every object is derived from some class. Classes, in turn, may be derived from other classes forming a classification hierarchy. Each level in the hierarchy inherits the messages available on the classes higher in the hierarchy (inheritance). Objects are encapsulated, hiding their structure. Access to the information in them is provided through the messages defined on their class. Named relationships are supported between objects. They form navigational links, as in the network or hierarchical databases, and predicate based links, as in relational databases [Zdonik and Maier 90].

The data model for Ada uses non-persistent, strongly typed, named objects (~entities). It can support objects as ab-

stract data types. Abstract data types are formed using records and packages, which contain an arbitrary collection of types. The fields of a record contain the object's information (~attributes). Access to an object (~relationships) can be direct, through the object's name, or navigational, through access pointers. One of the strengths of Ada's data model is its flexibility; its ability to support a variety of problem solving approaches. The Ada/SQL interface should not limit the application program to the relational data model, therefore it needs the ability to form an abstraction of the services provided by the database.

Views :

Views are a mechanism used to partition a database in order to hide information not needed or accessible to a user or group of users. For example, in a company's database each department would have their view of the database, i.e., the accounting department's view, the personnel department's view, etc. Each view reflects the users model of the world. It can be part of an existing relation or a join of several relations. The relations defined in a view are not stored in the database, only the definition of the view. As a result, modification to the database made through views cannot update some parts of the relations that the view is constructed from, and thus in most DBMS updates must be made through the original relations. Application programs are written against defined views. Since the view provides the access to the database from the application program, the vertical structure of the interface should be based upon the concepts of views. The view definitions form the modularization used in the Ada/SQL interface.

Typing :

The typing model in SQL is limited to a set of basic data types and a dynamic collection of tuples of those types. The ANSI standard defines eight data types, although different implementations provide slightly different interpretations. Ada on the other hand is a strongly and statically typed language. The language provides the user with the ability to define new types and subtypes at compile time, as well as structures. These features ensure that operations only occur on the intended type. Along with strong typing, Ada provides for data abstraction. Data abstraction is the defining of a type by specifying the operations that apply to the type while hiding the implementation from the user. Any interface solution must provide a mechanism for converting between the primitive types and tuples in the database, and the user defined types in the application program. Additionally, the interface should check that only valid data is inserted or modified in the database, therefore consistency constraints can be assured for any given type.

Structure of Entities :

The structure of entities in relational databases is first normal form. This means that all the attributes of an entity are atomic. Therefore, they do not have subparts and can be viewed as simple records of primitive types. Relationships between entities are just shared, named fields which provides a linking mechanism between different records. In traditional database applications this is sufficient, however there are application domains where structured entities are needed. One domain is CAD/CAM systems, another is programming environments and a third is command and control systems. An Ada program can easily support the structures used in these various application domains. However, they cannot be derived by simple, direct mapping from SQL tuples.

If a one-to-one mapping is not possible between two representations then there is a structure clash [Jackson 75]. To map between the structures, a process for transformation is developed. First the lowest common denominator between the two structures is determined, then algorithms are developed to decompose the first structure and then to build the second structures up from primitives. But, whereas Jackson allows the data structures to be visible to the users, Parnas hides as much of the details as possible [Parnas 72]. Parnas decomposes a system into modules based upon the concept of encapsulating design decisions into modules and then supplies the user with an interface to the module that only provides the information needed and nothing more. This technique of information hiding isolates the details of the mapping process when there is structure clash.

Abstract Interfaces :

Parnas also describes the use of abstract interfaces [Parnas 77]. By abstract, Parnas is referring to the representation of several actual objects without specifics to any particular object. The essence of the real world is captured in the abstraction which can then be reused to other objects that the abstraction represents. The interface is the format of the information exchange between two software components along with all the assumptions that each makes about the other. An abstract interface is defined as a set of assumptions that represent more than one possible interface. An abstract interface models those properties that the interfaces hold in common and hides the differences.

INTERFACE SOLUTION

Two Layer Interfaces :

One way to map the mismatch between multiple paradigms is to build a two layer abstract interface. The two layers are broken down into a high level interface for the semantic issues and a low level interface for the syntactic issues. The

high level interface provides the actual interface to the user. It uses the concepts of data abstraction to generalize the access to information crossing the interface. The low level interface hides the details of a given implementation, putting the information in the form expected by the other side.

For example, in UNIX, when a user requests a file be copied to the current directory, the command *cp* along with the path to the file are all that the user has to supply. The operating system determines from the path which device the file resides on and makes the call to the appropriate device driver. The device driver is concerned (if the file were on disk) with such things as: seek algorithms, inodes, motor speed, etc. The supplemental information and work involved in copying the file are transparent to the user. The upper level provides the file abstraction, the lower level the device read/write.

Another example is a proposed two level architecture for communication within multi-Ada program systems [CASS 89]. The interprogram interface is defined in terms of remote procedure calls (RPC). In other words, to the application program the location of the called procedure is unknown. Again, the lower level is concerned with the details of crossing the interface, such as, sockets and communication protocols. The upper level provides an abstract interface to the user, taking care of the bundling of the message with logical source and destination names. The upper layer provides a procedural interface defining services used by the application, the lower layer a message passing transport service.

A third example is the interface between the real world and a control system. In a control system certain observable characteristics of the external world are monitored by sensor subsystems which understand the meaning of those characteristics. In this example the lower layer of the interface receives the sensor data coming in from the external world while the upper layer interprets the information into a meaningful representation. An example of this interface is a stop light control system [Wheeler 86]. The system senses the presence of large metallic objects through sensors buried in the approach lanes of an intersection. The information from the sensors is passed to the lower layer of the interface. It is then sent to the upper layer which converts the information to a representation of traffic presence or absence in the various lanes. The upper level is concerned with control of the intersection, the lower level with the sensors and relays.

In each of the examples it is the lower level which takes care of the actual crossing of the boundary while the upper level converts the information into a usable form, providing the user program with a meaningful abstraction.

Ada/SQL Interface Architecture :

The architecture for the Ada/SQL interface is based on the concept of a two layer abstract interface (Figure 1). It consists of a low level Syntactic Layer and a high level Semantic Layer. The lower level provides the syntactic mapping while the higher level takes care of the semantic issues.

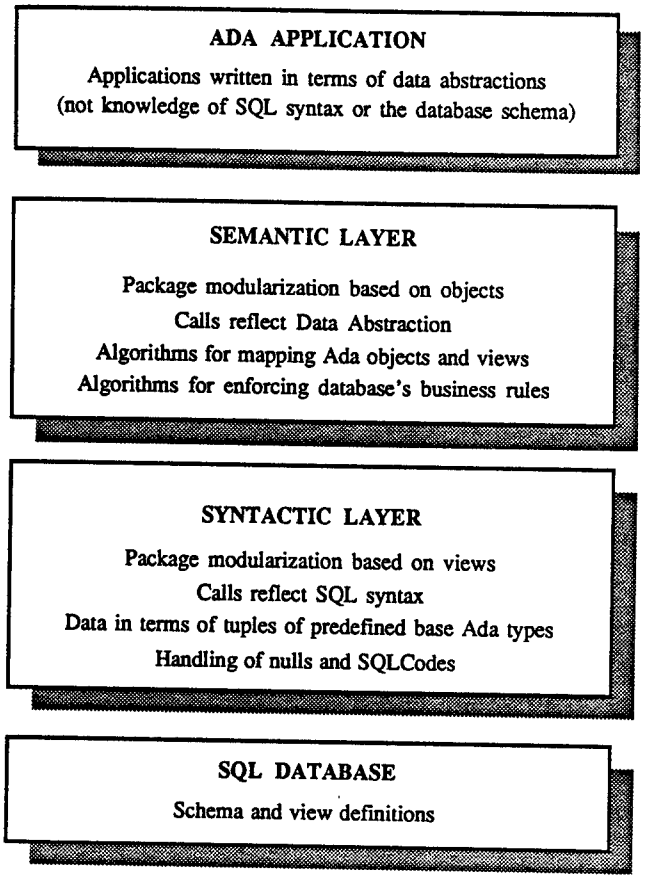


Figure 1
Interface Architecture

The Syntactic Layer contains the binding of SQL to Ada. It interprets the SQL error codes and takes care of null values. In general, it takes care of the implementation details associated with the chosen binding and hides them from the application program. This layer directly maps the types in the database to predefined Ada types. The procedure calls to the Syntactic Layer reflect the syntax of SQL. The implementation that we use of the Syntactic Layer is based upon the Module Language definition in Chapter 7 of the ANSI SQL Standard [ANSI 86].

The Semantic Layer is the visible part of the interface to the Ada application programmer. This layer converts the tuples from the Syntactic Layer to data structures forming Ada abstract types. The data abstractions reflect the information

needs of the Ada application program and are derived from view definitions in the database. This layer not only provides the application with access to the information contained within the data abstraction, but also maintains data integrity by enforcing the business rules governing modification of the database. It is the job of the interface to enforce the rules and modify the original tables giving the appearance of modification through views even when that is not supported by the DBMS. Calls to the Semantic Layer abstract away the SQL functions used in the lower level. The upper level of the interface can be viewed as a retriever and maintainer of a persistent abstract data type. This design takes advantage of Ada's ability to implement information hiding and data abstraction.

Use of this architecture gives Ada programmers the advantages of an SQL DBMS without requiring them to be proficient in SQL or understand the logical schema of the database. This approach not only segregates the two paradigms from a design standpoint, but also has practical advantages, since only the builders and maintainers of the interface need to know both worlds.

Design Method :

Designing a two layer interface, using the architecture described, consists of three phases: initial analysis, mapping of views and data abstractions, and business rule enforcement (Figure 2). The first step is a general analysis of the application in order to develop a conceptual model of the problem. This can be accomplished using the ER model to form an object based view of the problem domain. By using the ER model to describe the objects, there need not be a presumption toward a relational solution. At this point a functional analysis is done of the identified objects and the flow of information between objects. Once this initial analysis is completed the design of the interface can begin.

The development of the interface is accomplished by starting at each end and developing the details within each layer. The first stage is concerned with mapping the objects in the database to the objects in the Ada application. To develop objects in the database, a logical model is established from the conceptual model and developed into a normalized relational schema. On the Semantic Layer side, abstract data types and their underlying data structures are developed from the objects defined in the general analysis. Views are then derived from the normalized schema and the information needs of the data abstractions.

The modularization within the interface is based upon the views and data abstractions. At the Syntactic Layer a package is developed for each view. The syntactic packages provide open, fetch, and close operations on the tuples needed by the Semantic Layer. The Semantic Layer develops packages for each data abstraction. The specifications of these packages define the types and logical operations for the types. The syntax of these operations reflect the point of view of the application not the syntax of SQL. The bodies of the Semantic Layer packages define the underlying data structure and the algorithms for building them from the tuples supplied by the Syntactic Layer.

Once the data abstractions are defined and the support for establishing them is complete, the next step is to determine the rules or policies governing modification of the database in order to assure consistency. The modifications in SQL terms can take the form of inserts, updates, or deletes. Since the modification in most SQL DBMSs occur on the original tables not the views, the Syntactic Layer provides packages separate from the view packages for these calls. The Semantic Layer encapsulates the sequence of calls needed to modify the database according to the established rules. Within the Semantic Layer, as in the definitions of objects, the modification requests from the application are in terms of the data abstraction. By hiding the business rules within the interface, systems do not have to rely upon the applications to remember and enforce the rules.

EXAMPLE

Background :

In order to explore and illustrate our ideas for the interface, we made use of a contrived toy problem. We decided to choose a toy problem to allow us to concentrate on the issues of interest without getting bogged down in a lot of details. Also, by using a toy problem we could more easily generalize the essential concepts[Lomuto 87].

The toy problem is a computer management program for a fictitious toy company. The TWIT Toy Company is a small toy manufacturing company which builds wooden truck and train collectibles. The company sells the toys directly to customers through mail orders. The program provides support to four departments: Customer Service, Inventory Control, Manufacturing, and Shipping. Customer Service, Inventory Control, and Shipping provide straight forward traditional database application programs. The types in these Ada application programs are derived directly from each department's view of the database. The Manufacturing subsystem uses structured objects composed from several different queries on the database. The Manufacturing subsystem uses toy structures and parts lists stored in the database to gather the correct parts and assemble a toy. The serial number for each of a toy's parts is retained in a Manufacturing_Record. Therefore, the relative position of a part must be captured. Each department queries the database for work to be done. The work unit is an order. The information about an order is retained in an Order_Request which moves about the factory as an order is processed.

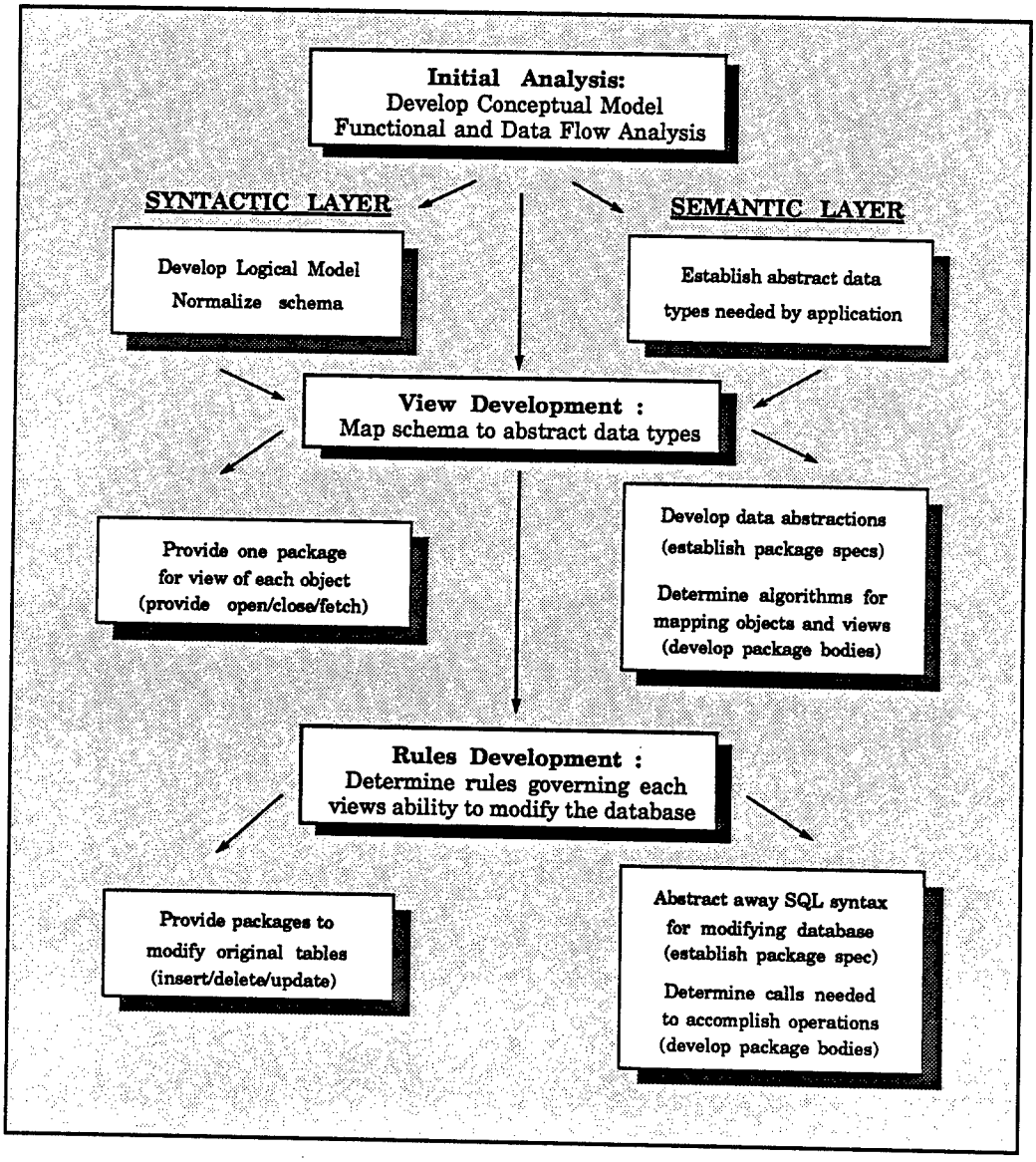


Figure 2
Design Method

The toy example was implemented on an AT clone running MS-DOS and using the ALSYS Ada compiler, XDB SQL engine, and Computer Associates' Module Compiler to provide the binding. Once it was running under this configuration, it was ported to the Army Tactical Command and Control System (ATCCS) configuration consisting of the Hewlett Packard 9000 Series 300 running UNIX System V and using HP's Ada compiler (derived from ALSYS), HP-SQL DBEnvironment, and HP's AdaSQL binding. Even though the bindings were entirely different for these two systems, the only changes made between the two versions were in the bodies of the Syntactic Layer, where the implementation details of a binding are handled.

Initial Analysis :

To analyze the toy problem we first developed a extended ER model of the ob'ects in the factory (Figure 3). The relationships between Customer_Account, Orders, and individual Toys are fairly straight forward. However, the relationship between a toy and the parts that make up the toy required a representation of the idea of *variety* (truck or train)

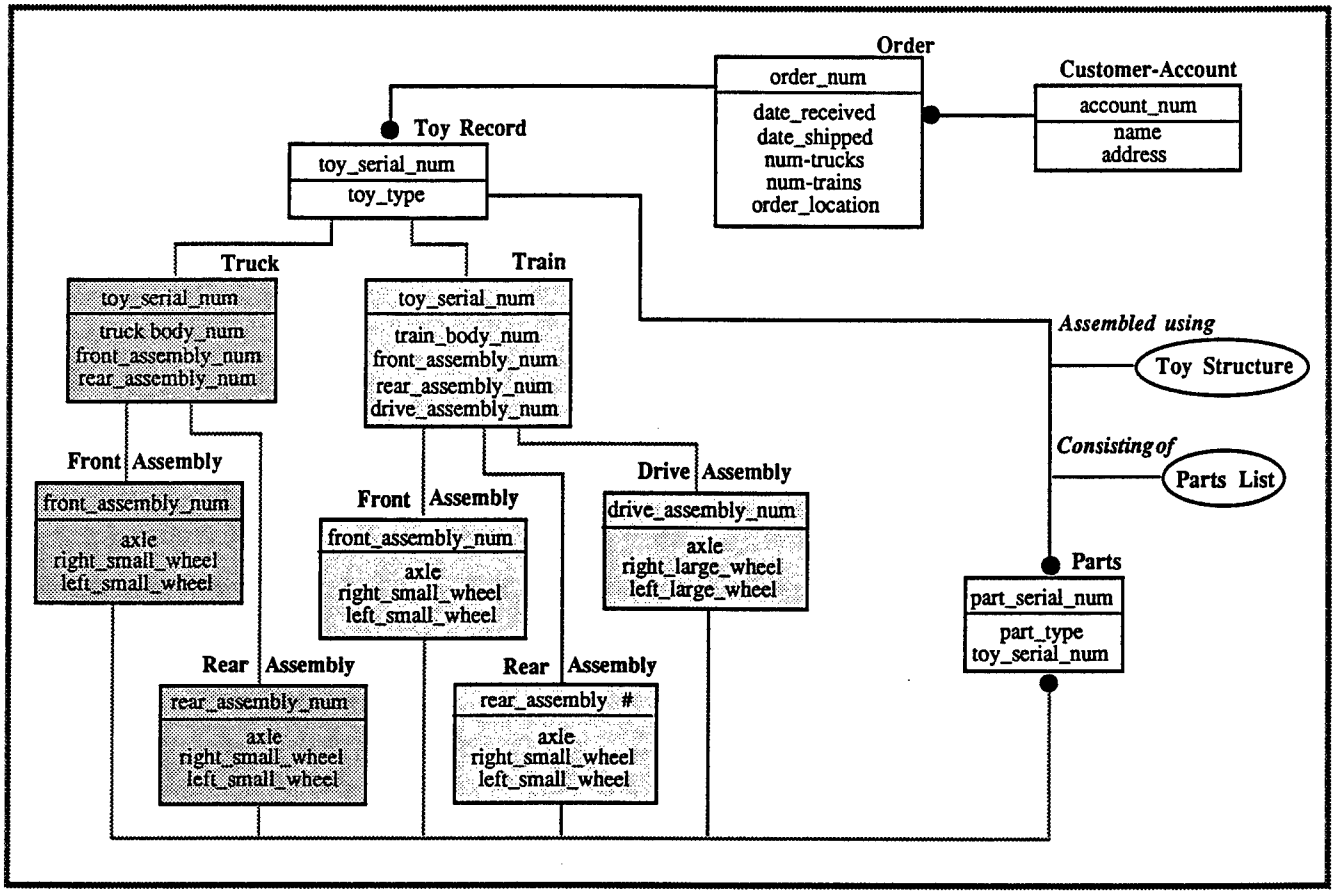


Figure 3
Extended Entity Relationship

and *made of* (toys made of multiple, interrelated parts). We wanted to capture in the Toy_Record the concept of relative position of a part within the assembled toy. In the figure, the white boxes, ovals, and dark lines represent the ER diagram. The ovals indicate attributes of the relationship. They represent the two ways the part-toy relationship can be thought in terms of; as a linear list of parts and as parts in a tree structure representing relative position. The two shades of grey indicate the sub-assembly structural information we wished to capture.

After we identified the major objects within the system we performed a functional analysis of the system (Figure 4). Each object derived from the database and used in a subsystem was identified as well as how information was passed through the system.

As the diagram shows, work enters the system through a New_Order which is received by the Customer_Service subsystem. Customer_Service calls up an existing Customer_Account_Record or generates a new one. It then generates an Order_Request which is placed in the New_Order_Bin.

The Manufacturing subsystem polls the New_Order_Bin for orders to be filled. By looking at the Parts_List for the toy variety, it gathers the unassembled parts from the Parts_Bin. Then it uses the generated Toy_Structure as a template to construct the toy. Once the toy is assembled, the information is stored in the Toy_Record and the toy and the Order_Request are placed in the Toy_Bin.

The Shipping subsystem gets the toys and Order_Request from the Toy_Bin and packs the toys into boxes; updating the Shipping_Record and placing the packed toy in the Shipping_Bin. Shipping then mails the packed toys and again updates the Shipping_Record. Shipping then files the Order_Request.

Periodically the Inventory subsystem counts the current in-

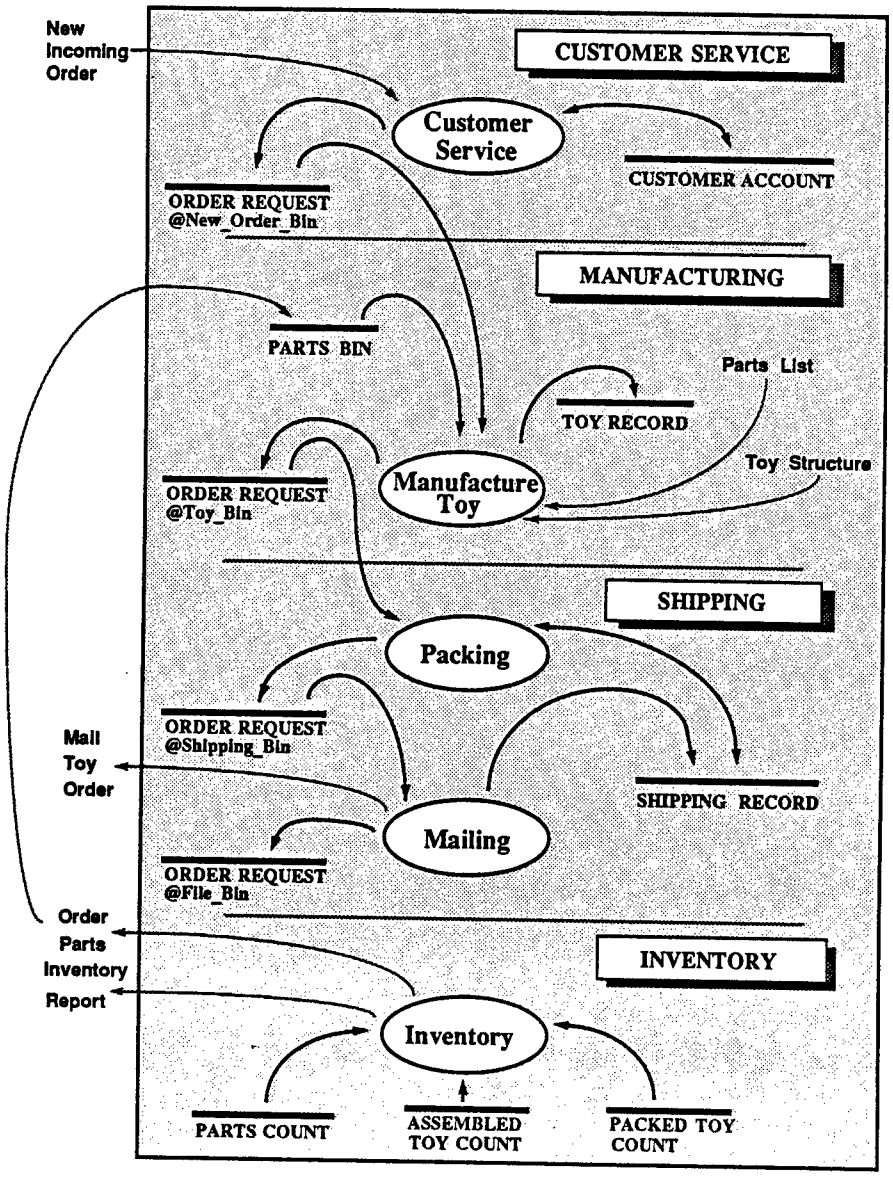


Figure 4
Funtional Diagram

ventory of parts, assembled toys, and packed toys and generates an inventory report. If the number of a certain part falls below a predefined threshold, new parts are ordered which replenishes the supply in Parts_Bin.

Mapping of Views to Data Abstractions :

At this point we began developing the interface. We started by developing the normalized relational schema needed to furnish the objects identified (Figure 5). We also established the type definitions to be used in the bodies of the Semantic Layer. From this we developed the view definitions. Then we determined the algorithms needed to map the objects and the view definitions.

Order_Request : An example of the mapping for a simple query function is the Order_Request. The view definition is derived from the Order table in the database. The Syntactic Layer gets the data across from the database and converts it to predefined Ada types declared in the package

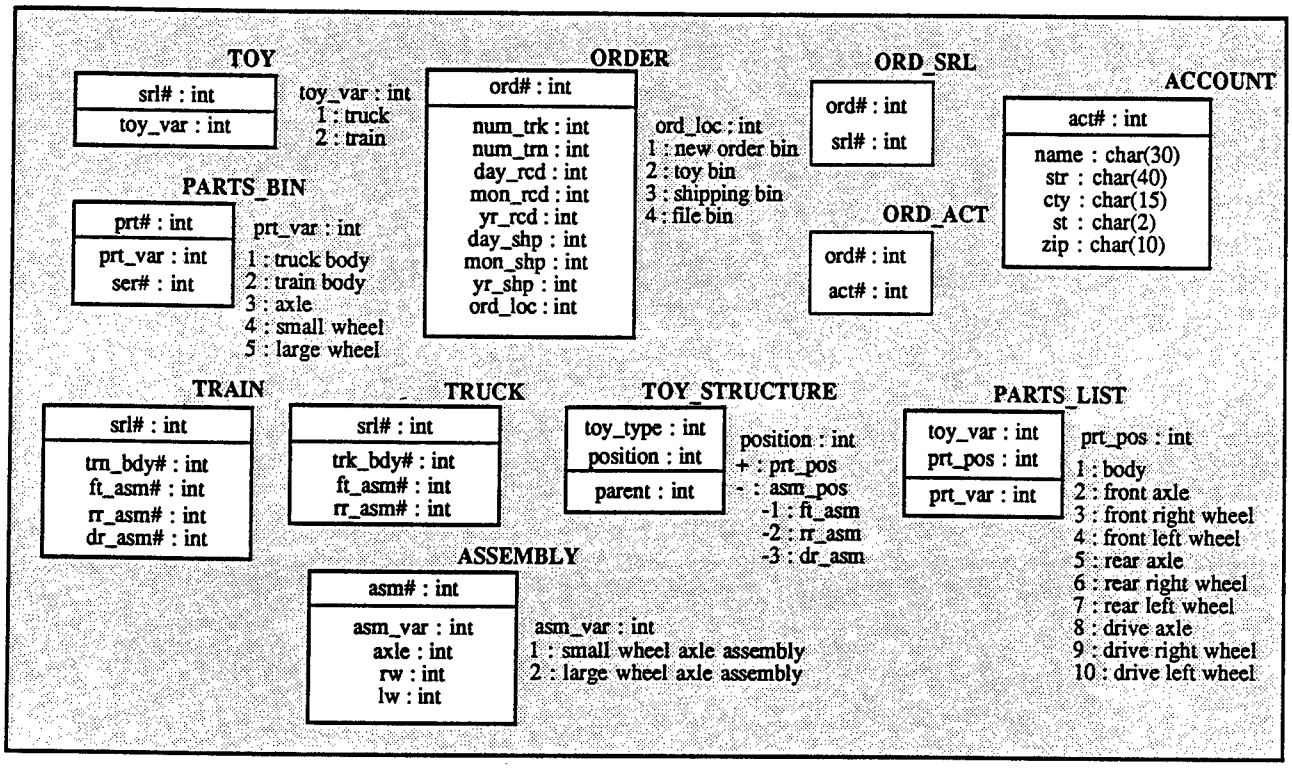


Figure 5
Database Tables

SQL_Standard. The Semantic Layer takes a record formed from the parameters returned from a fetch of the Order_Request view and builds the Ada object used by the application.

Order_Request
View Definition :

```
CREATE VIEW request_view
SELECT ord#, num_trk, num_trn, ord_loc
FROM Order ;
```

Order_Request
Fetch :

```
FETCH  SELECT ord#, num_trk, num_trn
FROM request_view
WHERE ord_loc = : Location ;
```

Order_Request
Syntactic Layer Tuple :

```
type Request_Tuple_Type is
record
  Order_Number : SQL_Standard.Int ;
  Number_Trucks : SQL_Standard.Int ;
  Number_Trains : SQL_Standard.Int ;
end record ;
```

Order_Request
Semantic Layer Object :

```
-----------------------------------------------
-- Semantic Layer enumeration types used in the examples

  type Toy_Variety_Type is ( TRUCK, TRAIN ) ;

  type Part_Variety_Type is
   ( TRUCK_BODY, TRAIN_BODY, AXLE,
     SMALL_WHEEL, LARGE_WHEEL ) ;
```

```
type Part_Position_Type is
 ( BODY, FRONT_AXLE, REAR_AXLE, DRIVE_AXLE,
   FRONT_RIGHT_WHEEL, FRONT_LEFT_WHEEL,
   REAR_RIGHT_WHEEL, REAR_LEFT_WHEEL,
   DRIVE_RIGHT_WHEEL, DRIVE_LEFT_WHEEL ) ;

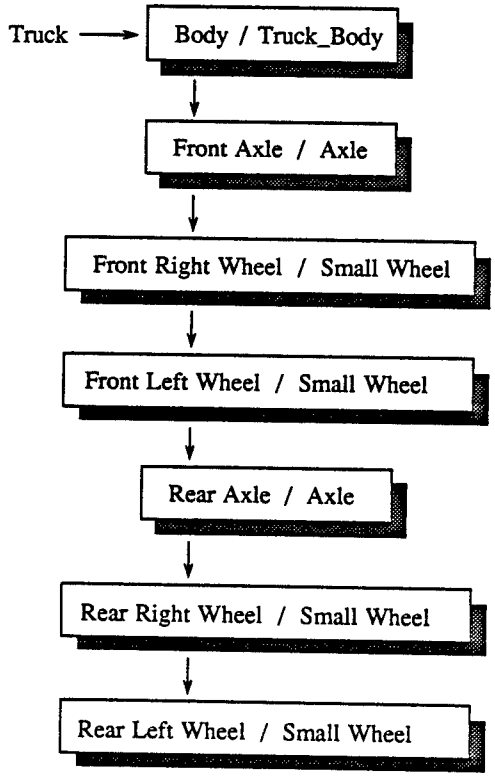
type Assembly_Position_Type is
 ( FRONT_ASSEMBLY, REAR_ASSEMBLY,
   DRIVE_ASSEMBLY ) ;

type Node_Variety_Type is ( PART, ASSEMBLY ) ;
-----------------------------------------
type Number_Ordered_Array_Type
 is array ( Toy_Variety_Type ) ;

type Order_Request_Record_Type is
record
 Order_Number : Order_Number_Type ;
 Number_Ordered : Number_Ordered_Array_Type ;
end record ;
```

Toy Record : Another example, which shows a more object-oriented use of a relational database, is the algorithm for building a Toy_Record by the Manufacturing subsystem. It uses two objects which are themselves stored in the database: the Parts_List and the Toy_Structure. The Parts_List is a linked-list created once for each toy variety. It contains the part variety and position name for each part in the toy. Below is the mapping from tuples to list object:

Parts List for a Truck :



Parts_List
Syntactic Layer Tuple :

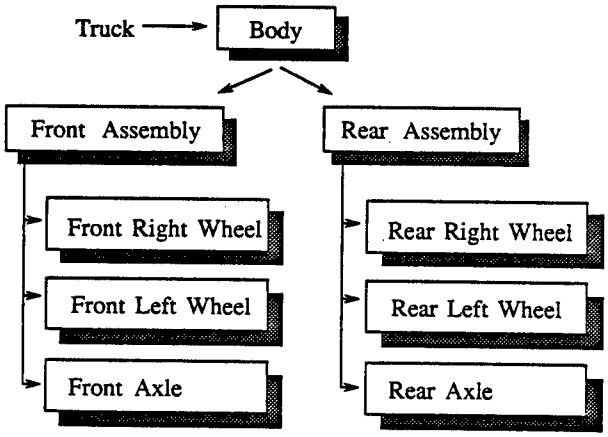
```
type Parts_List_Tuple_Type is
record
 Toy_Variety : SQL_Standard.Int ;
 Part_Position : SQL_Standard.Int ;
 Part_Variety : SQL_Standard.Int ;
end record ;
```

Parts_List
Semantic Layer Object :

```
type Parts_List_Type is
record
 Part_Position : Part_Position_Type ;
 Part_Variety ; Part_Variety_Type ;
 Next_Part : Part_List_Pointer_Type := null ;
end record ;
```

To build the Toy_Record the Manufacturing subsystem also uses a template of the structure of the toy. This template is built once for each toy variety from information in the database using a technique described by Hardwick and Spooner for support of CAD/CAM objects in relational databases [Hardwick and Spooner 87]. With this technique objects are built from a collection of tuples. Each object has one root tuple and any number of component tuples. The tuples are related by use of an attribute which links a tuple to its parent component, forming a tree hierarchy.

Toy Structure for a Truck :



Parts_List
Syntactic Layer Tuple :

```
type Toy_Structure_Tuple_Type is
record
 Toy_Variety : SQL_Standard.Int ;
 Position : SQL_Standard.Int ;
 Parent : SQL_Standard.Int ;
end record ;
```

Parts_List
Semantic Layer Object :

```
type Toy_Structure_Type
 ( Node_Variety : Node_Variety_Type := Part ) is
record
 case Node_Variety
  when Part =>
   Part_Position : Part_Position_Type ;
  when Assembly =>
   Assembly_Position : Assembly_Position_Type ;
 end case ;
 Peer : Toy_Structure_Pointer_Type := null ;
 Child : Toy_Structure_Pointer_Type := null ;
end record ;
```

The algorithm we used to construct the Toy_Record is derived from Wirth's generalized table-driven parser [Wirth 76]. This algorithm uses a syntax graph representing the grammar. The parser then simultaneously walks the syntax graph and the incoming program to generate the parse tree representation of the program. In our use of this technique, we "parse" the Parts_List into parts and "walk" the Toy_Structure generating a tree in the Toy_Record containing the part information. The Toy_Record can then be broken down into its component tuples and stored in the database.

Toy_Record
Semantic Layer Object :

```
type Toy_Tree_Type
 ( Node_Variety : Node_Variety_Type := Part ) is
record
 case Node_Variety
  when Part =>
   Part_Position : Part_Position_Type ;
   Part_Variety : Part_Variety_Type ;
   Part_Serial_Number : Serial_Number_Type ;
  when Assembly =>
   Assembly_Position : Assembly_Position_Type ;
   Assembly_Number : Assembly_Number_Type ;
 end case ;
 Peer : Toy_Tree_Pointer_Type := null ;
 Child : Toy_Tree_Pointer_Type := null ;
end record ;
```

```
type Toy_Record_Type is
record
 Toy_Variety : Toy_Variety_Type ;
 Toy_Serial_Number : Serial_Number_Type ;
 Order_Number : Order_Number_Type ;
 Toy_Tree : Toy_Tree_Pointer_Type := null ;
end record ;
```

Development of Packages :

Once we defined the SQL views and Ada objects, we could start developing the package specifications for each level. The Syntactic Layer packages are in terms of the view definitions, while the Semantic Layer packages are in terms of the data abstractions. The data abstractions make visible to the user only the information needed by the application, not the information needed for the mapping to views.

For example, in the Toy_Record, the assembly numbers are used for mapping the component tuples and are retained in the Toy_Record for use when the record is broken down and stored in the database. However, the application has no knowledge of the mechanism used to associate parts to positions in the database and so the visible part of the Toy_Record would not contain the assembly numbers.

Shipping Record : The following example shows the package specification to the Syntactic and Semantic Layers for a Shipping_Record as well as the type definition of the record in the Semantic Layer body.

```
package Syntactic_Ship_Pkg is

type Ship_Tuple_Type is
record
 Serial_Number : SQL_Standard.Int ;
 Toy_Variety : SQL_Standard.Int ;
 Order_Number : SQL_Standard.Int ;
 Order_Location : SQL_Standard.Int ;
 Ship_Date_Null : Boolean ;
 Month_Shipped : SQL_Standard.Int ;
 Day_Shipped : SQL_Standard.Int ;
 Year_Shipped : SQL_Standard.Int ;
 Name : SQL_Standard.Char ( 1 .. 30 ) ;
 Street : SQL_Standard.Char ( 1 .. 40 ) ;
 City : SQL_Standard.Char ( 1 .. 15 ) ;
 State ; SQL_Standard.Char ( 1 .. 2 ) ;
 ZIP : SQL_Standard.Char ( 1 .. 10 ) ;
end record ;

procedure Open_Cursor
 ( In_Order_Number : in SQL_Standard.Int ) ;
```

```
procedure Fetch_Cursor
 ( Out_Ship_Tuple : out Ship_Tuple_Type ) ;

procedure Close_Cursor ;

end Syntactic_Ship_Pkg ;
```

```
package Semantic_Ship_Pkg is

-- User visible type definitions

type Order_Info_Record_Type
   (Mail_Date_Null : Boolean := True ) is
record
 Order_Number : Order_Number_Type ;
 Order_Status : Order_Status_Type ;
 Customer_Name : Name_Type ;
 Customer_Address : Address_Type ;
 case Mail_Date_Null is
  when False =>
    Date_Mailed : Date_Type ;
  when True => null ;
 end case ;
end record;

type Toy_Info_Record_Type is
record
 Serial_Number : Serial_Number_Type ;
 Toy_Variety : Toy_Variety_Type ;
end record ;

-- Functions used for accessing information in object

function Find_Record
 ( In_Order_Number : in Order_Number_Type )
 return Boolean ;

function Get_Order_Info
 return Order_Info_Record_Type ;

function Get_Toy_Info
 return Toy_Info_Record_Type ;

-- Procedures and function for accessing the data abstraction

procedure Set_Head_Toy_List ;

procedure Next_Toy ;

function More_Toys
 return Boolean ;

end Semantic_Ship_Pkg ;
```

```
package body Semantic_Ship_Pkg is

-- Implementation of the data abstraction

type Toy_Record_Type is
record
 Toy_Info : Toy_Info_Record_Type ;
 Next_Toy : Toy_Record_Pointer_Type ;
end record ;

type Shipping_Record_Type is
record
 Order_Info : Order_Info_Record_Type ;
 Toy_Info_List : Toy_Record_Pointer_Type := null ;
end record ;
...
end Semantic_Ship_Pkg ;
```

Establishing Rules :

Finally we determined the rules governing a views ability to modify the database. To ensure the integrity of the database the Semantic Layer was used to enforce compliance with the established business rules.

For example in the Shipping subsystem, when an order is mailed, the location of the order is changed to the File_Bin. But, the database expects a filed order to have a date mailed. Therefore, the procedure in the Semantic Layer which changes the location of an order inserts the current date into the Date_Shipped field when the location is being changed to File_Bin.

CONCLUSION

This paper examined the issues in trying to resolve the impedance mismatch between Ada and SQL. It described a two layer interface architecture to deal with both the syntactic and semantic issues resulting from using multiple paradigms. It used an example to illustrate the interface and show how complex objects in an Ada application can be supported in an underlying relational database. Use of the architecture gives an Ada program access to retrievable, persistent data while hiding the details of the database structure, the SQL language and the binding implementation.

REFERENCES

[ANSI 86] ANSI, *Database Language-SQL*, ANSI X3.135-1986, American National Standards Institute, Inc., New York, 1986.

[CASS 89] Common ATCCS Support Software Working Group-Architecture Subcommittee, *Requirements for Inter-Task Communication*(Draft), 1989.

[Chen 72] Chen, P., "The Entity-Relationship Model-Toward a Unified View of Data", *ACM Transactions on Database Systems*, March 1976.

[Codd 70] Codd, E. F., "A relational model of data for large shared data banks", *Communications of the ACM*, June 1970.

[Graham 89] Graham, M. H., *Guidelines for the Use of the SAME*, Technical Report, Software Engineering Institute, Pittsburgh, PA, 1989.

[Hailpern 86] Hailpern, B., "Multiparadigm Languages and Environments", *IEEE Software*, January 1986.

[Hardwick and Spooner 87] Hardwick, M. and D. L. Spooner, "Comparison of Some Data Models for Engineering Objects", *IEEE Computer Graphics & Applications*, March 1987.

[Jackson 75] Jackson, M., *Principles of Program Design*, Academic Press, New York, 1976.

[Korth and Silberschatz 86] Korth, H. F. and A. Silberschatz, *Database System Concepts*, McGraw-Hill, New York, 1986.

[Lomuto 87] Lomuto, Nico, *Problem Solving Methods with Examples in Ada*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987

[Parnas 72] Parnas, D. L., "On the Criteria to Be Used in Decomposing Systems into Modules", *Communications of the ACM*, December 1972.

[Parnas 77] Parnas, D. L., "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems", *NRL Report*, Number 8047, June 1977.

[Wheeler 86] Wheeler, T. J., "An Example of the Developer's Documentation for an Embedded Computer System Written in Ada", Ada Letters, November-December 1986 and January-February 1987.

[Wheeler 87] Wheeler, T. J., *Memorandum for Record*, CECOM Software Technology Division, Information Processing Directorate, November 1987.

[Wirth 76] Wirth, N., *Algorithms + data structures = programs*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

[Zave 89] Zave, P., "A Compositional Approach to Multiparadigm Programming", *IEEE Software*, September 1989.

[Zdonik and Maier 89] Zdonik, S. B. and D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

ACKNOWLEDGMENTS

The authors wish to thank Rennetta McGhee and Hoang D. Lee for their efforts in porting to the Hewlett-Packard.

Dr. Thomas Wheeler is the technical leader of the Software Engineering Research Group, Center for Software Engineering, Ft. Monmouth, NJ and is a member of the Software Engineering faculty at Monmouth College. He received his Ph.D. in Computer Science from Stevens Institute of Technology in 1988 and has degrees in Physics, Electronic Engineering, and Computer Science. His current research interests are: Formal Techniques, Programming Languages and Principles, Databases, Distributed Systems and Environments/Operating Systems.

Judith Richardson is a Computer Scientist with Advanced Software Technology, Center for Software Engineering, Ft. Monmouth, NJ. She received her B.S. in Computer Science from the University of Maryland in 1980 and is currently working toward a M.S. in Software Engineering at Monmouth College. Her areas of interest include: Interfacing Software Standards, Database Technology, and Software Engineering Environments.

Arturo Ferrer is a Electronics Engineer with the Command and Control Directorate, Rome Air Development Center, Griffiss AFB, NY. He received his B.S. in Materials Engineering from Virginia Tech in 1985 and his M.S. in Software Engineering from Monmouth College in 1989. His areas of interest are Databases, Ada Applications, and Software Engineering Methodologies.

A PRACTICAL Ada/SQL IMPLEMENTATION

John L Schoenecker III

U.S. Army Information Systems Software
Development Center-Washington
Fort Belvoir, Virginia 22060-5456

ABSTRACT

Few Structured Query Language (SQL) applications demonstrate any of the current Ada/SQL binding methods; i.e, SQL in Ada, SQL Module, Extended Ada, Embedded SQL, or String-oriented. This paper presents an overview of an Ada application development within a segment of The Army Authorization Documents System-Redesign (TAADS-R) using IBM's DB2 Data Base Management System and a 'SIMULATED' SQL Module binding method.[1,3] It discusses how Ada was used to implement packages, a preprocessor, and interfaces to another language environment, COBOL/SQL. The use of Ada and COBOL/SQL together in a single load unit is presented. An important feature of this paper is the inclusion of actual examples of Ada applications showing how this methodology is incorporated in TAADS-R development.

INTRODUCTION

The structure of this paper is divided into five parts. The Introduction describes the overall content of the paper and areas addressed. In the Project Description, the reader is familiarized with the requirements of the TAADS-R application and the intended operating environments. A brief description of each binding method follows. The Project Solution addresses the application of Ada, COBOL/SQL and DB2 Data Base Management System. Coding examples in Ada and COBOL/SQL are presented to explain features and the implementation strategy used in the binding method. The Results section discusses application statistics produced during this development (e.g., programs, lines of code, etc.) along with an explanation of lessons learned and employment of solutions. The Summary reviews several major points addressed in the document and projects the future of Ada development within this project.

PROJECT DESCRIPTION[2,3,5]

Application Requirements

As it currently exists, The Army Authorization Documents System (TAADS-R), (see Figure 1), consists of three major components Installation TAADS (ITAADS); Vertical TAADS (VTAADS); and Force Development Integrated Management System - Authorization Sub-system (FORDIMS-AS) at Headquarters Department of the Army (HQDA). ITAADS is the system used by Army installations to document their manpower and logistics needs and to pass these documents to VTAADS for Major Army Command (MACOM) review. VTAADS is the system a MACOM uses to document its elements. When the MACOM has documented a unit, the document is forwarded to HQDA for approval. Once the document is approved, it is stored in the FORDIMS data base. Field units can only make changes to their authorization documents twice a year, during a Management of Change (MOC) window. Both VTAADS and ITAADS operate in a 'batch' processing environment. Document analysts must enter transactions into the system via card image and wait several days to see if these transactions were entered into the system correctly.

TAADS-R Interface Requirements

TAADS-R Output Interfaces (represented as part of the 'External Interface*' in Figure 2), initially began with over thirty Output Interface Modules identified to be constructed. Currently the number of Output Interfaces has grown to over fifty. Five of these modules were designated as critical for the initial deployment. These critical modules were designated for immediate construction because the MACOM functions they represented would not be available to these organizational levels in the first phase of deployment. Secondly, these five were also targeted to be our first involvement with the Ada programming language.

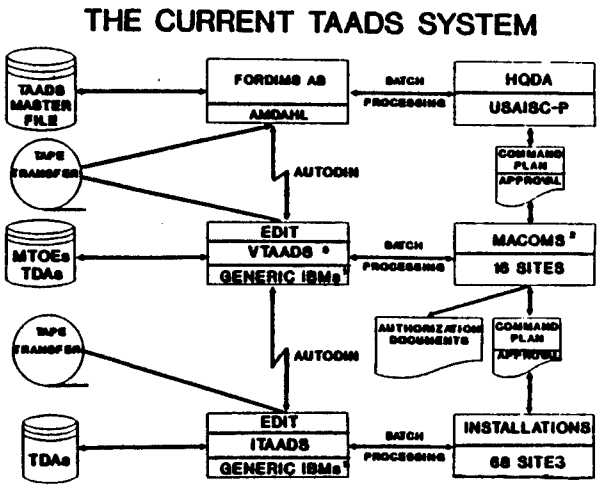


Figure 1. Current TAADS System.

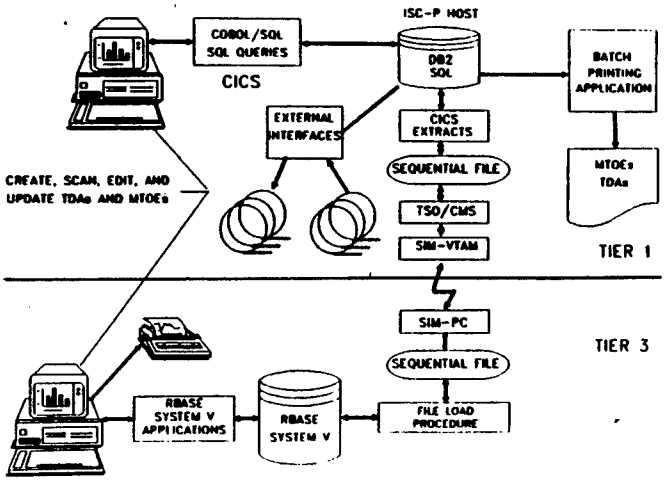


Figure 2. TAADS-R Design Flow.

Production Environment

The TAADS-R effort began in 1986 to provide the Army with a functional authorization documents system that would allow document analysts to update their documents interactively using Personal Computers (PC). The prototype or baseline TAADS-R architecture employs a centralized mainframe computer data base with distributed processing at PC workstations located at user sites. This is depicted in Figure 2.

Development Environment

The development environment for the Output Interfaces consisted of a Team Leader, a Computer Programmer/Analyst, and two Intern Computer Programmers. A preliminary prototype was completed on Z248 PCs, using Meridian and Alsys compilers, but the majority of coding was accomplished on an Amdahl 5680, under MVS/XA with an Intermetrics compiler. The remaining non Ada software was constructed using RBASE SYSTEM V at the PC level and COBOL w/CICS at the host level along with DB2.

Ada/SQL METHODS

Overview

The following paragraphs describe briefly the characteristics of the different methods presented on the subject of Ada SQL binding. They are not intended to present an in-depth examination of each method, but to give a basic understanding of the philosophy behind each, along with some associated positive and negative observations. A more complete description can be found in selected documents[4,3].

SQL in Ada

SQL syntax in this method is expressed in compilable Ada. With this method, programs may use the full data typing facilities of Ada. Objects would be strongly typed, allowing the full use of Ada specific tools, since all code is written in Ada. A draw back to this method is that its run time approach may result in poor performance with current compiler implementations.

Ada/SQL Module

A user defines Ada procedures which interface to precompiled SQL modules. This has an advantage of not conflicting between Ada and SQL syntax. This method does not require a preprocessor, and the Data Manipulation Language (DML) may be reused. A draw back to this approach is that all Ada programs are limited to SQL's data types (i.e. real, integer, and character).

* External Interfaces represent separate Input and Output Requirements.

SQL Ada Module Extensions(SAME)[8]

This method of program design and development utilizes support software of standard packages. SAME treats SQL just as Ada treats other foreign languages; that is, it imports complete modules, not language fragments. This method is currently in development by the Software Engineering Institute at Carnegie Mellon University. Its major advantage is that it constructs an Ada program which accesses the DBMS services.

Embedded SQL

The embedding of SQL statements in Ada or other host languages has been the traditional approach to data base access for application programming languages. The programmer prepares a single text containing statements from two different languages (i.e., the Host language and SQL). These two sub-texts are processed thru a preprocessor supplied by the particular DBMS vendor. The output programming language text replaces the data base statements with procedural calls to the DBMS. This text is processed by the programming language compiler. This simplifies the writing of programs with no conflict between Ada syntax and SQL syntax. It is the most commonly used method of binding programming languages (e.g. FORTRAN, COBOL, PL1, Pascal and Ada). This approach adheres to the ANSI Standard for Embedding SQL Support and Dynamic SQL.[8] A preprocessor is required and Ada programs are limited to SQL's data types (i.e., real, integer, and character).

Simulated Method[11]

This method is a 'module style' using an existing embedded syntax support. The difference between this method and the previously mentioned 'Ada/SQL Module', method is that any language which supports the embedded syntax, (e.g., COBOL, FORTRAN, ALC) is written as a subprogram. The programs developed are not pure SQL programs but, are encapsulated in the Host language. This method does not utilize a 'SQL Module processor' but, a preprocessor supported by the Host language. The use of subprograms and multiple entry points was a key concept with this method and was exploited in the solution to the TAADS-R development.

String Method

This method defines queries which are placed in strings and passed as parameters to a procedure. This method is sometimes termed a Dynamic SQL approach. This is a simple method to implement, and may use specific Ada tools but there is no compile time checking of the syntax on data types. This type of program construction is limited to the SQL data types.

'SIMULATED' SQL MODULE SOLUTION

Problem Overview

The development of the Output Interface application programs for TAADS-R required the use of IBM's DB2 with Ada as the application programming language. As stated previously, several approaches using Ada and SQL have been postulated and have proceeded through a gauntlet of scrutiny, but no definitive method has been totally agreed on. A forthcoming revision of the data base language SQL standard hopefully will change this situation.[8]

The embedded approach has been implemented in several vendors DBMS and host language, but DB2 with Ada/SQL is not one of them. The module approach (SAME), currently under development, seems to be a better alternative, but no extensive testing with this method has been accomplished with any major development effort.

Solution Implementation

Our initial coding effort started with a Meridian compiler, Version 1.5. This followed with an update to version 2.0. We then acquired an Alsys compiler. These two PC software compilers provided us with invaluable experience with the many features of Ada, and initiated our investigation into the capabilities of Ada compilers. Our efforts focused on implementing the features stated in Chapter 13 of the Ada Language Reference Manual (LRM), specifically, the 'Pragma Interface' statement. Our interest with the Pragma Interface statement stemmed from the fact that this was the only way Ada could talk to the 'outside world.' Unfortunately the PC software compiler versions we had at the time could only interface with 'C' and ALC. A Intermetrics compiler was identified as having an interface capability with COBOL. This compiler was loaded on our development system. Programs which were in testing using the PC compilers were successfully transmitted and recompiled on the host system.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
  EXEC SQL INCLUDE SQLCA END-EXEC.
  EXEC SQL INCLUDE MDOC END-EXEC.
  EXEC SQL INCLUDE MDOCU END-EXEC.
01  WS01-MHDRMUIC.            PIC X(51).
01  WS10-LITERALS-INDICATORS.
  03  WS10-FETCH-INDICATOR PIC X(5).
    88  FETCHING-MHDR1  VALUE 'MHDR1'.
  03  WS10-DOCNO-INFO      VALUE SPAS.
    05  WS10-CCNUM-MACOM PIC XX.
    05  WS10-DOCNO       PIC X(10).
    05  WS10-CCNUM-CHGNR-FY PIC XXXX.
  03  WS10-FE-INDICATOR  PIC X VALUE 'Y'.
EXEC SQL
DECLARE MHDRMUIC CURSOR FOR
SELECT MDOCHDR.CCNUM_MACOM, MDOCHDR.DOCNO
FROM MDOCHDR, MDOCUIC
WHERE (MDOCHDR.CCNUM_MACOM =
  MDOCUIC.CCNUM_MACOM)
ORDER BY UICOD, EDATE
END-EXEC.
LINKAGE SECTION.
01  LS01-MHDRMUIC             PIC X(51).
01  LS10-DOCNO-INFO.
    03  LS10-CCNUM-MACOM      PIC X(2).
    03  LS10-DOCNO            PIC X(10).
    03  LS10-CCNUM=CHGNR-FY IC X(4).
01  LS13-FE-INDICATOR         PIC X.
PROCEDURE DIVISION.
0100-MAIN-PROCESS.
    MOVE '0' TO WS10-FETCH-INDICATOR.
    PERFORM 10000-SEND-TO-ADA.
    ENTRY 'OPMHDR1'.
1000-OP-MHDR1.
    EXEC SQL
        OPEN MHDRMUIC
    END-EXEC.
    MOVE 0 TO WS10-FETCH-INDICATOR.
    PERFORM 10000-SEND-TO-ADA.
    ENTRY 'FEMHDR1' USING LS10-DOCNO-INFO.
2000-FE-MHDR1.
    MOVE SPACES TO WS01-MHDRMUIC.
    EXEC SQL
        FETCH MHDRMUIC INTO WS01-MHDRMUIC
    END-EXEC.
    MOVE 'MHDR1' TO WS10-FETCH-INDICATOR.
    PERFORM 10000-SEND-TO-ADA.
    ENTRY 'CLMHDR1'.
3000-CL-MHDR1.
    EXEC SQL
        CLOSE MHDRMUIC
    END-EXEC.
    MOVE '0' TO WS10-FETCH-INDICATOR.
    PERFORM 1000-SEND-TO-ADA.
10000-SEND-TO-ADA.
    IF FETCHING-MH
        PERFORM 11000-PREPARE-MHDR1.
    GOBACK.
11000-PREPARE-MHDR1.
    MOVE WS01-MHDRMUIC TO LS01-MHDRMUIC.
    MOVE WS10-FE-INDICATOR TO
        LS13-FE-INDICATOR.
```

Figure 3. COBOL SUBPROGRAM.

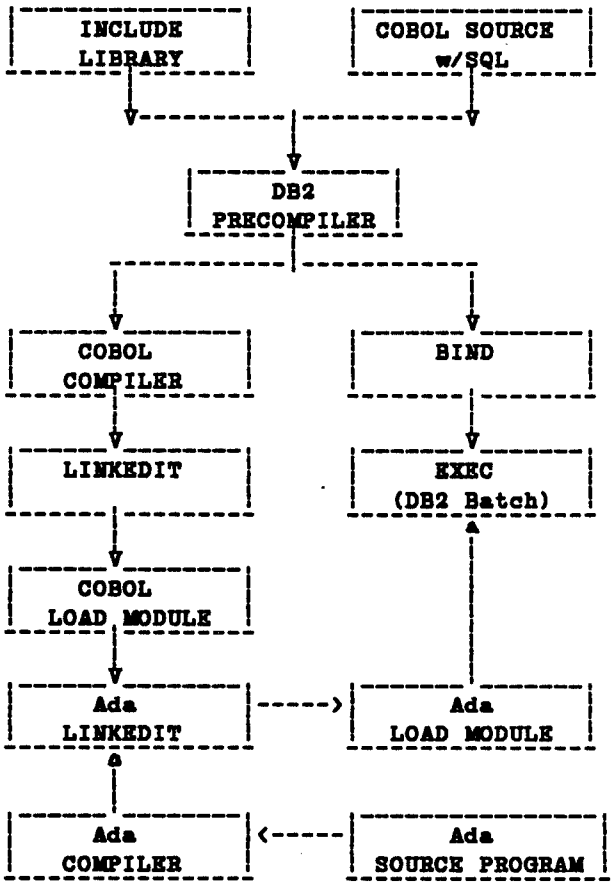


Figure 4. "Simulated" Binding Method.

The strategy used from this point was a combination of two Ada/SQL methods, the embedded approach and the module approach along with several key software engineering techniques. All SQL I/O functions for a given interface would be embedded in a COBOL subprogram. The COBOL subprogram would be used exclusively for accessing the DB2 data base tables (see Figure 3) and minimize any other data manipulation functions. The Ada package (see Figure 5a-5b) is the main controlling software of one Output Interface function and would call (via a Pragma Interface package (see Figure 6)) the specific data necessary to be manipulated by a particular Output Interface requirement.

Figure 4 is a graphical representation of the linking and final binding of these separate modules together to form one load module. The steps involved in designing each Output Interface initially started with the construction of the COBOL subprogram.

```
--*****************************************
--*          Main Ada Driver              *
--*****************************************
with Text_Io;
with Sequential_Io;
with Ebcdic;                    use Ebcdic;
with RTS_CICS_Packed_Decimal_Pkg;
  use RTS_CICS_Packed_Decimal_Pkg;
with ANJUTS01;                  use ANJUTS01;
with ANJ18S03;                  use ANJ18S03;
with ANJ18P01;                  use ANJ18P01;
procedure ANJ18A02 is
--*****************************************
--* Generic Package Instantiations        *
--*****************************************
    package Hdr_Io is new
     Sequential_Io(HDR_OUT_TYPE);
    package Int_Io is new
     Text_Io.Integer_Io(integer);
--*****************************************
--* Object Declarations                   *
--*****************************************
    END_OF_FILE_MHDRMUIC:boolean:= false;
    FIRST_TIME_MAIN      : boolean:= true;
    FIRST_TIME_MHDRMUIC  : boolean:= true;
    FIRST_TIME_MAIN_S    : boolean:= true;
    LAST_MHDRMUIC        : boolean:= false;
    MHDRMUIC_OPEN        : boolean:= false;
    HDR_OUT_CREATED      : boolean:= false;
    RUNNING_MTOE_OR_TDA
                : RUNNING_MTOE_OR_TDA_TYPE
                  := MTOE;
    MHDR1_IND           : REC_IND_TYPE:= 'Y';
    LOOP_CTR_MAX  : constant integer:= 3;
    LOOP_CTR            : integer:= 0;
    MHDRMUIC_POS        : integer:= 1;
    MHDRMUIC_POS_END: integer:= 1;
    PAK_OUT_LENGTH : constant integer:=2;
    FETCHED_MHDRMUIC_CTR : integer := 0;
    WRITTEN_HDR_CTR : integer := 0;
    MHDRMUIC_IN_REC : MHDRMUIC_IN_TYPE;
    HDR_OUT_REC     : HDR_OUT_TYPE;
    HDR_OUT_REC_DUMM: HDR_OUT_TYPE;
    HDR_OUT_FILE    : Hdr_Io.file_type;
    MHDRMUIC_ARRAY : MHDRMUIC_ARRAY_TYPE;
    UICOD_HOLD          : estring (1..6)
                          := SPACES(1..6);
    UICOD_MATCH         : UICOD_MATCH_TYPE;
    DOCNO_INFO          : DOCNO_INFO_TYPE;
--*****************************************
```

Figure 5a. Ada Main Package.

Following this step the subprogram is run through the DB2 Precompiler which inserts the necessary call statements to DB2 and performs the required binding of the COBOL subprogram. Next, the main Ada program is constructed, compiled and link edited with the bound COBOL subprogam, creating a single load module. The sequence of these steps depict a "bottom up" approach to our development effort.

```
--*****************************************
--*          Main Ada Driver con't        *
--*****************************************
--*****************************************
--* Procedures                            *
--*****************************************
    procedure P1000_FETCH_MHDRMUICS is
        CONTINUE         : boolean:= true;
    begin
      MHDRMUIC_POS := 0;
      if not MHDRMUIC_OPEN then
          OP_MHDR1;
          MHDRMUIC_OPEN := true;
      end if;
      while CONTINUE loop
        if (FIRST_TIME_MHDRMUIC) and
          (not FIRST_TIME_MAIN) then
            null;
        else
            FE_MHDR1
            (MHDRMUIC_IN_REC,MHDR1_IND);
            FIRST_TIME_MAIN := false;
        end if;
        if MHDR1_IND = 'Y' then
          FETCHED_MHDRMUIC_CTR
           := FETCHED_MHDRMUIC_CTR + 1;
          if FIRST_TIME_MHDRMUIC) or
           (MHDRMUIC_IN_REC.UICOD=
             UICOD_HOLD)
             then
             FIRST_TIME_MHDRMUIC := false
             MHDRMUIC_POS :=
               MHDRMUIC_POS + 1;
             MHDRMUIC_POS_END :=
               MHDRMUIC_POS;
             MHDRMUIC_ARRAY
             (MHDRMUIC_POS).MHDRMUIC_IN_REC
              := MHDRMUIC_IN_REC;
              UICOD_HOLD
               := MHDRMUIC_IN_REC.UICOD;
          else
              CONTINUE := false;
         end if;
        else
          CL_MHDR1;
          CONTINUE := false;
          LAST_MHDRMUIC := true;
      end if;
    end loop;
   end P1000_FETCH_MHDRMUICS;
--*****************************************
--* MAIN
--*****************************************
begin
   CALL_COBOL_MAIN;
    while not END_OF_FILE_MHDRMUIC loop
        P1000_FETCH_MHDRMUICS;
        if not END_OF_FILE_MHDRMUIC then
        end if;
        if LAST_MHDRMUIC then
            END_OF_FILE_MHDRMUIC := true;
        end if;
    end loop;
end ANJ18A02;
--*****************************************
```

Figure 5b. Ada Main Package.

```
package ANJ18PØ1 is

procedure CALL_COBOL_MAIN;
pragma interface
         (COBOL, CALL_COBOL_MAIN);
pragma link_name
         (CALL_COBOL_MAIN, "ANJ18CØ1");

procedure OP_MHDR1;
pragma interface (COBOL, OP_MHDR1);
pragma link_name (OP_MHDR1, "OPMHDR1");

procedure FE_MHDR1
        (MHDRMUIC : out MHDRMUIC_IN_TYPE;
         MHDR1_IND : out REC_IND_TYPE);
pragma interface (COBOL, FE_MHDR1);
pragma link_name (FE_MHDR1, "FEMHDR1");

procedure CL_MHDR1;
pragma interface (COBOL, CL_MHDR1);
pragma link_name (CL_MHDR1, "CLMHDR1");

end ANJ18PØ1;
--****************************************
```

Figure 6. Ada Pragma Package.

Solution Advantages

This solution incorporates a number of important principles of Software Engineering. First is modularity. In this solution each program has been constructed to be as loosely coupled as possible. Each COBOL subprogram, could be reused to extract the same information for a totally different requirement. Secondly, the collection of the "Pragma Interface" statements in the Pragma Package (see Figure 6) demonstrates the principle of localization, and facilitates modifications to a small set of functions for reuse. Lastly, it works! Future development of the remaining Output Interfaces is anticipated to progress rapidly as we continue to employ these techniques.

Solution Disavantages

First, many religious Ada purists would object to the fact that this solution does not use Ada in total as the implementation language. Secondly, it is required that the application programmer be knowledgeable in a number of different languages (e.g., Ada, COBOL, SQL) and of course the DBMS used. This is a point I do not believe most computer programmers would object to.

RESULTS

Application Status

The following table represents the current disposition of the Output Interface requirements and software development effort.

Phase of Development	Number Interfaces	Number Programs	* Coded Statements
Completed	12	35	27k
Analysis/Design	2		
Coding	2	2	4k
Testing	4	4	8k
To-be-assigned	32	69 *	121k *
Total	52	110	160k

*Estimate

One additonal metric worth noting is in regards to the number of Ada statements. The majority of original COBOL code, utility programs, and JCL use in the TAADS/VTAADS system, was 40-50 percent more then the new code developed. It was felt that a further reduction would be possible if not for the fact that the COBOL/SQL modules increased in size as the requirement for accessing additional tables increased. On the average five to fifteen tables were necessary per module.

A number of innovative solutions were developed to work around several limitations which surfaced in our application development. This additional programming included the following:

* development of Ada packages to handle packed decimal using bit manipulation and memory addressing.

* development of a technique to use strings and enumeration types to get same effect as moving spaces/zeros.

* establishment of a data position technique for COBOL record structures.

The results of this effort have been very encouraging. Personnel within our organization have had the opportunity to develop varying degrees of expertise in the Ada programming language. They have obtained the skills necessary to fulfill a greater commitment when Ada is the Army's dominant application development language.

Lessons Learned

The TAADS-R Ada Output Interface project offers some important lessons which can be applicable to other Ada projects. First, and most important is my contention that Ada can be used effectively in a Management Information System (MIS) environment. Secondly, most Ada compiler vendors do not fully support the capabilities defined in Chapter 13 of the Ada Language Reference Manual (LRM). I would strongly recommend the mandatory implementation of all features and components in the above stated LRM. As alluded to previously, we had a difficult time acquiring an Ada compiler with a COBOL interface. This situation, hopefully, will be rectified in future compiler improvements. Ideally, each compiler will have a "Pragma Interface" statement which will be able to communicate with a separate SQL Module compiler. Thirdly, the need exists for an Ada Programming Support Environment (APSE). Any organization anticipating Ada development work needs to include definitive policy statements on the usage of Ada within the intended environment. Lastly, in this development effort, learning the language is not extremely difficult but application programming personnel need more training in software engineering techniques before using the Ada programming language. Additional experience and advanced training in using generics and tasking abilities that the Ada language provides is also strongly recommended.

SUMMARY AND CONCLUSION

This paper has described an implementation method for using Ada and a DBMS which supports a host language and SQL. Until an SQL module compiler or SAME is completely developed and available, the method depicted in this paper and the few Ada/SQL embedded approaches available with a small number of DBMS vendors will be the only existing alternative for using Ada/SQL with DBMS.

The future for TAADS-R utilization of Ada looks bright. TAADS-R will deploy five MACOM Output Interface modules in January 1990 and a total of thirty Output Interfaces are scheduled for completion and deployment by the end of FY90. Plans are underway for the installation of Ada/CICS interface software at the host development site. Shortly, the Ada Team will be testing the installation and capabilities of a new version of the Intermetrics Compiler.

ACKNOWLEDGMENTS

The author would like to thank the members of the Ada Team, (Ms. Sheryl Edlund, Ms. Matrica Ware, and Mr. Samuel Lukschander) of the TAADS-R project for their valuable contributions to the development of this paper. I wish to gratefully acknowledge the encouragement and support given to me by the management of the Information Systems Software Development Center-Washington to complete this effort.

REFERENCES

[1] AJPO "Ada SQL Interface Workshop", November 1987

[2] Functional Description, "The Army Authorization Documents System Redesign (TAADS-R)", January 1988

[3] "System Design Plan for The Army Authorization Documents System Redesign (TAADS-R)", May 1989

[4] Brykczynski Bill, "Methods of Binding Ada to SQL: A General Discussion", November 1987

[5] Graham, Marc, "Guidelines for the Use of the SAME", May 1989

[6] ANSI X3H2-88, "Discussion of the Proposed changes to the SQL Module language for Ada", May 1988

BIOGRAPHY

John L Schoenecker III is a Senior Computer Specialist with the United States Army Information Systems Software Development Center-Washington. He is involved with the implementation and deployment of both Ada Software and The Army Authorization Documents System-Redesign. His interests are in Software Engineering, Ada and the use of Ada with Data Base Management Systems. He is a member of ACM, the Washington Chapter of SIGAda, and IEEE.



Creating a Database Application Using the SQL Ada Module Extension (SAME) Method

Thomas D. Fenton

US Army Information Systems Engineering Command
Software Development Center - Fort Lee

Abstract

This paper describes the development of a Management Information System (MIS) pilot project using Ada and SQL, in a modular compiler implementation, with a Relational Database Management System (RDBMS). The use of the modular compiler method has been mandated, by contract, as the preferred method for implementing Ada SQL binding to a RDBMS. This project took an existing process and converted it from another language with imbedded SQL to the modular binding method. A forms generation tool which produces Ada code packages was used for menus and data entry screens. Ada packages from the local reusable library were implemented at all possible locations. It was concluded that the modular compiler approach of Ada SQL binding to a RDBMS is both feasible and desirable. It produces Ada and SQL code which offer high potential for reuse, maintenance, and portability.

Background

Ada has, for some years, been the required programming language for development of Department of Defense automated systems. The US Army Information Systems Engineering Command (ISEC) has endorsed that policy. ISEC, with a major role in the development of Management Information Systems (MIS) for the Army, has also urged the use of RDBMS to allow for information access between systems through American National Standards Institute (ANSI) SQL. After much consideration, the decision was reached that the modular approach to Ada SQL binding would be the accepted ISEC policy. The implementation of the decisions to use Ada, SQL, and a RDBMS through a modular compiler approach has not been possible until recently. Few, if any, commercial Database Management Systems had implemented Ada access to their data structure except through an imbedded approach. A RDBMS was delivered to ISEC in July 1989, with modular Ada SQL binding allowing MIS development to begin meeting all the dictated standards.

The Software Engineering Institute (SEI) has proposed a modular compiler binding method for Ada and SQL. This method is called the SQL Ada Module Extension (SAME) Method. Through a government contract, the SAME methodology was implemented by a RDBMS vendor. The Ada/SQL binding used in this pilot project follows the SAME method with the exception of changes to ANSI SQL. Upon acceptance of the SEI proposed SAME method, the product will be modified through existing an contract.

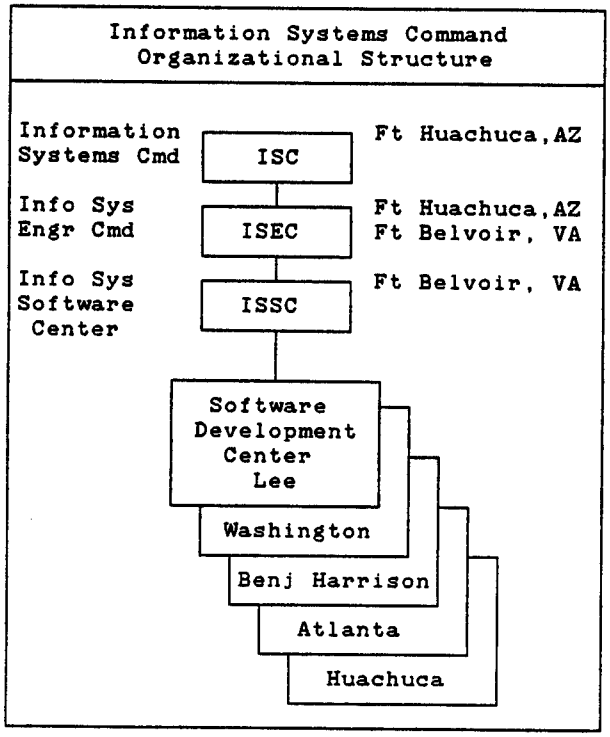
Project Goal

The goal of the pilot Ada/SQL binding project was to determine the feasibility of implementing a functional and technical database design using the combination of Ada and SQL. Although the RDBMS, with the Ada/SQL binding, had passed acceptance testing, no actual, functional MIS process had been developed using Ada, SQL and a modular compiler binding.

For the test project, a logistics functional process was selected which had been accomplished with another RDBMS using its proprietary procedure language (4GL) and imbedded SQL. This functional process had also been developed, in a test mode, using a government owned DBMS that is driven by unique Ada packages. The process is one where the user enters certain data from the keyboard, the program gathers additional data from files, and the results are written to a file and to the printer. The procedure must handle significant data editing with very little mathematical processing. It is typical of many MIS applications in its high degree of input/output activity and human engineering requirements and its low degree of internal processing activity.

Resources

This project was assigned by the Commander, US Army Information Systems Software Center, to its Software Development Center Lee, Fort Lee, VA. (See the Organization Chart below.) The project was accomplished, under command sponsorship, by one systems programmer analyst from the Directorate of Technical Support. This programmer analyst had experience with the functional process and with the 4GL/SQL solution. He had participated in five weeks of Ada training and had participated in DBMS evaluations.



Manpower Cost

One hundred eighty-four man-days was projected for completing the project. The actual man-days used were only 51, for a savings of 133 man-days. The use of locally written Ada reusable packages was instrumental in the reduced actual man-days. Of significant value in reducing the Ada code required to provide screen interface was the use of a locally written "Panel Designer" which generates Ada code and "Panel Driver" package which drives those panels. The implementation of the SQL modules was significantly less difficult than original projections.

Project Plan

Project Name : Ada/SQL Pilot Project
Project Manager: Thomas D. Fenton

Development of the Dispatch Process using XDB v2.2, Ada/SQL binding, and Alsys Ada Compiler V4.2 on Zenith Z-248 Microcomputer.

Task	Title
1	Decompose Existing Process
2	Analyze File Structure
3	Design File Structure
4	Download Test Data
5	Create Database and Tables
6	Upload Test Data
7	Create Screens
8	Survey Ada Reuse Module Library
9	Develop Ada Code for Process
10	Develop SQL Code for Process
11	System Test Process
12	Functional Validation of the Process
13	Comparison of the Methods

NOTE:
Very little attention will be given to tasks which are generic to the conversion of a process from one language and RDBMS to another. The discussion will focus on those areas involving an Ada and SQL binding.

Tasks

Tasks 4 and 6 - Download Test Data and Upload Test Data. Test data was downloaded from an existing set of tables under one RDBMS to ASCII delimited files. The files had to be modified slightly due to differences in the two RDBMSs.

Task 7 - Create Screens. Data Entry and Menu screens were created using the locally written facility "Panel Designer." This facility creates panels which are portable across microcomputer, minicomputer, and mainframe computer levels. Two panel description files are produced as well as an Ada package for each panel. (See samples following.) The panels are accessible to the programs through another locally written package, "Panel Driver." The use of these panels allows total control over data entered with complete data visibility.

Sample of Panel and Ada Code Generated from 'Panel Designer'

A Sample Panel Created with 'Panel Designer'
The Panel Provides Data Entry and Data Display Fields

```
Date: _________________ Motor Equipment Dispatch                AWAM20A4
-------------------------------------------------------------------------
                          Equipment/Unit Data

   DODAAC: ______          ____________________          UIC: ______
 Admin No: ______
Serial No: ______________  Noun: ____________________  Model: ___________
-------------------------------------------------------------------------
                        Additional Dispatch Data

 1st Operator: _____ ______________
 2d  Operator: _____


-------------------------------------------------------------------------
 Press <F1> to Continue     Press <Esc> to Exit
```

The Ada Code Generated from the Above Panel
by the Panel Designer Program

This code declares types for the fields defined in the panel. It also provides two internal functions. One function takes data contained in the record type (Panel_Descrition_Type) and converts it to the string type (awam20a4_String_Type). The other function converts the string type into the record type.

```
--External Name: awam20a4.ada

--Internal Name: awam20a4

package awam20a4 is

  type Panel_Description_Type is
    record
      System_Date           : string ( 1 ..    17 );
      ................
      Oprtr1_LName          : string ( 1 ..    15 );
      Oprtr2_Lic            : string ( 1 ..     5 );
      Termination_Key       : character;
    end record;

  subtype awam20a4_String_Type  is string ( 1 ..   130 ) ;

  Panel_Name : string ( 1 ..     8 ) := 'awam20a4';

  function Make_String ( Panel_Description : in     Panel_Description_Type )
    return awam20a4_String_Type ;

  function Make_Panel_Description ( The_String : in     awam20a4_String_Type )
    return Panel_Description_Type;

end awam20a4;

package body awam20a4 is

  ..... NOT SHOWN .....

end awam20a4;
```

Task 8 - Study Reuse Modules. All locally written reusable Ada packages were examined for applicability to the project. The following Ada modules were selected for reuse.

Case_Conv
SI_Conv
System_Date_Time
Time_Ops
Pnl_Driver

Case_Conv (Case Conversion) was used to convert all data entered through panels into upper case for database use.

SI_Conv (String_Integer_Conversion) was used to convert string data and integer data for use between panel types (string) and database types (integer).

System_Date_Time was used to obtain the system date from the computer in various formats desired.

Time_Ops was used to obtain a pause in operation where necessary.

Pnl_Driver (Panel Driver) was used to display panels and data, retrieve data, display and clear messages, and to clear the screen.

Task 9 - Develop Ada Code for the Process. The decomposition phase of the project included the separation of the functions to determine if they should be completed as Ada procedural code or as SQL database interaction. In a larger project, this separation would allow the maximum use of resources based on Ada skills and database (SQL) skills.

The Ada procedures and functions fell into the category of those required to accomplish the functions of the process and those required to pass data to and from the SQL modules. Type declarations created through the Panel Designer process were used wherever possible to reduce redundancy. The development of the Ada procedural code required to accomplish the functional requirements was, by far, the largest part of the entire development effort.

The Ada module below calls an SQL module which selects one row of a table. 'SQL_Standard.Indicator_Type' is used to receive NULL values without causing an Ada exception.

Sample Ada Code Calling an SQL Module

```
    ......................

-- ------------------------------------------------------------------------
-- Get_Vehicle_Oprtr_Data (SQL)
-- ------------------------------------------------------------------------

procedure Get_Vehicle_Oprtr_Data
  ( Dispatch_Record_1     : in out AWAM20A5.Panel_Description_Type;
    Operator_Record       : in out Operator_Record_Type           ;
    Oprtr_Is_Found        :    out boolean                        ;
    License_Has_Expired   :    out boolean                        ) is

  Date_Lic_Exp       : string ( 1 .. 8 ) := ( others => ' ' );
  Date_Lic_Exp_Ind,
  Last_Name_Ind      : SQL_Standard.Indicator_Type;

  begin

    ......................

    AWAM20S1_SQL.SQL_Get_Vehicle_Oprtr_Data (
      S_DODAAC             => SQL_Standard.Char ( Dispatch_Record_1.DODAAC ),
      S_Drvr_Lic           => SQL_Standard.Char ( Operator_Record.Drvr_Lic ),
      S_Date_Lic_Exp       => SQL_Standard.Char ( Date_Lic_Exp ),
      S_Date_Lic_Exp_Ind   => Date_Lic_Exp_Ind,
      S_Last_Name          => SQL_Standard.Char ( Operator_Record.Oprtr_Lname ),
      S_Last_Name_Ind      => Last_Name_Ind );

    ......................

end Get_Vehicle_Oprtr_Data;

    ......................
```

Task 10 - Develop SQL Code for the Process. The SQL code developed in the project required various normal functions found in any MIS application.

Select one row from one table.
Select one row from joined tables.
Select multiple rows from one table.
Select multiple rows from joined tables.
Insert one row into one table.

Sample SQL Module

Select one row from one table

```
            ..............
            ..............

-- ------------------------------------
-- SQL_Get_Vehicle_Oprtr_Data
-- ------------------------------------

procedure SQL_Get_Vehicle_Oprtr_Data

S_DODAAC              char (6)
S_Drvr_Lic            char (5)
S_Date_Lic_Exp        char (8)
S_Date_Lic_Exp_Ind    smallint
S_Last_Name           char (15)
S_Last_Name_Ind       smallint ;


select Date_Lic_Exp,
       Last_Name
  into S_Date_Lic_Exp
       S_Date_Lic_Exp_Ind,
       S_Last_Name
       S_Last_Name_Ind
  from AWAMX195
 where DODAAC    = S_DODAAC
   and Drvr_Lic = S_Drvr_Lic ;

-- ------------------------------------
            ..............
            ..............
```

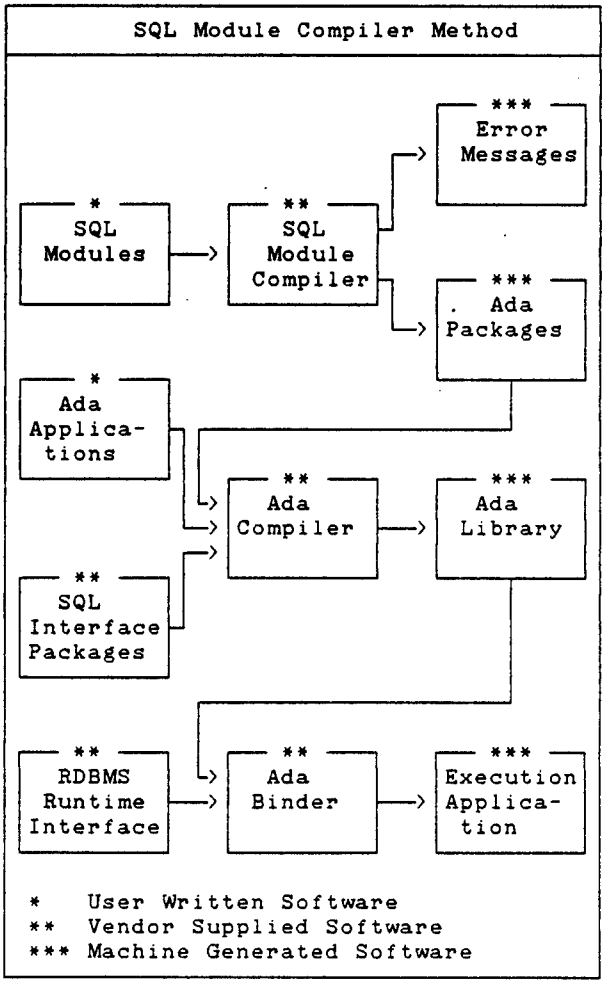
NOTE: '_Ind smallint' types are used to pass NULL values, if found in the data base, to the Ada module without raising an exception.

To enhance portability of both Ada and SQL code, all data was declared as a type available in the package 'SQL_Standard.' These types match only those allowed in the current ANSI SQL standard. Additional typing restrictions and display formats were accomplished in the Ada procedure. As an example, all 'date' fields were declared as char(8) in SQL and as a string (1 .. 8) in Ada.

The format, YYYYMMDD, allowed extensive use of the 'System_Date_Time' package and a date edit package. It will also allow the database structure and the Ada and SQL code to be ported to any DBMS and any Ada compiler using ANSI SQL standards and module compiler methodology.

SQL Module Compiler Method

The SQL Module Compiler implementation used in this project is an interim method pending acceptance of a final standard. The major differences between the method used and the proposed standard are that no changes to the SQL language are in effect nor is a pre-compiler used to add Ada typing to the SQL types. The concept of the two methods is, however, the same.



Task 13 - Comparison of Two Methods. Upon completion of the project, a comparison of certain aspects was performed between the Ada SQL module compiler solution and the 4GL imbedded SQL solution. The comparison was used to form a judgement whether is feasible and practical to develop MIS systems with Ada and a modular SQL approach for database systems.

Comparison of Two Methods		
Item	Ada/SQL (Modular)	4GL/SQL (Imbedded)
Executable (Bytes)	162,129	172,491
Source Code (Lines)	1,316	1,353
Generated Code (Lines)	1,040	N/A
'with'ed Packages	18	N/A
Execution Time (Seconds)	18	18
Coding Time (Man-Days)	30	Data Not Available

NOTES:

Executable. Both implementations contain the same functions.

Source Code.

Ada/SQL. The Ada/SQL code included the code in the Ada module and the SQL module. Ten procedures and two functions were written to accomplish the functions.

4GL/SQL. The SQL code was imbedded in the 4GL code to accomplish the functions. The 4GL/SQL code is pre-compiled into 'C' before final compilation.

Generated Ada Code. Ada/SQL Only.

Approximately 590 lines of Ada code were generated from the SQL module through the SQL module compiler.

Approximately 450 lines of Ada code were generated from the 'Panel Designer.'

'with'ed Packages. Ada/SQL Only.

The 'with'ed packages included the Ada packages generated from the SQL module compiler and the 'Panel Designer.' Other 'with'ed packages were those from the SQL/RDBMS binding and those selected from the local reusable library.

Execution Time. Execution time was computed through repetitive executions of the procedure. The time included only the actual processing time between selections and printing.

Coding Time.

Ada/SQL. Coding time included the time to write the Ada modules and the SQL modules. Additional time was spent on the code to ensure that it served as a good example for future projects regarding reusability and adherence to Army and local standards. Extra effort was also spent ensuring some level of local reusability for the procedures which were written.

Conclusions

The results of the test indicate that development of MIS database applications using a modular Ada SQL binding is both practical and desirable. The results of the project, combined with the inherent benefits to be derived from Ada and from modular components, show significant benefits to be gained from this method. These benefits can best be summarized into the areas of development efficiency, maintainability, portability, and adherence to standards.

Development Efficiency. The simple step of separating Ada code, SQL code, and menu/data entry panel development into self-contained modules creates significant advantages in development. The ability to apply the highest level of expertise to a given task speeds the entire development effort and injects a higher level of efficiency to the result.

Maintainability. Smaller pieces of a whole, which perform a specific function, increase the ease of maintaining any system. A higher level of understanding for a specific function can be attained more quickly. Additionally, the correct person can be applied to maintenance of the code for which he or she is best trained. Management of the entire maintenance process can then be vested in people with capabilities in broader areas.

Portability. The portability of source code between hardware and operating system platforms is a major goal in decreasing the cost of development and maintenance. Ada is one tool which increases that portability. Ada combined with an SQL modular compiler meeting government standards will increase that portability many times over. Applications written in this mode will allow the use of any relational

database management system which incorporates Ada/SQL binding according to standards. One baseline of application source code could then be run on many platforms. The changing of hardware and RDBMS vendors is sometimes required over the life cycle of a system. These changes are mandated by emerging technology, obsolescence of products and the evolution of requirements. Such changes, today, have a very high cost in dollars and human resources. Increased portability, through Ada/SQL binding can reduce these costs significantly. Another factor in achieving higher levels of portability is the ability to develop on one platform for fielding on one or more different platforms.

An additional step towards portability was taken in this project. The Ada/SQL binding used offered both ANSI SQL standard data types as well as data types unique to the RDBMS. Only ANSI SQL standard types were used in describing data. Conversion of these types for display or calculations were performed, as necessary, within the Ada code. Therefore, no conversion of data will be required, to move either the data or the code to another RDBMS product implementing ANSI SQL standards into its binding.

Adherence to Standards. Separation of SQL code from the Ada code into modules allows complete adherence to the standards of both languages. Adherence to standards increases the ease of training personnel, maintaining systems, and the portability of systems between platforms and RDBMS. This separation removes any requirement for either Ada or SQL to change while allowing either to change or grow as necessary.



Thomas D. Fenton is the Ada Technical Project Officer for the US Army Information Systems Software Development Center Lee, Fort Lee, VA. The center develops Logistics Standard Army Management Information Systems (STAMIS) for use throughout the Army. He has been involved in Ada development there for two years, concentrating on reuse and on the implementation of Ada and SQL with Relational Database Management Systems.

Fenton received his BA in English from Virginia Polytechnic Institute and State University, Blacksburg, VA, and his MA in Education from Pepperdine University, Malibu, CA. His computer training has been through the US Army.

Fenton's address is Thomas D. Fenton, USAISSDCL, ATTN: ASQBI-LTS, Fort Lee, VA 23801-6065.

FORWARD ENTRY DEVICE SOFTWARE ENGINEERING AND DOD-STD-2167A EXPERIENCES

Jag Sodhi *Guy Daubenspeck* *Thomas Archer*

TELOS Systems
Lawton, Oklahoma

ABSTRACT

This paper discusses the software development approach for the U. S. Army's Forward Entry Device (FED) system. The system is written in Ada. TELOS engineers have successfully used software engineering methodology while documenting and managing the effort in accordance with the DOD-STD-2167A (interim version dated April 1987). FED system development is being conducted for and managed at Fort Sill by the Fire Support Systems (FSS), Center for Software Engineering (CSE), U. S. Army Communication-Electronics Command (CECOM) of Fort Monmouth, New Jersey. We will share our software engineering and DOD-STD-2167A experiences.

INTRODUCTION

FED is a device used by the field artillery forward observer and fire support team chief to compose, edit, transmit, receive, store, forward, and display messages used in the conduct and planning of fire support operations at platoon, company, battalion, and brigade levels.

The customer required TELOS to write the system in Ada using DOD-STD-2167A (interim version dated April 1987). Recommendations for tailoring of the standard and Data Item Descriptions (DIDs) were submitted to the customer for approval.

FED is a part of the Army Tactical Command and Control System (ATCCS) as shown in figure 1. ATCCS provides five automated battlefield functions from Fire Control to Air Defense. ATCCS provides unique interfaces with each communications and control system. The purpose is to:

- Share critical battlefield information among the ATCCS systems
- Relay over Army tactical voice and data communications systems
  - Mobile Telephones
  - Combat Net Radios
  - Data Distribution
- Access by other services and theater-level forces engaged in the battle

TELOS' role has been to analyze the customer requirements and engineer the software for the FED system. This involved selecting a suitable methodology and Computer Aided Software Engineering (CASE) tool. The software engineering includes performing various development phases in accordance with DOD-STD-2167A. These phases are:

- Software Requirements Analysis
- Preliminary Design
- Detailed Design
- Coding and Computer Software Unit (CSU) Testing
- Computer Software Component (CSC) Integration and Testing
- Computer Software Configuration Item (CSCI) Testing

MANAGEMENT PERSPECTIVE

The management perspective for the software engineering is to plan, schedule, estimate, and utilize the available resources. An experienced task leader was assigned to manage the FED requirements. The first priority was to select a team for the project. Finding good Ada experienced professionals was, and is, difficult. New outside hiring for this type of personnel is expensive. The few experienced Ada professionals are hard to pry away from other Ada projects. There are many programmers who are potential Ada professionals if they can be provided proper training in software engineering and Ada. TELOS established an in-house facility to provide proper education and training in Ada and software engineering.

SOFTWARE ENGINEERING APPROACH

The major factors influencing software engineering for Ada software development under DOD-STD-2167A are the development methodology used and the software engineering environment which supports this methodology. The methodology consists of software development methods used to perform required activities. The software engineering environment provides tools which support the performance of these activities and the method used. The criteria for selection of these methods is that each should be structured, well defined, and tested. The methods used for FED are:

- Structured Analysis (SA), variation for real-time system
- Object Oriented Design (OOD)
- Structured Ada Programming
- Stepwise Refinement
- Bottom up Testing

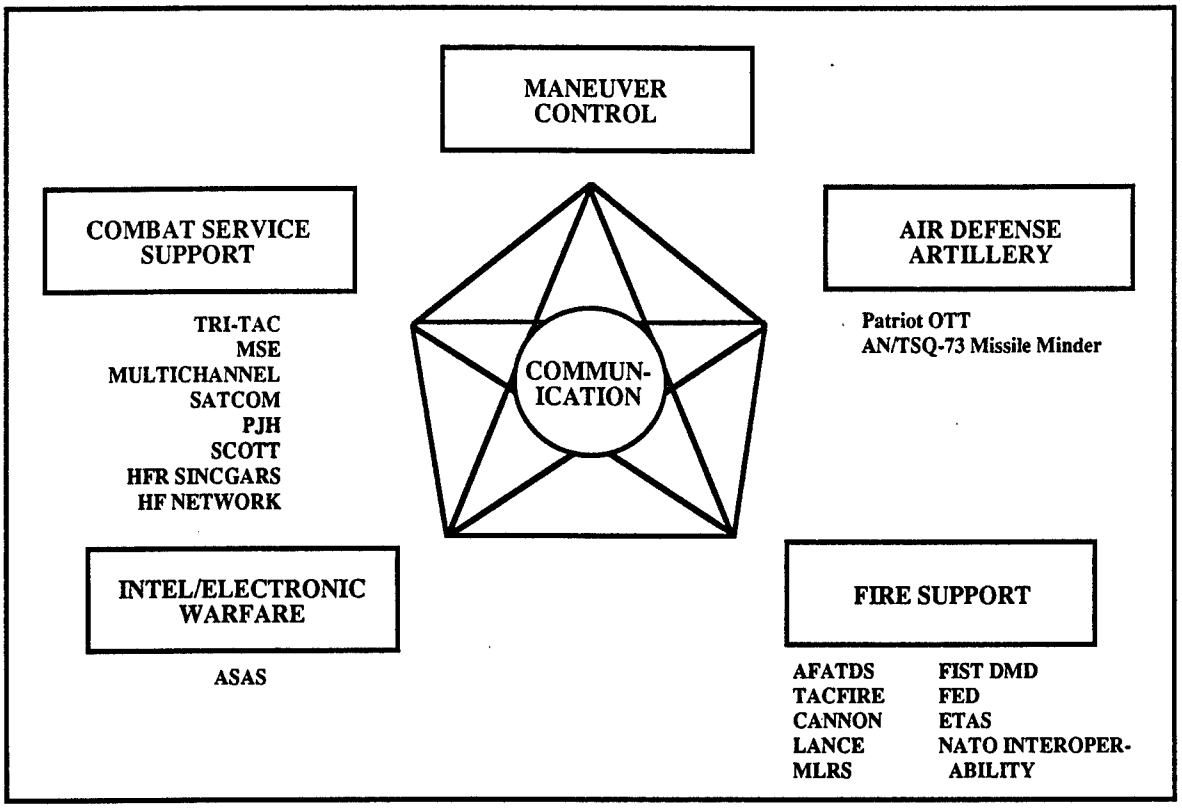


Figure 1: Army Tactical Command and Control System (ATCCS)

Mapping from Analysis to design has been a problem. The selection of OOD is favored because it does provide mapping from design to the Ada language and utilizes its features.

CASE TOOL

Two years ago, there were not many CASE tools available in the market. The selection criteria for a CASE tool was:

- User friendly
- Support 2167A and DIDs
- Support SA
- Support OOD
- Support requirements traceability
- Reverse engineering
- Vendor's support
- Vendor's reputation
- Vendor's training facility
- Support reuse repositories
- Support centralized database
- Support Configuration Management
- Provide automated source code generation
- Provide automated test cases
- Compliance with ANSI-MIL-STD-1815A
- Provide automated quality control
- Well documented
- Easy to learn
- Provide automated project management

We selected PROMOD on the basis that this CASE tool supports all phases of software developments. The vendor satisfies most of the established evaluation criteria. The CASE tool is geared towards Ada and DOD-STD-2167A. And the price of the CASE tool has been within our budget.

CHRONOLOGY OF EFFORT

The effort began on 15 December 1987. At the time the project began MIL-STD-2167A had not been approved. As the project progressed and official approval had not yet been granted it was decided to lock into the draft 2167A, dated April 1987. The project was originally planned to be a twenty-four month effort with the system to be delivered at the end of the eighteenth month. Final documentation was to be delivered within the following six months. Due to the accelerated system delivery, an agreement was reached to deliver only a subset of the documentation required by 2167A Draft. The remainder of this section of the paper will describe what was developed for the System Requirements Analysis Phase, the Software Requirements Analysis Phase, the Preliminary Design Phase, the Detailed Design Phase, the Coding and Unit Testing Phase, Computer Software Component (CSC) Integration and Testing Phase, and the Computer Software Configuration Item (CSCI) Test Phase.

SYSTEM REQUIREMENTS ANALYSIS

During the System Requirements Analysis Phase the principal tasks were to develop a Requirements Summary List (RSL), a Requirements Definition Document (RDD), a Software Development Plan (SDP), a Software Requirements Specification (SRS), a Interface Requirements Specification (IRS), and twelve Interface Specifications (ISs).

The SDP presented the plan for the development and management of software for the FED. The plan included: required resources and organization; development schedule, including milestones and deliverables; software development procedures addressing software standards and procedures; and software development tools, techniques and methodology. These portions of the SDP were developed with two basic goals in mind. Number one, the system had to be portable since at the beginning of the project the target processor and operating system were unknowns. The way we solved the portability issue was to isolate all hardware and operating system dependencies within our design. Chapter 13 of the Ada Language Reference Manual contains features that may or may not be implemented by a given compiler. Those features were either restricted or isolated and laboriously documented as specified by very stringent coding standards that were developed for the FED. In this way if the target compiler did not have any Chapter 13 features implemented, which we had used, it was only a matter of changing the affected units, as opposed to making wholesale changes throughout the system due to the ripple effect. All units or CSCs that had a direct hardware interface were encapsulated. Envision a circle with a line drawn through the middle, in effect dividing it into two halves. One half of the circle contains the known information, while the other half of the circle contains the unknown. The half with known information would be stable and would not change no matter what the target hardware. The only thing that would have to change would be the unknown half of the circle. The unknown portion of the circle would contain only the code necessary to convert the known to the format required by the unknown. This technique worked quite well on this project. This technique was the major reason for achieving a high degree of portability.

The second goal was to develop a system that was easily maintainable. The methodology chosen for requirements analysis and preliminary design was not nearly as successful as the technique employed for achieving portability. We traveled down a road others had tried and failed many times. We had read and heard all the horror stories concerning those brave souls who had attempted to use structured analysis during the requirements analysis and then magically transform from SA to OOD. We were sure we could do better, since we had a better plan for making that transformation. Within the SDP we defined a mechanism that we felt would make the transformation in the real world, not magically but with a predefined set of steps. The idea was to develop context diagrams, data flow diagrams, and state transition diagrams for the analysis portion, and to take those diagrams one step further. SA is intended to define the functional requirements of a system. The requirements diagrams described above were to serve as a baseline for the development of implementation diagrams. To put it simply, the implementation diagrams were to build on the requirements diagrams by adding-in the derived and implied requirements, along with any implementation decisions. The process symbols, which were by now actually packages or units, were then regrouped together with other packages or units which operated on or were operated upon by like entities. These groupings were then identified as objects and became packages or a package of packages. As many others before us had discovered we ended up with a quasi-functional object oriented system. We did however recover through perseverance and dedication. Although you could not by any stretch of the imagination say that our design was truly object oriented, we did achieve a design that will be easily maintainable and is extremely portable.

A RSL was developed during the first month of the project. The RSL is a list of all the user requirements stated in a series of one liner statements. The purpose of the RSL was to insure that all parties agreed upon the operational requirements.

The RDD and the ISs were developed in lieu of the System Segment Specification (SSS) which had not been developed by the user because of time constraints and was subsequently removed per requested tailoring. The actual functional baseline also included the Prime Item Specification for the Fire Support Team Digital Message Device (FISTDMD) since the FED was a modification and enhancement of the FISTDMD system. The FISTDMD Prime Item Specification along with the RDD served to detail the user requirements that would normally be described in the SSS. The twelve ISs served to define the interfaces to the twelve external systems with which the FED was required to communicate. The external interfaces would normally have been defined in the SSS.

SOFTWARE REQUIREMENTS ANALYSIS

The products developed during the Software Requirements Analysis Phase were a Software Requirements Specification (SRS), and a Interface Requirements Specification (IRS). The SRS defined the requirements, including the implied and derived requirements, for the FED. The requirements diagrams, along with associated mini-specifications (a textual description of the processes and states), previously mentioned were included in the SRS. The diagrams did not replace anything required by the Data Item Description for the SRS, but were included to further facilitate user understandability.

The IRS, as published during the software requirements analysis phase, was only a skeleton of a document. The IRS is intended to be the interface document for CSCIs within the whole system. Since we did not know what hardware or software we would be interfacing with, we could only establish a very preliminary document. We did however update and republish the document when details became known.

PRELIMINARY DESIGN

The document produced during the preliminary design phase was the Software Top-Level Design Document (STLDD). The STLDD contained the implementation diagrams previously discussed. Along with the implementation diagrams, the package specifications and bodies of all the CSCs were included. The STLDD is not required under the approved version of 2167A. The authors of 2167A made a truly wise decision when they decided to eliminate this document. It was the most troublesome deliverable on the whole project. I will relate all the grand and glorious problems associated with the STLDD in the following section.

DETAILED DESIGN

The Software Detail Design Document (SDDD) was the product created during the Detail Design Phase. The detailed design for the FED consisted of Ada Design Language (ADL).

We bowed to the proponents of ADL and decided to do our design using ADL instead of some form of Pseudo Design Language (PDL). The ADL was syntactically correct Ada statements. The intent was to use comments in place of Ada statements whenever details were not easily understood. This was also a problem during the project, although not easily discerned by the untrained eye. The problem with the use of ADL is the tendency by designers to become caught up in the details. It is too easy to think in terms of code instead of design. So, as an end result you end up with Ada code instead of ADL as we found out. The benefit, as proclaimed by the proponents of ADL, is truly evident. You do have a detail design that very closely resembles the end product. In fact, in many cases it is the end product. The ADL, however did constitute the bulk of the SDDD. For much of the remainder of the paragraphs of the SDDD we referred to the ADL section which contained the Program Management Shell. The Program Management Shell is a comment block at the beginning of each source file which contains cross reference information, identification information, declarations, procedures called, instantiations, and a description of the package or unit.

CODING AND UNIT TESTING

Nearly one year, and several large bottles of aspirins later, we finally arrived at the start of the code and unit test phase. There were no true deliverables for this phase. This phase is where we discovered if our analysis and design would hold water, and it was the smoothest phase of all. The coding was accomplished with relatively few problems. Development of unit test procedures was a little slower than we had anticipated, due to the excessive amount of time necessary to update and republish previously mentioned documents, but was accomplished with few problems. The system had been designed in such a way that many of the test cases were almost duplicates of others. The test cases were documented in the Software Development Folders (SDFs). A prototype of the system, executing on an AT compatible computer was made available as each piece of the puzzle fell into place. There were numerous releases of the prototype made available for user, test, and training personnel evaluation. We maintained a log of any problems identified by these evaluations. This was very beneficial to our test effort as problems were evaluated and fixed as discovered.

COMPUTER SOFTWARE COMPONENT INTEGRATION AND TESTING

This phase was accomplished through the use of the products developed in the previous phase. Since all of the CSCs comprising the FED system were either Ada packages or packages of packages the CSC testing was accomplished by linking together all of the units or packages together and executing all of the unit test cases for that package.

COMPUTER SOFTWARE CONFIGURATION ITEM TEST

The Software Test Description (STD) and Software Test Plan (STP) were delivered prior to the beginning of the Preliminary Formal Test (PFT). The PFT had not begun as of the writing of this paper. The PFT is the final test conducted by TELOS for the FED, with the system being delivered immediately following the completion of the PFT. The Final Qualification Test (FQT) to be conducted by an Independent Validation and Verification contractor will follow the PFT. An Operational Test conducted by user personnel will follow the FQT.

CONSTRAINTS AND PROBLEMS

There were many constraints placed on the FED project from the very beginning.

- Accelerated system delivery date
- Delays in award of the ATCCS contract
- Delays in approval of MIL-STD-2167A
- Documentation required for MIL-STD-2167A Draft
- Lack of experience with documentation standards, design methodology, and the Ada language

The accelerated system delivery date was necessitated by the user goal to field the FED concurrently with the fielding of Lightweight TACtical FIRE (LTACFIRE) Direction System.

The ATCCS contract, when the FED project began, was scheduled to be awarded in March of 1988, with the initial HTUs to be delivered 90 days after contract award. The contract award date was later slipped to June and finally awarded in September of 1988. The first production HTUs were then scheduled to be delivered in June of 1989. We dealt with this problem as mentioned previously by creating a system that is highly portable.

The delays in approval of MIL-STD-2167A forced us to use a non-standard version. This in turn created the biggest headaches since we had to develop a STLDD.

The STLDD was truly a nightmare. The STLDD is not required for the approved version of 2167A. Thank goodness. The STLDD was a document that required too much detail at a much too early stage. The STLDD was also the reason for a perceived traceability problem. Reviewers of the STLDD were expecting to find a direct correlation between the requirements defined in the SRS and the names of CSCs contained in the STLDD. They also were expecting to be able to see where all requirements, defined in the SRS, were implemented in the STLDD. Since the STLDD only defined and identified the CSCs with no consideration given to all of the units, it was impossible to show where all of the lower level requirements were implemented in design. We could at that point only identify where the requirement would be implemented, not show the actual implementation of each requirement. Further compounding the perceived traceability problem were the requirements diagrams and the implementation diagrams. It was very difficult to explain to outside agencies why the requirements diagrams did not match the implementation diagrams. The outside agencies did not fully understand the intent of showing a structured picture of the functional requirements and a similar, but different picture detailing the implementation decisions. Finally, through the course of several meetings, the picture became a little clearer and the reviewers understood what the different diagrams contained.

The lack of experienced personnel is always a problem. It was especially a problem for this effort since very few systems have been developed even with 2167, much less 2167A. The methodology was untested and no one had experience since we had come up with the idea shortly before the project started. While we had a core group of people with actual Ada experience and several others who had been trained beforehand, we were definitely lacking in years of actual

experience. The personnel involved in this effort were dedicated and attacked the problems we encountered with fervor. Those people are directly responsible for the success of this project and should now be considered very experienced. They make planning for the next version much easier, and eliminate a lot of the risks associated with this type of effort.

REFERENCES

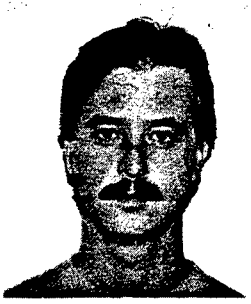
1. U.S. Dep. Defense, Military Standard, Defense System Software Development, DOD-STD-2167A, Washington, D.C., 29 February 1988.
2. Prime Item Development Specification for the Fire Support Team Digital Message Device (FIST DMD), AN/PSG-5, dated 15 October 1983, as modified by Specification Change Notice 002, dated 26 November 1985.
3. Software Requirements Specification for the FED, dated 30 October 1989.
4. Interface Requirements Specification for the FED, dated 21 December 1989.
5. Software Top Level Design Document for the FED, dated 7 December 1989.
6. Software Detailed Design Document for the FED, dated 14 December 1989.
7. Interface Specification for Forward Entry Device Interface with FED, dated 23 November 1988.
8. Interface Specification for Lightweight Tactical Fire Direction System (LTACFIRE) Interface with FED, dated 21 December 1988.
9. Interface Specification for Advanced Field Artillery Tactical Data System Interface with FED, dated 21 December 1988.
10. Interface Specification for Fire Support Team Digital Message Device Interface with FED, dated 23 November 1988.
11. Interface Specification for TACFIRE Digital Message Device (AN/PSG-2) Interface with FED, dated 23 November 1988.
12. Interface Specification for MLRS-Lance FDS (AN/GYK-29) Interface with FED, dated 21 December 1988.
13. Interface Specification for Battery Computer System (AN/GYK-29) Interface with FED, dated 7 December 1988.
14. Interface Specification for Tactical Fire Direction System at Battalion and FED, dated 7 December 1988.
15. Interface Specification for Tactical Fire Direction System at Brigade/Corps/Division Artillery (AN/GSG-10) Interface with FED, dated 7 December 1988.
16. Interface Specification for M109A3/E2/E3 Automated Fire Control System Interface with FED, dated 21 December 1988.
17. Interface Specification for Mortar Ballistic Computer (M23) Interface with FED, dated 23 November 1988.
18. Interface Specification for Airborne Target Handover System Interface with FED, dated 7 December 1988.
19. Software Development Plan for the FED, dated 29 April 1988.
20. Forward Entry Device Requirements Summary List (RSL), dated 21 January 1988.
21. Requirements Definition Document for Forward Entry Device, dated 31 March 1988.
22. Draft Equipment Publication, Operator's Manual Forward Entry Device, dated 15 November 1989.
23. System Test Plan for the Forward Entry Device, dated 6 November 1989.
24. Computer Software Configuration Item Test Plan for the Forward Entry Device Preliminary Formal Test, dated 27 November 1989.
25. Software Test Description for the Forward Entry Device, dated 6 November 1989.



JAG SODHI

TELOS Systems
P. O. Box 33099
Ft Sill, Oklahoma 73503-0099

JAG SODHI has a Master Degree in Mathematics, a Degree in Telecommunication Engineering, and is a Graduate of IBM in Data Processing. Jag has over 28 years of Data Processing experience in business, financial and scientific applications using various Electronic Data Processing (EDP) machines and languages. He has conducted numerous professional classes and seminars on these subjects. His publishing credits include two books: Computer Systems Techniques: Development, Implementation, and Software Maintenance, and Managing Ada Projects Using Software Engineering, numerous training courses on Ada and software engineering. Jag is a senior system engineer and is in charge of education and training at TELOS Systems, Fort Sill Project. He has since joined CECOM, Fire Support Systems, Center for Software Engineering, at Fort Sill. Jag is also an Adjunct Professor at Cameron University.



GUY DAUBENSPECK

TELOS Systems
P. O. Box 33099
Ft Sill, Oklahoma 73503-0099

GUY DAUBENSPECK has over 13 years experience in designing, developing, and implementing data processing applications on various computers and languages. He has worked on the Forward Entry Device system, which is written in Ada, since its inception. Guy is a senior system engineer, managing the Forward Entry Device system for the Command and Control Branch of TELOS Systems, Fort Sill Project.



TOM ARCHER

TELOS Systems
P. O. Box 33099
Ft Sill, Oklahoma 73503-0099

TOM ARCHER has a Bachelor's Degree in Mathematics with a minor in Computer Science and Applied Physics. He has over 27 years experience in designing, developing, and implementing Government applications of various EDP machines and languages. He has worked on numerous weapon system projects such as the E-3A Airborne Warning and Control System (AWACS), the M-X Missile Development, and currently is assisting in various Fire Direction Ada language system developments. Tom is a senior system engineer, currently involved in technical analysis and management support for Fire Direction Systems Branch of TELOS Systems, Fort Sill Project.

DEVELOPING AN IN-HOUSE VHDL CAPABILITY

Steve Adamus Karen Crossland

SAIC, COMSYSTEMS Division, San Diego, California

Summary

Software model developers with relatively little experience with a hardware description language are using VHDL, the VHSIC Hardware Description Language, to express their designs. Though many VHDL and Ada constructs are similar, the similarities occur primarily at the lower levels of a VHDL model. An initial comparison to Ada makes it difficult to accurately estimate the VHDL learning curve or the resources that will be required for a VHDL simulation. The VHDL model is both a behavioral model composed of procedural statements and a structural model consisting of concurrent components interconnected by signal lines. Developers of VHDL models must have knowledge of hardware design concepts and of the organization of a digital electronic system.

Introduction

VHDL provides designers of digital electronic systems with a method to describe their designs in a simulatable language. A system expressed in VHDL consists of a set of interconnected components. The components and their interconnections correspond directly to hardware and, when executed, exhibit the same behavior as the hardware would. This allows designers of custom made integrated circuits to determine the performance of a design before the component is put into production enabling them to avoid costly design errors. VHDL was adopted as IEEE standard 1076 in mid-1987; the DoD requires VHDL for documentation for all ASICs (Application Specific Integrated Circuits) developed in military contracts after September 1988 (MIL STD 454, Requirement 64).

This paper presents technical and management-related issues that first-time VHDL users should consider. These issues are the results of our first modeling effort using VHDL. Our task was to model the data link layer of the VHSIC Parallel Interface Bus (PI-Bus) and the Test and Maintenance Bus (TM-Bus). Each bus was implemented as a separate model that included all necessary support functions so that it would be a complete stand-alone system.

Models of the VHSIC bus protocols had previously been written using other simulations languages. What was needed now was a VHDL model to be integrated with the designs produced by the VHSIC contractors. Because of the significant differences between VHDL and languages that were previously used to model the bus protocols, translation of an existing model into VHDL was not feasible. The VHDL models were developed from scratch.

Our first impressions of VHDL were that the language was an Ada-like language. We have used Ada in simulations in the past and expected to be able to use VHDL similarly. The VHDL books and tutorials available to us stated that the targeted audience was expected to have "experience with hardware logic design," or "a working knowledge of digital hardware design." These statements were overlooked (after all, we were not going to be doing any hardware design).

As we started working with VHDL, we discovered the importance of these statements and the fundamental differences of VHDL from other high-level programming languages. VHDL is a hardware description language, and is intended for use in all phases of the creation of electronic systems. It supports the design, verification, synthesis, and testing of hardware, as well as communication of hardware design data.[3]

VHDL can be viewed as an extension of an Ada subset.[1] Some software constructs have been eliminated, but additional constructs have been added to support hardware design. Also, some Ada constructs have been extended to support the concurrent environment.

Development of the Models

We derived the requirements for the VHSIC bus models from the VHSIC Phase 2 Interoperability Standards PI-Bus Specification and TM-Bus Specification. Requirements for a communications path between the Bus Interface Unit and an application specific device were identified and compiled into back end bus interface requirements and design documents. Additionally, message reader and message generator functions were identified and placed within the application specific device. We did not identify any VHDL-unique issues during the requirements analysis phase.

We moved from requirements analysis to design, and decomposed the design into hardware components. Our goal was to provide a generic model of the VHSIC bus protocols, implying no particular hardware implementation. In line with this goal, we wanted to

limit the amount of structural decomposition as much as possible. Also, decomposition would add components and signals to the model.

Signals are time-oriented interface objects that are used to connect components. Execution of a model consists of the repetitive execution of a simulation cycle. On each simulation cycle, the VHDL simulator (1) determines any new signal values in the model, (2) propagates the new signal value throughout the design and identifies any processing that is to be started as a result, and (3) activates the identified processes. In addition, histories for all signals in the model are maintained. Minimizing the decomposition would keep the quantity of signals in the model small and help make the simulation more efficient.

Implementation

The decomposition process continued until all low-level components were identified. Specifying the behavior of the components was accomplished with the process statement.

The VHDL process statement defines an independent sequential process representing the behavior of some portion of the design.[2] The process presents itself as a concurrent element to the rest of the design, but consists of procedural statements. This places the process as the gateway between the concurrent environment and the sequential behavior of the design. Signals are used to coordinate communications between processes.

Variables vs. Signals.

VHDL provides for a class of objects called signals that are used to model the data pathways between components. Signals differ from variables in a number of ways. The signal is an abstraction of a wire and has a history of values. Variables have only a single value, have no direct hardware representation, and are used in algorithms. The effect of an assignment to a variable is felt immediately. In contrast, a signal assignment statement specifies the value of the signal after some delay. The symbol for the signal assignment operator is different from a variable assignment operator; this helps to enforce the conceptual differences.

The future values of a signal are stored in a driver for the signal. A different driver is created for each process that assigns a value to a signal. Thus, a signal may have multiple drivers. If a signal does have multiple drivers, different values may be contained in each of the drivers, and a resolution function is required to compute the value of the signal.

Resolution functions are written by the user for each signal type that has multiple drivers. A signal that is declared to be of a resolved type is called a resolved signal. When a resolved signal is to receive a new value, the resolution function is implicitly called to determine the new value of the signal. The resolution function is passed the values of all of the signal's active drivers and may perform the function of a Wired-And or Wired-Or gate to determine the new signal value.

Users that are new to VHDL can fall victim to misunderstandings with signals, and we were no exception. In an early version of our model, we needed to test the response of one component to a signal generated by another. The code was not yet completed for the source of the signal, so an initializer on the signal declaration was used to set the signal to the desired value. The explicit initializer had no effect on the value of the signal. Although we were convinced that a bug in the compiler was causing the problem, we consulted the language reference manual. An initializer for a signal does not determine the initial value of the signal, but rather the initial value for the driver of the signal. At the level of the hierarchy that our signal was declared, there were no drivers. The components that were included and interconnected by the higher level module had their own signals and drivers, and the values of those signals were being propagated.

Another time, a portion of a state machine set a state signal to indicate what the subsequent state would be, performed the processing associated with the current state, then tested the state signal that was set up earlier to determine a proper exit. The state signal had not yet received the value of the new state: it was still the same simulation cycle.

In still another case, a signal was set by an element to alert a process that an operation had to be performed. After the process had completed the operation, the process attempted to clear the signal, but the signal would not clear. The element that initially set the signal was still driving it. Only that element could clear the signal.

Decomposition.

During implementation, we found that VHDL would not allow us to express design features that were successfully implemented in a model written in Ada. For example, creating a queue as an interface object between two processes is not permitted because variables cannot be shared by more than one module. The only variables allowed in a VHDL design are those variables declared to be local to a process, procedure, or function. Also, the only variables that persist throughout a simulation are those that are declared within the process.

We decided to place all our queues, registers, and other data structures within a process. All operations that required direct access to the data in those structures would have to be local to the process. What resulted were rather large modules that lacked cohesion.

As the size and functionality of the processes grew, so did the need to decompose the process into more cohesive modules. Our initial impulse was to divide a process into multiple processes. The difficulty with this was the sharing of data stored in the local data structures. For example, a set of registers provided common data for the entire state machine.

Also, since all processes execute concurrently, communications protocols had to be implemented to allow the orderly exchange of data between processes. Further, a state machine is a sequential circuit, and decomposing the state machine into multiple concurrent processes seemed rather unnatural.

We chose a method of decomposition that removed operations from the process and placed that code into procedures. These operations were primarily the processing associated with individual protocol states of the bus state machine. Any object that needed to persist over time, such as state variables or data structures, could not be removed from the process.

With the functionality of the process relocated to procedures, interface lists grew quite large as all data structures that were formerly visible to the code now had to be explicitly imported. Most of the individual protocol states required access to the majority of the bus signal lines, and these also had to be imported into the procedures.

One of the problems associated with this method of decomposition was that much of the functionality of the model was now subordinate to a single process. The VHDL simulator provides visibility down to any signal line in the model, but not to any statement or variable local to the process. Complicating matters, the toolset that we used did not have a symbolic debugger.

The reason that we chose the toolset that we used is simple: at the time that our project was funded, only one toolset was available. To this date, it is still the only toolset that provides a full implementation of VHDL 1076. Because we have no symbolic debugger, execution of the processes and procedures, as well as visibility into all our data structures, was hidden from us. Our model had become a black box.

In retrospect, a more ideal method of decomposing the design would have been to represent the queues and registers in the model as hardware elements e.g., as a separate entity or as a block within an existing architecture. Additional signal lines such as address bus lines, data lines, and read and write control lines would have been required, but the resulting design would have been better organized, resulted in more manageable components, given us greater flexibility in implementation, and provided a model that was less of a black box.

Our lack of hardware design knowledge prevented us from implementing, or even conceiving, a more logical hardware decomposition. A digital electronic system may be broken down into microprocessors, RAMs, buses, etc., which in turn may be broken down into smaller hardware components such as ALUs (Arithmetic Logic Units), registers, multiplexers, etc. The VHDL designer needs to be able to conceptualize these components and then model them using entities, blocks, and processes.

Concurrency.

When processes or components need to communicate, the user must synchronize the two processes for the duration of the communication. In a sense, the developer must implement a parameter passing mechanism. For the hardware engineer, using a strobe to gate data into a component or performing a handshake between two components may be intuitive. The software engineer, however, is at a disadvantage. The software engineer lives in a sequential programming environment and uses a language that guarantees that actuals in a procedure call will be properly associated with the formal parameters. Hardware mechanisms that perform such functions are unknown to most software engineers.

One of the biggest difficulties that we as software engineers had with VHDL was adjusting to the concurrent environment. An Ada program is primarily a description of sequential activities. An Ada task may execute in parallel with the rest of the environment, but the environment is primarily sequential, and the developer may think of the tasks as an exception to the environment. Even if the program is primarily composed of tasks, each task executes independently of each other and sequentially within itself. Tasks "behave" themselves: interaction between tasks occurs only after a task call is made and the rendezvous occurs. The VHDL environment is a concurrent one, and the magnitude of the concurrency can be surprising. Tens, hundreds, or even thousands of concurrent processes may be executing. One process may assign a value to a signal, and that signal may activate the execution of dozens of other processes that are sensitive to that signal, sometimes inadvertently. For example, the output of an And gate is a product of its inputs - no one has to "call" the And gate for it to perform its function. Concurrency is the rule, not the exception.

Developers of a VHDL model must establish timing conventions, document those conventions, and adhere to them rigidly. For example, some of our timing conventions are as follows:

- all state transitions occur on the high-to-low transition of the clock,
- all processes write data to an interface on the high-to-low transition of the clock,
- all processes read data from the interface on the following low-to-high transition of the clock,
- each process guarantees that data will remain on an interface for one full clock cycle.

Adhering to these conventions and others helped us overcome some problems we encountered while implementing asynchronous communications between processes, including: (1) interfaces that locked and resulted in no data being passed, and (2) a second read that was activated from an interface (and the data was processed as a different object) while the source had not yet removed the first data object from the interface.

VHDL vs. Ada.

There were other fundamental differences between VHDL and Ada that caused our model to be

implemented differently than if it had been an Ada model. Ada allows many program objects to be dynamically allocated at run time. In an Ada model, a simulation can read in the number of nodes on a network at run-time and then instantiate those nodes. In a VHDL design, each node corresponds to a hardware component. Dynamic allocation of hardware components does not occur in the real world and is not allowed in a VHDL description. Model configuration information must be placed into a design unit and compiled prior to constructing an executable model.

The concept of generic compilation units exists in VHDL, but in a restricted form. Design entities can receive generic parameters which are then treated as constants. Thus, a clock component can be instantiated with the clock rate being passed in as a generic parameter. However, generic packages and procedures are not supported by VHDL. Because of this, we wrote separate queueing routines to store different data types.

VHDL Weaknesses.

VHDL formalizes the concept of object class and identifies three classes of objects: constants, signals, and variables. When an object is passed as a formal parameter to a subprogram, VHDL enforces class compatibility in addition to mode and type compatibilities, and from this inconsistencies and ambiguities arise. It seems redundant to say that a formal constant is of mode "in" (no other mode could be allowed). If a formal parameter is of class variable, then only a variable is allowed to be the actual designator in a procedure call (no literal or expression can be used as the designator in the procedure call even if the formal is of mode "in").

These inconsistencies are most evident in file I/O. Several TextIO procedures have the wrong mode associated with their formal parameters. Functions that need to accept a file object or other variable parameter are in conflict with the language, as formal parameters to functions must be signals or constants. With a strict interpretation of the language rules, a file object cannot be passed to function EndFile. A parameter "L" to function TextIO.EndLine is a constant (as enforced by the language), but needs to be a variable. In this last case, a recommendation was made by the VHDL Analysis and Standardization Group to delete procedure TextIO.EndLine, and in its place, use the expression "L'Length = 0". A problem exists even with this recommendation. "L" is an access type (pointer to a string), and the Length attribute is appropriate only on array types and not access types. In the latest version of our toolset, the language rules have been relaxed so that the expression would be allowed.

VHDL Toolsets

Compared to software development, hardware design tools are more specialized and hardware design in general is more automated. The VHDL toolsets on the market reflect these differences, and each toolset is tailored to a different use.

Some toolsets have a graphical interface to allow for schematic capture. Such a toolset may generate a VHDL description for the designer directly from the schematic diagram.

Other toolsets were developed primarily for synthesis: compilers that translate the VHDL directly into silicon. One implementation is a VHDL front-end to a proprietary language designed for a simulation accelerator that allows a great reduction in the simulation time of large gate-level models. Still other toolsets provide powerful symbolic debuggers.

Most of the toolsets have implemented a subset of VHDL 1076. We know of only one toolset that provides a full implementation of the language, and that is the toolset we are using. Unfortunately, it also compiles slower than most of the others.

None of the toolsets on the market can be considered mature tools, i.e., relatively error free. We lost about five weeks during development because of time spent debugging the toolset and finding suitable workarounds.

Model Documentation

Documentation of a software program often includes structure charts, data flow diagrams, and other illustrations that help to depict the organization and function of the program. A VHDL design has direct hardware representation in that it is a collection of elements interconnected by signal lines. The documentation of a VHDL model should include appropriate wiring diagrams, functional block diagrams, and state diagrams.

Unlike other software modeling where parameters are passed among components via the language, using VHDL to pass data among concurrent elements often requires the developer to create a mechanism providing orderly transfers. For example, one process may use a signal to strobe data from one process to another, or a more complex handshaking protocol may be used. In these cases, the synchronizing technique used must be clearly documented in a header to the process.

Testing

Testing the model required considerably more time than we originally anticipated. This was primarily caused by the inability to view the inner workings of the model. Test drivers often had to be built. Sometimes test code was embedded into the source. Every modification to the source code required a rebuilding of the model, and the long recompilation times that we experienced further impacted the length of testing.

Under out toolset, rebuilding the model and performing a simulation requires five steps:

- compiling a VHDL module to an intermediate form (analyzing),

- compiling the intermediate form and producing object code (model generating),
- linking and producing an executable kernel (building the model),
- running the kernel (simulating),
- generating a report to view the results.

Depending on the level of rebuilding that was required, building a new model consumed 10 to 90 minutes (13 - 27 minutes being typical).

Without the ability to view the operation of the model, several builds were often required to even discover where the bug was located. Extracting even trivial bugs sometimes required the better part of a day.

As the model was being tested, we realized an unexpected benefit from structural (hardware) decomposition. As the model was decomposed, additional signal lines were placed between the new components. As a result, the size of the black boxes was reduced, and the number of signal lines was increased providing added visibility into the model. The number of additional signal lines was small and did not noticeably affect the length of a simulation. The time we expended in the structural decomposition process was more than returned during the testing process. Structural decomposition, which we originally were opposed to, became one of our biggest allies.

Requirement for Alternate Test Methods.

No source level debugger was available, and the simulator provided no visibility into the processes and procedures in the model. When an error was detected, performing a code walkthrough was often the most productive method of finding the error.

An alternate debugging method that we used consisted of creating a temporary signal, assigning the value of a variable of interest to the signal, performing a simulation, and doing a signal trace on the temporary signal. If multiple elements of a data structure needed to be viewed, a wait statement would be inserted (e.g., "wait for 1ns;") between each assignment. A trace on the signal would then show each individual value that was stored.

We also used the VHDL assert statement as a debugging tool. The assert statement allows the designer to embed design information in the code. The design information might be operating constraints that would be expressed as a boolean expression in the assert statement. If an assertion is ever violated during the simulation, a message associated with the assert statement is printed. We used the assert statement in a very general sense to test any condition in the model. The simplest case was to assert the condition "False." This always resulted in an assertion violation when the statement was executed, and the developer would be alerted of the exact simulation time that the module containing the assert statement was executed.

Both of these alternate debugging methods suffered from having to modify (sometimes quite heavily) the source code. Once the bug was removed from the code, the debugging software itself had to be removed.

VHDL Test Benches.

Testing or simulation of a VHDL component is accomplished by a component known as a test bench. A test bench typically instantiates the component, stimulates the component by supplying input data, and records the component's responses. With global variables eliminated from the language, the test bench's visibility into the component under test is restricted to the outputs provided by the component. Thus, once a VHDL component is built, there is no way to perform an unanticipated test function unless the component's interfaces provide the means.

In order accomplish all necessary test functions, test requirements must be defined prior to implementation of the model. Interfaces can then be designed to accommodate the test requirements. Some components may require additional interfaces and circuitry that would be used only during an off-line test.

In a software model, test software (drivers, stubs, etc.) may be developed as throwaway code. Test software may be retained so that formal tests can be repeated. In the development of a VHDL model, the test benches are often part of the deliverable code. Further, the government's Data Item Description for development of VHDL models directs that a test bench be constructed for each component in the model. Test benches are often quite complex, and it is not unusual for the test benches to require as much effort to construct as the model itself.

Resources Required

The development of the VHSIC bus models consumed a large amount of computer resources. Early in our development, we had several system errors because primary and secondary storage was being exceeded. Our system manager increased our allocation of storage space until nearly the entire computer system was dedicated to us. The compilation times that resulted from having a dedicated system were significantly less, but still slow.

Development of our model was performed on a MicroVAX under version 4.7 of the VAX VMS operating system. Primary storage was 8 Mbytes RAM. Increasing RAM beyond 8Mbytes is recommended.

VHDL Libraries, managed by the VHDL Library System (VLS), contain design units, simulation runs (signal histories), and the run-time executables. One of our libraries contains 170 files and consumes 3618 kbytes of storage. The components that make up the toolset itself (VLS, Analyzer, Model Generator, Build, Simulator, Report Generator) use an additional 6574 kbytes.

Both the PI-Bus model and the TM-Bus models are still under development. The PI-Bus model, the

larger of the two models, currently is composed of 40 library units, 14 of which are for the entity declarations and architectures of the 7 components in the model. The remaining library units are package declarations and package bodies. Since project startup, 10 man-months of effort have been expended on the PI-Bus model. The source code for this model currently contains over 4700 executable lines of code.

Considerable processing resources are required to perform a simulation of the PI-Bus model even though the model is primarily behavioral. The model contains 169 behaviors and more than 4300 signals. For the simulator to initialize the model, over 2600 transactions are required. Running a 4-microsecond simulation of PI-Bus activity consumes in excess of 3 minutes running time and requires almost 28,000 transactions. From that one 4-microsecond simulation, 1064 kbytes of signal histories are generated and stored in the design library.

The Simulator program stores the signal histories from each simulation run in the design library. Because of the large size of the individual runs stored in the library, storage of multiple runs results in secondary storage being exceeded.

Lessons Learned

At the beginning of our effort, we made assumptions about VHDL and the amount of time required to become familiar with the language. Some of our assumptions proved erroneous. The lessons that we learned include:

- the learning curve associated with VHDL can be significant,
- the developer of a VHDL model needs to possess some knowledge of hardware design concepts,
- each VHDL toolset was written for a specific application, and each toolset will affect productivity differently,
- the tools that are available are not mature, and the availability of symbolic debuggers is limited,
- the testing and debugging of a VHDL model can be more difficult and take more time than an equivalent model written in Ada,
- in terms of the computer resources that are required, VHDL simulations are expensive.

We assumed that the VHDL learning curve would not be significant because we had experience with using Ada for simulations. We experienced reduced productivity for several months while we learned VHDL. Learning the syntax of the language was not difficult. One problem we had was understanding the hardware design concepts behind the language and then applying them to our environment. Another problem was the concurrency that was a product of our design. Even after building and debugging several of the model's components, the operations of those components were difficult to grasp because of the large amount of concurrency that those components exhibited.

At a recent VHDL User's Group meeting, the experiences of those involved in training both hardware and software engineers in the use of VHDL were related. It appears that electrical engineers adapt much more readily to VHDL than software engineers do. This is particularly true with the hardware design and concurrent aspects of the language. If the electrical engineers have a stumbling block with the language, it is with the sequential aspects of the language. "Hardware does not work that way."

As software engineers, we had considerable trouble with the language. We did not have experience with a hardware description language or with hardware design. We had some experience with concurrency, but that was within the context of a sequential language such as Ada.

VHDL is not a general purpose programming language. VHDL is a general purpose hardware description language, with the emphasis on hardware description. Our attempt to treat VHDL as a general purpose programming language led to misconceptions and misuse of the language. Providing VHDL descriptions of the bus protocols, to be integrated with other VHDL descriptions, was the overriding requirement for choosing to use VHDL in our effort. Excepting this requirement, VHDL was probably the wrong language to use for purely behavioral models such as ours.

VHDL does have its advantages. The abstraction of a signal object, complete with a timing model, greatly simplified the modeling of the physical level of the VHSIC buses. VHDL also provides the means to write a specification, such as the VHSIC bus specifications, and execute that specification to determine the existence of any ambiguities or inconsistencies. With such an executable specification, the impact of a proposed change can be evaluated prior to actually making the change.

Hardware design experience is desirable and knowledge of hardware architecture is required before beginning a design in VHDL. The needed knowledge can be acquired during the design effort, but stopping to learn how to design and implement the hardware components increases schedule time and costs.

The queues and registers in our models should have been represented as separate concurrent components using the VHDL block or entity constructs. A more organized, manageable, flexible design would have been produced.

The documentation of a VHDL model should include appropriate hardware illustration such as wiring diagrams, functional block diagrams, and state diagrams.

The VHDL toolset that is used for development will significantly affect productivity. We

underestimated the difficulties that we would have with the toolset we used. The slow speed of our toolset and the lack of a symbolic debugger contributed to unproductive testing and debugging.

Because of the concurrent nature of VHDL, developing a symbolic debugger presents significant challenges, and most vendors have not yet marketed a product.

A behavioral model can be simulated much more efficiently than a gate-level model, but even a behavioral model consumes a large amount of computer resources during simulation. Development and simulation of our model required considerable real time, CPU time, main memory, and secondary storage.

VHDL was developed primarily as a tool to document hardware designs. VHDL is also an Ada-like language that allows simulations to be performed. The distinction between hardware and software development methodologies is becoming blurred. Electrical engineers are programming and simulating their designs. Software developers are performing simulations in a language that allows a more accurate representation of the systems being modeled. We need to train electrical engineers and computer science personnel so that they are familiar with a broader range of engineering disciplines.

References

1. R. Farrow and A. G. Stanculescu. A VHDL Compiler Based on Attribute Grammar Methodology. In *Proceedings of the SIGPLAN 89 Conference on Programming Language, Design, and Implementation.* ACM. 1989.

2. *IEEE Standard VHDL Reference Manual.* IEEE Std 1076-1987. The Institute of Electrical and Electronic Engineers, Inc., New York, NY, March 31, 1988.

3. VHDL Tutorial for IEEE Standard 1076 VHDL (Draft). CAD Language Systems, Inc., Rockville, MD, May 1987.



Steve Adamus has been with SAIC COMSYSTEMS for two years, and has a B.S. in Computer Science and an M.S. in Software Engineering. He has a hardware background in communications and computer systems, and a software background in simulations and test programs. Since coming to SAIC, he has developed communications software using the C language, performed simulations of routing algorithms for an HF communications network using Ada, and has been involved in the development of the VHSIC PI-Bus model using VHDL.



Karen Crossland has been with SAIC COMSYSTEMS for over 5 years. During this time she has developed models of the HYPERbus, the VHSIC PI-Bus and TM-Bus, and a distributed routing protocol for radio networks focusing on channel access and routing issues. These models were developed using Simscript II.5, Simscript II.5 (and currently VHDL), and Ada, respectively. Additionally, she has been involved in the development of a model of a token-ring architecture studying survivability issues and a model incorporating voice data into a tactical network. Both models were developed using Ada.

Ada in an Embedded Real-Time Environment

Debra A. Schmidt

P.P. TEXEL & Co., Inc.
Eatontown, NJ

ABSTRACT

A real-time software project with the intent of implementing an Object-Oriented Design in Ada, which additionally meets DOD-STD-2167 standards, has to consider many software engineering features. During the design phase, the intended design should adhere to the software engineering principles of abstraction, information hiding and modularity. Yet at the point of implementation and testing, the design may need to be modified due to hardware limitations, compiler limitations, timing requirements or space constraints. This paper intends to show solutions and workarounds to some project specific situations encountered by this author.

SYSTEM OVERVIEW

The system model was a message passing system. A control processor sent messages to be processed by a firmware system. The firmware processed these messages and possibly sent these messages out to other locations.

The firmware consisted of three processors and associated software. The software running on the first processor was one Computer Software Configuration Item (CSCI), named CSCI1. The software on the other two processors made up the second CSCI, named CSCI2. See Figure 1.

CSCI1

CSCI1 consisted of 12 Ada tasks. One of the Ada tasks was a low priority scheduler task that "decided" which task was to be scheduled next according to which task had work to do. The tasks were scheduled in a particular order depending on the amount of time required for each task's job, to insure each task finished its jobs within a required amount of time.

CSCI2

CSCI2 processed messages it received from CSCI1 or the modem. It reformatted the messages received from CSCI1 to a known output format for either the transceiver or modem and then passed the messages to the appropriate device. The modem or transceiver then may have sent the messages to other locations. The modem received either responses to the messages it sent out or new messages. Once the modem received either type of message it forwarded this to CSCI2 to process. Once CSCI2 processed the messages from the modem it passed these messages to CSCI1.

CSCI2 contained the same scheduler mechanism as CSCI1. The only difference in scheduling was that instead of 12 Ada tasks to be scheduled, CSCI2 consisted of 9 Ada tasks, one being the low priority scheduler.

Summary

The software configuration of each CSCI was intended to be an Object-Oriented design conforming to the software engineering principles of information hiding, abstraction, and modularity. Due to hardware and compiler limitations, timing requirements, and space constraints, some of these principles had to be compromised. The problems involved with incorrect code generation, timing, and limited memory were investigated

through the Ada constructs where these situations were seen. The solutions or workarounds used are provided.

MACHINE REPRESENTATION SPECIFICATIONS

Machine Representation Specifications permit Ada software to talk to the hardware and to interface with other languages. Problems with two (2) of the Machine Representation Specifications and the associated work arounds are discussed below.

Record Representation Clauses

When interfacing with another language, the language might expect the fields of a record in a specific order. Because the method in which the Ada language specifies the order of objects in a record is nondeterministic, Record Representation Clauses were used to specify an order to the record objects. The order that another language requires for a record layout was forced onto the Ada record object so that both languages were referencing the same offset into the structure when a particular field in a record was accessed.

Using Record Representations Clauses resulted in problems which appeared in the code generation of functions which either passed parameters of the record type which had associated representation clauses and/or returned a value of the record type with a representation clause.

In this embedded Ada system many of the formal parameter types of functions had associated Record Representation Clauses. For functions the parameter values were copied into the subprogram. Due to the Record Representation Clauses the value was sometimes incorrectly evaluated. On some occasions the value was evaluated correctly, yet the return value of the function was returned to the wrong location. The reason the return value was corrupted was due to the fact that the method this implementation used to manipulate objects of types which had associated representation clauses in use by functions invalidated the stack.

The solution to the problem was not simple. To continue to conform to the specifications, Record Representation Clauses were not removed from the data types. Instead, all the functions whose parameters consisted of types which had associated representation clauses or returned types with associated representation clauses were changed to procedures with an out parameter for the value which should have been returned from the function.

For example, consider the following:

```
type Output_Control_Byte_Type is
       (UHF_Single_FF_Short,
        UHF_Single_FH_Short,
        UHF_Single_FH_Long,
        UHF_Single_FF_Long,
        VHF_Double_FF_Long,
        VHF_Single_FF_Short,
        VHF_Single_FH_Short);

for Output_Control_Byte_Type use
   (UHF_Single_FF_Short    => 16#53#,
    UHF_Single_FH_Short    => 16#54#,
    UHF_Single_FH_Long     => 16#55#,
    UHF_Single_FF_Long     => 16#56#,
    VHF_Double_FF_Long     => 16#63#,
    VHF_Single_FF_Short    => 16#64#,
    VHF_Single_FH_Short    => 16#65#);

type Message_Type is
record
    Tag      :  Tag_Type;
    Reserved :  Reserved_Type;
    Text     :  Text_Type;
end record;
```

Tag_Type, Text_Type, and Reserved_Type are not shown because these types are not relevant to the example.

```
for Message_Type use
record
    Tag        at 0 range 0..30;
    Reserved   at 0 range 31..31;
    Text       at 4 range 0..639;
end record;

function Output_CB
   (Message :  Message_Type)
   return Output_Control_Byte_Type;
```

The Change required to function Output_CB is :

```
procedure Output_CB
   (Message   : in Message_Type;
    Out_CB    : out
          Output_Control_Byte_Type);
```

Enumeration Representation Clause

Using Enumeration Representation Clauses, a problem appeared when the expression in a case statement was an enumeration type that had an associated Enumeration Representation Clause. Only if the values either began with a nonzero value and/or were nonconsecutive in the representation clause was there a problem. In general, the code generated for the case statement would only work correctly when the values for the representation clause started at zero and continued consecutively throughout the type. The code that was generated, executed as if no representation clause were associated with the type. The index into the case was by position in the type.

To work around this problem, all the case statements which involved variables of enumeration types that had associated representation clauses which either started with a nonzero value and/or had nonconsecutive values were changed to "if...then...elsif ...elsif...elsif...endif" statements.

For example, for the following which has a record type declaration and corresponding Record Representation Clauses for both the parameter and the return value type,

```
type Enumeration_Type_1 is
          (A, B, C, D, E, F, G);

for Enumeration_Type_1 use
          (A   => 6,
           B   => 7,
           C   => 8,
           D   => 10,
           E   => 14,
           F   => 15,
           G   => 19);

procedure Enumeration_Example is
    Enumeration_Value :
         Enumeration_Type_1
begin
    case Enumeration_Value is
       when  A | B   =>
          ...
       when  C       =>
          ...
       when  D       =>
          ...
       when others   =>
          ...
    end case;
end Enumeration_Example;
```

the required change is:

```
procedure Enumeration_Example is
       Enumeration_Value :
            Enumeration_Type_1
begin
    if Enumeration_Value = A or
       Enumeration_Value = B
    then
          ...
    elsif Enumeration_Value = C
    then
          ...
    elsif Enumeration_Value = D
    then
          ...
    elsif Enumeration_Value = E or
          Enumeration_Value = F or
          Enumeration_Value = G
    then
          ...
    end if;
end Enumeration_Example;
```

VARIANT RECORD

The structure of the messages passed between CSCIs was represented as a variant record with the discriminant of the variant record an enumerated type. The discriminant value determined the fields of that particular variant. This implementation attached header and trailer bytes to the storage area for each variant record. Because some fields of the variant records had associated record representation clauses and/or size clauses, these header fields were used to keep track of what bits in the record belonged to which identifier.

Modifying a variant record was required when either CSCI needed to pass a message to the other CSCI or to an external device. In either case each CSCI would have to modify the variant record according to the place it was to be sent.

Variant records posed several problems. These problems with variant records were complicated by the fact that the fields of the variant record were sometimes other variant records. Because Ada requires the discriminant to have a static value when being assigned, a variable could not be used to assign the discriminant a value. Therefore a case statement for each possible discriminant value was needed to format all the output variant records. This caused the amount of code required for handling variant records to be quite large.

Space Allocation

Additional problems with variant records were encountered. These problems were specific to this project's Ada compilation system. Allocation of space for each variant record on the system was always the size of the largest variant. This caused timing delays when assigning values. The delays were due to the fact that the maximum amount of bytes were always moved instead of just the amount used for that particular variant.

Assignment

When assignment to a variant record changed the discriminant, the entire record needed to be assigned. The implementation that was used for this project handled the assignment of variant records by moving values to be assigned to parts of the variant record to the stack and then moving them to the new location in the record. The amount of moves with the largest size variant increased the amount of time in handling a particular variant by the amount of time it takes to move the extra bytes. To reduce the amount of extra time, assignments to the entire variant record were kept to a minimum.

Code Generation

Code generated by the compiler to manipulate variant records was not always accurate. The header and trailer bytes sometimes were incorrect in referencing a particular field in the record whose type had a corresponding representation clause. Incorrect values were being retrieved from the variant. The user thought he had modified one field in the record and actually in memory a different one had been changed, if any fields were changed at all.

Solution

The solution to the problems encountered with variant records was to encapsulate each field that either contained a variant record and/or had representation clauses inside of another fixed record. This caused the programmer to have to use another level of referencing when accessing a field in the variant record. This enabled the project to continue to use variant records, because the other alternative was to remove them and use only fixed records. An example of this is shown below.

Example

In general, here is an example of the proposed solution. Consider the following:

```
type Part_1_Type is (A, B, C, D, E);

for Part_1_Type use (A   => 6,
                     B   => 7,
                     C   => 8,
                     D   => 10,
                     E   => 14);

type Part_2_Type is
record
   First_Field  :   First_Field_Type;
   Second_Field :   Second_Field_Type;
   Third_field  :   Third_Field_Type;
end record;
```

First_Field_Type, Second_Field_Type and Third_Field_Type are not shown because these types are not relevant to this example.

```
for Part_2_Type use
record
   First_Field     at 0 range 0..15;
   Second_Field    at 2 range 0..2;
   Third_Field     at 2 range 3..7;
end record;

type Discriminant_Type is
   (Message_Type_1,
    Message_Type_2,
    Message_Type_3);
```

```
type Variant_Record_Type
    (A_Discriminant :
       Discriminant_Type) is
record
   case A_Discriminant is
      when Message_Type_1 |
           Message_Type_2     =>
              Part_1          :
                 Part_1_Record_Type;
      when Message_Type_3     =>
              Part_2          :
                 Part_2_Record_Type;
   end case;
end record; )
```

Problems were encountered when trying to reference fields within objects of Variant_Record_Type. The proposed solution is as follows:

```
type Part_1_Type is (A, B, C, D, E);

for Part_1_Type use (A   => 6,
                     B   => 7,
                     C   => 8,
                     D   => 10,
                     E   => 14);

type Part_1_Record_Type is
record
   Part_1_Rec : Part_1_Type;
end record;

type Part_2_Type is
record
   First_Field  :   First_Field_Type;
   Second_Field :   Second_Field_Type;
   Third_Field  :   Third_Field_Type;
end record;

for Part_2_Type use
record
   First_Field     at 0 range 0..15;
   Second_Field    at 2 range 0..2;
   Third_Field     at 2 range 3..7;
end record;

type Part_2_Record_Type is
record
   Part_2_Rec : Part_2_Type;
end record;
```

Note that the only difference is that now Part_1_Type and Part_2_Type are now encapsulated in the fixed record types Part_1_Record_Type and Part_2_Record_Type respectively.

EXECUTION SPEED

The time for a message to pass from the control processor to CSCI1, to CSCI2, and finally from CSCI2 to either the modem or transceiver had to be within a requirement specified limit. Due to the speed of the processor the application was currently executing on and the amount of code that had to be executed in order to move the message along its path, the messages were not reaching their destination in the allotted amount of time.

Because the speed of the processor could not be changed, the amount of time to execute the Ada code needed to be reduced. To help reduce the amount of time required for code execution, some of the function calls were removed. Local objects, which resided in package bodies to conform to the software engineering principles of information hiding and abstraction, were moved to package specifications. These objects were previously accessed thorough a function interface. Once they were moved to the package specifications the function interface was no longer needed. The objects could be referenced directly. The amount of time saved was approximately 40 microseconds per function call removed.

PRAGMA INLINE

Pragma Inline was used to eliminate the time due to subprogram calls overhead. Those subprograms which were inlined had the code of the routine placed at every invocation rather than a procedure or function call. This saved about 40 microseconds for each call.

There were two problems in using pragma Inline. The first problem was that the code became too large with its use. Inlining everything caused the amount of RAM required to be larger than the available 128K. To offset this some of the previously inlined subprograms had to have the pragma Inlines removed. Because everything had been inlined everything to meet timing requirements, now some of the pragma Inlines had to be removed

with care, so as not to hinder the execution time of other areas of the system beyond an acceptable level.

Another problem in using pragma Inlines was that the subprograms could not be stubbed out and still be correctly inlined. All the subprograms that were to be inlined had to have the code for their bodies included in the package in which they were declared. This was an inconvenience because the packages were developed using body stubs to control the size of the package, to permit incremental development, and to reduce compilation/re-compilation time for change.

ADA TASKING, SYNCHRONIZATION, AND COMMUNICATION

Tasking

The tasks in each CSCI were all task types. The actual task was an access to a task type. In this way all the tasks could be started by creating a new object of the task type, in an appropriate order. This left the Scheduler task to be started last, thus insuring that the Scheduler would not try to schedule another task to do work before the Scheduler was started.

In each CSCI, the tasks were controlled by the lowest priority task, the Scheduler task. The rendezvous between the Scheduler and the other tasks in each CSCI consisted of a simple rendezvous. The time for the simple rendezvous was approximately 70 microseconds, this included the time for context switching.

Synchronization

The scheduler checked the counts of the task queues of each task in the system and in a round-robin fashion rendezvoused with each one that had a non zero queue count. The task which had gained control through this rendezvous would then process the buffers on this task's queue. Once the task had finished processing a buffer it continued dequeuing buffers from its queue until the count was zero. Once the queue count was zero this task "went to sleep" until another task put a buffer onto its queue and the scheduler woke the task up again.

Communication

There was a preallocated buffer pool consisting of 100 buffers for each CSCI. Each of the tasks within a CSCI contained a queue in which the task looked for work. When a part of the CSCI had work for a task, the task fetched a buffer from the buffer pool, a message was put into the buffer and that buffer was queued to a task's queue. The task that had work dequeued the buffer from its task queue and took the message out of the buffer. This task next processed the message. Once the task was finished processing the message, the message was either put into a buffer again and passed to another task or the buffer was given back to the buffer pool.

INTERRUPTS

There were several types of interrupts in this model. The type of interrupts in this system were interface interrupts (message complete, input/output between characters), frame sync interrupts and timer interrupts.

Interrupt entries were bound to a task entry by a physical address. The format of setting up the interrupt entries used on this project were the same as those specified in the Ada Language Reference Manual. The physical address specified in the interrupt entry became an entry in an interrupt vector table. Further explanation of the mechanics of how the interrupts were used will be provided in a future paper.

INTERRUPT SERVICE ROUTINES

The interrupt vector table (IVT) was a table of addresses that was built according to the entry addresses specified in the interrupt service tasks. The code branched to this address when an interrupt of a particular type was received in the system. This interrupt service task then invoked the interrupt service routines (ISR). The ISRs were the routines that were invoked by the task with an entry for a particular interrupt when this interrupt occurred. See Figure 2.

Overrun

All the ISRs were initially written in Ada. The ISRs for input were not quick enough to accommodate the baud rate of the interfaces. Characters were being lost and therefore the messages were not being processed correctly. This being the case the software would "get behind" and lose characters or in other words get "overrun". There was no problem of time in outputting because no allocation of buffers from the buffer pool needed to be made nor was any queueing to task queues done.

Solution

Many attempts were made trying to reduce the amount of time to execute an ISR. Some of these attempt included using pragma Inlines, machine code insertions, algorithm redesign, and finally converting the ISRs to assembly language and using pragma Interface. The mechanics and reasons for these steps will be described in a future paper.

Using assembly language ultimately resulted in the best execution time for an ISR. The ISR had been optimized at this low level after the algorithm had been optimized. This finally brought the time of execution of the ISR to an acceptable level, approximately 200 microseconds.

CONCLUSIONS

This paper has made an attempt to show some of the problems one needs to be aware of when using Ada on an embedded real-time software project. These types of problems may not always be as much of an inconvenience as they were on this project but one needs to be aware of the types of problems for which to be on the alert.

Using Ada on an embedded real-time project using good software engineering practices requires much hardware and software coordination. Once the coordination between hardware and software is successful then Ada will make the overall system easier to maintain. The system should be designed for reliability, utilizing the software engineering principles of abstraction and information hiding. In so doing, the interface between objects will be via procedures and functions. Optimizing for timing and memory constraints should be considered at a later stage, the testing stage, in the life cycle. This paper proves that this approach is viable.

Debra A. Schmidt
Victoria Plaza
Building 4, Suite 9
615 Hope Road
Eatontown, NJ 07724

Debra A. Schmidt is a Senior Software Engineer at P.P. TEXEL & Co., Inc., Eatontown, NJ. She has been with the company for about a year and a half. Most of this time has been spent working with Ada in an embedded real-time environment.

Prior to joining P.P. TEXEL & Co., Inc. she was a Software Engineer at Concurrent Computer Corporation, Customer Service Division, for over three years. Work at Concurrent Computer Corporation included solving customer problems concerning the operating system, as well as the languages of Ada, FORTRAN, Pascal and Assembly Language.

She earned her bachelor's degree in Computer Science with a minor in Mathematics and Chemistry from Mercer University, Macon, Georgia. She is currently pursuing a master's degree in computer science from Rutger's, The State University of New Jersey. Debra is a member of the Association for Computing Machinery national and local Princeton chapters.



SYSTEM OVERVIEW

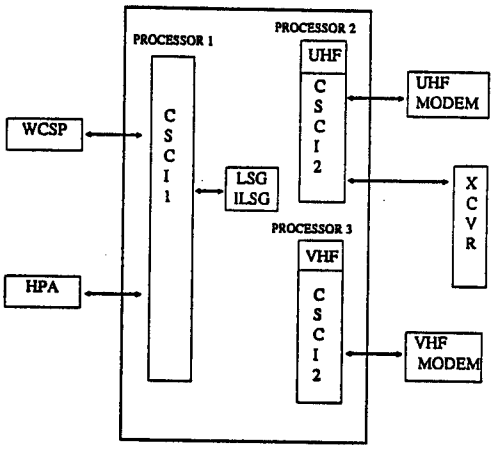


Figure 1

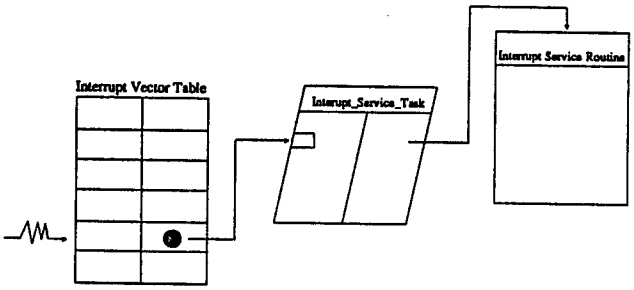


Figure 2

ON THE DESIGN OF A PROTECTED INTERFACE TO A C-2 CLASS SYSTEM

ROB MADSEN AND JOHN P. FITZGIBBON*

MARTIN MARIETTA
INFORMATION AND COMMUNICATION SYSTEMS

INTRODUCTION

Command and Control Systems are becoming more interconnected to provide enhanced utility, greater access of information and better correlation of intelligence. While this integration provides added value in terms of usefulness and usability, it increases the difficulty of insuring authorized access of information and protection. When systems that process information of different levels of classification and sensitivity are interconnected it is necessary to install more sophisticated security mechanisms.

Systems that process, store and protect information of different levels of classification or sensitivity are known as Multilevel Secure systems. The simplest of such systems are categorized as B-1 systems. Systems that provide comparable protection for information of a single level of classification and sensitivity are categorized as C-2 systems. The added sophistication required to meet B-1 requirements can cost 2 to 10 times as much as C-2 requirements and extend the development period unpredictably. Even worse, development may have to be abandoned when it is discovered that the cost of protecting data is greater than the value derived from the information.

This paper describes a cost effective alternative to B-1 implementation that can provide necessary protection. This approach is based on partitioning of system data flow and isolation of software components according to trustworthiness. Using a specific example, this paper describes the operational concept, the requirements analysis process and salient design features along with pertinent selection rationale. This paper goes on to describe measures that must be taken to justify trustworthiness of code, the effect of implementation in Ada and verification methods that will lead to final approval of the system.

OPERATIONAL CONCEPT

System Architecture

Figure 1 depicts the system architecture in relation to external systems. The target system receives and processes data from three mission areas (i.e., Air Defense, Missile Warning and Space). Users of the system include Canadian Forces as well as United States military forces. Security is an issue because the Space interface delivers US-Only data as well as information that is releasable to Canadian personnel (RELCAN). The US-Only data can not be disclosed to Canadians in accordance with the regulations of the Foreign Disclosure Office (FDO). There is no requirement to display any of the US-Only data or any information that is derived from it. The security architecture is further complicated by the policy of allowing uncleared visitors to regularly tour the facility.

* The opinions expressed in this paper are those of the authors and do not necessarily reflect those of the Martin Marietta Company.

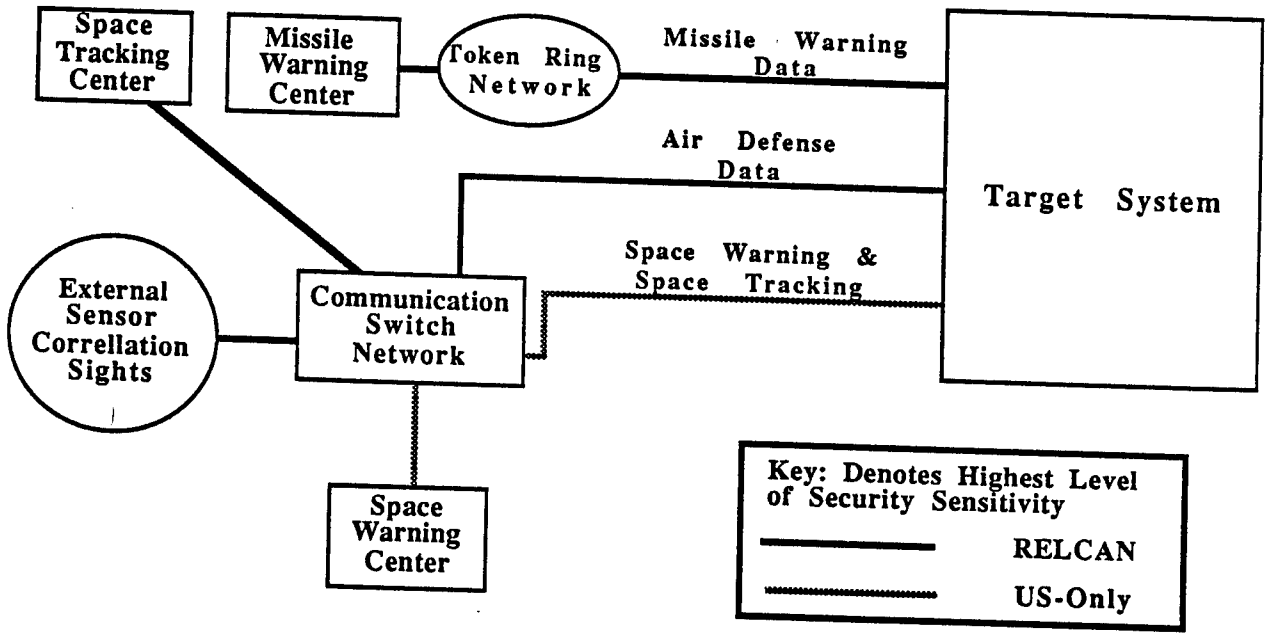


Figure 1: System Architecture

Users

Users having access to the Target system shall possess a minimum of a SECRET security clearance with authorizations to information releasable to Canada (i.e., SECRET/RELCAN data). Access to the system is granted to only those users who possess requisite security clearances. Any users who are granted access to sensitive data must satisfy two criteria: (1) The individual must have been granted a security clearance (SECRET) commensurate with the level of classification of the data; (2) The individual must have a verifiable need-to-know the information in the conduct of that individual's duty. This need-to-know shall serve as a basis for the discretionary access controls enforced on the Target system. The Target system shall provide specific roles to individuals who access the system. These roles shall define the type of accesses (e.g., read/write/execute access to specific files) required for each individual. The roles provided by the system shall include but not be limited to:

US Personnel

1. Computer System Security Officer (CSSO)
2. Computer Operations Personnel
3. Computer Maintenance Personnel

US or Canadian Personnel

1. Command Post User(s),
2. Air Mission User(s),
3. Space Mission User(s),
4. Missile Warning User(s) and
5. System Analyst(s)

The Target system processes three different sensitivity levels of information:

1. UNCLASSIFIED,
2. SECRET/US-Only and
3. SECRET/Releasable to Canada (i.e., SECRET/RELCAN).

The Target system mandatory access control mechanisms shall ensure that SECRET cleared United States users are granted access to UNCLASSIFIED, SECRET/RELCAN, and SECRET/NOFORN data. The Target system ensures that SECRET cleared Canadian users are granted access only to SECRET/RELCAN and UNCLASSIFIED data. Once mandatory access control checks have been satisfied,

subsequent access to data is granted on a strict need-to-know. The operational concept of use implies that the system must manage classified data at two levels of sensitivity as well as UNCLASSIFIED data. In accordance with guidance provided by CSC-STD-003-85 [Yellow], this leads to a B-2 class system design.

Environment

The Designated Approval Authority (DAA) considers the operational environment to be benign. This is based upon several characteristics of the system. The first is that physical security enclosing the system is very effective. The facility is protected by armed guards and badge readers authenticate access at facility portals and individual work centers.

Different functional work centers are separated by isolated Sensitive Compartmented Information Facilities (SCIFS). Individuals working in these centers have been cleared to appropriate levels and their need-to-know is verified. Foreign users are not allowed physical access to computer operations, security functions or maintenance functions.

Intercommunication among external systems and internal components is provided by fibre optic communication links. These links are protected against penetration from external sources. No remote access is allowed into the system from external sites except through dedicated fibre optic links between SCIFS.

RISK MITIGATION STRATEGY

Cost Control

One of the major concerns of the system relates to cost and schedule control of development. The system has been incrementally developed and security requirements have become increasingly more complex. As a result, security features necessary to support a Multilevel Security (MLS) system have not been incorporated into the system design infra-structure. Development of one system of comparable size (refer to [Fitz-1989]) required a one time cost of $10,000,000 to analyze the system architecture and develop a security policy and security architecture. This cost did not take into account the effort necessary to implement the security features or the continuing cost to support future system development. Moreover, there were additional problems related to staffing the program with qualified personnel and schedule delays associated with security certification.

Exploitation of Strengths

The design of the system exploits the fact that the environment is benign and resistant to penetration. These factors include the strong physical security, facility access control and the fact that the system is only operated by cleared personnel. Another exploitable feature is the fact that information flow within the system is separated to support isolated work centers. Points of access are secured in SCIFS.

Co-operation with the DAA

Another strategy to mitigate risk was to involve the technical staff of the Designated Approval Authority (DAA) from the very beginning of the program starting with system design. In this manner we were able to define and derive functional requirements of the system interface and map those requirements to design features. By getting early concept approval, the program greatly reduces the risk that the DAA might object to the implementation when the system is ready for acceptance.

Approval often hinges on the documentation of the system design as well as the quality of

TABLE 1: SUMMARY OF DESIGN TRADE-OFFS

CRITERION	OPTIONS Null	Grant	Custom	B2	Ext.	Int.
Essential						
FDO approval	no	no	yes	yes	yes	yes
Schedule	yes	yes	no	no	yes	yes
Cost	yes	yes	no	no	yes	yes
Staff	yes	yes	yes	no	yes	yes
Technical						
Cost					14	18
Security					12	12
Schedule					6	8
Staff					0	5
Reliability					4	8
Miscellaneous					-6	4
Total					30	55
Results	**Fail**	**Fail**	**Fail**	**Fail**	**Good**	**Best**

implementation. Working with the DAA'S technical staff provides better understanding of the documentation requirements to provide the necessary assurance that will allow approval. Addressing these aspects early in the development process mitigates risk because it is easier to assemble the proper documentation during the design process than afterwards, when key personnel may have left the program.

Verification

A final strategy involved planning for verification of the security interface. We planned a two-tiered approach that subject the security elements to verification by analysis as well as testing. Formal software inspections were performed at critical points during design and implementation. Customary procedures were augmented with special analysis methods that were applicable to security related software. Since the security related software was designated as critical Computer Software Components (CSC'S) the software development plan provided for early implementation of these components. This was done to provide addition time for security testing of the component in an isolated configuration prior to integration with the rest of the system.

SYSTEM DESIGN

Evaluation of Design Alternatives

This section identifies the design options and summarizes the results of the analysis. The options are identified as follows: (1) Null Option (i.e.,take no action); (2) Grant Canadians access to all data delivered through the Space interface; (3) Implementation of a customized message set to eliminate US-only data. (4) Full B2 level implementation; (5) Implement an external line filter; (6) Implement an internal filter using existing capabilities of target system.

Table 1 summarizes the advantages and disadvantages of each of the considered design alternatives. The most important criteria was cost of development. The results of the analysis show that only the external and the internal filter options met the essential criteria. Of those two alternatives, the internal filter option was significantly better due to the

fact that it is based substantially on existing capabilities of target system. It is significant to note that the study would identify this as the best solution regardless of the weighting factors assigned to each criterion. The internal filter option was rated equal or superior in all categories. Moreover, there was a substantial difference in cost ($1,000,000) in the two options and this was the most important criterion for the design.

Hardware Architecture Description

Figure 2 depicts the physical architecture of the target system. The system Automatic Data Processing (ADP) architecture comprises three different Hardware Configuration Items (HWCI). The Gateway HWCI provides connectivity to mission data sources and hosts the security filter conceived during design analysis. The Mission Processors host the application software that correlates, stores and manages data for the system users. The Workstation HWCI is the primary interface to the user and is responsible for providing representations of the real world situation at any current instant. The situation is inferred from the data base managed by the Mission Processors.

The ADPE HWCI'S are interconnected via an Ethernet communication network. The network provides interconnectivity among the processors and allows all users to have access to any of the information stored by the mission processors. This architecture distributes processing of information to allow concurrent computation while allowing correlation and fusion of the separate mission are data bases. It also supports reliability and availability of the system in concert with backup processing units. The shaded items in Figure 2 represent redundant processors. These processors shadow the active units so that the system may recover immediately from hardware errors.

Software Architecture Description

Figure 3 depicts the software architecture of the target system. The software architecture comprises the Computer Software Configuration Items (CSCI'S) of Gateway, System, Space, Air, Missile, Workstation and Support. The Gateway provides connectivity to external system and message routing. Space, Air, and Missile process, manage and store mission data. The System CSCI maintains system status information and provides a platform independent infrastructure in support of mission processing. The Workstation CSCI provides mission displays supporting situation assessment and decision making.

The target system is being developed incrementally and future system capabilities are not clearly specified until the development

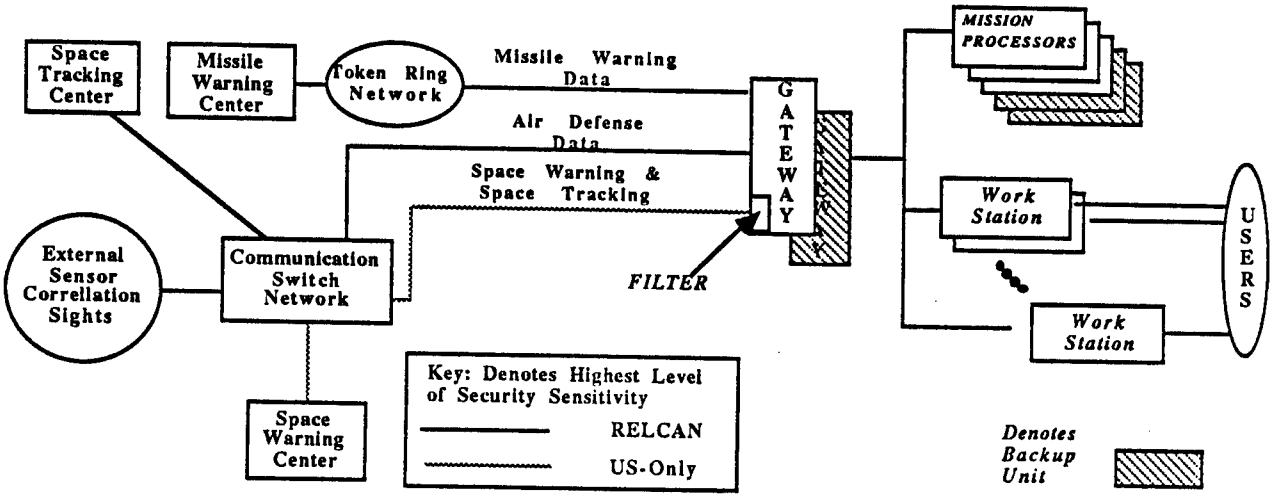


Figure 2: Hardware Architecture

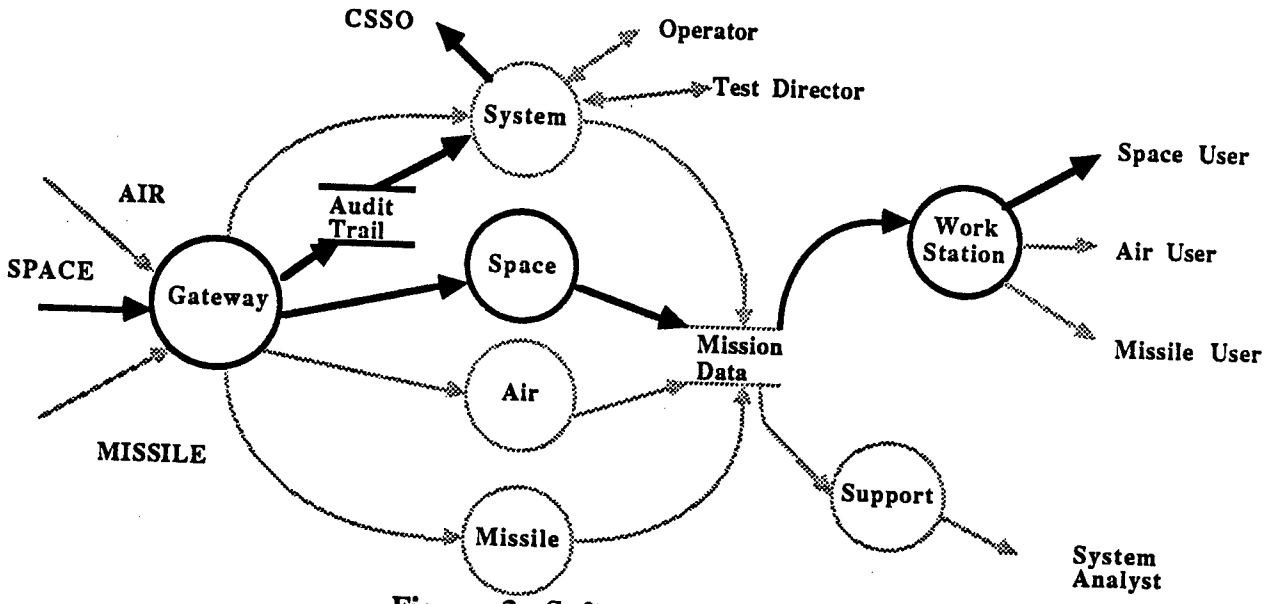


Figure 3: Software Architecture

phase in which they are implemented. The highlighted portions of Figure 3 represent new capabilities for the current phase while the other features represent existing capabilities. This development phase provides enhancements related to the Space mission and security enhancements required for interfacing to the Space system center.

The interface of the target system to the Space center is supported by the Gateway CSCI. This CSCI also extracts security sensitive data from the mission data stream by means of a software filter. Security events related to filter operation are stored in a security audit trail so that they may be reviewed by the Computer System Security Officer (CSSO). Sanitized Space traffic is processed by the Space CSCI and stored in the correlated mission data base. The Workstation has been enhanced to provide addition mission displays related to the new mission data.

REQUIREMENTS ANALYSIS

Specified Requirements

The System Specification for the target system provided very guidance in the area of functional security requirements. The explicit requirements were as follows:

1. The categories of information stored in the system shall be SECRET/Releasable to Canadians with a strict need to know.

2. Two displays shall be provided that are UNCLASSIFIED.

The system level requirements provided little guidance in determining the allocated software requirements. The fact that the system accepts data of different levels of sensitivity and must produce UNCLASSIFIED displays as well as SECRET displays implies that provision for Multilevel Security must exist. To develop a better refinement, it was necessary to derive functional requirements from the system design concepts and the operational concepts. Using the filter concept and the fact that Canadian personnel are precluded access to the system operation center, the following allocated requirements were derived:

1. The Gateway CSCI extracts and disposes of data not releasable to Canadians.

2. The Gateway CSCI shall not store on non-volatile media, any data not releasable to Canadians,

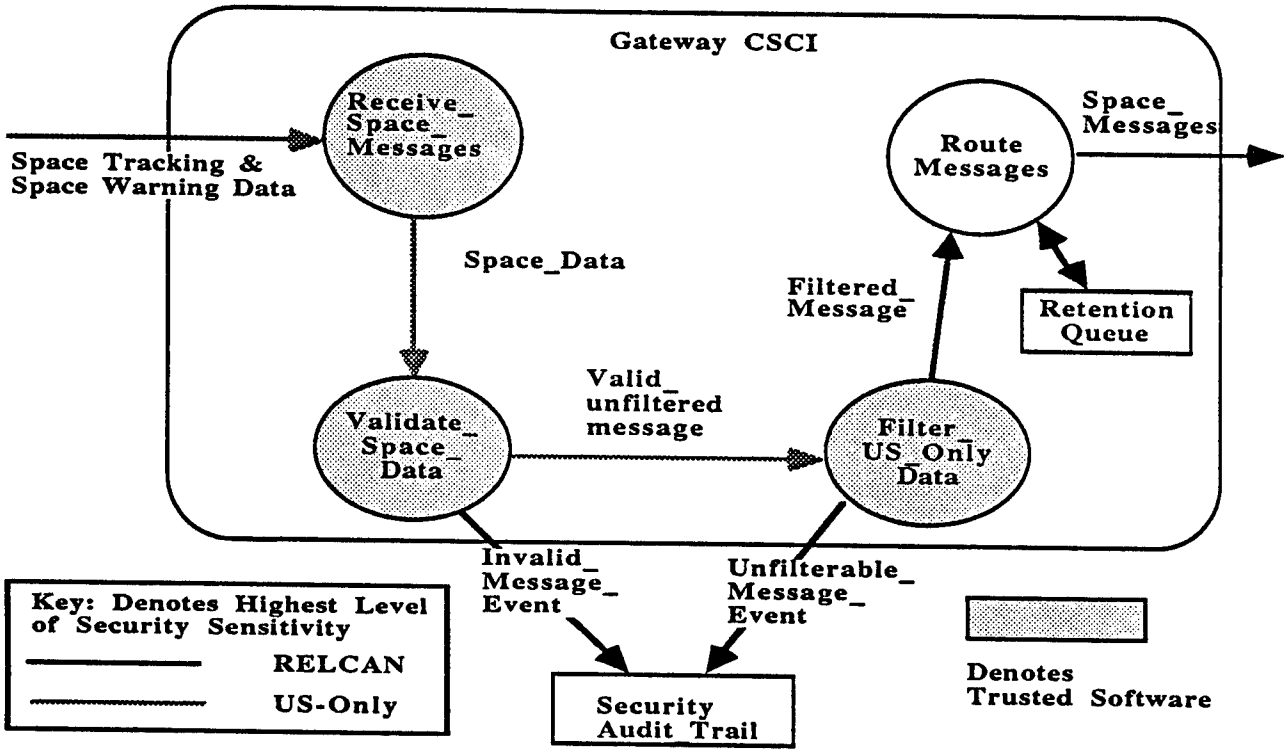


Figure 4: Gateway Functions

3. The Gateway CSCI shall store unfiltered data in memory accessible only by the Gateway CSCI.

4. The Gateway CSCI shall not display on any display device, any data not releasable to Canadians,

5. The Gateway CSCI shall not print on any print device, any data not releasable to Canadians,

6. The Gateway CSCI shall not output from the Gateway CSCI, any data not releasable to Canadians,

7. The Gateway CSCI shall forward only data classified at the sensitivity level of SECRET/RELCAN (Releasable to Canadians) to the Space CSCI,

8. The Gateway CSCI shall forward only data classified at the sensitivity level of SECRET/RELCAN to the System CSCI,

9. The Gateway CSCI shall validate the integrity of each received message prior to removing any data not releasable to Canadians,

10. The Gateway CSCI shall execute in protected memory such that only the System CSCI and Gateway CSCI can influence its control.

11. The Gateway CSCI shall record all errors detected during validation or filtering of message blocks.

12. After extraction of US-Only data, the Gateway CSCI shall package mission critical information in output messages that are forwarded to the Space CSCI or the System CSCI.

Functional Analysis

Figure 4 depicts the logical functions composing the Gateway CSCI. This CSCI comprises the functions of "Receive Space

Messages", "Validate Space Data", "Filter US-Only Data", and "Route Messages". Three of these functions, denoted by shading, are part of the Trusted Computing Base (TCB). The design and implementation TCB software must be subjected to special analysis and verification to ensure that sensitive data is not inadvertently disclosed to unauthorized users.

The "Receive Space Messages" function accepts messages from the Space center and implements the low level protocol of the interface including guaranteed delivery. These messages are transmitted in groups of varying size and message content. Each message group is in turn segmented into fixed length blocks. This function passes the received blocks on for further processing and validation.

The "Validate Space Data" function identifies the message type and verifies the syntax and format of each message fragment. In addition to syntax verification, this function is responsible for reconstituting the message structure from the block fragments. Space messages are subjected to filtering. Any message that fails to pass syntax verification is erased according to security procedure requirements. This generates a security event which is subsequently recorded in the Security Audit Trail.

The Filter function extracts any US-Only data from Space messages and reformats the message according to internal format. In addition to sensitive data, this function is responsible for eliminating any superfluous data that is not required by the Space Mission. Any memory locations containing sensitive data are erased using an approved security procedure. If an error or anomalous condition occurs during the filtering process, a security event is recognized and recorded in the Security Audit Trail.

The "Route Messages" function is responsible providing safe data storage and guaranteed message delivery. Space event messages are distributed to the Space CSCI for correlation, processing and storage. The purpose of the Retention Queue is to store messages in the event that a hardware or software failure causes the Gateway or Mission processor to halt. The backup processor may then recover by inspecting the Retention Queue and resuming message processing where it left off.

SOFTWARE DESIGN

Figure 5 depicts the software design of the Space Service Capability. The Space Service comprises the "Filter Process" and the "Delivery Process". The "Filter Process"

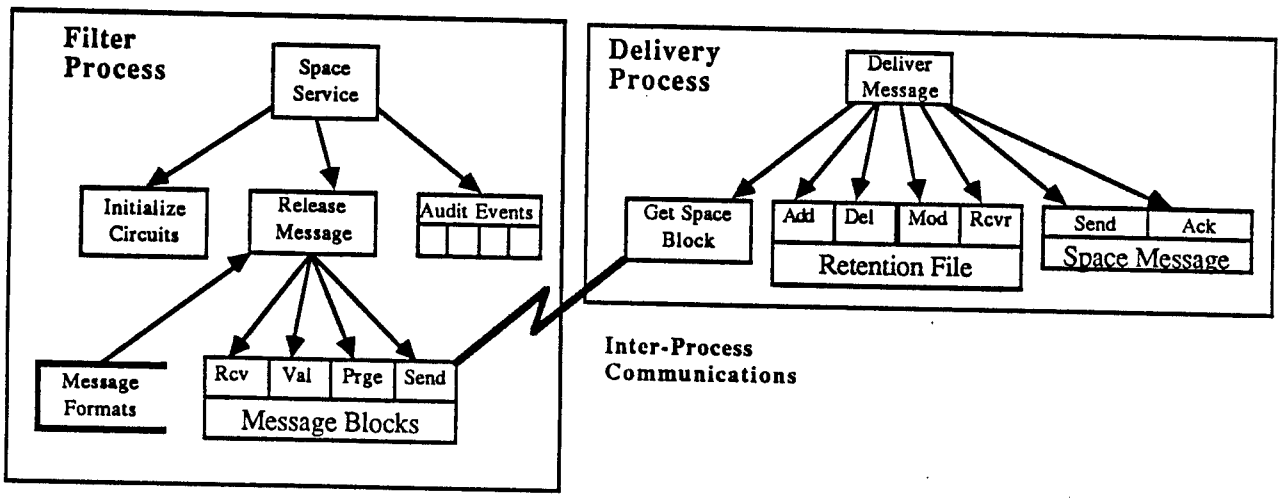


Figure 5: Software Design

includes the security related software components of the Gateway CSCI and is part of the Trusted Computing Base (TCB). The host operating system, through process management, effectively isolates the execution of the TCB code from non-trusted software. This process has been defined as non-pageable and non-swappable so that sensitive data cannot be inadvertently recorded on disk, thereby compromising security policy. Moreover, operating system crash dumps have been disabled on Gateway processors to prevent disclosure resulting from hardware or software failure.

Filter Process

The first module in this process controls the initialization of Space circuits. The low-level circuit protocol is provided by a micro-programmable communication board. This function loads the micro-code, verifies version and configuration, and establishes a communications session with the Space Warning center.

The message release module accepts message blocks, identifies and validates the blocks, filters out sensitive data, and purges the block after sending it to the Delivery process. One of the significant design features is the use of an informational cluster, as defined by Parnas [Parn-1971], to control access to the message processing functions. The implementation of an informational cluster maps directly to the concept of packaging in Ada. This has been done to promote security by localization of access to the message block. The message block information cluster is part of the program's static memory structure and therefore derives access control from the process protection attributes (e.g., non-pageable).

A second feature is the use of a message format file. This file contains a syntactic representation of each allowable message. This allows message formats to be changed, added or deleted without changing the executable code structure. In addition to affording extensibility and reusability, this decision makes it easier to re-certify the security related software with subsequent maintenance releases.

Several designs were considered for the Audit Control module. The first was to use existing capabilities that record mission events. A second option was to create special purpose code to record security events. These approaches suffered from two basic flaws: (1) Explicit recording to disk is required from the TCB process; (2) Auditable events appear in two separate security audit files impairing both monitoring and correlation of events. The selected approach was to create several named data objects, one for each auditable event, and to set access privileges to each object so as to preclude reference by the Gateway software. Whenever a security event occurs, the Filter process attempts to reference the appropriate data object causing an access violation to be recorded in the operating system Audit Trail. This approach has the favorable side-effect of obviating the need to create addition software (i.e., TCB software), allowing the CSSO to review the Filter journal.

Delivery Process

This process receives message blocks and stores them in a Retention File until the completely assembled message is delivered to the Space CSCI. The Retention File contains partially assembled message and completed messages that have been transmitted to the Space CSCI. The Retention File contains partially assembled messages and completed messages ready to transmit to Space. These messages are retained until the Space CSCI acknowledges receipt.

The Retention File has been implemented as an informational cluster to hide implementation details and to simplify the application level interface. The informational cluster provides the expected primitives of adding and deleting a message. The modify primitive is necessary to support message assembly. The Recover primitive supports resumption of processing following failure of the Gateway or the Space CSCI. This primitive is activated whenever the process is restarted. It returns process state information including the last message acknowledged by

Space, outstanding message transmissions and the point where message assembly was interrupted.

USING ADA FOR SECURITY FEATURES

Program and Data Abstraction

The development of security related software to augment the TCB of a vendor supplied operating system (e.g., trusted processes) requires that the design documentation and implementation to provide both understandable and maintainable protection. Depending on the level of security of the system, the security related software may require verification. Normally, this is only a requirement of B3 and A1 systems. The current technology is lagging in provision of code verification tools for these systems. [Gass-88] For security-related software in class B1 systems it is sometimes sufficient to subject the code to thorough analysis to establish a level of trust in the logical structure of the software and the interfaces to external software components. The following principles are critical to developing security related software that can be analyzed:

1. The TCB must be both simple in design, yet provide full implementation of the allocated requirements.

2. As the amount of code in the TCB increases, the ability to express confidence in the full implementation of security requirements decreases.

3. The interfaces to the TCB must be completely identified along with their functionality, in order to provide a complete evaluation of the TCB software.

The design of the Ada language was strongly influenced by the need to make software readable. Readability of code is one of the paramount attributes of security related code since confidence in this code is derived from the fact that the code can be shown, through analysis (for B and C class systems), to correctly implement the security policy. Ada provides features such as strong type checking, exception handling, and packaging that promote deterministic behavior of the software. These language features assist in the implementation of requisite attributes for security related software. The relevance of these features is discussed below.

Ada is a strongly typed language. The compiler ensures that program variables are used correctly and that predictable results are guaranteed. During execution, any operation that attempts to assign a variable a value that is inconsistent with its declared usage generates a program exception. The program's response to an exception is in turn, controlled by the implementor of the algorithm. Consequently, the program will always execute from a known state. For example, if a program exception occurs during the processing of security object, control is transferred to an exception handler. This exception handler can insure that sensitive information is purged from the system prior to the resumption of execution so that the system is not vulnerable to compromise.

The packaging feature of the Ada language enables a program to prevent unauthorized access to data by any other software subject or agent. Any inadvertent attempt to access a protected object is discovered during compilation and the program is never transformed into executable code. When this is coupled with operating system features, such as file access mechanisms, this achieves a well constructed security shell around sensitive data. Furthermore, attempts to penetrate this shell are detectable by the operating system features and can be recorded in the system audit trail.

The Ada tasking feature provides a standardized, well ordered methodology to implement well behaved program execution of concurrent operation. This provides predictability of operation and promotes analysis of applications that require concurrent processing. However, the ease of implementation of concurrent algorithms that Ada affords can lead to abuse by programmers. This can inject concurrency, and hence complexity, into an algorithm that

does not require it. The solution to this problem is the enforcement of programming and design standards to ensure that unnecessary complexity is not injected into the system. A second problem with the Ada tasking mechanism is that the execution of a task rendezvous is not deterministic.

Reliability

For the development of security related software there exists a control criteria from DoD 5200.28-STD [Orange] for assurance. Simply stated, security related software incorporated into a Trusted Computing Base (TCB) must be designed, implemented, and operationally guaranteed to accurately interpret the security policy (i.e., security requirements) for the system. Assurances are provided through analysis of requirements, design, and implementation and through testing. A critical and often overlooked development point is the transition from implementation to testing. Up to the point of implementation (i.e., writing code) all phases of system development are in the form of documentation. Documentation is relatively straight forward to interpret and analyze.

Once code is compiled into executable code, the documentation analysis is replaced with testing. At this critical stage, a high level of trust must be placed in the correct translation of the analyzed code to compiler generated executable code. Thorough analysis of requirements, design and code can be circumvented by compiler generated errors that go undetected during the testing process. Exhaustive hands-on testing of operational software to test compliance with a security policy cannot guarantee correctly generated code. Consequently, this gap in assurances must be filled by assurances provided by a thoroughly proven compiler.

The process of building trust in the implementation language begins with a carefully specified definition of the language. During the interval between 1975 through 1980, the requirements and design of the Ada language was subjected to an unprecedented analysis and verification process. Literally thousands of the best and brightest computer scientists and engineers participated in design and review [Barn-89]. Ada compilers are subject to rigorous testing and verified against the ACVC test suite to ensure correct translation into executable code. Hundreds of millions of lines of code have been developed, tested and deployed in Europe and the United States. This constitutes an amalgam of software that is arguably the largest body of code developed in a single language dialect. To this extent, Ada represents the most thoroughly validated computer language in existence. Based on this reasoning, it is easier to build a trust case for security software implemented in Ada than other common languages such as FORTRAN, Pascal or BASIC.

VERIFICATION

Software Inspections

Software design and code inspections are regularly applied to trusted code and untrusted code to eliminate defects prior to test and deployment. For trusted code additional attention is necessary to meet certification requirements. Trusted code must satisfy special criteria as well as all criteria imposed on untrusted code. The following sections provide a high level summary of those requirements.

Untrusted code is inspected to detect defects that could cause the software to malfunction, to enforce compliance of design and coding standards, to insure that resulting implementation conforms to system design and to provide management visibility into the quality of software implementation and design products. This is accomplished by having qualified peers review code and design documentation in accordance with a comprehensive set of criteria that address twelve (12) different aspects of the software.

Additional procedures were created to deal with Security related software (TCB) in accordance with criteria for B1 class systems defined in DoD 5200.28-STD. This standard recommends that at least two of the inspectors have a bachelor's degree in Computer Science and one inspector have a master's degree. In

addition, at least one team member must have previous experience with security testing and evaluation. Any piece of software that has identified flaws must be re-inspected after the flaws have been removed or neutralized. Special attention is given to aspects of the software that might allow the imposition of access or control by external software agents. The checklist incorporated by project Engineering Practices follows:

1. Are there any execution paths within the TCB software that bypass the intended security mechanism?

2. Under what conditions is a "jump" made from the beginning of the software module to the end, which bypasses the intended security check? If existing, does such a path conform to security requirements?

3. Is there design evident in the security software that could place this mechanism in a state (e.g., suspended, deleted, unknown) such that this security software is unable to respond to received events?

4. Are there conditions where unauthorized access to the audit or authentication data would be permitted?

5. Are there any conditions where sensitive data is read, changed, or deleted by any user or process other than TCB software?

Testing

A Security Policy defines the set of laws, rules, and practices that regulate how an organization manages, protects, and distributes sensitive information. In this instance, the Security Policy is stated as specific security requirements in the System Specification, System Design Document, Gateway SRS, and System SRS. The TCB is the totality of security protection mechanisms within a computer system - - including hardware, firmware, and software -- the combination of which is responsible for enforcing a security policy. Specifically, the TCB is defined as:

1. Operating System software that enforces the requirements for :

   a) Discretionary Access Control (e.g., Access Control Lists),

   b) Identification and Authentication (e.g., Login with username and password),

   c) Object Reuse (e.g., the "zeroing" of memory/storage locations after a file or buffer have been deallocated),

   d) Auditing (e.g., the auditing of specific events that are security relevant),

2. Message Filtering (e.g., a Granite Sentry function that removes fields from messages for which the user does not have authority to access).

3. The hardware that is part of the TCB is the Gateway HWCI.

For Security testing purposes, the area requiring test are the TCB elements allocated for the Gateway CSCI. More specifically, the elements of the TCB as described above need to be tested against the requirements allocated to each area. For Operating System Software it is not necessary to test for the correct operation according to operating system design documentation. We must assume that the operating system operates correctly. What we must test for the operating system is that for each of the TCB elements within the Operating System that they are "configured correctly" (e.g., access permissions are appropriately restricted, auditing turned on for security events) as stated in the System SRS. For the Message Filtering function of the Gateway CSCI must be tested/analyzed to ensure it executes only as required by design documentation and coded in source code.

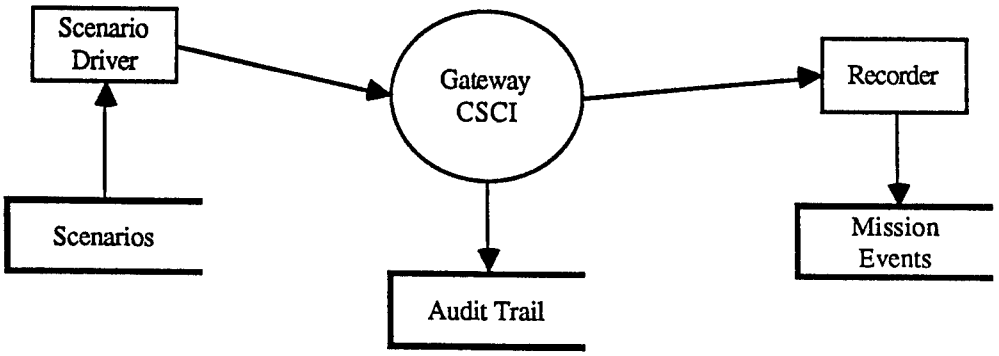


Figure 6 CSCI Testing

The message filter software must be tested both in isolation and integrated with the rest of the CSCI'S allocated to the Gateway HWCI. To accomplish isolated testing, we created a scenario driver that stimulates the Gateway and a message recorder that responds as the Space CSCI. The scenario driver can replay a stored scenario of messages or generate a random sequence of messages. The capabilities of the Gateway are verified by comparing the contents of the Audit Trail and the Message Event data store with the selected scenario. Once the Gateway has been demonstrated to work in isolation, the system is integrated and tested against system level requirements.

CONCLUSIONS

The approach described in this paper provides a cost effective alternative to relying exclusively on software architecture to implement security policy. This approach is suitable for a wide variety of applications given the system environment provides adequate physical security and a careful mapping of data flow. This approach has the added advantage that it can often be applied to existing system to implement additional security requirements.

Increasing emphasis is being placed on system security and developers must have a greater awareness of the consequences implied by interconnectivity of systems. This paper attempts to distill much of the security issues to fundamental system and software engineering design principles. With better understanding of these principles and their relationship to system security we will be able to build more effective and secure systems with less reliance on security specialists.

BIBLIOGRAPHY

Parn-71 Parnas, D., "Information Distribution Aspects of Design Methodology", *Proceedings of the 1971 IFIP Congress, Booklet TA-3,* Amsterdam: North-Holland, 1972.

Fitz-89 Fitzgibbon, J. P., "Cost Analysis of Security Filter Concept", Martin Marietta internal technical report, October, 1989.

Yellow CSC-STD-003-85, Computer Security Requirements -- Guidance for Applying the Department of Defense Trusted Computer System Evaluation Criteria in Specific Environments, June, 1985.

Orange DoD 5200.28-STD, Department of Defense Trusted Computer System Evaluation Criteria, December, 1985.

Barn-89 Barnes, J. G. P., *Programming in Ada*, Third Edition, Addison-Wesley, Workingham, England.

Gass-88 Gasser, Morrie, *Building a Secure Computer System,* Van Nostrand Reinhold, New York, 1988.

AUTHORS

Mr. Fitzgibbon is a senior group software engineer for Martin Marietta Information and Communications Systems. His current assignment is in Colorado Springs supporting Air Force Space Command in the development of systems for NORAD. In this capacity, he manages system engineering, design, analysis and integration testing. Mr. Fitzgibbon is a member of the Institute of Electrical and Electronics Engineers and the Association of Computing Machinery. He graduated from Northwestern University in 1972 with a BSEE. Mr. Fitzgibbon can be reached at 4180 E. Bijou, Colorado Springs, Colorado 80909 or telephoned at (719) 591-3800.

Mr. Madsen is a staff system engineer for Martin Marietta Information and Communications Systems. His current assignment is in Denver performing security analysis and architecture for systems requiring Multi-Level Security. Mr. Madsen is a member of the Institute of Electrical and Electronics Engineers. He graduated from University of Iowa in 1981 with a BSEE. Mr. Madsen may reached at P.O. Box 1260, Denver, Colorado 80201-1260 or telephoned at (303) 977-6883.

Verification of Generics in Ada[1][2]

Fuyau Lin, George W. Ernst

Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

Abstract

The work is to develop methods for verifying that generic procedures or functions implemented in Ada are correct. The methods will be modular in the sense that an implementation can be verified once, and then be used by other Ada packages without any reverification of the implementation. The methods will also be logically sound and relatively complete. Verification rules and an example of verification are included.

1 Introduction

The cost of software development has become a major expense in the application of computers. The problem of being able to develop correct as well as clear code also requires the application of a systematic procedure. Proving even simple programs correct turns out to be a far from easy task. Program verification refers to the application of mathematical proof techniques to establish that the result obtained by the execution of a program with arbitrary inputs are in agree with formally defined output assertions.

The generic mechanism in Ada allows packages to be parameterized by types, functions and procedures. The specifications of such procedure type parameters are needed for verifying a generic because its implementation will invoke these procedures, and it is their specifications which are used in verification to determine the net effect of their invocations. Our method for specifying these procedures uses predicates which are also parameters to the generic. Such predicates are not implemented by Ada code, but rather are part of the specification language. The precondition of the generic specifies the properties of these predicates which are needed by the generic's implementation.

The work is to develop methods for verifying that generic program units implemented in Ada are correct. The methods will be modular in the sense that an implementation can be verified once, and then be used by other Ada packages without any reverification of the implementation. The methods will also be logically sound and relatively complete; soundness guarantees that any program which can be verified is, in fact, a correct program. Completeness also has practical importance because it guarantees that any correct program can be verified by our methods, at least in principle; i.e., the methods are not just designed for special classes of programs.

In this paper, section 2 describes the background and the basic approach. Section 3 and 4 present the verification of generics and the rules, respectively. We give example in section 5.

[1]Ada is a registered trademark of the U. S. Department of Defense.

[2]This work is supported by DoD under grant MDA904-89-C-6018.

2 Verification of Generic

2.1 Background

Modular verification allows one module of a program to be verified independently of the other modules. Since one module can reference other modules, specifications of the latter is need for verification of the former. However, the implementation of a module is only available for its verification, whereas its specification is available for verification of any module which references it. Many researchers, including the authors, feel that any practical verification method will be modular; for example, see [14] and [26].

There is much less literature on the verification of Ada generics. Young and Good [26] is the only work that we know of whose main topic is verifying Ada generics. They propose a modular method for a special class of generics, but such methods are inherently incomplete, unlike the methods that we are developing. Young and Good [26] conclude that modular verification of Ada generics is very difficult in the general case, since modularity precludes methods based on removing generics by macro-expansion.

Ada generics allow procedures and packages to have parameters whose types may be *procedure* or *type*. There is some literature on the verification of procedures with parameters of type procedure. This is relevant to the verification of generics even though most of this literature precludes parameters whose types are type. We will not describe this literature here because it is described in [7].

Most theoretical research on verification does not provide for modularity, nor can it be easily modified to do so. The reason is that modularity requires the validity of a program to depend on the specifications of externally defined quantities, such as procedures and data abstractions, but not on their implementations. Most definitions of validity, such as those in [2], [3], [5], [12], and [22], do not provide for such external specifications; rather the *validity of a program* only depends on its input and output assertions. In a valid modular program the effect of invoking an externally defined procedure is determined by its specification; for example, its precondition must be true before the invocation. Usually validity does not use such specifications to determine the effects of a procedure call, but rather uses the body of the procedure for this purpose. Our definition of validity uses such external

specifications to provide for modularity, but changing such fundamental definitions as validity has a large impact on many aspects of the meta-theory.

2.2 Basic Approach

We assume that a generic or a package P can reference items defined externally, for instance, a procedure defined in some other packages. The verification of P depends on the specification of such externally defined items, but not on their implementations. In addition, the things exported from P can be used by other packages under the assumption that they have been (or will be) verified, and thus no part of P needs to be reverified in the verification of code which uses something exported from P. This is just an explicit statement of modular verification, but it points out the essential role of the specification of externally defined items.

In our verification process, formulae have the form $SC; GC \backslash P$, where P is a program fragment to be verified and $SC; GC$ is the context in which P executes. The context consist of two parts:

- The **specification context** SC is a list of the specifications of externally defined procedures, functions, packages and generics that can be used by P.
- The **generic context** GC is the conjunction of the preconditions of all of the generic declarations whose scopes enclose P.

  The reason for this part of the context is that P can reference the formal parameters of these generics, and their preconditions describe properties of their formal parameters.

The information in the context is assumed to be *true* when verifying P. The code to be verified P has one of two forms:

- It is declaration which contains its own specification, for example, a package declaration.

  We have added a formal specification mechanism to Ada and we require specifications to occur at appropriate places in the code. Thus, the programs to be verified are a mixture of code and specifications.
- The other possibility is a formula of the form, **assume** Q; A; **confirm** R.

  A is a sequence of program statements and declarations with precondition Q and post-condition R. Thus, it is written as $Q\{A\}R$ in Hoare's [15] notation.

The verification rules are formulated so that they are mechanical in the sense that they prescribe an algorithm for converting a verification formula to a statement in predicate calculus. This is achieved by the following property of the verification rules: any formula $SC; GC \backslash P$ to be verified will match the bottom line of precisely one rule which essentially selects the rule to apply to the formula. Not only does this make the rules easy to use, for either a human or a machine, but the mechanical property has some nice theoretical properties.

The rules will be logically sound if they all preserve validity when applied in the forward direction, and they will be relatively complete if they preserve validity when applied in the reverse direction.

The semantics of our verification formulae are different from the usual semantics found in the literature. Most of this difference stems from the context which is needed for modular verification. This allows our definition of validity to depend on specifications which is necessary for modular verification. The next subsection discusses issues about verifying packages and generics.

3 Verifying Ada Generics

The generic mechanism in Ada allows packages to be parameterized by types, functions and procedures. This is obviously a desirable feature of Ada because one may want to define. If T is a type parameter of a generic, its implementation will need procedures in which some of their parameters are of type T in order to manipulate objects of type T. These procedures must also be parameters to the generic because different instances of it will require different procedures for this manipulation, since some of their arguments are of type T. Although *equality* and *assignment* are defined for all types in Ada which are not limited , it is desirable, in many cases, to use other operations to manipulate the elements of type T. This is particularly true if T is an abstract data type because there may be several realizations for a single abstract value in which case equality and assignment do not work. Thus, the use of type parameters often gives rise to parameters of type procedure.

The specifications of such procedure type parameters are needed for verifying a generic because its implementation will invoke these procedures, and it is their specifications which are used in verification to determine the net effect of their invocations. Our method for specifying these procedures uses predicates which are also parameters to the generic. Such predicates are **not** implemented by Ada code, but rather are part of the specification language. The precondition of the generic specifies the properties of these predicates which are needed by the generic's implementation.

As an example suppose that a generic has three parameters: T is a type; P is a predicate; and Q is a procedure with a parameter X of type T. The specification of Q will be statements about X, but since T is a parameter to the generic, X must be an argument of P in this specification because it is the only predicate which can have arguments of type T, i.e., the only predicate parameter of the generic.

The precondition of the generic will give the properties of P which are necessary to verify the generic. For example, the precondition may specify P to be transitive and non-reflexive. Then, $P(I, X)$ in the post-condition of Q is true if I is "less-than" and not equal to X. We are assuming that this occurrence of i in the post-condition is a quantified variable of type T. Verification of the generic can assume that P is transitive and non-reflexive, and an instantiation of the generic must verify that the actual parameter corresponding to P has these properties. For example, if T is **integer** , P may be $<$, but T may be vectors of integers in which case an appropriate actual parameter would have to be used for P.

We have deliberately omitted **in** and **in out** parameters of generic declarations because program variables can be used as actual parameters for such formals. The difficulty is that some references to such parameters in the specification of a generic refer to their values at the point of instantiation while other references refer to their values when a procedure, defined in the generic, is invoked. This is not a conceptual difficulty because it can be dealt with by renaming certain variables. We have omitted such parameters because these renamings are somewhat counter-intuitive and

cloud the description of how the other kind of parameters are processed, which is the conceptually difficult part of verifying generics.

A major difficulty with specifying and verifying generics is that each of them potentially extends the specification language with a new theory. The objects of this new theory are the values of type T which is a parameter to the generic. The specification of the generic's implementation will be statements about these objects, and hence will contain predicates whose arguments are of type T. The precondition of the generic defines the properties of these predicates so that one can reason about the objects of type T in verifying that the implementation is correct. Such specifications essentially constitute the axioms for the new theory.

Extending a specification language with such a new theory may cause it to lose its expressive property, assuming that it was expressive before the extension. The reason is that nothing is known about the new theory; it is part of the generic's specification, and the person formulating the specification has considerable freedom as well as power, since he has the ability to specify new theories. For example, the specification of a generic may be a very weak characterization of its new theory which prevents the expression of the post-conditions of certain procedures in the extended specification language. In addition, different generics will probably define different theories, and thus extend the specification language in different ways. Such extensions to the specification language appear to be unavoidable, but our method for making them preserves expressiveness. Not only does this allow all programs which use a new theory to be specified, but it is a necessary condition for the verification system to be relatively complete, as shown by [3].

The method used to keep the specification language expressive is to allow specifications to reference strings of any type of objects, e.g., strings of integers. A string is an ordered list of elements and in that sense is like a sequence. Strings are introduced into the specification language by the string constructor which constructs new types out of old types in the same way as the array constructor. String type objects can only occur in the specification language and not in the programming language; thus, we are not proposing an extension to Ada, but rather a method for specifying Ada programs.

4 Rules of the Game

This section describes verification rules in connection with the generic program unit. SG was chosen because it typifies generic program specifications in the sense that it has one of each different kind of parameter, e.g., one type parameter, one predicate parameter, etc. Of course, in general there may be zero or multiple parameters of each kind, but the verification rules show how each different kind is to be processed.

In figure 1, UnitSpec is the specification of a procedure or a function including **require** and **ensure** clauses. SG shown above has five parameters:

- T is a type parameter.
- PSL is a predicate of the specification language with two arguments.
- FSL is a function of the specification language whose argument is of type T3 and whose value of type T4.

```
generic type T is private;
   with predicate PSL(X : T1; Y : T2);
   with specification function FSL(X : T3) return T4;
   with procedure P(X : in out T5; Y : in T6);
      require PreP;
      ensure PostP;
   with function F(X : in T7) return T8;
      require PreF;
      ensure PostF;

   require PreG;
UnitSpec.
```

Figure 1: The specification, SG, of a typical generic program unit.

- The procedure-type parameter P has PreP and PostP as its pre- and post-conditions.
- Finally, F is a function-type parameter with PreF and PostF as its pre- and post-conditions.

Note that any of the T_i above may be T, the type-type parameter of SG. The pre-condition of SG is PreG.

A declaration like SG will be correct in some contexts but not in others since it may contain external references such as a call on a procedure declared outside of SG. We denote this by formulae of the form $SC; GC \backslash P$ where P is the program fragment to be verified and $SC; GC$ describes the context in which P executes. P is either a *declaration* like SG which contains its own specification or a *formula* of the form, `assume Q; A; confirm R;`.

In the latter, A is a sequence of program statements and declarations with pre-condition Q and post-condition R. The context consist of two parts: the specification context SC is a list of the specifications of externally defined procedures, functions, packages and generics. By specification[1] we mean the declaration without its implementation.

The generic context GC is the conjunction of the preconditions of all of the generic declarations whose scopes enclose P. The reason for this part of the context is that P can reference the formal parameters of these generics, and their preconditions describe the properties of their formal predicates and specification language functions. This information can be assumed to be true in verifying P.

```
procedure P(X : T1; Y : in out T2);
   require PreP;
   ensure PostP;

procedure P(X : T1; Y : in out T2) is
D2;
begin A4; end P;
```

Figure 2.a: The declaration, DP, of a typical procedure.

In the case of generic (regular) procedure or function specifications, we have the following three verification rules with slightly difference:

[1] In the Ada manual, a procedure specification only describes the parameters and their types. In addition to this our procedure specification contain a pre- and a post-condition.

```
function F(X : in T) return T2;
   require PreF;
   ensure PostF;

function F(X : in T) return T2 is
D2;
begin A4; end F;
```

Figure 2.b: The declaration, DF, of a typical function.

```
[SR1]

   C, SP \ A1; declare D1 begin A2 end; confirm Q1;
   ------------------------------------------------
   C \ A1; declare SP; D1 begin A2 end; confirm Q1;
```

where SP is the specification of either a regular or generic procedure or function.

```
[SR2]

   SP, C \ DB
   SP, C \ A1; declare D1 begin A2 end; confirm Q1;
   ------------------------------------------------
   Sp, C \ A1; declare DB; D1 begin A2 end; confirm Q1;
```

where DB is the body of either a regular or generic procedure or function.

```
[SR3]

   C \ DP
   C, SP \ A1; declare D1 begin A2 end; confirm Q1;
   ------------------------------------------------
   C \ A1; declare DP; D1 begin A2 end; confirm Q1;
```

where DP is either the declaration of an instance of a generic procedure or function, or the declaration of a procedure or function which contains both specification and body.

The next rules we discuss, [GDVR], is the rule for generic declaration verification; i.e., for verifying generic body, GB. GB is correct if the externally defined quantities meet their specifications in SC and the formal parameters of the enclosing generics satisfy GC. All this rule says is that the declaration must be correct in an extended context which has the precondition of GB **anded** onto the old generic context. In addition, the specifications of actual procedures and functions are part of the new specification context so that they can be processed as ordinary, instead of formal, procedures and functions, in verifying generic body.

```
[GDVR]

   NSG, SP, SF, SC; GC & PreG \ GB;
   --------------------------------
   SG, SC; GC \ GB;
```

where GB is the body of the generic with specification SG. SG is the specification part of generic procedure or function. SP is the specification of procedure P given in SG. SF is the specification of function F given in SC. NSG is SG with the generic part removed.

The information in the generic context is used whenever a pure predicate calculus verification condition is generated, i.e., one which contains no program statements nor declarations. The generic context is in the antecedent of each such verification condition. Thus, all of the usual verification rules found in the literature must be extended so that the generic context is used in this way.

An instance of a generic declaration is given by a declaration of the form: `kind GI is new G(AT, APSL, AFSL, AP, AF);` The verification rule for such an instance is given below:

```
[GIVR]

   GC -> PreG [APSL/PSL, AFSL/FSL, AT/T];
   GC & PreP [APSL/PSL, AFSL/FSL, AT/T]
                    -> PreAP [X/U, Y/V];
   GC & PostAP [X/U, Y/V, #X/#U]
                    -> PostP [APSL/PSL, AFSL/FSL, AT/T];
   GC & PreF [APSL/PSL, AFSL/FSL, AT/T] -> PreAF [X/U];
   GC & PostAF [X/U]
              -> PostF [APSL/PSL, AFSL/FSL, AF/F, AT/T];
   ------------------------------------------------------
   SAP, SAF, SG, SC; GC \ kind GI is
                          new G(AT, APSL, AFSL, AP, AF);
```

where SG is the specification of generic unit G. SAP and SAF are the specification of AP and AF, respectively. These specifications look like:

```
procedure AP(U: in out T5; V : in T6);
   require PreAP;
   ensure PostAP;

function AF(U : in T7) return T6;
   require PreAF;
   ensure PostAF;
```

The specification context of this instance contains the specifications of the generic unit and the actual procedure and function parameters, AP and AF. SC is a list of the other items in the specification context while GC is generic context.

The bottom line of the rule in [GIVR] is verified by verifying all of the lines above it. The first line requires the PreG to be *true* after substituting the actual types, predicates and specification language functions for formals. We use the following notation for substitution: `Q[e1/X1,...,en/Xn]` denotes the result of simultaneously substituting expressions e_i for symbols X_i in formula Q. In addition, we use the convention that substitution has higher precedence than logical connectives.

An **in out** parameter in the post-condition of a procedure refers to its final value; its initial value is denoted by its name preceded by a sharp sign (#). Hence, #U and #X refer to the initial values of U in PostAP and X in PostP, respectively.

These verification rules are the only rules needed for generics. However, to show how these rules interface with other verification rules we will describe how to verify a block which contains the declaration of an instance of a generic procedure. Three more rules are included for this purpose: procedure and function verification rule, [PFVR]; procedure invocation rule, [PIR]; and function invocation rule, [FIR].

```
[PFVR]

   C, SP \ assume PreP & #Y = Y & #Z = Z;
          declare D2 begin A4 end; confirm PostP;
   ----------------------------------------------
   C \ DP;
```

where DP can be replaced by DF with minor change.

```
[PIR]

   C, SP \ A1; confirm PreP [e/X, U/Y];
   C, SP \ A1; confirm forall U' Z'
          (PostP [e/X, U'/U, Z'/Z, #Y/Y, #Z/Z]
                -> Q1 [U'/U, Z'/Z]);
   ------------------------------------------
   C, SP \ A1; P(e, U); confirm Q1;

[FIR]

   C, SP \ A; confirm PreF [e/X];
   C, SP \ A; confirm forall U (PostF [e/X, U/F] -> Q);
   ------------------------------------------------
   C, SP \ A; U := F(e); confirm Q;
```

5 Example: Euler Path

In this section we look at the problem of verifying the correctness of programs which use or define generics. In this example, we would like to solve the *Euler Path Problem*. Given an undirected connected graph $G = (V, E)$, there exists an eulerian path P such that each edge of E appears in P exactly once. To solve this problem, we will apply the theorem below:

Theorem (Euler): Let G be a connected graph. Then G contains an eulerian path if and only if G has exactly zero or two odd vertices.

Before we start to verify our program, we need to introduce part of the statements which comprise the theory of the specification type graph of NODE0.

```
Types:
   NS0 is sequence of NODE0;
   GRAPH0 is graph of NODE0;

Axioms:
   forall i, j : NODE0 (edge(i, j, G) -> edge(j, i, G))
        where the type of G is GRAPH0;

Definitions:
   path(p : NS0; G : GRAPH0) iff forall i : NODE0
        (1 <= i < len(p) -> edge(p(i), p(i+1), G);

   connect(G : GRAPH0) iff
        forall i, j : NODE0 -> (exists p : NS0
        (path(p, G) & p(1) = i & p(len(p)) = j));

   degree(v : NODE0, G : GRAPH0) =
        card({k : NODE0 | edge(v, k, G)});

   e_path(p : NS0; G : GRAPH0) iff
        forall i, j : NODE0 ((edge(i, j, G) ->
           exists k : int ({i, j} = {p(k), p(k+1)} &
           forall n : int (n /= k ->
              {p(n), p(n+1)} /= {i, j})));

   odd_nodes(G : NODE0) = card({x : NODE0 |
                               degree(x, G) mod 2 /= 0});

Theorems:
   connect(G) -> ((odd_nodes(G) = 0 or odd_nodes(G) = 2)
        iff exists p : NS0 (e_path(p, G))
        where the type of G is GRAPH0;
```

Base on the specification definitions and theorem mentioned above, we have three modules: a module to manipulate graph; a generic EULER function; a block that invokes other modules.

```
[MOD1]

 1. package PG1 is
 2.    type NODE1 is range 1..N;
 3.    type GRAPH1 is limited private graph of NODE1;
 4.    function SUCC1(I1, J1 : NODE1; G1 : GRAPH1)
 5.                 return BOOLEAN;
 6.    ensure SUCC1 = edge(I1, J1, G1);
 7.
 8.    procedure ADD1(I1, J1 : NODE; G1 : in out GRAPH1);
 9.    ensure edge(I1, J1, G1) & for all p, q
10.          ({p, q} /= {I1, J1}
11.          -> (edge(p, q, #G) iff edge(p, q, G)));
12.   ...
13.
14.   private
15.   ...   -- implementation details
16. end PG1;
```

The generic function EULER will import three different types and a successor function. There are two loop invariants. ODD_DEG has no **require** clause, in this case we assume it is true.

```
[MOD2]

 1. generic
 2.    type NODE2 is range <>;
 3.    type GRAPH2 is limited private graph of NODE2;
 4.    type NS2 is sequence of NODE2;
 5.    with function SUCC2(I2, J2 : NODE2; G2 : GRAPH2)
 6.                          return BOOLEAN;
 7.    ensure SUCC2 = edge(I2, J2, G2);
 8.
 9. function EULER(G2 : GRAPH2) return BOOLEAN;
10.    require connect(G2);
11.    ensure EULER = exists p : NS2 (e_path(p, G2));
12. function EULER(G2 : GRAPH2) return BOOLEAN is
13.    function ODD_DEG(W : NODE2; G2 : GRAPH2)
14.                       return BOOLEAN is
15.    ensure ODD_DEG = degree(W, G2) mod 2 /= 0;
16.       M : INTEGER;
17.    begin
18.       M := 0;
19.       for K in NODE2 loop -- inv_M is
20.       -- card({j|j : NODE2 (1<=j<K & SUCC2(W,j,G2))});
21.          if (SUCC2(W, K, G2)) then
22.             M := M + 1;
23.          end if;
24.       end loop;
25.       return (M mod 2 /= 0);
26.    end ODD_DEG;
27.
28.    N : INTEGER;
29. begin
30.    N := 0;
31.    for V in NODE2 loop -- inv_N is
32.    -- card({j|j : NODE2 (1 <= j < V & ODD_DEG(j,G2))});
33.       if (ODD_DEG(V, G2)) then
34.          N := N + 1;
35.       end if;
36.    end loop;
37.    return (N = 0) or (N = 2);
38. end EULER;
```

The best way to illustrate the verification procedure is by example. This module is a block to show *generic instantiation* and *function invocation*.

```
[MOD3]

 1. declare
 2. use PG1;
 3. G3 : GRAPH1;
 4. type NS1 is sequence of NODE1;
 5.
 6. function EULER1 is new function EULER(NODE1,
 7.                              GRAPH1, NS1, SUCC1);
 8. begin
 9.    A;                    -- initialization of G3
10.    b := EULER1(G3);      -- b is a global variable
11. end;
12.
13. confirm b = exists p : NS1 (e_path(p, G3));
```

Verify Generic Declaration

Three kind of verifications are involved: generic declaration, generic instance, and instantiated function. First, we verify the generic function EULER:

```
\ D_GE
        where D_GE is MOD2 line 1-38

   => [SR1]

S_S2 \ D_E
        where S_S2 is MOD2 line 5-7 without the "with"
              D_E is MOD2 line 9-38

   => [PFVR]

S_S2; S_E \ assume connect(G2); declare E_Body;
        confirm EULER = exists p : NS (e_path(p, G2))
        where S_E is MOD2 line 9-11
              E_Body is MOD2 line 14-38

...
```

Verification Conditions

In our generic EULER function, we get 8 verification conditions. The validity of these *VC*s are left to readers.

```
(1) M = card({j | j : NODE2 (1 <= j < K &
              SUCC2(W, j, G2))}) & ~(K in NODE1)
        -> M = degree(W, G2)

(2) 0 = card({j | j : NODE2 (1 <= j < 1 &
        SUCC2(W, j, G2))});

(3) inv_M & (K in NODE2) & SUCC2(W, K, G2) ->
        M+1 = card({j | j : NODE2 (1 <= j < K+1 &
                    SUCC2(W, j, G2))});

(4) inv_M & (K in NODE2) & ~SUCC2(W, K, G2)
        -> inv_M [K + 1 / K];

(5) inv_N & ~(V in NODE2) -> (N = 0) or (N = 2) iff
        exists p : NS2 (e_path(p, G2));

(6) connect(G2) -> 0 = card({j | j : NODE2 (1 <= j < 1 &
        ODD_DEG(j, G2))});

(7) inv_N & (V in NODE2) & ODD_DEG(V, G2) ->
        N+1 = card({j | j : NODE2 (1 <= j < V+1 &
                    ODD_DEG(j, G2))});

(8) inv_N & (V in NODE2) & ~ODD_DEG(V, G2) ->
        inv_N [V + 1 / V];
```

Verify Generic Instantiation

This subsection, we show the verification steps of verifying a block with a generic instantiation and a function invocation.

```
S_GE \ GIB
        where GIB is MOD3 line 1-13
              S_GE is MOD2 line 1-11

   => variable declaration rule (including the "use")

S_GE, S_S1, RSG1 \ declare D_E1; begin A; b := EULER1(G3);
        end; confirm b = exists p : NS1 (e_path(p, G3));
        where S_S1 is MOD1 line 4-6
              D_E1 is MOD3 line 6-7
              RSG1 is the rest of specification of PG1
                      except SUCC1

   => [SR3]

1. S_GE, S_S1, RSG1 \ function EULER1 is
        new function EULER(NODE1, GRAPH1, NS1, SUCC1);

2. S_GE, S_S1, RSG1, S_E1 \ declare begin A;
        b := EULER1(G3); end; confirm
        b = exists p : NS1 (e_path(p, G3));
        where S_E1 is S_E [EULER1/EULER, GRAPH1/GRAPH2,
              NS1/NS2] yields
              function EULER1(G2 : GRAPH1) return BOOLEAN;
              require connect(G2);
              ensure EULER1 = exists p : NS1
                     (e_path(p, G2));

From 1:

S_GE, S_S1, RSG1 \ function EULER1 is
        new function EULER(NODE1, GRAPH1, NS1, SUCC1);

   => [GIVR]

1a. (none)

1b. SUCC1 = edge(I1, J1, G1) [I2/I1, J2/J1, G2/G1]
        -> (SUCC2 = edge(I2, J2, G2)) [SUCC1/SUCC2];

=> SUCC1 = edge(I2, J2, G2) -> SUCC1 = edge(I2, J2, G2)

From 2:

S_GE, S_S1, RSG1, S_E1 \ declare begin A;
        b := EULER1(G3); end;
        confirm b = exists p : NS1 (e_path(p, G3));

   => declare removal rule

S_GE, S_S1, RSG1, S_E1 \ A; b := EULER1(G3);
        confirm b = exists p : NS1 (e_path(p, G3));

   => [FIR]

2a. S_GE, S_S1, RSG1, S_E1 \ A; confirm connect(G3);
```

```
2b. S_GE, S_S1, RSG1, S_E1 \ A; confirm
      forall Z = exists p : NS1 (e_path(p, G3))
      -> Z = exists p : NS1 (e_path(p, G3));

  ...
```

As you can see, we did not show all the details of the verification steps after we move to certain verification stage. The rest of the verification process had been well developed. Our work is focus on the generic part.

6 Summary

Verification rules for the generics in Ada are presented in this paper together with the discussion of the difficulties. Our work is to develop methods for verifying that generic procedures implemented in Ada are correct. Though we didn't show them here, these rules are proven to be *logically sound* and *relatively complete* with respect to our semantics [8]. An important property of both the verification rules and the semantics is that they provide for *modular verification*. The allows the implementation of a generic unit to be verified once, independent of the units which use it. Any unit can reuse a verified generic unit, G, and the verification of this unit can assume that G is correct and does not involve reverifying any part of it.

We have already developed semantics and modular verification rules for generic procedures, as described in [7]. It is an important part of our research plan which includes the verification of generic packages, as described below. The reason is that the proposed semantics for packages is, for the most part, consistent with our semantics for generic procedures. The only major deviation is that the proposed semantics of procedure implementations is relational. We propose to modify our semantics for generic procedures so that it also uses relational semantics for procedures. This should allow the two semantics to be combined together in developing semantics for generic packages and procedures.

The development of modular verification rules for generic packages and procedures will essentially combine the two separate sets of verification rules that we will have developed for packages and generic procedures. The combination of the semantics described above should not be difficult, but more research is needed on the meta-theory of this combination of packages and generic procedures. However, we believe that we have a good way to deal with a major difficulty: the specification language for generic packages needs to be expressive in order for the verification rules to be relatively complete.

References

[1] Chang, E., Kaden, N. E. and Elliott, W. D., Abstract Data Types in Euclid, SIGPLAN Notice, March, 1978.

[2] Clarke, E. M., Programming Language Constructs for Which It is Impossible to Obtain Good Hoare Axiom Systems, Journal of the ACM, January, 1979, pp. 129-147.

[3] Cook, S. A., Soundness and Completeness of an Axiom System for Program Verification, SIAM Journal of Computing, February, 1978, pp. 70-90.

[4] Dahl, O. J., Myhrhaug, B. and Nygaard, K., The SIMULA 67 Common Base Language, Publication S-22, Norwegian Computing Center, Oslo, 1970.

[5] Donahue, J. E., Complementary Definitions of Programming Language Semantics, Report CSRG-62, Computer Systems Research Group, Univ. of Toronto, Nov. 1975.

[6] Ehrig, H. and Mahr, B., Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics, EATCS Monographs on Theoretical Computer Science, vol. 6, Springer Verlag, 1985.

[7] Ernst, G. W., Hookway, R. J., Menegay, J. A. and Ogden, W. F., Modular Verification of Ada Generics, Report CES-86-1, Computer Engineering and Science Dept., Case Western Reserve Univ., 1986.

[8] Ernst, G. W., Hookway, R. J., Menegay, J. A. and Ogden, W. F., Semantics of Programming Languages for Modular Verification, Report CES-85-04, Computer Engineering and Science Dept., Case Western Reserve Univ., 1985.

[9] Flon, L. and Misra, J., A Unified Approach to the Specification and Verification of Abstract Data Types, Proc. of Specifications of Reliable Software Conf., IEEE Computer Society, 1979, pp. 162-69.

[10] Gannon, J. D., Hamlet, R. G. and Mills, H. D., Theory of Modules, IEEE Trnas. on Software Engineering, July, 1987, pp. 820-829.

[11] Geschke, C. M., Morris, J. H. Jr. and Satterwaite, E. H., Early Experience with Mesa, Comm. of the ACM, Aug., 1977, pp. 540-553.

[12] Gorelick, G. A., A Complete Axiomatic System for Proving Assertions about Recursive and Non-recursive Programs, Report 75, Computer Science Dept., Univ. of Toronto, Feb., 1975.

[13] Guttag, J., Horning, J. J. and Wing, J. M., The Larch Family of Specification Languages, IEEE Trans. on Software Engineering, Sept., 1985.

[14] Hailpern, B. and Owicki, S. S., Modular Verification of Concurrent Programs, Symposium on Principles of Programming Languages, January, 1982, pp. 322-336.

[15] Hoare, C. A. R., An Axiomatic Basis for Computer Programming, Comm. ACM, 1969, pp. 576-581.

[16] Hoare, C. A. R., Proof of Correctness of Data Representations, Acta Informatica, 1972, pp. 271-281.

[17] Kemmerer, R.A., Verification Assessment Study, Final Report, Vol. II, The Gypsy System, National Computer Security Center, Fort George G. Meade, MD, 1986.

[18] Lampson, B. W., Horning, J. J., London, R. L., Mitchell, J. G. and Popek, G. J., Report on the Programming Language Euclid, SIGPLAN Notices, Feb., 1977, pp. 1-79.

[19] Liskov, B. and Guttag, J., Abstraction and Specification in Program Development, MIT Press, 1986.

[20] London, R. L., Guttag, J. V., Horning, J. J., Lampson, B. W., Mitchell, J. G., and Popek, G. J., Proof Rules for the Programming Language Euclid, Acta Informatica, Jan., 1978, pp. 1-26.

[21] Luckham, D. C., von Henke, F. W., Krieg-Brueckner, B. and Owe, O., ANNA A Language for Annotating Ada Programs, Preliminary Reference Manual, Computer Systems Laboratory, Stanford, Univ., 1984.

[22] Olderog, E. R., Sound and Complete Hoare-like Calculi Based on Copy Rules, Acta Informatica, Vol. 16, 1981, pp. 161-197.

[23] Scott, D. and Strachey, C., Toward a Mathematical Semantics for Computer Languages, Computers and Automation, John Wiley, pp. 19-46, 1972.

[24] Smith, M. K. and Good, D. I., Software Verification in Gypsy, Verification Assessment Study, Final Report, Vol. II, The Gypsy System, Kemmerer, R. A. (ed.), National Computer Security Center, Fort George G. Meade, MD, 1986, pp. 5-20.

[25] Wulf, W. A., London, R. L. and Shaw, M., An Introduction to the Construction and Verification of Alphard Programs, IEEE Trans. on Software Engineering, Dec., 1976, pp. 253-64.

[26] Young, W. D. and Good, D. I., Generics and Verification in Ada, Proc. of the ACM-SIGPLAN Symposium on the Ada Programming Language, December, 1980.

Reuse Engineering and Procedure Types in Ada

Larry Latour and Curtis Meadow
University of Maine
Department of Computer Science
Orono, Maine, 04469

Abstract

Reuse engineering is the discipline of constructing "highly parameterized" components, thus encapsulating component content while parameterizing contextual information supplied by the environment. Procedure types are a language construct that have been suggested to help in this regard, but are noticably absent from Ada. We explore various language and implementation issues of procedure types, providing a number of examples demonstrating how procedure types facilitate the abstraction of control and how they can be used to construct abstract data type "variables", providing run-time parameterization capability.

1 Introduction

Reuse engineering is the discipline of constructing components which embody content (the implementation of a content), and are parameterized by context (information supplied by the environment of the component). A language supports reuse engineering to the extent that it supports the separation of content from context. One particular language construct that has been suggested to help in this regard is the procedure type, but it is noticably absent in Ada. We discuss issues related to procedure types, along with the ramifications of adding such a construct to Ada.

One form of polymorphism in programming languages refers to the ability of a language to support type parameterization [Card 85], and as such is a language property highly desirable for reuse engineering. Examples of such parameterized components include abstract data types whose components can be any type with an assignment operator (e.g., stacks and queues), and functions such as a sort of elements, where elements are restricted to those types that have an assignment operator and a linear ordering operator.

There currently exist facilities in Ada to partially support polymorphism, notably generic packages and task types. Unfortunately each has its drawbacks. Generic parameters are statically bound, hence they don't provide the run-time flexibility desired in a number of applications. Since the original intent was that the generic package be simply a code saving mechanism, such inflexibility is understandable. This, unfortunately, provides no solace to application designers. Task types, on the other hand, can be used to provide parameters that can be bound at run-time, but the overhead of doing so is potentially large.

Procedure types are more flexible than generics, since procedure objects can be bound at run-time. This facilitates the abstraction of control, as well as the ability to create abstract data type "variables" with procedure objects bundled in records. We present a number of examples to highlight this usefulness.

An important issue when considering procedure types for a language is the choice of procedure type implementation and how it is influenced by language issues such as binding time and structural vs. name equivalence. We discuss the impact of both language design and implementation issues on procedure types.

Our paper is organized as follows. We first present an historical review in section 2, followed by a rationale for procedure types in section 3. After a discussion of language issues in section 4, we present examples in section 5 that demonstrate how procedure types facilitate the abstraction of control. Section 6 introduces the notion of reuse engineering, and includes an example of a quick sort algorithm using procedure types to provide implementation context. We then discuss in section 7 how procedure types can be used to represent abstract data type "variables". We complete our paper with a section on implementation issues, followed by a summary and research directions.

2 Historical Perspective

Languages have allowed procedures to be treated as values as far back as Lisp [McCa 65] and Algol60 [Naur 63] . The ways in which procedures as values have been treated are varied and can be characterized by "degree of first-classedness", i.e., by the ability of objects to be created, destroyed, and assigned as variable values at run-time.

Language support for procedure objects varies from simply allowing subprograms to be passed as parameters as in Fortran 77, to the support of "persistent" procedure variables as in PS-Algol [Atki 85]. Other languages in which procedure variables can be declared and assigned values include Algol68 [vanW 75], C [Kern 88], Modula-2 [Wirt 83], ML [Harp 86] , CLU [Lisk 86], SmallTalk [Gold 83], as well as a number of Pascal [Reyn 81] dialects.

Although the Ada language [Ada 83] does not have procedure types, and does not allow for procedure parameters, it does provide the facility to parameterize compile-time generics with procedure parameters, and some of the same design choices for these generics (e.g., actual-formal parameter correspondence, parameter syntax) can be used in adding procedure types to Ada. In addition, Ada task variables can be used to simulate procedures [Lamb 83].

Our view of procedure types is similar to that proposed in [Hilf 82] , an excellent general discussion of Ada language design issues. Our view is that a procedure type specifies the number and type of parameters, and the type of the result, if any. For example, consider a function to reduce over a vector:

```
type vector is array range <> of element;

function reduce
  (v: vector;
   f: function(L,R:element)return element)
                          return element;
```

We can execute reduce for various function values of f as follows:

```
sum     := reduce(v,"+");
product := reduce(v,"*");
hashVal := reduce(v,XOR);
```

Unlike existing types in Ada whose equivalence is based upon the notion of name equivalence, a procedure type is based on the notion of structural equivalence (see the Language Issues section). In this sense, actual-formal procedure type parameter correspondence follows the same rules as for generic parameter correspondence.

3 Rationale for Procedure Types

Simply adding procedure types to Ada adds very little semantic power to the language. That is, any program using procedure types can be equivalently implemented without them. What procedure types do provide is a level of expressiveness previously lacking.

Since our perspective is abstraction and reusability, we are primarily interested in the following criteria:

1. **Expressiveness**: we should be able to easily express the "generic" properties of an algorithm along with parameters for providing the "context" in which it executes.
2. **Orthogonality**: we should be able to add a construct to the language in a consistent, uniform way, without affecting the semantics or usability of existing constructs.
3. **Usefulness**: the added construct should become an integral part of our model for building parameterized programs.
4. **Simplicity**: the added construct should not "gratuitously" complicate the expression of a program.
5. **Safety and Style**: the added construct should promote good programming style and safety.

As expressed in [Atki 85,Harl 84,Kers 88], and as we will see in the examples in upcoming sections, procedure types:

1. enhance abstraction capabilities through encapsulation of control,
2. reduce the need for generic formal subprogram parameters,
3. express some problems naturally,
4. support reuse through enhanced parameterization capabilities, and
5. can be implemented efficiently.

4 Language Issues

In this section we discuss issues dealing with features of procedure types visible in the language interface and therefore of primary concern to the application writer. Specifically we look at a comparison of type equivalence rules and the role of execution environments.

4.1 Type Equivalence Rules

By type equivalence rules we mean those rules that determine whether or not two objects are of the same type. The two most commonly used classes of such rules are:

1. **Name Equivalence**: two objects are name equivalent only if they have been declared using the same unique identifier associated with some type.

2. **Structural Equivalence**: two objects are equivalent if their respective type definitions are isomorphic with respect to some well-defined set of rules.

The general rule in Ada for type equivalence is name equivalence. While such equivalence is easy to use and understand, and supports a strong typing system, it can be at times awkward. It can lead to messy naming problems, and to a proliferation of package dependencies.

Structural equivalence, although more difficult to understand and contributing to a weaker typing system, supports our notion of a procedure type well, and in fact is the rule used to match generic formal subprogram parameters. Specifically, structural equivalence of procedures [Hilf 82] means:

1. the sequence of subtypes of the formal parts, in order, are pairwise identical, and

2. the return types denote the same subtype.

Note that the type equivalence of formal part subtypes is based upon name equivalence. We discuss, in section 7, how this rule for equating formal part subtypes might "get in the way" of the "flexibility" of a procedure type.

In addition to providing a flexible notion of procedure type, structural equivalence also allows for the straightforward usage of procedure literals, i.e., unnamed procedure constants. Such literals provide default parameter values, initialized variables, and anonymous procedure values. For example:

```
type Predicate is function (I: Item)
                            return Boolean
              := (begin return True; end);
```

4.2 Execution Environments

A major implementation problem is determining the execution environment of formal procedures, essentially the same problem as determining where the free (or global) variables of a scope are bound [Hilf 82,Fish 88,Seth 89]. Two common approaches to providing execution environments for procedure types are *dynamic binding*, where free variables are bound at run-time in the activation environment, and *static binding*, where free variables are bound at compile-time in the definition (or lexical) environment.

Dynamic binding of procedure variables is non-orthogonal with the existing binding rules of Ada, but the lexical environment of a procedure parameter or variable cannot be statically determined. The actual execution of a formal parameter can take place outside of the lexical environment of the actual procedure.

For example, consider the following procedure:

```
procedure Main is
   procedure R(FP: procedure) is
   begin FP; end;

   procedure Q is
      X,Y: Integer := 0;
      procedure P is
      begin X:= Y+1; end;
   begin R(P); end;
begin Q; end;
```

Note that when P is actually executed, its lexical environment is invisible to R. One solution is to "package" the environment with the procedure. This packaging is called a *closure*.

The closure of a procedure includes all information required to correctly execute the procedure. The actual form of a closure depends upon whether static chains or displays are used to implement scope rules (see section 8).

5 Procedure Types in Support of the Client-Server Model

In this section we present two examples, one of a Set package, and the other of a Menu package. They are analogous in that in both cases the application writer supplies procedures to be executed *under control of the respective package*. This *client-server* model is widely used in window management systems [Jone 89], command interpretors [Whee 84], and many other components that encapsulate control.

Consider first the following partial specification of a set package in standard Ada [Ada 83]:

```
generic
   type Item is private;
package Sets is
   type Set is private;

   -- the usual operations Add, Delete,
   -- New, Is_In, etc.

   -- an iterator over a Set
   function First(S: Set) return Item;
   function Next(S: Set) return Item;
   No_More_Items: exception;
   .
private
   type Set is {implementation dependent}
end Sets;
```

Using the following type definition and package instantiation:

```
type Person is record
   Age, Height, Weight: Positive;
end record;

package P_Sets is new Sets(Item => Person);
```

we can explicity implement an iterator over the set in our application package.

```
function Select_Short (People: P_Sets.Set)
                       return P_Sets.Set is
   P: Person;
   Short_People: P_Sets.Set;

   function Short(P: Person)
                       return Boolean is
   begin
      return (P.Height < Normal.Lower);
   end Short;
begin
   Short_People := P_Sets.New;
   -- the iterator
   P:= P_Sets.First(People);
   loop
      if Short(P) then
         P_Sets.Add (Short_People,P);
      end if;
      P:= P_Sets.Next(People);
   end loop;
exception
   when P_Sets.No_More_Items =>
      return Short_People;
end Select_Short;
```

A diagram of this form of client-server interaction appears in figure 1.

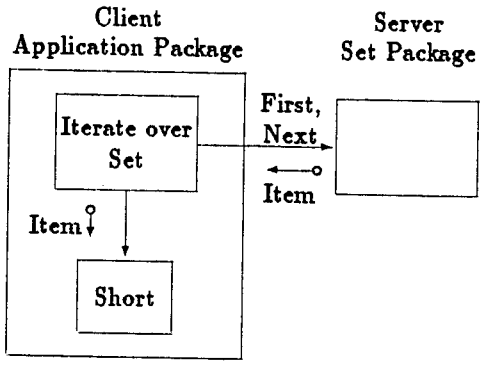


figure 1

There are a number of problems with this approach:

1. The explicitly coded iterator leads to a "cluttered" implementation.

2. The set is iterated over in the abstract, negating any efficiency gains that can be had by iterating over the representation.

3. Multiple predicates would add significantly to the complexity of the implementation.

We can deal with these problems by abstracting away the iterator control program, placing it inside of the Set package (see figure 2).

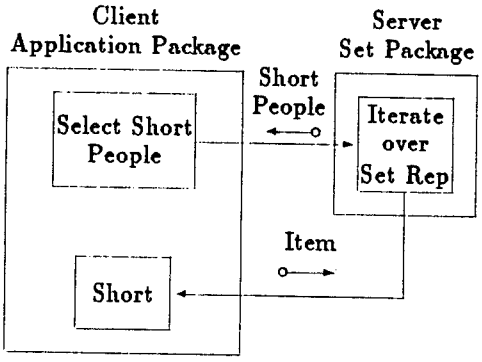


figure 2

One way of doing this in Ada would be with a generic iterator procedure, provided by the set package specification. Examples of such generic iterators can be found in [Booc 87].

First, we define the generic procedure itself:

```
generic
   with function(I: Item) return Boolean;
function Select(S: Set) return Set;
```

With this generic template, we can instantiate a procedure for each predicate of interest to us:

```
function Short (P: Person)return Boolean is
begin
   return P.Height < Normal.Lower;
end Short;

function Tall (P: Person) return Boolean is
begin
   return P.Height > Normal.Upper;
end Tall;

function Select_Short
                is new Select(P => Short);
function Select_Tall
                is new Select (P => Tall);
```

These procedures can then be executed as follows:

```
Short_People := Select_Short (People);
Tall_People := Select_Tall (People);
```

Unfortunately, this method requires explicit instantiation of the generic Select operator at compile time for each predicate. Not only is this overly restrictive, but a proliferation of names results when many predicates are used in the same application.

Procedure types provide the above capability in a more flexible way. Consider the following type definition and Select operation in the Set package specification:

```
type Predicate is function (I: item)
                                return Set;
function Select (S: Set;
                 P: Predicate) return Set;
```

We can use the previously defined functions Short and Tall as parameters to Select, as follows:

```
Short_People := Select (People, Short);
Tall_People := Select (People, Tall);
```

Using the same model, we can provide a variety of "higher level" services in the Set package, all based on the encapsulation of control. The partial specification of such a set package is now presented.

```
generic
   type Item is private;

package Sets is
   type Set is private;

   -- the usual operations Add, Delete,
   -- New, Is_In, etc.

   type Predicate is function (f: Item)
                              return Boolean;

   function Select(S: Set; P: Predicate)
                               return Set;
   -- returns a set containing those
   -- items for which p evaluates to true

   function Reject (S: Set; P: Predicate)
                               return Set;
   -- returns a set containing those
   -- items for which p evaluates to false

   function For_All (S: Set; P: Predicate)
                             return Boolean;
   function Exists (S: Set; P: Predicate)
                             return Boolean;

   function Detect (S: Set; P: Predicate)
                               return Item;
   -- returns an arbitrary item for which
   -- p evaluates to true

   procedure Remove_Subset (S: in out Set;
                            P: Predicate);
   -- equivalent to S:= S - Reject (S,P);

   procedure Iterate (S: in Set;
       Process: procedure (I: Item;
                           Done: out Boolean));
   -- iterates over the items of the set,
   -- executing P for each item

   procedure Add_Item (S: in out Set;
                       I: in Item;
       If_Present: procedure (I: in Item));
   -- executes P with parameter I if I is
   -- already in the set

   procedure Remove_Item (S: in out Set;
                          I: in Item;
        If_Absent: procedure (I: in Item));
   -- executes P with parameter I if I is
   -- not already in the set
   .
private
   type Set is {implementation dependent}
```

```
end Sets;
```

Notice that in the Set package example we can directly deal with such "higher level" abstractions as iteration, selection, and quantification. In addition, procedure types can be used to implement "resumable exceptions", allowing us to parameterize exceptional behavior as well as normal behavior [Hilf 82].

We have just seen how the procedure type allows us to provide a collection of flexible, high level operations on sets that can be used with a variety of application-supplied procedures. In the next example, a Menu package, we see how procedure types help us to address two problems common to such systems.

1. User Interface Management Systems are concerned with the man-machine interface, and not with "back-end" processing.

2. Due to the asynchronous nature of user interfaces, the user interface management system needs to be in control as the server in the client-server model, with applications invoked from the server as clients.

We present the Menus package specification below:

```
with Lists;
package Menus is
   type Return_Code is (Quit, ReDrawMenu,
      ReDrawSelector, DoNothing);

   type Application is
        procedure (R: out Return_Code)
            := (begin R:= DoNothing; end);

   type Menu_Selector is record
      Message: String;
      Do_Action: Application;
   end record;

   package Menu_Lists
       is new Lists(Item => Menu_Selector);
   use Menu_Lists;

   procedure Activate(M: List);
   procedure Terminate(M: List);
end Menus;
```

The application uses this package in essentially the same way in which the Sets package is used. A list of menu selectors is created and provided with applications to be invoked by the menu package control program (possibly in response to a mouse click, although this design decision is abstracted away from the application and encapsulated in the Menus package). The control program is then activated using this list.

Although this package is an overly simplified specification of an actual Menu package, it highlights the client-server mode of interaction also present in more complex systems.

Using procedure types in this way has obvious advantages for building generalized reusable components. In the next section we discuss the basic ideas of "reuse engineering", along with an example of how procedure types may be used to support the discipline.

6 Reuse Engineering

A recent workshop on Reuse in Practice [SEI 89] focused on the problem of how to engineer components for optimal reusability. A framework for thinking about such components was proposed which, though still in the early stages of development, captures their essence nicely. The goal of this section is to explore how procedure types help support such a framework.

A component can be thought of as being composed of three basic elements:

1. **The concept** - the abstract semantics of the component, roughly equivalent to its specification.

2. **The content** - the implementation of the component, i.e., the "how" of the behavior described by the concept. There is typically a one-many relationship between the concept of a component and its content.

3. **The context** - those parts of the environment needed to "complete" the definition of the concept or content within that environment. The context can be considered to be the parameterization of the component.

Possible kinds of context include encapsulated types, functionality parameters, and usage constraints.

In this section we explore how procedure types can be used to provide functionality parameters, allowing one to build webs of interrelated components that can be mixed and matched in a variety of ways. Procedure types can also be used as building blocks to provide encapsulated types as parameters. The issue of whether or not the procedure type mechanism is sufficient in this regard or whether a package type is desirable is an interesting question but is beyond the scope of this paper.

6.1 Example: Quick Sort

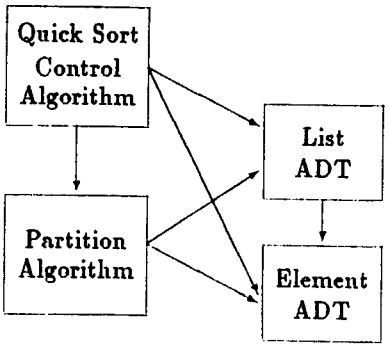
The approach we use to design an integrated collection of components is based on the *separation of concerns* principle [Hest 81]. Essentially a system is designed as a collection of components, each with a particular concern implemented as *content*.

When engineering for reuse, our goal is to define a collection of components, each of which has content that is in a sense *minimal*. The content should be an implementation of one concern and one concern only. This is not always easy to achieve, nor is it always desirable for efficiency reasons.

Consider the design and implementation of a set of components used to implement a quick sort algorithm. We begin by separating the algorithm design into four concerns:

1. A base element abstract data type that supplies a total ordering operator.
2. A list abstract data type providing "low-level" operations on a list to be sorted.
3. A partition algorithm to partition a list into a lower and upper part based upon the base element total ordering operator.
4. A quick sort algorithm to recursively invoke the partition algorithm on a list until it is completely ordered.

A diagram of the dependencies between these four concerns is presented in figure 3.



Quick Sort Algorithm Concerns
figure 3

At this point we compare an implementation using Ada generics entirely with an implementation using a combination of Ada generics and procedure types. Since we are primarily concerned with the interfaces between the components, we omit the quick sort and partition implementations.

Note that by explicitly using an array implementation for the list concern, we establish the design decision of the array as part of the content of both the partition and quick sort algorithms. In other words, if we were to change the array implementation to a linked list implementation, the content of both the partition and quick sort algorithms would need to be changed. For a nice example of a partition algorithm in which the choice of list implementation is separated out as context, see [Muss 88]. Included in [Muss 88] is also a comprehensive example and discussion of how a generic algorithm can be verified to be correct without first being instantiated with context.

The following are the specification parts for the generic quick sort and partition algorithms.

```
generic
   type Element is private;
   type El_Array_Ind is (<>);
   type El_Array is array
        (El_Array_Ind range <>) of Element;
   with Procedure Partition
                  (Els: in out El_Array;
                   Split: out El_Array_Ind);
procedure Quick_Sort (Els:in out El_Array);

generic
   type Element is private;
   with function "<="(L,R: Element)
                              return Boolean;
   type El_Array_Ind is (<>);
   type El_Array is array
        (El_Array_Ind range <>) of Element;
procedure Partition(Els: in out El_Array;
                    Split: out El_Array_Ind);
```

An example of the use of these two generic templates appears in the following application program. Notice that. while we would like to instantiate both the quick sort and partition algorithms in one expression, the limitations of Ada generics force us to first instantiate the partition algorithm, and then use the instantiation (which we call Splitter) to instantiate the quick sort algorithm.

```
with Partition,Quick_Sort;
procedure QS_driver is
   type Int_Array is array
            (integer range <>) of Integer;
   X: Int_Array(1..100);

   procedure Splitter is new Partition
                  (Element => Integer,
                     "<=" => "<=",
```

```
                El_Array_Ind => Integer,
                    El_Array => Int_Array);

   procedure Sort is new Quick_Sort
                     (Element => Integer,
                El_Array_Ind => Integer,
                    El_Array => Int_Array,
                   Partition => Splitter);
begin
    .
   Sort(X);
    .
end QS_driver;
```

We now consider how this example changes when we use a procedure type to represent the partition parameter to the quick sort procedure.

```
generic
   type Element is private;
   type El_Array_Ind is (<>);
   type El_Array is array
        (El_Array_Ind range <>) of Element;
procedure Quick_Sort
         (Els:in out El_Array;
          Partition: procedure
                (Els: in out El_Array;
                 Split: out El_Array_Ind));

with Partition,Quick_Sort;
procedure QS_driver is
   type Int_Array is array
            (integer range <>) of Integer;
   X: Int_Array(1..100);

   procedure Splitter is new Partition
                    (Element => Integer,
                        "<=" => "<=",
                El_Array_Ind => Integer,
                    El_Array => Int_Array);

   procedure Sort is new Quick_Sort
                    (Element => Integer,
                El_Array_Ind => Integer,
                    El_Array => Int_Array);
begin
    .
   Sort(X,Splitter);
    .
end QS_driver;
```

Notice that this application is almost identical to the previous application using only generics, except for the subtle but important difference that in this case the partition algorithm context is applied at run-time rather than at compile time. This "late binding" of context allows us a greater degree of flexibility. One can envision implementing a quick sort algorithm where the choice of partition algorithm is made dynamically, based upon "sortedness" properties of the list that might change even within a single quick sort execution.

An interesting issue arises when combining generics with procedure types. The quick sort example suggests the notion of a generic procedure type such as:

```
generic
   type Element is private;
   type El_Array_Ind is (<>);
   type El_Array is array
        (El_Array_Ind range <>) of Element;
type Partition_Type is procedure
                  (Els: in out El_Array;
                   Split: out El_Array_Ind);
```

which would then be used as the type definition for the procedure parameter to the quick sort algorithm:

```
generic
   type Element is private;
   type El_Array_Ind is (<>);
   type El_Array is array
        (El_Array_Ind range <>) of Element;
procedure Quick_Sort
         (Els:in out El_Array;
          Partition: Partition_Type);
```

Important here is that the actual/formal procedure parameter correspondence is between the instantiations of the respective generics.

7 Procedure Types and ADTs

In this section we explore how procedure types can be used to represent abstract data types (for a similar approach, see [Atki 85]). We first consider abstractions that encapsulate a single object, followed by abstract data types, i.e., modules that provide a type definition along with operations on that type.

At first glance it would seem straightforward to construct abstract data types using records and procedure types. Unfortunately a scoping problem arises. That is, the collection of procedure types used to construct an abstract data type record must all have access in their internal structure to the global data that is the abstract data type representation. This can be done when a single instance of an object

is encapsulated in a package, but additional language facility is needed to export a type definition along with its operations.

7.1 Example: A Stack Module Type

Consider the following example of a record representing a stack data abstraction:

```
type Stack_Object is record
   push: procedure(I: in Item),
   pop: procedure(I: out Item),
   empty: function return Boolean;
   full: function return Boolean;
end record;
```

Suppose we now have two separate implementations of a stack package, one using arrays, which we shall call *Bounded_Stack*, and one using linked lists, which we shall call *Unbounded_Stack*. Assume for simplicity that these packages each encapsulate one stack of the data type Item, and that the operation names are the same as those in the above Stack_Object record.

Suppose we now create an application Calculator needing a stack in its implementation and taking the Stack_Object record as a parameter.

```
procedure Calculator
                    (Expression: in String;
                     Result: out Integer;
                     S: in Stack_Object);
```

After declaring an object of type Stack_Object:

```
Stack: Stack_Object;
```

one can bind Calculator to the two different implementations of Stack_Object in successive executions as follows:

```
Stack.push  := Bounded_Stack.push;
Stack.pop   := Bounded_Stack.pop;
Stack.empty := Bounded_Stack.empty;
Stack.full  := Bounded_Stack.full;
Calculator(E,R,Stack);

Calculator(E,R,  (Unbounded_Stack.push,
                  Unbounded_Stack.pop,
                  Unbounded_Stack.empty,
                  Unbounded_Stack.full));
```

Notice that the first invocation of Calculator uses a record "Stack" to hold the binding to the Bounded Stack package whereas the second invocation of Calculator uses a record aggregate to bind Unbounded Stack to the formal parameter. Both methods achieve the same effect.

7.2 A Stack ADT Type

Now suppose we want to import a private type Stack in both a Bounded_Stacks package and the Unbounded_Stacks package. Before dealing with this example, consider again Ada type equivalence rules. We have already discussed the usefulness of structural equivalence as the basis for procedure type equivalence in Ada. Unfortunately, the name equivalence rules for existing type resolution in Ada "get in the way" of the flexibility provided by this structural equivalence.

Even though structural equivalence allows us to "plug in" any procedure with the "same" interface as that in the procedure type definition, "sameness" is defined by parameter correspondence using name equivalence rules. For example, consider again the Set package predicate function. Since this function depends upon the generic instantiation of Item, any procedure of this type must "see" Item in the scope of its definition.

Consider now how we package the Item concern, which in the example is the type Person. More than likely we will have packaged the Person type with its operations, including Short and Tall, making it available as a private type. Of course, Short and Tall needn't necessarily know the representation of Person (as they do in their implementations in this paper), but they do need to be aware of the Person package in order to have access to the type definition for Person.

Generics give us a clue to how to handle this issue. Note that, in general, generic actual/formal parameter correspondence is based on structural equivalence, not name equivalence. This is true not only for subprogram parameters but also for object and type parameters. We've seen that the difference between generic subprogram parameters and procedure types is essentially a binding time difference. In the same way we should be able to treat object and type parameters also as variables to be manipulated at run time.

With this notion of a "type of types", we look again at the Stack example. We can extend the Stack_Object record type to include an Item type "variable", which holds not an object of a type, but a type itself.

```
type Stack_ADT is record
   Stack: type is private;
   new: function return Stack;
   push: procedure(S: in out  Stack;
                   I: in Item),
   pop: procedure(S: in out Stack;
                  I: out Item),
```

```
   empty: function(S: Stack)return Boolean;
   full: function(S: Stack) return Boolean;
end record;
```

We can create a Stack and assign values to it as follows:

```
Stacks: Stack_ADT;

Stacks.Stack := Bounded_Stacks.Stack;
Stacks.new:= Bounded_Stacks.new;
Stacks.push := Bounded_Stacks.push;
Stacks.pop:= Bounded_Stacks.pop;
Stacks.empty := Bounded_Stacks.empty;
Stacks.full := Bounded_Stacks.full;
```

Although this facility gives us the flexibility to pass abstract data types as parameters and assign them as variables at run-time, it raises a number of concerns that we are currently exploring. These include:

1. **Safety**: there is no language support for "proper" assignment of operations to an ADT record.

2. **ADT semantics**: "improper" binding of type to operations can cause unforseen behavior and run-time errors.

3. **Efficiency**: binding each operator of an ADT individually might cause unnecessary overhead at run-time.

One solution would be to use the generic parameter structural equivalence rules as the basis for a *package type* [Kapu 89]. Specifically, our procedure types and "type" types could be "bundled" into a package type, allowing us to address some of the issues raised above.

8 Implementation Issues

In this section we briefly discuss implementation issues related to procedure types. These include closure implementation using static chains vs. displays, efficiency considerations, and practical implementations of procedure types in block-structured languages.

Recall the discussion of closures in the section on language issues. In a static chain implementation, closures are typically represented as an [entry point,environment] pair of pointers [Fish 88,Seth 89]. Formal procedure call are quite efficient since procedure calls are simply indirect calls using these pointers. Displays on the other hand can lead to a fairly complex implementation. This is due to each procedure usually requiring a copy of the entire display active for that procedure.

Procedure variables are rarely used because of perceived inefficiencies, when in fact some Ada compilers already have efficient mechanisms used to implement code-sharing generics. Hardware support for static scoping is of course also helpful.

Some language implementors have elected, because of efficiency concerns, to allow only subprograms declared in the outermost scope to be passed as parameters or assigned to variables [Wirt 83]. This guarantees that "dangling subprograms" do not occur, and greatly simplifies the construction of closures.

9 Summary and Research Directions

We have presented a number of examples which we hope will convince readers of the usefulness of procedure types. In particular, the abstraction and parameterization of control is a common practice, and needs language support. We have also suggested that procedure types can be used to construct abstract data type "variables" that can be used to delay the binding of ADT implementation decision until run-time, allowing for a fairly flexible style of programming.

Procedure types can also provide support for the following language features, and further research into these areas is needed.

1. Distributed systems (remote procedure calls)
2. Object oriented programming
3. Persistent Systems
4. Dynamic linking
5. Heap based languages
6. Package Types

References

[Ada 83] U.S. Dept. of Defense [1983]. *The Ada Language Reference Manual*, ANSI/MIL-STD-1815A.

[Atki 85] Atkinson, M.P., and Morrison, R. [1985] Procedures as Persistent Data Objects. *ACM Transactions on Programming Languages and Systems* **7**:4,539-559.

[Booc 87] Booch, G. [1987] *Software Components with Ada.* Benjamin/Cummings Publishing Co.

[Card 85] Cardelli, L., and Wegner P., On Understanding types, data abstraction, and polymorphism. *ACM Computing Surveys* **17:4**, 471-522.

[Fish 88] Fisher, C.N., and LeBlanc, R,J. [1988]. *Crafting a Compiler.* The Benjamin/Cummings Publishing Company, In., Menlo Park, Cal.

[Gold 83] Goldberg, A., and Robson, D. [1983]. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass.

[Harl 84] Harland, D.M. [1984]. *Polymorphic Programming Languages.* Ellis Horwood Ltd., Chichester, England.

[Harp 86] Harper, R., Milner, R., and Tofte, M. [1988] The definition of Standard ML, Version 2. ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, Univ. of Edinburgh.

[Hest 81] Hester, S.D., Parnas, D.L., and Utter, D.F. [1981]. Using Documentation as a Software Design Medium. *The Bell System Technical Journal*, **6:8**.

[Hilf 82] Hilfinger, P.N., [1982]. *Abstraction Mechanisms and Language Design.* MIT Press, Cambridge, Mass.

[Jone 89] Jones, O. [1989]. *Introduction to the X Window System.* Prentice-Hall, Inc.

[Kapu 89] Kapur, D., Musser, D., Olthoff, W., Snyder, A., Stepanov, A., Szymanski, A. [1989]. A Prototyping Language for Rapid Reuse: Technical Proposal. Software Technology Laboratory, Hewlett-Packard Laboratories, Report STL-89-10.

[Kern 88] Kernighan, B.W., and Ritchie, D.M. [1988] *The C Programming Language*, 2nd Ed. Prentice-Hall, Englewood Cliffs, N.J.

[Kers 88] Kershenbaum, A., Musser, D.R., Stepanov, A.A. [1988]. Higher Order Imperative Programming. Computer Science Department Rep No. 88-10, Rensselaer Polytechnic Institute, Troy, N.Y.

[Lamb 83] Lamb, D.A., and Hilfinger, P.N. [1983] Simulation of Procedure Variables Using Ada Tasks. *IEEE Transactions on Software Engineering*, **SE-9,1**, 13-15.

[Lisk 86] Liskov, B., and Guttag, J. [1986]. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, Mass.

[McCa 65] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., and Levin, M.I. [1965]. *Lisp 1.5 Programmer's Manual*, 2nd Ed. MIT Press, Cambridge, Mass.

[Muss 88] Musser, D.R., and Stepanov, A.A. [1988]. Generic Programming. Proceedings of ISSAC-88 and AAEC-6, Rome, Italy.

[Naur 63] Naur, P. [1963] Revised report on the algorithmic language Algol 60. *Comm. ACM* **6:1**, 1-17.

[Reyn 81] Reynolds, John C. [1981] *The Craft of Programming*, Prentice-Hall, N.J.

[SEI 89] Implementation working group report [1989]. *Reuse In Practice Workshop.* Software Engineering Institute, Pitt, Pa.

[Seth 89] Sethi, R. [1989]. *Programming Languages: Concepts and Constructs.* The Addison-Wesley Publishing Co., Reading, Mass.

[vanW 75] van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G.L.T., and Fisker, R.G. [1975]. Revised report on the algorithmic language Algol 68. *Acta Informatica* **5**, 1-236.

[Whee 84] Wheeler, T. [1984]. A Command Interpretor for Ada. *Ada Letters*, **3:4**.

[Wirt 83] Wirth, N. [1983]. *Programming in Modula-2*, 2nd, corrected Ed. Springer-Verlag, New York. 3rd corrected Ed. (1985).



Larry Latour received the B.B.A. degree in Statistics from Baruch College/CUNY in 1973, the M.S. degree in Operations Research from Polytechnic Institute of New York in 1978, and the Ph.D. degree in Computer Science from Stevens Institute of Technology in 1985. He is an Assistant Professor of Computer Science at The University of Maine, Orono, Me., 04469. His research interests include database transaction systems, software engineering environments, and reuse engineering.



Curtis Meadow received the B.A. degree in Computer Science from the University of Maine in 1988, and is currently pursuing in M.S. degree in Computer Science from that same institution. His research interests include object oriented software development, programming languages and compilers, and reuse engineering.

Ada Revision Issues in Concurrent Programming

Joseph L. Linn and Cy D. Ardoin

Institute for Defense Analyses
Alexandria, Virginia 22311

Abstract

The Ada 9X Project is responsible for revising the current Ada programming language, ANSI/MIL-STD-1815A. During the revision process, the Ada 9X Project has gathered a large number of revision proposals from many different sources. One of the main issues raised in these revision proposals is the lack of support for concurrent programming in the current Ada standard. More than one-hundred revision proposals have been submitted on this topic alone. The purpose of this paper is to overview the breadth of revision issues concerned with concurrent programming and provide a technical discussion of how Ada could be improved to more nearly realize its goals with respect to only on of these issues, namely monitors and nested monitors.

Introduction

The Ada programming language was developed by the Department of Defense (DOD) with the major intent of enabling the use of effective software development methods for producing embedded systems applications. Indeed, the DOD has determined that Ada embodies appropriate software development mechanisms so well that Ada is now strongly recommended for essentially all DOD software projects.

Nevertheless, there are a relatively small number of minor problems in the language that compromise this intended goal; these problems are the (almost inevitable) result of the process by which Ada was developed. Specifically, Ada came into being as the result of the successive refinement of initial requirements culminating in a competition from which the final selection emerged. The refinement of requirements and the subsequent language design process required numerous individual technical decisions. Further, these decisions were based almost exclusively on analysis and experience without practical experimental feedback. Subsequent experience with the language now affords the opportunity to observe that slightly different low-level decisions would significantly improve the usefulness and safety of the language in certain domains.

Substantial momentum is growing in support of Ada. However, the intense passion in the Ada community frequently hinders dispassionate technical discussion[1]. That is the goal of this paper–examine the breadth of revision issues concerned with concurrent programming and provide a technical discussion of how Ada could be improved to more nearly realize its goal with respect to only one of these issues, namely monitors and nested monitors. The discussion of monitors does not represent official changes that will be made to the language nor does it represent official requirements. The discussion is presented to enlighten the reader.

1.1 Revision

Currently, the Ada 9X Requirements Team has over one-hundred request concerning concurrent processing and Ada tasking. The requests are received from several sources: 1) Ada Issues (AIs) submitted to the International Standards Organization, Working Group 9 (ISO/WG9), 2) the Ada 9X Destin workshop [Workshop 1989]), 3) special studies sponsored by the Ada 9X Project Office, and 4) individuals submitting requests to the Ada 9X Project Office.

Of particular interest to this discussion are the revision proposals concerning tasking, scheduling, and queuing. To begin with, there are more than one-hundred revision requests dealing with various aspects of concurrent programming. Several dozen of these proposals are concerned with the complexity, inefficiency, and inappropriateness of Ada tasking for different problem areas, such as mutual exclusion and semaphores, dispatching processes, monitors, asynchronous communications, and interrupt processing. The basic problem in all of these cases is that the design of the programmer is not easily and efficiently mapped into the Ada programming language.

This work was sponsored by the Strategic Defense Initiative Organization and the Ada 9X Project Office.

[1] In fact, those readers introduced to the program structuring paradigms discussed in this paper by the Ada literature are strongly cautioned to read the definitions (and even the citations) carefully as there is often considerable variance of a concept as embodied in the Ada literature and the same concept in "general computer science". Consequently, benefits demonstrated or claimed for the *concept-in-general* often do not apply to the *concept-in-Ada*.

For example, two revision requests submitted to the Ada 9X Project Office specifically ask for semaphores, [RR-0185, RR-0461]. The complaint in both cases is the cost incurred by implementing mutual exclusion with tasks and rendezvous. These proposals also point out the fact that some vendors are providing interfaces to semaphores; however, the vendors' solutions are not portable across implementations, and some vendors are providing optimizations to recognize tasks that are only used as semaphores.

Another large group of issues concerns scheduling and queuing. In this group of issues the users would like control over the scheduling and queuing decision currently made by the Ada run-time environment. Once again, the problem is that the design of the programmer is not easily and efficiently mapped into the Ada programming language. Excellent examples of these requests are given in [RR-0016, RR-0379, and RR-0170]. All three requests ask for control over the scheduling decisions. This particular issue was also raised by the Parallel and Distributed Systems Working Group at the Destin Workshop [Workshop 1989, pp 68-69].

Finally, there is a significant group of issues concerning parallel and distributed systems. In this case, the users are concerned with the general lack of language support for partitioning programs, modeling communications, heterogeneous architectures, and the semantics of tasking operations. Current projects are both required to use Ada and to implement a distributed or parallel processing architecture. Since this is the current state of embedded systems the programming language should provide some useful support for this environment. This was the primary concern of the Parallel and Distributed Systems Working Group at the Destin Workshop and of some concern to the Real-Time Embedded Systems Group [Workshop 1989, Sections 4 and 5]. There are several impediments to distributed programming in Ada. A few of the problems cited in revision requests include package SYSTEM, package STANDARD, STORAGE_UNIT, CLOCK, DURATION, timed and conditional entry calls, shared memory, and volatile memory [RR-0071; RR-0109; Workshop 1989, Section 5; Workshop 1989, pp 54-55]. All of these issues are being examined by the Requirements Team and more studies are likely before the final requirements are derived.

1.2 Constraints

In this paper, we are examining only one of the problems facing concurrent programmers. In our examination, we are looking for ways to map programs designed for concurrent operation into Ada; the goal is to find mappings that are efficient **and** portable **and then** highly safe without resorting to compiler optimization techniques that remove programmer-specified synchronization points. Within this constraint, a secondary goal is that the fewest number of new mechanisms be introduced into the language. Each of these concepts deserves further clarification.

The first constraint requires the separation of the concept of a program design from the concept of encoding that program in a programming language. Ideally, program designs are language independent. As an example, consider the development of an Ada program intended to implement a simple translator based on a finite state machine (FSM). One does not design the state machine in terms of its Ada representation; rather, the Ada code representing the FSM is a result of mapping the design into Ada. Given a design in terms of a state machine, several different Ada images are possible.

If this is an important way of designing programs and the programs are to be realized in Ada, then Ada should be designed so that at least one possible mapping produces code that is portable and efficient. This is the essence of language design, to ensure that the language supports those paradigms for which it is intended so that the resulted code has the desired properties. According to [STEELMAN 1978], the desired properties for Ada are

1) Generality only to the extent necessary for embedded systems,
2) Reliability,
3) Maintainability,
4) Efficiency,
5) Simplicity,
6) Implementability,
7) Machine Independence, and
8) Complete Definition.

Thus, the goals sought for the mappings of the target problem for this study are in a large sense fundamental to the design of the language itself.

"Without resorting to optimization techniques...", means that efficient object code should not require a compiler that performs complex recognition and restructuring operations such as *passivation* or *idiom-substitution* as presented in [Haberman and Nassi 1980] and [Hilfinger 1982]. While these optimization techniques may be crucial in allowing some Ada programs to achieve efficient operation, there should be an inherent efficiency in Ada programs produced by mapping from important design paradigms.

1.3 Proposed Changes

The proposed changes are presented here without any explanation. There are two purposes for this. The first is to show how few changes are being proposed and the (arguably low) complexity of these changes. The second is that many readers familiar with programming languages in general and with Ada in particular are aware of the arguments to be advanced without further elaboration. For others, if the nature of the proposed change is not clear from the description in this section, then the reader should refer to Chapter 2 where the changes are given in greater detail. The two changes listed are intended to support monitors and nested monitors.

1) that a mechanism be added for dynamically creating private entry queues and that the accept side of the rendezvous mechanism be generalized to permit requeuing of entry calls after the input parameters have been read.
2) that the point-of-declaration discrimination be extended (a) to allow all static-sized types as discriminant types and (b) to apply to task types as well as record types.

1.4 Reasons for Omission

One may reasonably ask why these facilities are not already present in the language. The following attempts to answer this question; however, some of the rationale is subjective and anecdotal.

The Ada mechanisms for concurrency suffer from immaturity; Ada took a bold step here in adopting a concurrency model that was roughly contemporary with the Ada requirements/design process. In retrospect, it seems that the basic model is tenable but that a programmer needs more control of the service order than can be achieved at reasonable cost by the current facilities. The immaturity of these facilities produced a very weak form of task type parameterization. The designers obviously felt that it was far better to err on the conservative side [Ichbiah et al. 1979, Section 4.6].

In summary, the major reason that these changes are not included in the original language is that (1) the designers felt that their inclusion was too dangerous to justify by benefits, (2) the designers did not have the benefit of actually applying Ada to a large, structurally complex problem, (3) the designers did not have the benefit of hindsight, or (4) some combination of the above, probably heavily weighted towards (3). In the paper, the benefits and costs of the proposed changes are presented so that the reader can judge whether the changes are justified.

Representing Monitors within an Ada Context

The synchronization facilities of Ada are patterned after the work of Hoare called "Communicating Sequential Processes" (CSP) [Hoare 1978]. However, there are a number of problems that are more easily structured using Hoare's earlier "monitor" concept [Hoare 1974]. A survey discussion of the issues of concurrent programming in Ada may be found in [Burns 1985]; [Hoare 1985] is an excellent discussion of concurrent programming in general. When monitors are used as the high-level programming style for concurrency control, the Ada synchronization primitives are somewhat lacking with respect to efficiency. The discussion here follows [Silbershatz 1984] except that the precise problem solution is different because the solution proposed by Silbershatz requires the addition of a new and rather different synchronization mechanism whereas the solution proposed here utilizes existing execution-time mechanisms. [Gehani and Roome 1988] contains a similar discussion of the problem and a somewhat different solution that has been defined for Concurrent C.

The problem, in general, is that monitors are generally used to delay clients using some priority scheme where the priority is calculated based on the exact client request. This definition of a monitor is rather different from the one that one finds in the Ada literature where monitors simply service clients in a FIFO order but is entirely in accord with [Hoare 1974] and [Hoare 1985]. The key difference is that a monitor must generally be able to control the order in which clients are served based on information that the client provides at the time of call. For example, a disk scheduler implemented as a monitor would determine the service order based of which requests refer to cylinders close to the current position of the disk heads; thus, the "priority" of a client changes as the head moves. (This is among several examples with built-in scheduling that appear in [Hoare 1974].) One may notice the similarity of this problem to other problems in which a task may need to determine service order based on some shared notion of priority.

Another example is the one discussed in this chapter, a monitor that implements a scheduling discipline called "shortest-hold-time." The scheduler allocates the resource (i.e., allows a client to continue from (possible) suspension) to the client that promises to hold the resource for the shortest duration; this duration is called the *hold time* for the request. We will assume that the hold times are are drawn from some *duration* type and that cardinality of *duration* is at least as large as *integer*. Further, the promised hold time is dynamic, that is, a client potentially computes a different hold time for each call. Since, in Ada, a client can only be delayed on entry to the scheduler, the computation of the priority and the delaying of the client must be divided into separate steps. This essentially doubles the overhead of using the monitor. In very Ada-specific terms, the problem is (1) that a calling task can only be held at the entry to a rendezvous, (2) that in the situation under discussion the scheduler cannot know whether to suspend the client until the scheduler has discovered the client's priority by reading the parameters to the entry call and (3) that the scheduler cannot discover the client's priority (i.e., read the parameters of the entry call) without accepting the rendezvous. Thus, two rendezvous are required, one for transmitting the client's priority via the entry parameters and a second where the client is delayed if necessary.

There are, of course, sophisticated optimization techniques by which a compiler can generate efficient object code by using complex recognition and restructuring operations such as *passivation* or *idiom-substitution* as presented in [Haberman and Nassi 1980] and [Hilfinger 1982]. While these optimization techniques may be crucial in allowing some Ada programs to achieve efficient operation, one might hope that the language would be designed to achieve an inherent

efficiency without resorting to intermodule optimization techniques. Here, we assume that intermodule optimization is not performed.

2.1 Simple Monitors

Assume then that we wish to define a task whose function is to implement shortest-hold-time scheduling. Again, the hold time is supplied as a parameter to acquire the resource; in general, the parameter represents any priority with small values indicated higher priority. This may be solved as shown in Figure 2-1 and Figure 2-2[2]. The scheduler operates as follows. When a task tries to acquire the resource, the scheduler obtains a free entry from the set of *Wait_to_acquire* entries which is then returned to the calling task. The calling task then enqueues on that entry and waits to acquire the resource. Meanwhile, the scheduler has obtained the requested duration and records this information in a priority queue. When the calling task is the waiting task with the shortest hold time and the resource is free, the scheduler accepts on the entry where the calling task is waiting and the task thereby is released from suspension. Actually, this package only approximates shortest-hold-time scheduling; the approximation is exact whenever no more than *N* processes are trying to acquire the resource at the same time.

Silbershatz objects to this solution on two grounds (besides readability): (1) it "artificially" divides the acquiring of a resource into two parts, and (2) it is very inefficient because it requires two rendezvous. Since one might easily argue that it is not artificial to separate the computation of the priority from the actual acquisition of the resource, our concern here is on the efficiency issue. Doubling the overhead of the monitor could be a great concern in an execution environment either where interprocess communication times are long or where task context switching times are long. The former situation is prevalant in non-shared memory systems; however, this is not the focus in this paper. However, the latter situation, long context-switching times, is quite common with current Ada execution environments. Thus, the initial goal is to remove one of the rendezvous.

A mechanism that solves this problem is to allow the handler for the first rendezvous to enter the second rendezvous without returning control to the caller, i.e. the caller continues to be delayed until the end of the second rendezvous. Thus, the accept handler for one entry would be able to requeue the caller on another local entry where the second entry has the same parameter profile as the original. The handler for the original accept would be able to perform this requeuing operations after first reading any in parameters to the entry call. This solution is presented in Figure 2-3 and Figure 2-4. There is not a great deal of difference in the two codes. The difference is one of efficiency; to

[2] We assume the existence of a library package providing any common data structures that we may need.

```
package Shortest_Hold_Time_1 is
    procedure Acquire(t: duration);
    procedure Release;
    pragma inline(Acquire, Release);
end Shortest_Hold_Time_1;

package body Shortest_Hold_Time_1 is
    N: constant:= 100;
    type sch_entry is new integer range 1..N;

    task scheduler_1 is
        entry notify(t: duration; j: out sch_entry);
        entry wait_to_acquire(sch_entry);
        entry release;
    end scheduler_1;

    procedure Acquire(t: duration) is
        j: sch_entry;
    begin
        scheduler_1.notify(t, j);
        scheduler_1.wait_to_acquire(j);
    end Acquire;

    procedure Release is
    begin
        scheduler_1.release;
    end Release;

    task body scheduler_1 is separate;
end Shortest_Hold_Time_1;
```

Figure 2-1. Package Structure for a Monitor in Ada.

```
with Basic_Data_Structures; use Basic_Data_Structures;
separate(Shortest_Hold_Time_1)
task body scheduler_1 is
    package PQ is new priority_queue(Sort_type => Duration,
      Data_type => sch_entry);
    package Unused_Slots is new set(Data_type => sch_entry);
    Free: Boolean:= true;
    my_j: sch_entry;
begin
    for i in sch_entry'first..sch_entry'last loop
        Unused_Slots.add_element(i);
    end loop;
    loop
        select
            when Unused_Slots.not_empty =>
                accept notify(t: duration; j: out sch_entry) do
                    my_j:= Unused_Slots.remove_arbitrary_element;
                    PQ.enter_into_queue(t, my_j);
                    j:= my_j;
                end notify;
        or
            when Free and PQ.not_empty =>
                accept wait_to_acquire(PQ.first_element.data) do
                    Free:= false;
                    Unused_Slots.add_element(PQ.first_element.data);
                    PQ.remove_first_element;
                end wait_to_acquire;
        or
            when not Free =>
                accept release do
                    Free:= true;
                end release;
        end select;
    end loop;
end scheduler_1;
```

Figure 2-2. Scheduler Body for a Monitor in Ada.

```
package Shortest_Hold_Time_2 is
    procedure Acquire(t: duration);
    procedure Release;
    pragma inline(Acquire, Release);
end Shortest_Hold_Time_2;

package body Shortest_Hold_Time_2 is
    N: constant:= 100;
    type sch_entry is new integer range 1..N;

    task scheduler_2 is
        entry acquire(t: duration);
        entry release;
        entry wait_to_acquire(sch_entry) (t: duration);
    end scheduler_2;

    procedure Acquire(t: duration) is
    begin
        scheduler_2.acquire(t);
    end Acquire;

    procedure Release is
    begin
        scheduler_2.release;
    end Release;

    task body scheduler_2 is separate;
end Shortest_Hold_Time_2;
```

Figure 2-3
Package Structure for a Monitor using Requeuing

repeat, the first solution requires two rendezvous between the caller and the scheduling task whereas the second solution only requires one rendezvous plus one requeuing. Since the requeuing does not result in a context switch, a requeuing should be somewhat more efficient than an extra rendezvous. Actually, at least two context switches are saved. Since context switching is normally a rather expensive operation, the savings may be considerable.

This proposal does not introduce additional mechanisms into the language since the execution-time support for the requeuing operation must exist in virtually all implementations since it is essentially identical to the operation of enqueuing external calls prior to reaching an accept statement. The additional compilation requirements are small: (1) the recognition and processing of another type of statement (the **exit-accept-with-entry** statement), and (2) a context check to determine if a requeuing operation is legal. The requeuing statement is legal only if it occurs within an accept statement and the target entry has the same parameter profile as the entry of the enclosing accept. Last, the execution of a requeuing statement automatically completes the execution of the accept statement, i.e., in

```
with Basic_Data_Structures; use Basic_Data_Structures;
separate(Shortest_Hold_Time_2)
task body scheduler_2 is
    package PQ is new priority_queue(Sort_type => Duration,
      Data_type => sch_entry);
    package Unused_Slots is new set(Data_type => sch_entry);
    Free: Boolean:= true;
    j: sch_entry;
begin
    for i in sch_entry'first..sch_entry'last loop
        Unused_Slots.add_element(i);
    end loop;
    loop
        select
            when Unused_Slots.not_empty =>
                accept acquire(t: duration) do
                    j:= Unused_Slots.remove_arbitrary_element;
                    PQ.enter_into_queue(t, j);
                    exit accept with entry wait_to_acquire(Next);
                end acquire;
        or
            when Free and PQ.not_empty =>
                accept wait_to_acquire(PQ.first_element.data) (t: duration)
                    do
                    Free:= false;
                    Unused_Slots.add_element(PQ.first_element.data);
                    PQ.remove_first_element;
                end wait_to_acquire;
        or
            when not Free =>
                accept release do
                    Free:= true;
                end release;
        end select;
    end loop;
end scheduler_2;
```

Figure 2-4
Scheduler Body for a Monitor using Requeuing

```
accept A do
  <statement-1>
  if <Boolean-expression> then
    <statement-3>;
    exit accept with entry B;
  end if;
  <statement-2>;
end;
```

<statement-2> will not be executed if the <Boolean-expression> is true.

Actually, a second language change should be considered: that a task body have private entries. Obviously, this has no utility without the requeuing mechanism. However, the ability to have private entries eliminates the need to define procedures to hide the internal details of the scheduler as was done in the first case. Also, the solution is much safer since it eliminates any possibility that a procedure internal to *Shortest_Hold_Time* but external to the scheduler will interfere with scheduler's use of the family of entries. Further, this solution seems to be clearly within the existing run-time mechanisms. Using private entries, the structure of the solution might be changed as in Figure 2-5. Still, this solution is not without difficulty. The problem is that the maximum number of clients in the monitor has to be selected a priori. The only solution to this problem is to allow a program to create a dynamic number of entries. The standard way of conquering this problem in Ada is to declare an extra task for each task to be controlled in order to create an entry for each client. The solution is shown in Figure 2-6 and Figure 2-7.

This monitor *Shortest_Hold_Time_4* actually implements shortest-hold-time scheduling for an arbitrary number of clients. But, there is tremendous overhead; now three rendezvous are needed and the monitor creates tasks dynamically just to get a dynamic number of entries[3]. A far better solution would be obtained if there were a mechanism for creating entries dynamically. Such a dynamically created entry would be patterned after some existing local entry. That is, dynamically created entries would be typed in the same way as procedure types. Using this new idea, the final solution emerges, as depicted in Figure 2-8 and Figure 2-9.

[3] Actually, the number of entries per task can be significantly increased so that far fewer tasks are needed. However, the overhead to create the tasks is still present and the number of rendezvous cannot be reduced without intermodule optimization.

```
package Shortest_Hold_Time_3 is
    task scheduler_3 is
        entry Acquire(t: duration);
        entry Release;
    end scheduler_3;
end Shortest_Hold_Time_3;

package body Shortest_Hold_Time_3 is
    N: constant:= 100;
    type sch_entry is new integer range 1..N;
    task body scheduler_3 is separate;
end Shortest_Hold_Time_3;

with Basic_Data_Structures; use Basic_Data_Structures;
separate(Shortest_Hold_Time_3)
task body scheduler_3 is
    entry wait_to_acquire(sch_entry)(t:duration);
    package PQ is new priority_queue(Sort_type => Duration,
      Data_type => sch_entry);
    package Unused_Slots is new set(Data_type => sch_entry);
        .
        .
        .
begin
        .
        .
        .
end scheduler_3;
```

Figure 2-5
A Monitor using a Mechanism for Private Entries.

2.2 Nested Monitors

In [Hoare 1985, 230], Hoare identifies a potential safety problem with simple monitors[4]. The problem is that simple monitors place upon the programmer the burden not only to finish using the "monitored" resources within appropriate time bounds but also to free the resources. If Ada supported access objects for procedures and functions[5], then a solution to this safety problem is for a client to encapsulate the work to be accomplished in a procedure and to pass an access value for the procedure to the monitor; the monitor then assumes the responsibility for acquiring and freeing the resource. Hoare's term for this is a *nested monitor*.

2.3 Parameterization of Tasks

There is a second problem with Ada versus Concurrent C reported in [Gehani and Roome 1988]: there is no point-of-initialization parameterization for task types. A normal use for such a facility occurs when a task is part of a larger structure; for example, one might desire to group a task and a buffer together into a record and to manipulate them as a unit. Figure 2-10 depicts such a situation; the problem is to inform each of the tasks of the of the identity of its

[4] One is tempted to speculate that this is the problem that has led some language designers to conclude that using monitors as a concurrent programming paradigm is less safe than using CSP.

[5] There are numerous Revision Requests for such a capability; also, two separate groups submitted this idea at the Ada 9X Project Requirments Workshop.

```
package Shortest_Hold_Time_4 is
     procedure Acquire(t: duration);
     procedure Release;
     pragma inline(Acquire, Release);
end Shortest_Hold_Time_4;

with UNCHECKED_DEALLOCATION;
package body Shortest_Hold_Time_4 is
     task type sleeper_task is
          entry scheduler_signal;
          entry client_signal;
     end sleeper_task;
     type sleeper_type is access sleeper_task;
     function new_sleeper_type return sleeper_type;
     procedure free_sleeper_type is new
       UNCHECKED_DEALLOCATION(sleeper_task,sleeper_type);

     task scheduler_4 is
          entry notify(t: duration; sleeper: out sleeper_type);
          entry release;
     end scheduler_4;

     procedure Acquire(t: duration) is
          sleeper: sleeper_type;
     begin
          scheduler_4.notify(t, sleeper);
          sleeper.client_signal;
     end Acquire;

     procedure Release is
     begin
          scheduler_4.release;
     end Release;

     task body sleeper_task is
     begin
          loop
               accept scheduler_signal;
               accept client_signal;
          end loop;
     end sleeper_task;
     function new_sleeper_type return sleeper_type is
     begin
          return new sleeper_task;
     end new_sleeper_type;

     task body scheduler_4 is separate;
end Shortest_Hold_Time_4;
```

Figure 2-6
Package Structure using the Three-Rendezvous Method.

enclosing structure so that it can address the correct buffer. The normal method to achieve parameterization (as is pointed out in [Gehani and Roome 1988]) is to pass in the parameterization in an initial rendezvous. However, this solution does not provide general, point-of-declaration initialization and is therefore somewhat error-prone. Further, tricks that work for nontask types such as initializing dummy variables using functions that perform the intended initialization by side-effect do not work for the initial rendezvous method. The reason is that tasks declared in a declarative part and not dynamically allocated do not activate until the corresponding body. Thus, it is impossible for a point-of-declaration initialization procedure to rendezvous with such a task; such a rendezvous if attempted would result in deadlock.

Fortunately, the discriminant mechanism already present in the language would provide an appropriate

```
with Basic_Data_Structures; use Basic_Data_Structures;
separate(Shortest_Hold_Time_4)
task body scheduler_4 is
     package PQ is new priority_queue(Sort_type => Duration,
       Data_type => sleeper_type);
     Free: Boolean:= true;
     Current_Sleeper: sleeper_type;
begin
     loop
          select
               accept notify(t: duration; sleeper: out sleeper_type) do
                    PQ.enter_into_queue(t, new_sleeper_type);
               end notify;
          or
               when not Free =>
                    accept release do
                         Free:= true;
                         free_sleeper_type(Current_Sleeper);
                    end release;
          end select;
          if Free and PQ.not_empty then
               Free:= false;
               Current_Sleeper:= PQ.first_element.data;
               PQ.remove_first_element;
               Current_Sleeper.scheduler_signal;
          end if;
     end loop;
end scheduler_4;
```

Figure 2-7
Scheduler Body using the Three-Rendezvous Method.

```
package Shortest_Hold_Time_5 is
     procedure Acquire(t: duration);
     procedure Release;
     pragma inline(Acquire, Release);
end Shortest_Hold_Time_5;

package body Shortest_Hold_Time_5 is
     task scheduler_5 is
          entry acquire(t: duration);
          entry release;
     end scheduler_5;

     procedure Acquire(t: duration) is
     begin
          scheduler_5.acquire(t);
     end Acquire;

     procedure Release is
     begin
          scheduler_5.release;
     end Release;

     task body scheduler_5 is separate;
end Shortest_Hold_Time_5;
```

Figure 2-8
Package Structure using the Proposed Mechanisms.

capability *if it is extended to task types*. In the example that follows, the discriminant mechanism extended to apply to tasks is used to pass in the structure access value; a second extension is that the construct "**record**.*bp*" is used to access the value of the *bp* field during initialization.

Figure 2-11 shows the structure using the proposed facilities.

```
with UNCHECKED_DEALLOCATION;
with Basic_Data_Structures; use Basic_Data_Structures;
separate(Shortest_Hold_Time_5)
task body scheduler_5 is
    type acquire_entry is entry acquire(t:duration);
    type acquire_entry_ptr is access acquire_entry;
    package PQ is new priority_queue(Sort_type => Duration,
      Data_type => acquire_entry_ptr);
    Free: Boolean:= true;
    Current_Sleeper: acquire_entry_ptr;
    this_Sleeper: acquire_entry_ptr;
    procedure allocate is
    begin
        Free:= false;
        Current_Sleeper:= PQ.first_element.data;
        PQ.remove_first_element;
    end allocate;
    function new_acquire_entry return acquire_entry_ptr is
    begin
        return new acquire_entry;
    end new_acquire_entry;
    procedure free_acquire_entry is
        new UNCHECKED_DEALLOCATION
            (acquire_entry,acquire_entry_ptr);
begin
    loop
        select
            accept acquire(t: duration) do
                this_sleeper:= new_acquire_entry;
                PQ.enter_into_queue(t, this_sleeper);
                exit accept with entry this_sleeper;
            end acquire;
        or
            when not Free =>
                accept release do
                    Free:= true;
                    free_acquire_entry(Current_Sleeper);
                end release;
        end select;
        if Free and PQ.not_empty then
            allocate;
            accept Current_Sleeper(t:duration);
        end if;
    end loop;
end scheduler_5;
```

Figure 2-9
Scheduler Body using the Proposed Mechanisms.

2.4 Summary of Changes

[Gehani and Roome 1988, 1549 and 1547 respectively] identify two significant ways that Concurrent C is stronger than Ada: (1) "Entry arrays must used in Ada to compensate for the fact that parameters cannot be examined prior to the call", and (2) "Unlike Concurrent C processes, Ada tasks cannot be parameterized." This is in spite of the fact that both Concurrent C and Ada are based upon rendezvous as the synchronization/communication mechanism. Here we have shown solutions to both of these problems. The requeuing facilities are as powerful as the dequeuing selection facilities of Concurrent C; in fact, they are more powerful because they allow a programmer to design custom search structures whereas it would be more difficult for the compiler to do so. Unfortunately,

```
package Buffer_n_Task is
    type buffer_type is array(0..511) of character;
    type buffer_ptr is access buffer_type;
    task type buffer_monitor is
        entry read(b: out buffer_type);
        entry write(b: buffer_type);
        entry initialize(bp: buffer_ptr);
    end buffer_monitor;

    type Buffer_n_Task;
    type Buffer_n_Task_ptr is access Buffer_n_Task;
    type Buffer_n_Task is record
        next: Buffer_n_Task_ptr:= null;
        bp: buffer_ptr:= new buffer_type;
        mon: buffer_monitor;
    end record;

    system_buffers: array(0..11) of Buffer_n_Task;
end Buffer_n_Task;

package body Buffer_n_Task is
    task body buffer_monitor is
        my_bp: buffer_ptr;
    begin
        accept initialize(bp: buffer_ptr) do
            my_bp:= bp;
        end initialize;
        loop
            select
                accept read(b: out buffer_type) do
                    b:= my_bp.all;
                end read;
                accept write(b: buffer_type) do
                    my_bp.all:= b;
                end write;
            end select;
        end loop;
    end buffer_monitor;
begin
    for i in system_buffers'first..system_buffers'last loop
        system_buffers(i).mon.initialize(system_buffers(i).bp);
    end loop;
end Buffer_n_Task;
```

Figure 2-10. A Task as Part of a Larger Structure.

the use of requeuing is also more cumbersome than the Concurrent C facilities, i.e., for simple cases the Concurrent C facilities are easier to program.[6]

In this section, the following actual changes required in the language have been discussed:

1) a new sort of limited type (in the Ada sense) - entry types.
2) the requeuing mechanism.
3) extension of discriminants to task types.
4) a construct to refer to the record being allocated during its initialization.

These changes are straightforward. The additional compiler support needed is minimal; it seems that only the following would be needed (besides scanning and parsing):

[6] The fact is that designers of Concurrent C learned from the problems in the 1983 version of Ada and the users of Concurrent C are the better for it. Hopefully, the same will be true of Ada 9X.

```
package Buffer_n_Task_2 is
    type buffer_type is array(0..511) of character;
    type buffer_ptr is access buffer_type;
    task type buffer_monitor(my_bp: buffer_ptr:= null) is
        entry read(b: out buffer_type);
        entry write(b: buffer_type);
    end buffer_monitor;

    type Buffer_n_Task;
    type Buffer_n_Task_ptr is access Buffer_n_Task;
    type Buffer_n_Task is record
        next: Buffer_n_Task_ptr:= null;
        bp: buffer_ptr:= new buffer_type;
        mon: buffer_monitor(record.bp);
    end record;

    system_buffers: array(0..11) of Buffer_n_Task;

end Buffer_n_Task_2;

package body Buffer_n_Task_2 is
    task body buffer_monitor(my_bp: buffer_ptr:= null) is
    begin
        loop
            select
                accept read(b: out buffer_type) do
                    b:= my_bp.all;
                end read;
                accept write(b: buffer_type) do
                    my_bp.all:= b;
                end write;
            end select;
        end loop;
    end buffer_monitor;
end Buffer_n_Task_2;
```

Figure 2-11
Buffer_n_Task with Parameterization/Initialization

1) support for entry types.
2) minimal changes in the handling for **accept**.
3) checking for type conformance in the requeue (**exit-accept-with**) statement.
4) checking whether any **accept** for an external-visible entry contains a requeue; this would be needed to give the correct semantics for conditional and timed entry calls. (The correct semantics is that a conditional or timed entry call to an entry that requeues should always "take the else-part".)
5) allocation of a static-sized block to hold a task discriminant and arranging for access to it. This differs from records in that the compiler will likely choose to allocate space for the task's discriminant somewhere other than on the stack of the declared task. Frequently, the parent's frame is a good place, in which case the compiler must arrange for the addressing (i.e., a sort of second static link).
6) availing the programming of an access value for a record being initialized.

An approximately three-fold improvement in efficiency (actually, more is possible given the potential elimination of extra tasks) could result from the requeuing mechanism; although a performance improvement is also available for parameterization; safety and understandability are the more compelling issues. The global allocation of entries to be served by multiple servers (in our example that there is exactly one task that may accept on any entry, exactly as in current Ada) may seem to be an attractive next step. However, there are synchronization issues for the servers in this case. This extension is not required for efficient implementation of nested and non-nested monitors.

Summary

Ada took a bold step in adopting a concurrency model that was roughly contemporary with the Ada requirements and design process. As a result, Ada does much better than its predecessors in terms of concurrency mechanisms *but only by default*. Specifically, C and Pascal have no concurrency mechanisms in their "standard" forms and none of the early concurrent forms of these languages has really caught on. PL/I has provisions for concurrent tasks with synchronization primitives very close to the machine level. However, many (if not most) PL/I programming systems do not implement these provisions; for example, the Multics PL/I compiler and its derivatives[7] do not implement tasking.

Comparing Ada with newer languages, it does not fare as well. [Gehani and Roome 1988, 1549 and 1547 respectively] identify two significant areas where Concurrent C is stronger than Ada: (1) "Entry arrays are used in Ada to compensate for the fact that parameters cannot be examined prior to the call", and (2) "Unlike Concurrent C processes, Ada tasks cannot be parameterized." This is in spite of the fact that both Concurrent C and Ada are based upon rendezvous as the synchronization/communication mechanism. In Chapter 2, requeuing facilities are proposed to solve the first problem and parameterization to solve the second. The requeuing facilities proposed are as powerful as those of Concurrent C; in fact, they are more powerful because they allow a programmer to design custom search structures whereas it would be more difficult for the compiler to do so. Unfortunately, the use of requeuing is also more cumbersome than the Concurrent C facilities, i.e., for simple cases the Concurrent C facilities are easier to program. Further, [Gehani and Roome 1988] is an analytical comparison, not a criticism of Ada. The fact is that designers of Concurrent C learned from the problems in the 1983 version of Ada and the users of Concurrent C are the better for it.

The challenge facing the Ada 9X Project Office is to update the Ada programming language to meet the needs of concurrent programmers by utilizing the experience of Ada over the past years. The most difficult constraint in performing the language revision is also the overall goal of the Ada 9X Project, "...to revise ANSI/MIL-STD-1815A to reflect current essential

[7] According to [Anklam et al. 1982], the derivatives of the Multics PL/I compiler include PL/I compilers for DEC, WANG, Data General, Control Data, Honeywell-Buil CII, and Stratus machines.

requirements with minimum negative impact and maximum positive impact to the Ada community." [Anderson 1989]. Clearly, since current embedded systems are required to use Ada in a distributed environment some of the revision issues, dealing with parallel and distributed systems, submitted to the Ada 9X Project Office will represent "current essential requirements." The task at hand is to identify these issues and develop requirements that satisfy the overall goal and constraints of the revision process as stated in [Anderson 1989]. The conclusion to be drawn from the monitor discussion with respect to the contemplated 9X revision of the Ada standard is that minimal changes in the language would make it technically competitive without wholly obsoleting the current compiler base.

References

[AdaLRM 1983]
"The Ada Programming Language Reference Manual", ANSI/MILSTD 1815A, US Dept. of Defense, US Government Printing Office, 1983.

[Anderson 1989]
Christine M. Anderson, *Ada 9X Project Requirements Development Plan*, Office of the Under Secretary of Defense for Acquisition, Washington, D.C., August, 1989.

[Anklam et al. 1982]
Anklam, Patricia, David Cutler, Roger Heinen, Jr. and M. Donald MacLaren, 1982, *Engineering a Compiler*, Digital Press, Bedford, Massachusettes.

[Booch 1983]
Booch, Grady, 1983, *Software Engineering with Ada*, Benjamin/Cummings Publishing Company, Menlo Park, California.

[Booch 1987]
Booch, Grady, 1987, *Software Components with Ada*, Benjamin/Cummings Publishing Company, Menlo Park, California.

[Brooks 1987]
Brooks, Frederick P., Jr., 1987, "No Silver Bullets: Essence and Accidents of Software Engineering", *Computer* 20:4, (April): 10-20.

[Burns 1985]
Burns, Alan, 1985, *Concurrent programming in Ada*, Cambridge University Press, Cambridge.

[Cohen 1986]
Cohen, Norman H., 1987, *Ada as a Second Language*, McGraw-Hill, New York, New York.

[Gehani and Roome 1988]
Gehani, Narain H., and William D. Roome, 1988, "Rendezvous Facilities: Concurrent C and the Ada Language", *IEEE Transactions on Software Engineering* 14:11, (November): 1546-1553.

[Haberman and Nassi 1980]
Haberman, A.N., and I.R. Nassi, 1980, "Efficient Implementation of Ada Tasks", Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

[Hilfinger 1982]
Paul N. Hilfinger, 1982, *Proceedings of the AdaTEC Conference on Ada*, October, 1982, Arlington, Virginia, 26-30.

[Hoare 1974]
Hoare, C.A.R., 1974, "Monitors: an operating systems structuring concept", *Communications of the ACM* 17:10, (November): 549-557.

[Hoare 1978]
Hoare, C.A.R., 1978, "Communicating sequential processes", *Communications of the ACM* 21:8, (August): 666-677.

[Hoare 1985]
Hoare, C.A.R., 1985, *Communicating Sequential Processes*, Prentice/Hall International, Englewood Cliffs, New Jersey.

[Ichbiah et al. 1979]
Ichbiah, J.D., J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brueckner and B.S. Wichmann, 1979, "Rationale for the Design of the ADA Programming Language", *ACM SIGPLAN Notices* 14:6, (June).

[Ichbiah et al. 1986]
Ichbiah, J.D., J.G.P. Barnes, R.J. Firth, and M. Woodger, 1986, "Rationale for the Design of the Ada Programming Language", ALSYS, La Celle St. Cloud, France.

[Knuth 1969]
Knuth, Donald E., 1969, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Second Printing, Addison-Wesley Publishing Co., Menlo Park, California.

[Knuth 1975]
Knuth, Donald E., 1975, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Second Printing, Addison-Wesley Publishing Co., Menlo Park, California.

[Linn et al. 1988]
Linn, Joseph L., Cy D. Ardoin, Cathy Jo Linn, Stephen H. Edwards, Michael R. Kappel, and John Salasin, 1988, "Strategic Defense Initiative Architecture Dataflow Modeling Technique Version 1.5", IDA Paper P-2035, Institute for Defense Analyses, Alexandria, Virginia.

[RR-0016]
Kent Power, "Alternate Ada Task Scheduling," submitted to the Ada 9X Project Office.

[RR-0071]
J. Barnes, "Applicability to distributed systems," submitted to the Ada 9X Project Office.

[RR-0109]
ARTEWG, "Distributed Systems," submitted to the Ada 9X Project Office.

[RR-0170]
J. R. Hunt, "Scheduling Algorithms," submitted to the Ada 9X Project Office.

[RR-0185]
J. R. Hunt, "Rendezvous Leads to Unacceptable Performance," submitted to the Ada 9X Project Office.

[RR-0379]
Jeremy James, "Scheduling Algorithms," submitted to the Ada 9X Project Office.

[RR-0461]
Bryce M. Bardin, "Provide a standard package for semaphores," submitted to the Ada 9X Project Office.

[Silbershatz 1984]
Silbershatz, Abraham, 1984, *Proceedings of the Seventeenth Annual Hawaii International Conference on System Sciences*, Honolulu, Hawaii, January, 1984, 290-293.

[STEELMAN 1978]
Department of Defense Requirements for High Order Computer Programming Languages– "STEELMAN", June, 1978.

[Workshop 1989]
Ada 9X Project Requirements Workshop, Office of the Under Secretary of Defense for Acquisition, Washington, D.C., June, 1989.

Dr. Joseph L. Linn is a member of the research staff at the Institute for Defense Analyses and the senior technical member of the Ada 9X Project Requirements Team. As a Requirements Team members, Dr. Linn is completing a study of programming paradigms and participating in the development of the Ada 9X Requirements Workshop analysis and DoD waiver analysis. Dr. Linn is also one of the authors of the Preliminary Analysis of the Ada Issues and Commentaries that is now being reviewed by the Ada 9X Distinguished Reviewers. Dr. Linn is a recognized expert on microprogramming and firmware engineering who received both his B.S. and Ph.D. at Vanderbilt University. Dr. Linn is the past chairman of the IEEE Technical Committee on Microprogramming and vice-chairman of the ACM Special Interest Group on Microprogramming.

"SOFTWARE ENGINEERING, SOFTWARE DEVELOPMENT METHODOLOGIES"

WILLIAM H. PITTS

SOFTWARE DEVELOPMENT CENTER-BEN HARRISON

With the emergence of Ada Business Information Systems (ADABIS), software engineers will have the technology needed to faciliate development and maintenance of cost effective Management Information Systems.

A contract awarded by the U.S. government was used to assist in the development of the Standard Army Financial Systems Redesign (STANFINS-R). This was the government's first attempt at designing a large business information system taking advantage of Ada technology. Approximately 1.8 million lines of Ada code using CICS and a relational database will support world wide U.S. Army Accounting functions operating on large IBM compatible mainframe systems. This paper provides an overview of some of ADABIS problem areas encountered.

INTRODUCTION

After a brief look at background information about the early ADABIS project developments, this paper will discuss lessons learned in the area of:

-DEVELOPMENT ENVIRONMENT

-TRAINING

-DESIGN

-STANDARDS and REUSABILITY

Research of former Ada projects revealed that the earlier Ada development projects may not have had adequate development environments and that available Ada tools were immature. Experience proved that ADABIS development and its objective target environments would provide lessons in: compiler immaturity, lack of a mature transaction processing monitor, relational database interfaces; and the need for a more efficient operating system, for some targeted environments.

-In the case of Ada compilers (some more mature than others), some were developed for a fixed environment. The target environment for them as well as compiler certification was questionable for use in the ADABIS environment. The government contracted to have a compiler prepared for its environment and again management made an excellent decision in doing so. Here then was another "CASE" tool that was not in place for development but was developed while the project was ongoing.

-The transaction processing monitor (a variable depending on the environment in question) was identified at the beginning of the project in "UP-FRONT" planning. It was necessary after this decision was made to develop the appropriate hooks in the compiler discussed earlier to accommodate the transaction processor.

-The relational database interfaces including CICS required consideration while developing the aforementioned compiler.

-The operating system required separate consideration during compiler build to successfully work within the environment discussed above.

DEVELOPMENT

It is important to understand from the outset of any Ada project that Ada represents more than just another coding language. Ada in fact is a software engineering tool which promotes performing a large portion of what we know as "configuration management" UP-FRONT while continuing these efforts through all phases of development. "Configuration management" is key to a successful design installation.

When we discuss "configuration

management", we are referring to the following functions as a minimum:

-OBJECT ORIENTED DESIGN(OOD):

OOD represents only one of the methods currently available for use in Ada design and the decision on whether to use this method or not will depend on the designer's own requirements and target environment.

-HARDWARE CONFIGURATION:

Configuration management requires having in place and possibly on-line all of the required Mainframe and PC hardware and software. It also requires the communication lines necessary to bring to bear the network control required to achieve a successful installation.

-TECHNICAL SUPPORT:

WE must be sure that vendors and employees are positioned to support the efforts of the development personnel.

-MANAGEMENT SUPPORT:

Management must be totally committed to support the development effort, particularly when a new software engineering language such as Ada is used.

-CONFIGURATION MANAGEMENT:

In order for us to effectively manage these potentially complex automated business requirements we need to recognize immediately what the target hardware platform looks like. Additionally we need to identify the short and long term actions necessary to develop the proper support network to successfully accomplish the assigned tasks.

-COLLOCATION:

It has proven to be an extremely wise decision on the part of management to collocate the physical developers and functional designers. This collocation has proved itself to save an immeasurable amount of time resolving issues pertaining to a clearer understanding of the requirements set forth by the functional community and ensuring that all regulatory requirements remain compliant.

TRAINING

When we consider training in the environment described above, there are several issues that become paramount. Some of these are:

-TRAINING PROGRAM:

Due to the nature and immaturity of Ada training it is necessary to develop, fund and manage a comprehensive training program for the project at hand. It takes approximately 3-6 months to train an Ada programmer whether or not the individual has previous programming experience. Following the formal training, individuals will arrive at some level of proficiency based on how they are managed and utilized. This training program is part of the "configuration management" planning that must take place "UP FRONT:, prior to any coding development.

STAFFING:

Effective staffing is key to a successful development effort. Current experience has indicated that STAFFING at all levels of an Ada project is a critical and sensitive function. It is imperative to ensure that everyone is aware of and party to information having any impact on the overall effort.

-RESOURCE LEVELING:

One of the most difficult areas of the training effort will be to level resources while continuing education. Training will be required on HARDWARE, SOFTWARE, PCs, STANDARDS, FUNCTIONAL REQUIREMENTS, REPORTING PROCEDURES, TESTING and MAINTENANCE.

DESIGN

It is essential for an effective system design that a strong physical and software engineering oriented approach be utilized. Lessons learned include: issues related to the need for a requirements/process design database; physical design; and a sound development foundation. The software engineering oriented approach and the use of a structure/object oriented design mix may be acceptable.

STANDARDS AND REUSABILITY

The need for design and development standards and the use of Ada Generics have provided several additional lessons learned and these are further discussed

below:

-With the advent of the Ada language it became painfully apparent that there existed little or no standards under which to control and direct a specific Ada project. This presented a unique situation in that there appeared to be more than one avenue open for the development process. One avenue allowed for creating standards as development proceeded, a very risky and quite unacceptable situation, albeit quite tempting at times. Another approach appeared to be to take existing industry standards, government regulations, previous successful experience: review, update and enforce those. This second alternative proved to be most acceptable and successful. This has proven to be another sound management decision. In spite of the fact that Ada does not always lend itself to the same development criteria as other coding languages, there are enough similarities to adjust accordingly.

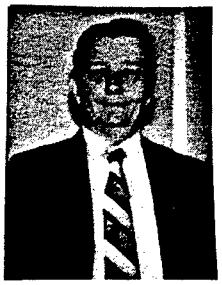
-Ada Generics presented a whole new set of design challenges related to reusable code, i.e; what is reusable, where, and how much? An immediately recognizable advantage of Ada Generic and reusable code is in the area of interactive screens and reports I/O. In the system described above with 1.8 million lines of Ada code, approximately 65% is made up of reusable Ada Generic code. A direct result of this reuse is represented by the fact that only 35% of functional design code was written by an Ada programmer. Herein lies resource planning information.

In summary, some of the lessons learned from the first ADABIS software development will provide software engineers of the future knowledge that may be used to overcome potential ADABIS problems in the areas discussed above. As we are just now working through the Testing environments and have not yet encountered the benefits of Ada maintenance, we expect to update our findings as soon as the results of these milestones are behind us.

BIOGRAPHY

WILLIAM H. PITTS
DEPARTMENT OF THE ARMY
U.S. ARMY INFORMATION SYSTEMS
SOFTWARE DEVELOPMENT CENTER-
BEN HARRISON
INDPLS, IN 46249-0901
TELEPHONE: COMM(317) 543-6595
AUTOVON 699-6595

CHIEF, Field Accounting Systems Division
ASQBI-W-SSF-SRD II



An MIS Ada Practicum

by Kenneth Fussichen

Computer Sciences Corporation

ABSTRACT. "An MIS Ada Practicum" is a brief overview of a major MIS Ada implementation, the first known attempt to deliver Ada to an IBM mainframe.

The project has its roots in a typical Yourdon-DeMarco functional description which was essentially converted "on the fly" to an Object Oriented structure as a response to a DoD mandate that the system be written in Ada. The mandate was received as the project phase moved from the functional definition phase to the physical definition phase.

This paper is intended to open discussion on Ada's basic suitability to the MIS environment. Although some technical insight is provided, the target audience is the Ada project manager and the MIS project manager who has been asked to look into Ada.

INTRODUCTION. I am a warrior. I live for the battle. I have survived many battles over the years because my warlords were wise and brave. I am still a warrior, but now I am a warlord too. I will do battle for any king who will pay my price. My loyalty is only to the battlefield. And once engaged, I will not abandon a king until the battle is won.

The battlefield has always been an IBM mainframe battlefield. Even its detractors speak its name with respect. Those who would never consider doing battle on an IBM mainframe battlefield would not argue there is almost always cause for particularly bloody engagements.

Three years ago a minstrel sang a story of a battle to be won. The song was, to me, of a traditional battle. The sorcerers were accountants, the battlefield was an IBM mainframe, and the weapons included COBOL, CICS and Datacom/DB. This was to a battle of magnificent proportions, greater than any battle I had yet seen. The battlefield will encompass approximately 150 records, over 2000 data items and 500 programs. It was said that the king was most wise. Excitement was the blood that raced through my veins.

I decided to present myself to that king and offer my services. He invited me to sit with him and drink wine, sample cheese and discuss past and future battles.

He spoke of a king of kings, a king to which he had sworn loyalty known as "The Client". "The Client" was great and powerful but was besieged by too many advisors of dubious loyalties. Our task was to guide "The Client" to a better world, in spite of bad advice and conflicting requirements. As he spoke, my appetite for battle grew. The minstrel was correct, for this king was very wise. The battle could be won. We agreed on a price for my services and I swore an oath of loyalty to my new king.

It is important for you, my dear reader, to note that at the time these plans were drawn I knew not of your existence. It was a revelation to me that there existed in the outlying provinces, a whole race of warriors that understood other weapons, weapons with strange names like Ada and Object Oriented Design, capable of real time, embedded system development and support.

It took much effort on my part and on the part of my brethren warriors to understand the existence of this other race of warriors. To be able to communicate with you has been more difficult still. Your words sound the same but seemed to have slightly different meanings, or innuendos. Values were disparate. We put great value on the word "implement". Outlyers put great value on the word "engineer". Discussions with the outlyers often became enraged and confused. (It seems few warriors of either race have much skill at diplomacy.)

As time passed, we came to understand each other and respect each others views. But, in truth, we never embraced as brothers might. Still, those pioneer warriors from the outlying provinces taught us much and endured great hardship. It is only right that as the tide of battle turns in our favor that we acknowledge their efforts and attempt to share the spoils of war.

Be assured that our strategies are identical, our weaponry is equivalent and our terminology is similar. And although our battlefields may be different, we each wear our battle scars with equal pride.

I am therefore invoking my right, as one warrior to another, to demand the same respect and attention from you as I have afforded your brethren. I gladly will explain variances in weaponry and terminology at the appropriate time just as your brethren have taught me.

INTRODUCTION TO Ada. A scant six weeks after I swore loyalty to my new king, he came to us and spoke powerful words to all warriors. "The Client" had decided that our primary weapon, COBOL, was old and rusted. Therefore, all traces of COBOL would be tossed into the inferno. We would immediately draw a new weapon called Ada.

Ada was new and powerful and supported exalted principles such as portability, maintainability and reusability. All warriors spoke with a single voice. Ada only supported exalted principles "in theory". They had never been tested in battle, except in the outlying provinces called embedded systems. Even there, the proclaimed victories sounded shallow.

My king was firm. "The Client" swore vengeance on any who spoke against the weapon called Ada. Retribution would be swift and final. Any warrior would be released from his oath of loyalty, but any warlord or warrior electing to stay must assume the role of an Ada standard bearer. There were no waivers or exemptions.

As time passed, each warrior in turn, spoke privately with my king. As fears were admitted, reassurances were given and sins of weakness were forgiven, and in turn, a new oath of loyalty was sworn to our king. All warriors understand bravery and wisdom.

Preparations for battle began immediately. Analysis of the enemy began in earnest. A very large reconnaissance team was formed to study the enemy and report its nature. A database team was formed to map the terrain using existing technology, particularly database theory, factoring information from the reconnaissance team as it became available. Serious issues regarding data storage capacities were raised. Mercenaries specializing in database were retained. A technical support team was formed to study battlefields, including the target environment and any intermediate environments necessary for development and documentation purposes. Alliances were formed with warriors of the outlying provinces. An emerging technology team was formed to study the new weaponry called Ada. And a documentation team was engaged to document and report our progress across teams and ultimately report back to "The Client".

Initially, only the emerging technology team was concerned with the new weaponry. They were tasked with establishing initial trade relationships with the arms merchants with names like Intermetrics, Rational, Alsys, Meridian and others. All other teams had traditional tasks. The impact of the mandated change in weaponry had been isolated.

As knowledge was gained teams were refined, new teams were introduced and other teams were disbanded or renamed. Slowly, knowledge of the weapon known as Ada spread.

REAL TIME AND SCIENTIFIC SYSTEMS. In an effort to highlight the differences between the provinces I will describe what I see. This is not an academic description of what Real Time and Scientific battles are, only one warriors observations of another's battlefield.

Real-Time Warriors and Scientific Warriors are concerned with time intervals, tolerances, software durability and reliability. Microsecond timing is important and can turn the tide of battle. Real-Time Warriors and Scientific Warriors have few weapons other than language and thus have great difficulty understanding that other weaponry could exist. For a Real-Time Warrior or a Scientific Warrior to be totally dependent upon a particular compiler vendor

seems to be neither unusual nor a cause for concern.

Real-Time and Scientific Warriors are less concerned about aesthetic reporting and state beyond the life of a program, which is a significant departure from the MIS Warrior's view.

MANAGEMENT INFORMATION SYSTEMS. MIS Warriors are concerned with time too. But our sense of time is much different. A two second response time is often considered respectable. Even a five second response time is considered tolerable in some situations.

This must truly seem inconceivable to you. What processing could possibly require five seconds? In that answer comes the essence of the MIS battlefield. The primary recipient of the MIS warriors efforts is a human being. A half second variance to a human interfacing with a computer is not relevant to the human. It is very relevant to the computer. Therefore the computer uses available resources to service hundreds or thousands of other humans in the same cycle. This is accomplished by permitting unlimited human access and releasing all possible computer resources encumbered while the human reflects upon the output.

Specialized warlords of various titles police these aspects of MIS battles. They teach and explain resource utilization and balance. They have three weapons beyond language. They are the full blown Operating System; the Teleprocessing Monitor or TP Monitor and the Database Management System (DBMS).

It came as a surprise to me when I learned that in a Real Time environment or a Scientific environment a Run Time System may exist in lieu of an Operating System. In an MIS environment, a full function Operating System must exist. Only the Operating System owns the hardware. Operating System is responsible for Input/Output (I/O) control, and resource management. All other software are tenants, attempting to coexist peacefully.

The TP Monitor is like a subordinate operating system of its own. All application terminals are connected to the TP Monitor and all programs are written as subroutines to it. The TP Monitor is responsible to respond to interrupts for process time or to poll all terminals looking for work. The TP Monitor is additionally responsible for its own I/O, which is specialized in nature. The effect is that widely varying applications can be accessed from the same clusters of terminals. The dominant TP Monitor in the IBM mainframe environment is IBM's Customer Interface Control System, or CICS. CICS is the dominant TP monitor primarily because of its association with IBM, rather than for any outstanding features.

All warriors are concerned about maintaining state. But the MIS sorcerers (accountants, actuaries and other financial types) require state to be maintained over years. It is quite reasonable to request a report of last year's conditions contrasted against this year's conditions and again contrasted against next year's forecast. This requires the use of additional weaponry called Database Technology.

Database Technology is a weapon of magnificent proportions. Properly designed and calibrated, data can be retrieved, organized, summarized and presented rapidly. Improperly designed databases have caused more MIS casualties than changing specifications late in the battle.

Database Technology is so powerful that specialized warlords have evolved whose function is different from those who maintain Operating System and TP Monitor. There are Logical Designers and Physical Designers. Logical Designers begin their work with the system functional requirements. Data items are grouped where they will be used together. Keys are identified to retrieve these records. No concern is given towards specific technologies. This would be characterized as designing with a concern towards micro efficiency.

As Logical Designers conclude their work, Physical Designers begin. The Physical Designer aims his efforts towards the available or mandated technology. In other words, given this logical model, and this proprietary DBMS, what do we really have to do to make it work? Tailoring efforts continue throughout the life cycle of the battle on into peacetime, or maintenance. Physical designers often populate tables (as they are now called) and are responsible for refreshing and refining the

data as battle conditions change. Physical designers are often tasked with issues such as interfacing with Operating System and TP Monitor.

It is common that Logical Design Warlords and Physical Design Warlords are the same individuals. It is just as common that these warlords have no taste for the programming language. It may also prove quite startling to you, Warriors of the Real-Time and Scientific provinces that these Database Warlords are often most highly prized and that their counsel is sought by king and warrior alike.

Since a human interface is the focus for the MIS warrior, he is obligated to expend a great deal of effort in making attractive presentations. Reports, whether on a screen or hard copy, must have an aesthetic value sufficient for the reader to continue. Headings, aligned decimal points and commas, page breaks and numbered pages are simply requirements.

Finally, since our sorcerers must also be appeased and most of those sorcerers are accountants (actual or would be), a column of numbers must total precisely, regardless of magnitude. Absolute precision is required. In this case, a tolerance of zero percent is considered standard.

APSE. In order to learn and use the new weapon called Ada, traditional development environments were discarded as far too hostile. An Ada development simply could not be done on an IBM mainframe with respect to a calendar using the traditional tools of the MIS warrior.

After a brief search, it was decided that we would use four Rational R1000's to spearhead the development. We named our tools, in order, Bilbo, Frodo, Gandalf and Strider. Later we added Pippin and a she-warrior named Ayla.

The naming was not a trifle, nor was it a whim. Each name was chosen symbolically, representing heros and reluctant heros responsible for bringing about a new and better age.

COMPILER VARIANCES. We elected to fight a two front war. We used the APSE on the development machines to gain experience that would later be used on a more hostile battlefield. While most warriors were training within the APSE, a small group of volunteers worked the target environment, looking for compiler and other variances that needed to be resolved, before we could claim victory.

It was interesting to note that exploring the target environment was considered by some (from the outlying provinces) to be initiated prematurely. It is naive to expect different compilers for different architectures to act identically, given identical source code, even if both compilers passed the validation suite.

Generics were implemented differently. Nested generics usually would not work on the target even though they tested perfectly on the development machine. Executable code sharing, desirable in our CICS environment, was not implemented on the target which forced executable sizes to balloon. Since we found tremendous advantages using generics to support our mainline logic, it became difficult to implement generics anywhere else in the system.

Representation specifications were not supported on the development machine because field order was guaranteed to be as specified. No such guarantees were provided with the target compiler. Representation specifications were therefore required because existing utility programs did not understand the compiler's penchant for re-ordering fields within a record. Fortunately, we were able to disable compilation checking for that particular feature on the development machines.

Unchecked_Conversion was simply not fully implemented in the earlier versions of target compiler. Certain types and certain locations within a record caused problems that were unaddressable until compiler corrections were made available. Try as we might, we could not totally eliminate the requirement for Unchecked Conversion, but we were able to limit its impact.

Pragma Inline, in addition to forcing additional dependencies, simply does not work on the target compiler.

Simple renaming a function in a specification worked in the development environment, but not in the target environment. It compiled as expected, but it would not execute.

Compiler variances, whether or not a deficiency, is a barrier that "The Client" does not understand, even if there is an interest.

When a king raises an army to fight a battle, and that battle has a nature that is dissimilar to any battle ever engaged, a warrior cannot go back to the king and say, "We cannot implement victory because the weapon doesn't work the way the [Language Reference] Manual says it should." Rather, you test the weapon, find out what it does do, and make the weapon do your bidding. A weapon as sophisticated as a spear, can, in some cases be used as a club, and still achieve the ultimate goal.

At some point, it becomes necessary to move source code from the development environment to the target environment. It should be noted that IBM mainframes are particularly hostile to other architectures, to occasionally include other IBM architectures. Porting source from the Rational to the IBM was an early important task assumed by the Technical Support Team. This was initially accomplished with a minimum of support from the vested arms merchant.

Moving source code from one environment to another assumes different flavors. Once the capability was established it then became necessary to move entire subsystems. This is a far more complex task because of the compilation dependencies introduced by Ada. Ada source code scripts had to be built in dependency sequence in a manner to support compilation on the target machine. This process was called the Target Builder. Merchant support here was better. But, since no one had done any real testing of this specific target, the Technical Support Team eventually wrote or enhanced most of tools necessary to support Target Builder.

TELEPROCESSING MONITOR. When "The Client" mandated use of the new weapon called Ada, the primary obstacle that fell upon deaf ears was the lack of an interface to a production quality teleprocessing monitor.

There was no interest in providing Ada support to CICS from the IBM provinces. Further, the target environment compiler used a memory management technique that directly conflicted with CICS's memory management techniques.

What had previously been described as "an implementation detail" is now a show stopper. An Ada compiler that does not support teleprocessing has no practical MIS application. Since we were not engaged to write or modify a suitable monitor, we could only continue under the assumption that a production quality teleprocessing monitor would arrive soon enough for us to use it.

Eventually, the alliance with the Intermetrics provinces was reviewed and upgraded. The Intermetrics warriors were to deliver a version of Ada that would support the teleprocessing environment. The delivered version of the compiler was identical to the previous versions we had seen at the source level. Only the compilation was different. We now had two Ada weapons, one for the on-line environment with an on-line Run Time System (RTS), and the original Ada weapon with its batch RTS. They did their job well, maintaining appropriate supply lines open after they had secured their section of the battlefield.

DATABASE INTERFACE. Input/Output (I/O) requirements of an MIS battle are so significant that entire strategies are built around those requirements. Part of the job of the Database Warlord is to facilitate I/O by working with the selected Database Management System (DBMS) and enhance the design based on the characteristics of that DBMS. "The Client" had already mandated the use of Datacom/DB by the merchant of that day called Applied Data Research, now Computer Associates. This was an entirely reasonable alternative, except that it had no Ada interface, and none were likely to be supplied by the original DBMS merchant.

All major proprietary DBMS's available on the IBM mainframe fully support COBOL. None support Ada. Further, Ada's own I/O handling is far too primitive to be taken seriously on an MIS front.

Therefore, a "hook" was necessary to access Datacom/DB from Ada. This problem was less severe than the TP monitor problem since the talent existed locally to build that "hook".

However, it was decided to use the same ally who delivered Ada to CICS to deliver the necessary DBMS "hooks". There were several reasons for this, including a desire for consistency, sup-

port, and the fact that "The Client" promised availability of this weapon prior to the start of the battle.

DEVELOPMENT ENVIRONMENT. Weapon support from the merchant of the development environment was an absolute requirement from the first day. That support was provided by that vendor at all levels. Our battlefield stressed his weaponry like no other had, and that contribution was understood.

Testing support was less, and that was understood. Since the target environment was as hostile to this arms merchant as it was to us, support for DBMS simulation and TP monitor simulation was crude, at best. Surprisingly, most other testing tools were also primitive. Virtually all testing capability was manufactured by our own warriors. Some component testing capability was built upon provided skeletons. Component testing tools, although useful, were only practical for testing the most simplistic packages. Any package of substance was too complex to test at a component level.

Porting from the development environment to the target environment was also supported at a lessor level. The arms merchant for the development equipment seemed to work at avoiding developing a skill or a bias towards any particular target environment.

TARGET ENVIRONMENT. Weapon support from the arms merchant of the target compiler was also an absolute requirement, from the first day. That support was also provided.

This arms merchant was also tasked to provide a CICS interface and a Datacom/DB (DBMS) interface and support those products at the same level that support was provided for the compiler.

The merchant for the targeted equipment had the larger and the more difficult task. In addition to mandated weaponry corrections and improvements one would expect from a new model of an existing weapon, both a CICS interface and a Datacom/DB interface had to be built and maintained. Further, the target environment was as hostile for warrior and arms merchant as well. And, as was the case in the development environment, no one had adequately stress tested their weaponry.

There were other problems. This arms merchant had an interest in developing an SQL interface to Ada. This was beyond our scope, and provided no real assistance to the task at hand. It required additional persuasion to focus this merchant correctly so we could complete our task.

We lacked a source level debugger. It was delivered after it could be practically applied to this task. Version 1.0 of the debugger was, for all practical purposes, a beta version, still buggy, bulky and rough handling. This product may be useful for the next battle.

Executable sizes were horrendous. A forensic team was established to determine what caused ballooning executable sizes. Some of the more memorable findings that resulted in excessive executable sizes were nested generics, private types and excessive packages. Each of these features was cautiously considered before it was used. Eventually, executable sizes improved to within usable, but never truly comfortable, limits. Curiously, execution speed was much quicker than expected.

Near-panic was observed when variances in source and target weaponry occurred. Paradoxically, they were relatively easy to solve. Forensic warriors were normally able to isolate the problem reasonably quickly. The next step was to build the minimum case that supported the event. When each arms merchant was notified with this minimum case, a work around was provided in short order, until the next weaponry release corrected the problem.

Where some of the problems were fairly formidable, the level of support provided by each of the arms merchants rendered this potentially disastrous situation to that of a minor inconvenience.

VISITING "EXPERTS". As our knowledge in the weaponry increased, so did knowledge of our ignorance. The more we learned, the more we realized how little we knew. To fill this increasing knowledge gap, we contacted mercenaries or "experts". This class of mercenary would come in, disburse what was claimed to be great knowledge and leave one or two or three days later. It takes little time to smell out "great knowledge". It has

the distinctive scent of a rose. Other "great knowledge" had a different scent, more like fertilizer. We used both.

A great deal was learned about "experts". We invited sorcerers who wrote books that made sense, conjurers referred to us by sorcerers and arms merchants, and magicians of both honorable and dubious distinctions.

Some sorcerers, conjurers and magicians were charlatans. Others collected coffee cups and a few even shared genuine interest and offered valuable assistance at critical times. But when all was said and done, none had to face the reality of battle.

CONTRACT CONSULTANTS. A second type of mercenary was hired. This second type would live on the battlefield and not disappear in scant days. These mercenaries were powerful and well respected. They would lead patrols into areas no MIS warrior had ever ventured. It became evident that the terrain was new to the mercenaries as well. Although their background was different, we came to understand that our battlefield, that called MIS, is not the lessor battlefield. It is a different battlefield marked by slightly different problems and a slightly different dialect.

Anguish prevailed between the Real Time and Scientific engineers and the MIS "implementors" until dialectal differences could be traversed. It turns out that visibility is a flavor of addressability and that a generic is new macro instruction. The new technology was built firmly on principles long used voluntarily on MIS battlefields.

Eventually, there were no pure engineers as there were no true "implementors". There were only warriors intent on delivering not just victory, but victory with style.

It is also appropriate to note that a single, special mercenary was located in the shadows of the mountains and retained. He would appear for three or four days periodically. Although his skills with the particular weaponry and that of teacher are of the first magnitude, that was only a portion of the role this mercenary filled.

This mercenary was essentially given a fast horse and a complete run of the battlefield. He had no authority except some formidable powers of persuasion and an audience with the great and powerful. He would use this freedom to locate and solve battlefield problems before significant resources were expended. This lone mercenary was granted the unprecedented title of "Master and Friend", an honor well-deserved.

PORTABILITY. The illusion of portability was propagated. Ada is Ada is Ada. Although compiler variances caused significant design changes early in the coding phase, those variances were identified early enough to prove non-fatal. What compiled on the Rational R1000 essentially compiled and executed on the IBM mainframe using OS/MVS. We were able to change certain low level code to simulate both a TP monitor and the DBMS on the testing machine, but both were so functionally limited that only the illusion of portability was supported. The theory that the we could run the exact same system on a VAX battlefield by simply changing a few low levels and porting to a VAX is, of course, ludicrous.

By definition, physical database design mandates that the features of the database management system be incorporated into the design. If the DBMS will not port, then even though the source code ports, the weaponry will not run except as a simulation.

This was demonstrated clearly when testing in the development environment was compared to the testing in the target environment. The development environment could only support certain features, with drastically limited data. System stress testing on the development battlefield was not possible.

REUSABILITY. The issue of reusability is so important and so complex that the mere use of the word "reusable" can often generate an emotional response.

There does not appear to be common agreement on a precise definition of reusability. There does seem to be common agreement that whatever the definition, reusability is one of the higher goals of software engineering. Conversely, any warrior who does not take an oath to the god named "Reusability" is forever condemned to the wander eternity in darkness.

I have no wish to wander eternity in darkness. I therefore take this oath of loyalty to the god that defies definition. And although I have no desire to impose my definition of reusability on the rest of humanity, I can offer my own limited perspective.

Writing code is easier than testing that same code. It then follows that if I can write code one time, test that code one time and use that same code multiple times, then that code, for our purposes, is reusable code. Continuing with this logic, generics are reusable code; reusable packages are reusable code and tested code templates are reusable code.

We were able to use generics for our mainline logic. This decision caused every module to conform to a certain standard that was accepted project-wide. Timing issues were reserved for mainline logic, written by the most experienced staff. This permitted relatively inexperienced programmers the opportunity to produce a great deal of code without consideration for timing issues, which had already been correctly handled by the mainline generics. Approximately eight variations of mainline generic supported 500 programs.

We were able to use a fair number of reusable packages. But reusable packages are old news with a new name. MIS warriors have long used "common modules" in systems development and maintenance. We had standard problems of isolating reusability into packages that were substantial enough to warrant separate packaging.

Sometimes, a planned reusable package required a database retrieval or an update. Frequently, the closed code of a planned reusable package tampered with pointers set by the application. The solution was to remove the database code from the planned reusable packages. What was left was often too trivial to warrant a separate package. This code was pulled in line.

Both generics and common packages can live in something called a "Reusable Library". This is, perhaps, a classification of reusability.

Another class of reusable code might be "Reusable Tools". Tools that are used extensively and even across projects might fit in this category. These "Reusable Tools" are not necessarily part of the delivered system, but integral to the development effort would exist in a separate set of libraries. Tools such as Unit Test Generators, Code Generators and Porting Tools could conceivably live here.

Another less sophisticated library called "Reusable Templates" could exist. This source code is designed essentially for "cut and paste" operations. If this code is tested and used in multiple places, it qualifies as reusable.

We made extensive use of templates. Much of the screen handling and the database I/O was similar enough to support templating.

We became quite expert at templating. We wrote templates for all mainlines, screen handling, data item typing and application views of the database. Every data item in the system was stored in it's own database, along with it's attributes. Screens were defined in terms of their data items. Application views of the database were stored in terms of the records that were required and the fields within the records that were used by this particular application.

As a matter of pure survival, we wrote code generators to manage our templates and implement 100% of our screen packages, types packages and database views. Programmer changes to generated code were prohibited. Eventually, 71% of the almost two million line of code in STANFINS/R was generated code.

MAINTENANCE. It is estimated that this system of almost two million lines of code has been completely rewritten at least four times. Rewrites have been necessitated as knowledge was gained, both functionally, and as skill was gained with the weapons set.

The system is complex and yet maintenance seems to be a much smaller task than has been seen in other battles. Although final judgement cannot be made until maintenance becomes more prominent than development, it does appear that the goal of maintainability has been met.

The promise of Ada that attracts the MIS warrior is the promise of maintainability. Virtually every other major software engineering principle pales in its image. Maintenance is such an expen-

sive chore, that if it can be practically overcome, Ada would be assimilated quickly by the MIS provinces. Preliminary findings indicate that our implementation of Ada may be significantly more maintainable than its COBOL predecessors.

PERFORMANCE. Crude Ada/COBOL benchmarks were performed. The good news is the Ada version appeared to run significantly faster than its COBOL counterpart. The bad news is that the Ada executable sizes are two to ten times larger than the COBOL executables.

MIS systems tend to be more I/O bound than CPU bound. This means that a significant performance gain can be attained by concentrating on physical database design primarily, and on process issues secondarily.

This gives Ada essentially equal footing with COBOL, if performance speed is an issue, since database design is language independent. However, if performance is to include actual executable size, the weapon known as Ada has some problems.

Some success in reducing executable size has been achieved by limiting the number of packages and by judicious use of private types and generics.

TESTABILITY. Testing was difficult. Testing in the development environment was difficult, consumed excessive system resources and was usually inconclusive. In response to the question "Does it work?" one would hear "It worked on the Rational".

Testing in the target environment was worse. No one has ever exèrcised the Intermetrics compiler to the extent we had. Compiler variances and inconsistencies would complicate issues and slow compilation scripts.

Actual testing was equally unpleasant. No tools existed to support testing. Source level debuggers were delivered too late to do much good and abends would point to Assembler programs of Ada code, which clarified nothing. Testing was a painstaking process.

One potential bright spot was that once a program compiled we were pretty sure it was close. This is attributed primarily to Ada's compile time parameter and type checking and extensive use of generated code.

OVERALL PROJECT SUCCESS. The battle is won. The original functional goals of "The Client" have been met. The system performs faster than anticipated, even though executable sizes are still exceed comfort levels.

The battle has been a greater success in human terms. The aggregate level of learning for warriors is among the highest I've ever seen. More warriors attend classes in the evening, write professional papers, belong to professional organizations and book clubs than any other campaign I've seen. The knowledge of Software Engineering principles is the highest of any campaign I've participated. And it was done on time and within budget.

THE DEATH OF COBOL. Sounding the death-knell of COBOL would be premature. The fact that the weapon called Ada worked at all under battlefield conditions surprised many, including some who should have known better. As implementors, we are highly skilled at making systems work.

That the weapon called Ada worked as well as it did is also a surprise. This is as much a tribute to the teamwork of various merchants as anything else.

Finding qualified Ada programmers is difficult. Training existing staff's in Ada is difficult and expensive. These two considerations alone make COBOL a superior choice over Ada. Given that MIS considerations were omitted from the language design, making DBMS and TP monitor interfaces difficult, and limitations in arithmetic and pretty printing, no sane MIS warlord will embrace Ada.

But given that this serious MIS implementation of Ada succeeded, and that it appears to be more maintainable than its COBOL counterpart and Ada's potential as a Program Design Language, well, no sane MIS warlord can ignore that either.

The COBOL death-knell is certainly premature. But it does appear that something better may be in the offing.

EPILOGUE. Ada has been implemented on the MIS battlefield. Further, it was accomplished using IBM's OS/MVS, CICS and state of the art database technology. Every major software engineering prin-

ciple has been enhanced without sacrificing functionality to "The Client". The demons that plague us, such as ego, ignorance and ruthless ambition have been temporarily chained. The surviving warriors were significantly strengthened by the battle. We have learned many new strategies and techniques.

As we transition to peacetime (maintenance), there is more to do, particularly in the areas of refining development technologies and testing. There is also the opportunity for technology transfer. Experiences with reusability and portability were difficult and important gains that need to be shared.

It is also time to extend hands in friendship to our Real-Time and Scientific Allies, to our reliable arms merchants and to he who is called "Master and Friend". We look forward to new battles, refreshed alliances and above all, the opportunity to serve "The Client".

Thank You!

BIOGRAPHICAL SKETCH. Kenneth Fussichen is a Computer Scientist for Computer Sciences Corporation in Indianapolis. He may be contacted at 317/543-6580 or P.O. Box 16008, Indianapolis IN 46216.

COMMERCIAL MIS BUSINESS SYSTEMS, OBJECT-ORIENTED DEVELOPMENT & ADA - - CAN THEY FIT TOGETHER?

Randy A. Steinberg

Andersen Consulting
Chicago, Illinois 60602

ABSTRACT

The MIS data processing community has no disagreements where systems support and development are concerned; software development, as currently practiced, does not adequately create systems that fully support business needs. Object-oriented software engineering disciplines appear to provide one of the first major breakthroughs for cracking the software development problem. Recent technology advances in hardware, systems software, and programming languages have made such disciplines more possible to consider. The Ada programming language appears to be well-suited for an object-oriented approach. Its success on development projects for the military make it the first such language on the market that fits the object-oriented paradigm and can be used to build large systems. While there are other languages available that can be used for object-oriented development, none has yet proven itself for building very large systems with hundreds of thousands of lines of code. This paper attempts to present how Ada may fit into the object-oriented paradigm using a small business example. Two radical issues are discussed: object-orientation in business systems development and using Ada as support for that development.

Overview

Suppose the following lines of code exist in 58 modules used for a large commercial billing application for a business organization:

```
READ BILLING-ACCOUNT INTO WORK-RECORD.
MOVE BILLING-BALANCE TO WORK-BALANCE.
```

Suddenly, during an EDP audit of that company, an auditor recommends that the billing balance should not be taken directly from the billing file. Instead, the balance should be calculated from the total of all ledger entries for each customer. This means reading, from a ledger file, all the ledger records that belong to a particular customer and summing the balances found in each record.

The auditor's request was fairly simple. The immediate effect on the application is not. Each of the 58 modules will have to be changed, re-compiled, and re-tested.

The true nature of commercial business systems are not as static as conventional development methods might lead one to believe. Businesses are in a constant state of change due to competitive needs, governmental regulations, mergers, acquisitions, user politics, and changing technologies. In many large development projects, the business needs may have changed so much since project inception that when finally delivered, users complain that the system is not what they really need. Most development methodologies require a full static understanding of every system component or possible change effect at design time, well before these unpredictable changes take place. In the above example, who could have known in advance what an auditor was going to recommend?

What would be more desirable, would be a development approach and architecture that is more "flexible" to the constant change that is experienced by businesses today. Object-oriented approaches appear to offer much greater flexibility towards that change. One current problem is that while any development programming language may be used with an object-oriented approach, my experiences with using conventional development languages in an object-oriented manner have resulted in the following:

- Application performance is greatly affected.
- Code bulk and application size greatly increase.
- Data type dependencies still exist between modules.
- Language constructs represent the business problem even less than they do without an object-oriented approach.
- Other modules being written with logic dependencies have little or no protection.

Generally accepted object-oriented development languages, such as SMALLTALK and OBJECTIVE-C, carry all or most of the features required by an object-oriented architecture. However, they have not yet proven their ability to build large systems with hundreds of thousands of lines of code.

The Ada language appears to support, directly and indirectly, many of the concepts desired in an object-oriented approach. The language is supported on all types of hardware platforms and operating systems. In addition, systems built by the military using Ada have shown that considerably large programs can be built successfully. The remainder of this paper attempts to look at some of the major components of an object-oriented architecture and describe how Ada fits into those requirements.

Object-Oriented Development

Object-oriented development is a software engineering approach that attempts to closely model the structure of a computer application after the structure of the real (business) problem. Objects are the fundamental units of computer system construction. Each object corresponds to an actual business entity in the real world. This is the fundamental difference from conventional development approaches where the unit of systems construction is based on modules representing abstract business functions and logic. Using the billing account problem described earlier, an object-oriented approach would involve constructing objects such as Billing Account, General-Ledger, and Customer.

Object-oriented development concepts and technology have evolved from research in Artificial Intelligence and Advanced Software Engineering. Many of the basic concepts were developed by researchers at Xerox PARC in the early 1970s. This research team was building complex applications for dedicated intelligent workstations with rich human-to-computer interfaces. Object-oriented development was a result of their pioneering efforts in coping with the application complexities involved. The most universally-accepted terminology for describing object-oriented structures are:

- Objects
- Messages
- Methods
- Classes
- Inheritance
- Polymorphism

Objects

An object can be viewed as a pool of private data with procedures or methods that perform operations on that data. An object can loosely correspond to the conventional notion of an application module or subprogram. While conventional modules tend to represent actions or "verbs," such as IO-HANDLER or COMPUTE-BILL, objects represent real-world entities or "nouns," such as Billing_Account or Customer.

The intent of an object is to hide or make inaccessible the data and related procedures for a business entity that will change state over time. In the billing account problem presented earlier, it would be more desirable to provide the billing balance to other program modules without forcing them to be dependent on the format of the billing data or how that data is obtained. In this way, the effects of a change in the code or data related to the billing balance can be isolated to a small part of the application.

An object can be logically viewed as being composed of the following three parts:

- o Data
- o Methods
- o External View

The Data part describes the format or structure of data elements that belong to the object. The Methods section contains code that manipulates or accesses this data. Data and Procedure parts cannot be accessed directly by other programs. Instead, the External View part provides the only interfaces available to other programs for accessing or changing the object.

Using our billing account problem, an object could be constructed that represents a billing account. The data part of this object would consist of a record description of a billing account. Application code that manipulates this data, such as logic to create a new billing account or obtain the billing balance, would be in the methods part. Interfaces to invoke those methods, as well as external access to the account data would be in the external view part.

With Ada, the Package construct and user data types may be used to represent an object. Data types can be viewed as slightly different kinds of objects usually referred to as Classes. This topic is discussed in more detail later. An example of the billing account object as represented by an Ada package would appear as follows:

```
Package Billing_Account is

    type ACCOUNT is                                  |
        record                                       |
            Name      :  STRING(1..30);              |  Data
            Address   :  STRING(1..40);              |  Section
            Amount    :  FLOAT;                      |
        end record;                                  |

    function New                                     |
        (Customer     :  ACCOUNT)                    |
        return ACCOUNT;                              |
                                                     |  External
    function Get_Balance                             |  View
        (Customer     :  ACCOUNT)                    |  Section
        return FLOAT;                                |

end Billing_Account;                                 |
```

```
package body Billing_Account is

    function New                                     |
        (Customer  :  ACCOUNT)                       |
        return ACCOUNT                               |
    is                                               |
    begin                                            |
        ...code to create a new account..            |
    end New;                                         |
                                                     |  Methods
    function Get_Balance                             |  Section
        (Customer  :  ACCOUNT)                       |
        return FLOAT                                 |
        is                                           |
        begin                                        |
            ...code to get customer balance..        |
        end Get_Balance;                             |

end Billing_Account;
```

A real billing account object would have many more types of data functions, but a simplified example helps illustrate the concepts.

Messages

A message is a request to an object from another object or program to perform one of its methods. It is only through messages that an object may change state or return results. All objects operate by sending and receiving messages. The allowable messages that the object responds to are limited by what is described in the External View part of the object.

If the Ada package Billing_Account represents a billing account object, then messages could be sent to that object using Ada call and function references. To obtain the billing balance, the function reference:

```
Billing_Account.Get_Balance
```

may be used. To create a new customer, the message:

```
Billing_Account.New
```

would be used. The Ada package specification determines what messages are available for manipulating the billing account object. These messages are enforced by the Ada compiler such that there would be no other way to access the Billing_Account package.

Classes

Within the billing account object, "templates" that represent a customer billing account and a billing balance are needed. These "templates" are not unlike a manufacturing factory where a mold or cast is used repeatedly to stamp out many parts. In object-oriented terminology, the mold or cast is referred to as the Class and the individual parts created by that mold as Instances. The code:

```
type ACCOUNT is
    record
        Name     : STRING(1..30);
        Address  : STRING(1..40):
        Amount   : FLOAT;
    end record;
```

represents the class of a billing account for a customer. To create customer objects using this class, the code:

```
declare

    Work_Customer      :   Billing_Account.ACCOUNT;
    Work_Balance       :   FLOAT;

begin

    Work_Customer      : = Billing.-Account.New;
    Work_Balance       : = Billing_Account.Get_Balance
                           (Customer => Work_Customer);

end;
```

can be used to create a new customer object and return a working balance that could be used by other program code. Should the logic of how the balance be computed have to change, only the code for the Get_Balance method in the Billing_Account package body has to be changed. The message used to get that balance by other programs would remain the same.

One problem that still exists with the billing account object is a dependency on the billing balance data type passed between the object and other programs. Currently, the Billing Balance is of type FLOAT. What if the data type needs to be changed to something such as INTEGER? In this case, protection from change for program code outside the object does not exist. All code that accesses the billing account balance must be changed to the correct type, re-compiled, and tested.

It would be more desirable to have a special data type that represents the billing account balance that is controlled by the Billing_Account object but whose structure is inaccessible by other objects and programs. In this way, data type changes can be isolated to only the object that owns them without affecting other objects or program code that uses them.

Ada private types can be used to create classes for types whose structure should remain inaccessible to other objects and programs. We can change the Billing_account package to define the protected class called BALANCE as follows:

```
Package Billing_Account is

    type BALANCE is private
        record
            Names   : STRING(1..30);
            Address : STRING(1..40);
            Amount  : FLOAT;
        end record;

    function New
        (Customer   : ACCOUNT)
        return ACCOUNT;

    function Get_Balance
        (Customer   : ACCOUNT)
        return BALANCE;

end Billing_Account;
```

Data Section

External View Section

```
package body Billing_Account is

    function New
        (Customer   : ACCOUNT)
        return ACCOUNT
    is
    begin
        ...code to create a new account..
    end New;

    function Get_Balance
        (Customer   : ACCOUNT)
        return BALANCE
        is
        begin
            ...code to get customer balance..
        end Get_Balance;

        private
        type BALANCE is INTEGER;

end Billing_Account;
```

Methods Section

Data Section

To create and use an instance of class BALANCE, other programs would use the following:

```
declare

    Work_Balance : Billing_Account.BALANCE;
    Work_Customer :  Billing_Account.ACCOUNT;

begin

      Work_Balance  : = Billing_Account.Get_Balance
        (Customer => Work_Customer);
```

The internal representation of class BALANCE can now be changed without having to make coding changes to other program code using that type.

Inheritance

Inheritance is the ability to specify objects that are similar to other objects but without having to create duplicate code to represent the similarities. Usually, inheritance is used to create objects that behave similarly to other objects but with some differences. An example would be to take our billing account object and create other objects, such as Customer_Account and Corporate_Account, that have similar functions but may compute their billing balances differently. The Billing_Account object becomes a superclass and Customer_Account and Corporate_Account become subclasses that will "inherit" those similar methods. In this way, redundant code to perform similar functions is eliminated; Customer Account and Corporate_Account objects need only define what is specific to their properties.

Ada does not provide direct support for inheritance. However, several Ada language features can be used to perform inheritance-like functions:

- o User Types
- o USE and WITH clauses
- o GENERICS

In the package Billing_Account, both user types, BALANCE and ACCOUNT, represent classes used by other objects and program code.

Because these other programs and objects use the class reference instead of the internal structure, the properties of BALANCE and ACCOUNT can be viewed as being "inherited" by these other programs. In fact, one of the requirements of a class object is the ability of subclasses and other objects to inherit from it.

By using:

```
with Billing_Account;
use Billing_Account;
```

A program may reference a method in the Billing_Account object without reference to the object (or package) that contains it. The program merely specifies:

```
Work_Balance : BALANCE;
Work_Customer : ACCOUNT;

Work_Balance := Get_Balance
    (Customer => Work_Customer);
```

In this way, program dependencies on specific objects can be reduced. If the object Customer_Account is now used, only the USE and WITH statements need to be changed. In this case, a kind of implied inheritance occurs (but not real inheritance) between subclass Customer_Account and superclass Billing_Account. To work, an identical method Get_Balance and user types ACCOUNT and BALANCE must exist in the Customer Account package.

Use of Ada generics can allow for creation of subclass objects that more closely represent true inheritance properties. If we change Billing_Account to:

```
generic
package Billing_Account is
        .
        .
        .
end Billing_Account;
```

A subclass object of Billing_Account can be created with:

```
package Customer_Account is new Billing_Account;
        .
        .
        .
end Customer_Account;
```

The subclass can then be used as follows:

```
with Customer_Account;
use Customer_Account;

declare

    Work_Customer : ACCOUNT;
    Work_Balance : BALANCE;

begin

    Work_Balance : = Get_Balance
        (Customer => Work_Customer);
```

The method Get_Balance can be viewed as "inherited" through subclass Customer_Account, which "inherits" from its superclass, the generic package Billing_Account, which contains the actual Get_Balance logic.

Polymorphism

Polymorphism is the ability to send the same message to multiple objects with the appropriate object responding by types used in the message. In this way, the sender does not have to be dependent on the object that is to receive the message.

Suppose the following is used:

```
with Customer_Account, Corporate_Account;
use Customer_Account, Corporate_Account
```

If each package has the function Get_Balance, a program could issue:

```
declare

    Work_Customer : CORPORATE_ACCOUNT;
    Work_Balance : CORPORATE_BALANCE;

begin

    Work_Balance : = Get_Balance
        (Customer => Work_Customer);

end;
```

In this case, the Get_Balance function in the Corporate_Account package would be invoked. Most conventional languages identify only called functions and procedures by name. Ada determines the package to be invoked based on a function and its parameters and types. The appropriate package can be referenced without having to create dependencies between specific package names and data types. In the above example, if we change to:

```
    Work_Customer : CUSTOMER_ACCOUNT;
    Work_Balance : CUSTOMER_BALANCE;
```

the Get_Balance function within the Customer_Account package would be invoked.

Another way to achieve a kind of polymorphism is to use procedure overloading. In this case, if we had two Get_Balance procedures in the Billing_Account package, one using type CORPORATE_ACCOUNT and one using type CUSTOMER_ACCOUNT:

```
Function Get_Balance
    (Customer : CORPORATE_ACCOUNT)
    return CORPORATE_BALANCE
is
begin...

function Get_Balance
    (Customer : CUSTOMER_ACCOUNT)
    return CUSTOMER_BALANCE

is
begin...
```

procedure overloading would invoke the correct procedure based on the data types in the parameters used with the call reference. True polymorphism is not achieved in Ada because the language requires duplicate functions to be created (as in the above example) to handle each data type that is to be processed.

Conclusions

In general, the following Ada constructs may be used to represent these object-oriented entities:

- Objects - Ada Packages and user types
- Messages - Ada call and function references
- Classes - Ada user types and generic packages
- Inheritance - Ada generics, WITH and USE clauses, user types
- Polymorphism - Ada procedure overloading, call and function references

Ada was not designed to support a pure object-oriented architecture. Some of the above entities work easily with Ada, but others may be somewhat awkward to use. A key element for proper object-oriented design is deciding how granular objects should be to protect against change. Ideally, the answer to this would be to have one object or Ada package for each implementation decision. This would create numerous objects that would have to be managed, controlled, and maintained.

Use of Ada and object-oriented development is rare in the business community. Business applications using these technologies are still few and far between. Businesses tend to avert development risk by buying proven market-accepted technologies. Because of a lack of Ada and object-oriented experience in the marketplace, their use is viewed by businesses as additional risks to development success.

Early candidates for these types of technologies will most likely be situations where the risk of change is a highly critical consideration. The requirements for future complex computing needs will require radical change in how systems are developed and maintained. The evolution of the technologies presented here will eventually determine their usefulness and success in developing future business systems.



Mr. Steinberg is a manager in the Technology Services division of Andersen Consulting. He is currently working in the firm's Practice Services area where he specializes in capacity planning and performance for large IBM mainframes. Mr. Steinberg has worked in research areas of Andersen Consulting that have investigated advanced architectures that will support business practices in the 1990s.

References

[Seidewitz 87]
Ed Seidewitz, Object-Oriented Programming in Smalltalk and Ada, Conference on Object-Oriented Programming Systems, Languages and Applications(OOPSLA), October 1987.

[Booch 83]
Grady Booch, Software Engineering With Ada, Benjamin/Cummings, 1983

[Ledgard 83]
Henry Ledgard, Ada, A First Introduction, 2nd Edition, Springer-Verlag, 1983.

[Goldberg 83]
Adele Goldberg and David Robson, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.

Procedures for Assessing Ada CASE Tools For Tactical Embedded Systems

Arvind Goel

UNIXPROS Inc.
Colts Neck, NJ

John T. LeBaron

US Army CECOM
Center for Software Engineering
Advanced Software Technology
Ft. Monmouth, NJ

Abstract: *Computer-Aided Software Engineering tools (CASE) for tactical embedded systems that support Ada are beginning to play an increasingly important role in the software engineering process. The proper selection of tools can tremendously benefit organizations involved in the development of tactical embedded systems. This paper presents a selection criteria for use during the assessment of CASE tools that support Ada. Based on those criteria, a procedure has been defined for the assessment of CASE tools that support Ada in order to enable an organization to select the most adequate tools or toolset for their embedded applications.*

1.0 Introduction

Computer-Aided Software Engineering (CASE) tools for Ada are playing an increasingly important role in the software engineering process and the proper selection and use of these tools can tremendously benefit organizations involved in the development of tactical embedded systems. Due to the ever increasing number of currently available Ada CASE tools, it is imperative for an organization about to purchase a tool to gain an understanding of these tools and assess their attributes. In this paper, selection criteria for use during evaluation of Ada CASE tools has been developed. Based on those criteria, a procedure has been developed for the assessment of CASE tools in order to enable an organization to select the most adequate tools or toolset for their embedded applications [10].

2.0 Criteria For Ada CASE Tools Assessment

Ada CASE tools can be assessed individually as well as in the context of the overall computing environment in which they will operate. Individual Tool evaluation deals with understanding the functionality and assessing the attributes of CASE tools in a stand alone environment.

Since an effective CASE environment is made up of many tools operating together to increase software productivity and quality, it is necessary to assess each tool not only on its individual merits, but also within the context of it's overall computing environment. How well these tools complement each other, how easily they interface to share data, and whether they communicate with a common development database are key determinants of overall effectiveness. The following sections presents a comprehensive set of Ada CASE tool selection criteria. The sections are in the format of a section heading, question which should be addressed when assessing tools, followed by an explanation, when necessary.

2.1 Life cycle coverage: *Which phase of the software development life cycle does the tool fit in: requirements generation, analysis, design, code, test, or maintenance?*

Many Ada CASE vendors, that address embedded system issues, offer a complete toolset that is comprised of individual tools, each of which may address a particular phase of the software development life cycle. These tools work in cohesion with each other and more often than not will not work with other vendor's tools. Due to the fact that Ada CASE tools for embedded systems are still evolving and that there are no set interface standards for inter-tool communication, an organization may decide to go with a vendor that offers a full Ada CASE environment rather than trying to integrate individual tools from different vendors.

2.2 Hardware/Software Platform: *Which hardware/ software platforms does the tool/toolset support? Does the tool support both a mainframe (centralized development system) and workstation environment?*

For development of tactical embedded systems, the trend is moving towards workstations networked with each other as well as networked with a mainframe computer. This is due to the fact that prices of workstations are coming down which allows for the addition of increased processing power at little extra cost. However, a major factor which must be considered is the availability of existing resources.

2.3 Support for Requirements Tracing

Requirements traceability is an extremely important activity for a successful development. Ada CASE tools/ toolset must help ensure that the systems meet the specified requirements. Given a particular requirement, the tool should show where the requirement originated, which design component(s) address it, where the approach to meeting the requirement is documented, and where and how it was tested. An example of the importance of requirements tracing to improve the quality

of software is the Department of Defense's (DoD) MIL-STD-2167A and 2168. Due to the need for extremely high quality and reliability in many computerized defense systems, such as battlefield command and control systems and combat flight systems, DoD has mandated the use of requirements tracking for all software used in such systems.

2.3.1 Requirements Tracing to Design, Test, and Documentation: *Does the tool have the capability of tracing the requirements to system design, test, and documentation?*

This is essential because once the system has been developed and tested, the developer is faced with the task of proving to the users that the system meets all of the requirements in the maintained baseline. To maintain the baseline, the requirements management tool must be able to track correspondence of requirements to design structures and components, test suites, and documentation outlines.

2.3.2 Requirements Tracking Enforcement: *Does the tool enforce requirements tracking?*

Some project managers prefer a system that absolutely will not allow them (or anyone else) to overlook a requirement or to skip a formal or informal test. Others, however, may resist using a system that does not provide flexibility in its use.

2.4 Analysis and Design

The criteria for assessing analysis and design tools have been considered together. This is due to the fact that many criteria overlap with each other because analysis and design activities are closely coupled as are the tools which support them.

Analysis: Analysis deals with the "what" of system development. At this phase, the textual system requirements are analyzed and the output is a structured specification that explicitly defines the system requirements with graphics, text descriptions, and a complete data dictionary.

Design: Design deals with the "how" of system development. At this phase the output from the analysis phase is evaluated to determine how a module will function, its relationship to other modules, and how it communicates with them. The outputs of this phase are less abstract high-level programs describing hardware and software technology to implement the system.

Ada CASE tools that support tactical embedded systems analysis and design can be compared in several ways. The assessment criteria have been divided into a number of logically separate areas.

2.4.1 Analysis/Design Techniques and Methodology

2.4.1.1 Specific Representations and Methodologies: *What specific representations are used for the representation of data, process and control during the analysis phase? What methodologies are used? Does the tool have real-time modeling capabilities?*

Different methods emphasize different aspects of analysis and design as appropriate to the intended application. As a result CASE vendors usually provide several options or a facility to customize the toolset to any variant of a basic method.

In general, the major extension for embedded systems design commonly found in Ada CASE tools today is behavioral modeling, that is, the ability to express control algorithms that define how the system will react to its environment under various conditions. Consequently, the complete design representation for an embedded system at the analysis stage can be viewed as having three dimensions: data, process, and control. Real-time extensions add notations to describe control mechanisms and synchronization with external events. Control mechanisms include:

- Languages with embedded state diagrams (Ward-Mellor, Hatley, ESML, StateChart/ActivityChart).

- Languages with internally synchronized interprocess communication (IORL).

- Language-specific or operating-system-specific inter-task communication constructs (text-based high-level languages, Pamela, Buhr diagrams).

In the design phase a variety of different methodologies are used (Constantine, Myers, Warnier-Orr, Jackson, SADT, Page-Jones, etc.) but the basic objective is to partition the system into a set of single-function modules with well-defined data and control interfaces. The essential capabilities common to most toolsets include methods to define program structure, module definitions, module data and control interfaces, calling relationships, data coupling, program logic, algorithms and performance specifications.

2.4.1.2 Integration of process, data, and control representations: *How are alternate representations for process, data and control integrated?*

The multiple dimensions of the design representation is one of the fundamental difficulties in real-time embedded design. The expression of these three modes is not fully integrated into a single design document in any current technique. Typically, there are multiple design views, each combining aspects from two of the three modes. For example, a data flow diagram depicts aspects of data and process representations. Control flow diagrams mix process and control considerations. Often the data and control flow diagrams are displayed together, while the details of the control algorithms are expressed in a separate representation, such as a finite state machine or state transition matrix.

While many CASE vendors have incorporated the Ward-Mellor and Boeing-Hatley notations, neither of these approaches address all the design decisions needed to fully-specify the behavior and performance of a generalized embedded system. A real-time technique called StateCharts has been developed that can be used to represent control, concurrency, timing, and sequencing and provides a robust set of features to manage the

complexity of tactical embedded systems.

2.4.1.3 Concurrent Tasks at Design Phase: *Does the tool allow for the specification of concurrency (tasks, task prioritization), critical path timing, queuing effects etc. at the design phase?*

In many cases there is a wide gap between the level of abstraction represented by data flow diagrams, control flow diagrams, and state transition diagrams, and the detailed design description necessary to begin coding. Consequently, the designer is forced to introduce elements such as tasks, task prioritization, critical path timing, queuing effects, etc. through manual extensions to the CASE design model, or worse by incorporation at the coding level. Because these low-level details have a critical effect on performance, and hence the adequacy of the design solution, they cannot be divorced from the abstract design of tactical embedded systems.

2.4.1.4 Architectural Representations: *Does the design tool generate an architectural representation (diagram of the design modules) of the system?*

Many Ada CASE design tools create a graphical representation of the system. They can portray graphical pictures of the system at any level: the total system structure, individual modules, or single functions. This can help the user in determining where potential bottlenecks may be evolving or what areas are not yet complete. Development costs are drastically reduced by the ability to view potential pitfalls and unnecessary complexities before implementation begins.

2.4.2 User Interface and Editors

2.4.2.1 Type of User Interface: *What type of user interface is used: textual or graphic?*

Graphical user interfaces along with the capability to enter text when needed provide the best user interface. Graphic representations are easy to understand and along with the textual inputs, a user can enter specific information about objects, e.g. a process. Most of the CASE design tools provide graphical methodologies for representing proposed systems design.

2.4.2.2 Mouse, menu driven, and windowing: *Does the tool use a pointing device such as the mouse or is the tool menu driven. Is the tool capable of executing with windowing capabilities?*

Although some users prefer a mouse driven interface and others prefer a menu driven interface, both are desirable and should be available. An advantage of the windowing capability is that multiple portions of the design can be displayed simultaneously and can therefore compensate for weaknesses in embedded explosion capabilities. As a result, the levels for explosion will not be restrictive and the comprehensiveness and integration of CASE design and development specifications should improve.

2.4.2.3 Intelligent Text and Graphics Editors: *Does the toolset have intelligent graphics and text editors?*

Many CASE tools include powerful graphics editors to improve the user interface and to transform the software development into a visual process. The information entered by the analyst or designer includes descriptions of the functions or processes, data entities and their relationships, and in embedded applications, the control mechanisms and timing required by the system. Besides simply allowing the user to draw diagrams or type text, the toolset's graphics and text editors have knowledge about the semantics of the particular structured analysis or design methodology being used. These intelligent editors may prompt the user for needed information, restrict the input to eliminate syntactical and obvious semantic errors, and automatically fill in data that can be derived from previously entered information. For example, an editor may automatically provide hierarchical process numbering and the required input and output dataflows whenever a lower-level diagram, or child, is created.

2.4.3 Static Model Analysis: *Does the tool support static model analysis to detect and report errors such as data elements that are never used, and inconsistent naming of data elements?*

This concerns the capacity of the Ada CASE tool to analyze design documentation and determine if the specifications entered by the analyst conform to prescribed methodological rules. The analysis should also indicate where design dictionary entries are incomplete. For example, a DFD with a free-standing block should be highlighted as violating one of the rules of structured methodology. In addition, blocks on a DFD, not having a corresponding dictionary, should be highlighted. Thoroughness of checking should include:

- In-document - checking for consistency within a single representation style
- Cross-document - checking one representation (e.g. control flow) against another (e.g. state transition diagram)
- Cross-project - checking documents from different development team members within a central project database.

2.4.4 Dynamic Model Analysis

Most Ada CASE tools commercially available for tactical embedded systems design, model certain aspects of tactical embedded systems, but a user can do little with the resulting description beyond testing them for syntactic consistency and completeness. Dynamic model analysis provides a mechanism that can allow the user to validate the behavior of the system under development early in the project.

A sufficiently complete analysis or design model is a program in a very high level language. Dynamic model analysis exploits this fact by executing the entire model or a portion of the model thus simulating the behavior of the system. An executable specification is a rigorous graphic description of the system's behavior and its software that can be executed. Before a line of code is written, the system can be simulated to see how its software and hardware modules work together. Combinations of

simulated external stimuli can be thrown at the system to see how it reacts, where internal conflicts arise, and where the bottlenecks that could affect performance exist. Dynamic model analysis includes:

2.4.4.1 Scenario-based Model Execution: *Does the toolset allow the analysis/design specification to be executed dynamically with user-specified sequence of inputs as input data?*

In this kind of dynamic model analysis, the specification of the system under development is run under a wide range of user specified data. This allows the user to simulate the actual system under development (including external events), check the specification for time-critical performance and efficiency, and in general to debug it and identify subtle run-time errors.

2.4.4.2 Exhaustive Model Execution: *Does the toolset provide for exhaustive, brute-force sets of executions to test crucial dynamic properties such as reachability, nondeterminism, deadlock, and potential race conditions?*

In this kind of dynamic model analysis, the specifications are executed to test for some of the crucial dynamic properties of the system under development - those the user desires to satisfy as well as those the user does not. These include reachability, nondeterminism, deadlock, and potential race conditions.

2.4.4.3 Man-machine Interface Prototyping: *Does the tool provide a means of prototyping man-machine interface to resolve human factors issues?*

In this type of analysis, a) analysis or design data structures are assigned to screens and b) data element ranges are interpreted as editing rules and a potential end-user may use a mock-up of a proposed screen. This can help in simulating the man-machine user interface.

2.4.5 Reusability: *Does the tool support reusable components?*

Due to the enormous cost involved in the development of software for tactical embedded systems, there is a major thrust in the CASE industry for reusable components. Ada CASE tools should enhance reusability such that in design and development specifications and code for one system may be reusable in the design and development of future systems.

2.4.6 Analysis/Design Reports: *Does the tool have the capacity to generate analysis/design specification reports automatically?*

The specifications created during logical and physical design activities serve as a source of documentation for the system. While they are permanently stored on disk devices, it is often advisable to get hard copy printouts of the design specifications for reference.

2.5 Database

Beyond simply capturing requirements and design specifications, analysis and design tools organize design elements in a number of ways and provide rapid access to them from a design database, alternately referred to as data dictionary, design repository, or encyclopedia. Typically, either a relational or object-oriented database, the design repository acts as the core for the set of modules that make up the toolset.

2.5.1 Centralized Database: *Is the database centralized to allow several users consistent views of system information? Does the database allow multi-user access and control?*

A typical tactical embedded project involves many different developers working on many parts of the project at the same time. It is essential that the database be centralized so that the different developers get consistent information from the database. Also, the database should allow access by several users at the same time and have control on updating of information in the database.

2.5.2 Facilities For Entering Information To Database: *Does the database allow the user to enter additional information such as screen and report formats, status and audit information, personal notes, review comments, test results, source code, code execution time, cost and schedule estimates and so on?*

Many CASE tools provide a mechanism that can allow a user to enter additional information in the database. For example, during the analysis/design phase of tactical embedded systems, a user may need to enter details that do not normally show up in the graphics, such as lengthy definitions of compound events and conditions or design rational.

2.5.3 Database Openness to External Access: *What is the degree of openness to external access? Can the data base be utilized by other tools?*

Some database only allow access through the toolset itself, thus limiting the views of the design data to those predetermined by the CASE vendor. Design databases with an open architecture allow access through well-defined interfaces so that an organization can use additional tools to access the database.

2.5.4 Query Capabilities: *Are there query capabilities that allow an analyst or designer to view a selected portion of the total system?*

Many CASE tools provide a query language with which the user can retrieve information from the database, thus effectively querying the model of the system under description as described in the modeling languages.

2.5.5 Access Requirements Information: *Does the database allow the user to access requirements information based on several database keys, such as system requirements, functions, and keywords?*

This capability is useful in tracking requirements throughout the life cycle. It is also useful in assessing the impact a change in one requirement would have on the rest of the system.

2.6 Code Generation

At the end of the implementation phase, a user has a specification of each program module from which coding can be started. It is at this point in the design sequence that one finds big gaps in the Ada CASE technology. An ideal toolset would be able to access all the design data that has been created and from it, automatically generate a complete set of Ada production code that would implement the design. Ada CASE vendors addressing the needs of embedded systems are creating their own integrated work-benches based on a proprietary design technique or program specification language. Even the new tools being built specifically for engineering and embedded systems cannot yet generate 100% of the Ada production code required for a complete system. They still require the user to develop some portion of the code by hand.

2.6.1 Design Issues

2.6.1.1 Traceability: *How much does the code generation process aid in the ability to trace module designs, interface specifications, system components, and objects to the code and vice-versa?*

This is a key requirement for useful Ada code generation in the engineering environment where implementation must be validated against requirements. If Ada CASE tools are used during the analysis/design phase, then gains are realized when the generated code is related back to the higher-level design specifications. All code generator vendors make some attempt to tie the design elements with actual code modules. This link between design and code facilitates verification and validation, two important software quality assurance techniques, and also simplifies software maintenance.

2.6.1.2 Information feedback from coding phase to design specifications: *Does the code generator automatically feed back information about the coding phase to the design specifications?*

An Ada code generator that offers requirements tracing capability automatically indicates and updates where in the design specification a new code segment was injected. Thus, until code generators for the engineering environment can reliably create 100% of the required code, it is paramount that the manual implementation efforts of programmers be reflected back into the system design as well as into documentation. To emphasize this requirement even further, defense contractors are now mandated by DoD-Std 2167A to trace requirements through all phases of the life cycle in all mission critical systems.

2.6.2 Prototyping Support

Prototyping is the process of developing a scaled-down version of a system to use in building a full-scale system. The consequences of not building prototypes in tactical embedded projects are many. Software projects are often late and the end product fails to meet the design goals. A prototype would help estimate the size of the project and ensure that the requirements are met.

2.6.2.1 Prototype Code Execution: *Does the generator support prototyping code execution to evaluate functional adequacy and assess potential performance problems?*

In prototype code execution, code in the target high-level language is created from the design model. It is possible that many detailed considerations are hidden from the user to make it easy to specify complex software. The code may be run in the development environment, in the target environment, and to evaluate functional adequacy and to assess potential performance problems. Feedback from prototype code execution can be used to further refine the design and generate better performing, more optimized code.

2.6.2.2 Performance Simulation Code Execution: *Does the generator support performance simulation code execution to highlight potential timing problems in the proposed system?*

In this type of analysis, special code to do a queuing network simulation of the proposed system is created from the model. This provides the ability to assign timing characteristics to individual modules and model the overall system performance. Execution of this code can highlight potential timing problems in the proposed system.

2.6.3 Support for Embedded Systems

2.6.3.1 Cross-Development: *Does the generator support cross development?*

This question is of particular importance if a user is designing embedded systems, because the target machine may be very different from the host on which the Ada CASE tools run and some code generators generate code only for the host or a very similar machine.

2.6.3.2 Code Concurrency: *Is the code produced by the code generator concurrent*?

Code produced by a code generator may be:

Non-concurrent, produced by a purely hierarchical design model

Concurrent, produced from a design model with network characteristics

Hierarchical code is of only limited usefulness in a tactical embedded environment. For tactical embedded systems, a model that views systems as collections of concurrent, prioritized tasks is needed. These tasks can be scheduled for periodic execution or can be triggered by asynchronous events.

2.6.4 Percentage of Finished Code: *Does the code generator produce Ada code frames or the final Ada production code? If it produces production code, what percentage of the finished code can the generator produce*?

Code generators must at least be able to produce Ada code frames from a graphic or textual design model.

2.7 Correlation and Testing

When the coding of the modules is completed - automatically or by hand - the straightforward part of the job is over. Now the task of correlation and testing begins.

Correlation is the process of verifying that every function in the system requirements is implemented by some program module.

Testing includes performance-analysis procedures to ensure that the code executes and also that every part of the code is executed at least once. The real-time requirements for new systems have pushed the burden of verification onto run-time software testing. Even with a formal design created by structured analysis and design tools, the validation of performance objectives is usually addressed during testing in the target hardware environment. Embedded software must meet the same fail-safe reliability expected of hardware. The challenge for testing is amplified because software problems are often related to performance rather than logical structure, and latent performance problems often remain in the code, waiting for the right set of circumstances to cause a malfunction.

2.7.1 Testing Against Baseline Requirements: *Does the testing tool test against the complete baseline requirements and does it provide test cases to exercise every requirement for 100% function coverage?*

2.7.2 Test Case Design: *Does the tool automatically design test cases, or help the user design them in a computer-aided mode?*

2.7.3 Test Case Execution: *Does the tool support test execution by providing an on-line, pass/fail recording facility?*

2.7.4 Boundary Conditions: *Does the test tool generate test cases that test the boundary conditions of the software?*

2.7.5 Timing Deadlines Testing: *Are the time-critical portions of the software tested to make sure that a real-time system can meet its timing deadlines?*

2.7.6 Stress/Load Testing: *Are stress and load test cases developed by the testing tool?*

2.8 Re-Engineering and Maintenance

Today, although Ada CASE tools are being developed for new applications, most of the software professionals work on existing systems that were developed without the use of CASE tools. These systems not only need to be maintained but in a lot of cases need to be modified as the system evolves over its lifetime. There is a scarcity of CASE tools that help in the maintenance and modification of existing systems.

In order to perform maintenance and modification on existing systems that were developed without the use of CASE tools, it may be necessary to re-engineer those systems. A major difference between re-engineering and maintenance tools is that maintenance tools enable a programmer to fix parts of code in order for the code to do what it was originally intended to do. Maintenance tools are more interactive and deal more with the code level. On the other hand, re-engineering tools provide a system description to enable an existing system upgrade or modification to future versions of the program. Re-engineering tool selection criteria are as follows:

2.8.1 Level of Re-Engineering without Introducing Ambiguity: *How far back or how high can the re-engineering tool go in describing the structure, content and meaning of the code without introducing ambiguity?*

For example, some tools mechanically extract all the details in the code and display it in a more structured representation, whereas others interpret information in the code and present it in a more abstract format leaving out distracting detail. No current re-engineering tools are capable of determining a complete high-level specification (data flow or E-R diagrams for example) from the code without some level of human intervention. In general, the higher the level of abstraction required, the more human assistance is required to remove ambiguities in the code.

2.8.2 Completeness of Design Notation: *How much of the information detail of a particular design notation is recaptured from the code?*

For example, if a re-engineering tool creates a Yourdon-DeMarco representation of the code, does it include all aspects that would have been used to generate the code during forward engineering (leveled data flow diagrams, data dictionary and mini specifications), or a subset of the specification.

2.8.3 Human Interactiveness: *How much interactiveness does the tool allow?*

Interactiveness refers to the amount of influence the user has on the process of generating the specification that represents the existing code, and what facilities are provided to manipulate that specification.

2.9 Documentation

All the fundamental data necessary for proper documentation of a project is contained in the central design database originally created by the front-end tools. This data is in the form of drawings and data-element specifications, process specifications, interface specifications, and program-module specifications.

Documentation tools can present integration problems, partly because they are in themselves complex systems that may be too large to be included in a CASE toolset, and partly because the best ones come from non-CASE vendors and may use an internal format incompatible with the one the CASE toolset uses.

2.9.1 MIL-STD-2167A Compatibility: *Does the tool produce documentation that is compatible with required standards?*

Some Ada CASE tools provide documentation that fully conforms to the standards, others do not. For DoD

applications, the MIL-STD 2167A and 2168 describe the rules for developing software for "mission critical" systems and tools intended for DoD applications must support these standards.

2.9.2 Automatic Generation of Design, Operations and End-User documentation: *Can the tool automatically generate design, operations, and end-user documentation? Does there exist a capability to combine graphics and text on a single document?*

As systems are designed and developed with Ada CASE tools, documentation concerning components and users of the system are entered into the dictionary. Thus, the majority of design, operations, and user documentation required for documentation manuals is available from these dictionary entries. CASE systems should provide this documentation in either on-line or hard copy form, with little additional work required from the project development team.

2.9.3 Interface with Standard Desktop Publishing Systems: *Does the tool interface with standard desktop publishing systems?*

2.9.4 Exception Reports showing Unmet Requirements: *Does it have the ability to produce exception reports showing unmet requirements and model segments not matched to requirements?*

2.10 Support for Ada

Tools that are built for commercial applications in mind generally do not have the capabilities to support the development of scientific and DoD applications. Also, these tools generally support Cobol programming, whereas scientific and DoD applications require either C or Ada. With the advent of Mil-Std 1815A which mandates the use of Ada on DoD projects, Ada has become the predominant language that many vendors are trying to support.

2.10.1 Ada Specific Constructs: *Does the tool have Ada language-specific inter-task communication constructs and other constructs that represent structures within the language during the design phase?*

For example, text-based high-level languages, Pamela, Buhr diagrams. These languages use text constructs or equivalent graphic symbols or icons that represent structures within the language, for example Ada tasks and packages. This makes the task of representing a program structure in a Ada-like fashion much more easier, and the transition from design to Ada code is much smoother.

2.10.2 Code Concurrency: *Does the tool/toolset support concurrency and parallelism?*

For tactical embedded systems coded in Ada, a model that views systems as collections of concurrent, prioritized tasks is needed. These tasks can be scheduled for periodic execution or can be triggered by asynchronous events. Data and control flows passing between tasks should be handled such that all tasks can be safely computing in parallel. Hence, concurrent code represents applications that can be targeted to uniprocessor environments as well as multiprocessing and distributed environments.

Many CASE tools that support Ada enable users to specify system design in terms of Ada design elements such as tasks so that Ada code generation from such a design is straightforward. This enables one to minimize dependencies between components, perfect high-level design , isolate reusable components, and avoid race conditions and deadlock between tasks, before committing to implementation details

2.10.3 Low-level Design Information: *Does the tool allow the software engineer to provide low-level design definitions via a specification language, Ada compatible program definition language, target hardware description libraries, and/or a library of previously coded low-level routines?*

The full-code generators usually require that a user define hardware-specific features of the target system by means of a program-definition language (PDL) and that the user supply target hardware description libraries and a library of previously coded low-level routines. Many CASE tools also provide the facility so that an Ada compatible language can be used to insert virtual code in the generated code by the user.

2.11 Project Management

Project management spans all phases of the software development life cycle. It is essential that the software development activity be managed effectively otherwise the project is more than likely to fail. The essential activities performed during project management include determining cost, time, and complexity estimates; monitor process flow, trigger milestone reviews, and management reporting. The output from this process includes cost and schedule estimates, complete project history, project status, deliverables status, and deviation reports.

2.11.1 Project Planning: *Does the tool help in doing project planning. Does it define personnel, resources, work packages, and schedules. Do the tools in the toolset have the combined ability to manage and control the actual deliverables produced by the project and to allocate and manage the resources needed to build the system?*

2.11.2 Status Reporting, Progress Tracking, Resource Usage and Cost Tracking: *Does the tool do status reporting and progress tracking. Does the tool perform resource usage and cost tracking?*

At any stage of the development life cycle, it is essential for managers to be aware of the status of the project. If the project is behind schedule, then management has the responsibility to investigate where the problem is and add additional resources as needed. In order to facilitate day-to-day supervision of a development project, an Ada CASE environment must:

- collect data on "who did what to which part of the model when"

- provide standard reports on current model contents and changes to the model since a given point in time

• provide query facilities to allow ad-hoc reporting on the model

2.11.3 Milestones Tracking and Critical Path Analysis: *Does the tool track milestones and perform critical path analysis?*

Some Ada CASE design and development tools can generate reports on the progress of individual project assignments and some can interface to existing project management software systems. Currently, this interface is a temporary exit from the CASE tool into the project management system, but the interface will become much stronger in the future and provide more automatic updating of the project schedule.

2.12 Configuration Control

Many tactical embedded systems involve:

• Multiple releases of a system with additions of functionality in later releases

• Multiple versions of a system for different target environments

• Complex integration testing with changes to code being made simultaneously by many developers

Configuration management tools provide features to assist in managing these situations. The configuration-control process begins during the design and implementation phase, and continues into the maintenance phase.

2.12.1 Multiple Versions/Generations of a Project: *Does the tool track multiple versions/generations of a project?*

2.12.2 Tracking of Revisions to Code: *Does the tool track revisions to units of code?*

2.12.3 Reconstruction of Previous Versions: *Does the tool reconstruct previous versions on demand?*

2.12.4 Changes Across Configurations: *Does the tool allow for selective changes to be applied across configurations thus resulting in labor savings?*

2.12.5 Difference between Two versions of a Code unit: *Does the tool help in determining the differences between two versions of a code unit?*

2.12.6 Integrity and Security: *Does the tool define different levels of authority and functional responsibility for different users ensuring integrity? Does it control who may have access to what information ensuring security?*

2.13 Common Work Environment Support

A typical tactical embedded project involves many different developers working on many parts of the project at the same time. Ada CASE environments that deal with this situation include toolsets that allow a number of developers to work on a model concurrently. Criteria to judge for support of common work environment include:

2.13.1 Team Development: *Does the tool support team development by permitting distribution of design/development responsibilities?*

Ada CASE design and development tools must provide a serviceable means of segregating job responsibilities and interfacing the individual efforts into a single system project. For example, team development can be supported through the decomposition of a system into a hierarchy of subsystems, each of which may be developed independently and then integrated.

2.13.2 Exchange Portions of Design Models: *Does the toolset allow developers to exchange portions of design models?*

If an Ada CASE tool allows developers to exchange portions of design models as well as gives instant access to all portions of a design, a user can evaluate and correct the interfacing information that will be needed to ensure that his/her section integrates accurately and smoothly to the other portions of the design.

2.13.3 Concurrent Access of an Analysis/Design Specification: *Does the toolset allow more than one developer to concurrently access the same copy of a analysis/design specification?*

Again, when different developers working on many parts of the project at the same time, it is quite possible that more than one developer is concurrently accessing the same copy of an analysis/design specification. The toolset should allow this while providing safeguards to maintain model integrity.

2.13.4 Central Repository for Design developed on Multiple Workstations: *Does the tool provide a central repository for a design model being developed concurrently on multiple workstations?*

It is essential that there be a central repository that contains all the information about a design model being developed by many developers on multiple workstations. This central repository contains the most up to date "official" version of the complete design model and serves as the central repository from which users can access this design model when needed.

2.13.5 Networking: *Does the Ada CASE toolset use the communication facilities provided by the workstation or host operating system such as Ethernet, TCP-IP, NFS, Domain, DECnet, SNA etc.?*

Generally, in a team CASE environment (where a number of people are working on a design simultaneously) the project design database resides on a central mainframe or on a file server within a local area network. The CASE toolset should be able to use networking facilities to communicate with other components (e.g. the database) resident on another workstation.

2.13.6 Export Data: *What facilities are available for exporting data from one CASE tool to another CASE tool from a different vendor?*

2.13.7 Open Architecture: *Does the tool's manufacturer have an open architecture philosophy? What are the external tools with which the toolset interfaces?*

The arguments for an open architecture are powerful. Few Ada CASE vendors support the entire life cycle at present, and experience in other CAD environments, such as computer-aided electronic engineering (CAEE) and computer-aided manufacturing (CAM) has shown that users seldom can meet all their requirements with products from a single vendor. Open architectures make it easier for the user to assemble a comprehensive Ada CASE environment by integrating several products.

3.0 Procedures for Assessing Ada CASE Tools

Checklists and evaluation guides for Ada CASE products often list the key elements that a user seeks within a CASE tool. Although that approach suggests that there's a complete set of product attributes constituting a model or ideal Ada CASE toolset, in reality, the ideal set of attributes differs from one group of users to the next, depending on their software development requirements. Different market segments require different tool capabilities, interfaces, methods and hardware platforms. Users' needs become specialized by their own development activity and environment.

The procedures for assessing an Ada CASE tool involve the following steps [1]:

- Needs and Environment Analysis
- Define the CASE Requirements
- Contact vendors for information
- Develop a Preliminary list
- Trial Use and Evaluation
- Apply Criteria and Purchase Tool

3.1 Needs and Environment Analysis

3.1.1 Needs Analysis: The first step in selecting any Ada CASE tool is that users need to look at their organization's needs and current practices and then use that reality as a guide for evaluation of CASE tools. Every organization that produces software has, in effect, an operational system to build systems. This operational system has to be analyzed and modeled in order to determine which areas are most appropriate for the introduction of CASE tools.

In order to create a model for existing software development practices in an organization, structured analysis and design methodology such as the data flow techniques developed by Yourdon and DeMarco can be used. Using data flow diagrams, systems development can be viewed as the transformation and communication of design data.

This model helps define how an organization typically captures requirements and transforms them into computer software using the organization's current technology. It identifies which portions of the process are good candidates for automation, given the available CASE tools. Using this model, an organization can identify the problems that exist in its current environment and develop a list of the types of the tools that can be purchased or developed to solve the problems identified.

3.1.2 Environment Analysis: Along with the needs analysis, an organization should also perform an analysis of the hardware/software environment in which the tool will operate. For example, it is useless considering a tool that works in a VAX/VMS environment for a UNIX environment. What computers does the organization use (mainframe, workstation etc.)? Does it use multiple hardware platforms? With what operating systems are the personnel of the organization most familiar? These question must be answered before one can even begin seriously looking at available tools.

3.2 Define the CASE Requirements

The needs analysis process identifies the areas which are good candidates for automation based on an organization's current software development practices. Besides identifying the areas which need to be automated, it is essential to define the CASE requirements. These requirements are based on the needs of the developers and can be categorized into hard and soft requirements. Hard requirements are an absolute must for the organization and CASE tools that can not meet the hard requirements have to be rejected. Soft requirements are wish-list items or are items that may satisfy future needs. The CASE requirements are based on the criteria that have been discussed in this report for each phase of the software life cycle.

3.3 Contact Vendors For Information

After performing needs analysis and defining the CASE requirements, the next step is to contact vendors for information or to attend one of the CASE conferences or expositions. There are several publications within the CASE industry that provide addresses, telephone numbers, brief product descriptions, and other information on Ada CASE tool vendors.

3.4 Develop a Preliminary List

Using the information received from the vendors and the published Ada CASE tool evaluations and comparisons, select the vendors, that, on this first appraisal, appear to best meet the results of the needs and environment analysis phase, as well as the CASE requirements. Again, a number of tools can be screened out based on other constraints such as cost of the tool.

3.5 Trial Use and Evaluation

Contact the vendors of the tools that have been selected in the preliminary list. A first step would be to send them a copy of the CASE requirements and ask them to respond in writing to each item listed. Vendors who refuse to respond in writing to the written CASE requirements should be eliminated from further consideration.

Evaluate the written responses based on the criteria

developed in this report and select the final list. Contact the finalists and ask for a 30 to 90 day loan of a copy of their Ada CASE tool product for trial use.

3.6 Apply Criteria and Purchase Tool

The final step in the tool selection process involves the application of the criteria that have been developed in this report during the trial use of the product. During the selection step, technical and management staff define a set of criteria for each software engineering element of the tool. Based on an organization's needs, the relative importance of the criteria can be different. It has to be kept in mind that the criteria that have been discussed in this report may have a different degree of importance depending on who is conducting the evaluation as well as the needs of the organization. A candidate Ada CASE tool is considered in the context of these criteria and is "graded" in relationship to other candidates. Again, it needs to be reemphasized that tool selection is highly subjective and is totally dependent on an organization's needs.

4.0 Conclusions

By applying the criteria and following the steps outlined in this paper, a software development organization can assure that it selects an Ada CASE tool that is appropriate for the type of software that it develops, its physical development environment, and for its software development culture.

It has to be kept in mind that just when an organization has zeroed in on a selection, a new and superior tool comes on the market. If you constantly look at all the new tools before selecting one for use, the process can go on forever. Selecting tools for software engineering is complicated by the options available for each tool and the fact that a lot of selection criteria are subjective. However, the productivity and quality gained by using the Ada CASE tools far outweigh the effort involved in the selection of the appropriate CASE tools.

For real-time systems, software engineers need more than conventional structured analysis, design and code generation tools. Ada CASE vendors are and must keep moving ahead on extensions for behavioral modeling, architectural modeling, explicit concurrency notations, performance analysis, simulation, and rapid prototyping.

References

1. Robert Firth et al., "A Guide to the Classification and Assessment of Software Engineering Tools", Technical Report CMU/SEI-87-TR-10, ESD-TR-87-111, Software Engineering Institute, August 1987.

2. Case Outlook, published by Case Consulting Group, Portland, Oregon, November, 1987.

3. CECOM Center for Software Engineering, US Army, Ft. Monmouth, NJ, "Evaluation of Existing CASE Tools For Embedded Systems", Technical Report, May 1989.

4. Tom DeMarco and Tom Lister, "Peopleware: Productive Projects and Teams", Dorset House Publishing, New York, 1988.

5. G. Booch, "Software Engineering with Ada", Benjamin/Cummings Publishing, 1983.

6. Carma McClure, "CASE is Software Automation", Prentice Hall, Englewood Cliffs, NJ, 1989.

7. Roger Pressman, "Making Software Engineering Happen", Prentice Hall, Englewood Cliffs, NJ, 1988.

8. Albert Case, Jr., "Information Systems Development: Principals of Computer-Aided Software Engineering", Prentice-Hall, Englewood Cliffs, NJ, 1986.

9. Paul Ward and Stephen Mellor, "Structured Development for Real-time Systems", Prentice-Hall, Englewood Cliffs, NJ, 1985.

10. CECOM Center for Software Engineering, US Army, Ft. Monmouth, NJ, "Procedures for Computer-Aided Software Engineering Tools Assessment", May 1989.

About the Authors:

Arvind Goel received his B. Tech. degree in Electrical Engineering from IIT, Kanpur, India in 1980 and his MS degree in Computer Sciences from the University of Delaware in 1982. He is the founder of Unixpros Inc. where he is working on developing composite benchmarks to model a class of real-time systems. He is also working on CASE tools and their application to real-time systems. His interests include programming languages, Ada compiler evaluation, APSE research and evaluation, and developing software for embedded application and distributed targets.

John T. LeBaron is a Software Engineer with CECOM, Center for Software Engineering, Advanced Software Technology, Fort Monmouth, NJ. He received his BS in Computer Sciences from Kean College in 1974. He is currently the Acting Chief of the Software Engineering Technology Program investigating the process, methods and tools used to create and evolve software for Mission Critical Defense Systems.

CASE TOOLS SUPPORTING Ada REVERSE ENGINEERING: STATE OF THE PRACTICE

M. Cassandra Smith Diane E. Mularz Thomas J. Smith

The MITRE Corporation, McLean, VA

Abstract

The purpose of this paper is to discuss the methodology and results of a study of Computer-Aided Software Engineering (CASE) tools that reverse engineer Ada programs. The paper discusses a list of features that support reverse engineering with Ada, criteria for interpreting vendor information on tools to identify candidates for further assessment, the results of exercising four candidate tools with sample Ada programs, and observations on the state of the practice of reverse engineering Ada in the CASE tools industry. The predominant finding is that the tools are in their infancy--they are primarily graphics-based and weak in scaling up.

Introduction

CASE technology is being used to increase programmer productivity and reduce software life cycle costs. Ada is being used increasingly for software development. There is a growing body of code that already has been written and could be targeted for reuse and revitalization. Given these considerations, a project was undertaken to evaluate commercially-available CASE tools claiming to support reverse engineering of Ada.

This paper discusses a list of features that support Ada reverse engineering, criteria for interpreting vendor information to find candidates for further assessment, the results of exercising four candidate tools with sample Ada systems, and recommendations for new features that CASE vendors should consider for future enhancement of their products. The purpose of this study has not been to identify the best tool but rather to identify what capabilities exist in CASE tools to support reverse engineering; to assess the extent to which these capabilities exist now in CASE products; and finally, based on the exercising of CASE tools with both real and textbook applications and the knowledge of what is needed for reverse engineering, to assess the state of the practice of Ada reverse engineering in the CASE tools industry.

Study Approach

Figure 1 gives an overview of the study approach. First, a working definition of reverse engineering was formulated. Second, based on the working definition, selection criteria were defined. Third, using the selection criteria and vendor literature on tools, four tools were selected on which to perform experiments. Next, experiments were performed using the tools. Finally, using the experiment results and a set of evaluation criteria, the four tools were evaluated and the current capabilities of the tools and recommendations were reported. The following sections discuss the major elements of the study.

Definition of Reverse Engineering

Bachman,[2] CASE Outlook,[8] and Wilde and Thebaut[24] offer alternative definitions of reverse engineering. Wilde and Thebaut,[24] for example, consider the recovery of program requirements to be a goal of reverse engineering. We feel that currently this is not achievable. Ideas from the three above-mentioned sources have been used to synthesize our working definition of reverse engineering, which follows:

> *Reverse engineering is the process of analyzing an existing program to obtain an understanding of that program. This is accomplished by producing representations of the program in forms different from the implementation language. These representations may take a graphical form, such as structure charts or structure diagrams, or they may be a subset of the information that presents a particular view of the software system, such as compilation dependencies.*

This definition is assumed throughout the following discussions.

As a point of clarification, we distinguished re-engineering and reverse engineering: It is often advantageous for a reverse engineered program to be enhanced further. The product of reverse engineering might be refined by a human and entered into the forward engineering process of a tool. This process--reverse engineering followed by forward engineering of the product--is considered re-engineering.

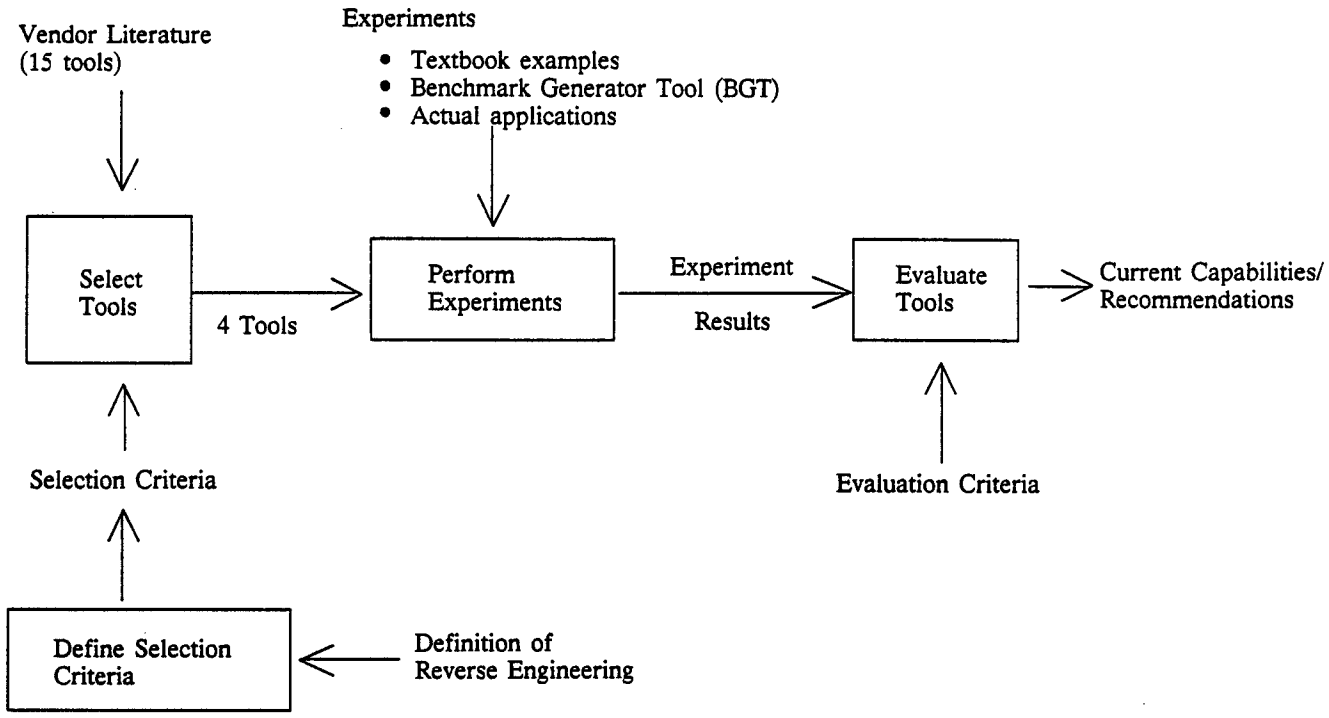


FIGURE 1
STUDY APPROACH

Reverse Engineering Tools Selection

We formulated a set of criteria for selecting Ada-oriented reverse engineering tools using Bachman,[1] Bush,[5] Campbell, et al.,[7] Roman,[17] Scarborough,[20] Schneidewind,[21] Wilde and Nejmeh,[23] and Wilde and Thebaut.[24] A reverse engineering tool:

1. Accepts general Ada code that was not necessarily generated by the tool (some tools require that the program input to reverse engineering be the output from the forward engineering of the tool).

2. Provides calling tree structure, structure charts, structure/Buhr/Booch graphs, or detailed design specifications (e.g., graphical program design language (PDL)).

3. Recognizes Ada structures (control, data, input/output)--tool should indicate hierarchy of modules and data, and indicate concurrent and sequential processes.

4. Presents compilation dependencies.

5. Provides batch analysis--the tool should allow the entire set of code to be analyzed without interruption.

6. Runs on common hardware platform--the tool may be used on generally available hardware, preferably multiple platforms.

7. Is available commercially.

A list of fifteen potential reverse engineering tools was compiled and checked against the reverse engineering selection criteria. This list was compiled from CASE Outlook,[8] Mimno,[14] Kerner,[11] and vendor presentations at the MITRE-Washington CASE Symposium.[6] When early information indicated that a tool did not currently support Ada or was not currently available, additional information was not pursued vigorously.

After the evaluation of the fifteen tools based on the selection criteria, four tools were selected to be subjected to the experiments described in the following section.

Reverse Engineering Experiments

A set of experiments was conducted with selected CASE reverse engineering tools as the second stage of the tool evaluation process. Reverse engineering, according to our definition, is the "process of analyzing an existing program to obtain an understanding of that program". Therefore, these experiments were conducted to determine the ability of a given tool to accurately import various Ada source programs into

an internal tool representation, and the tool's ability to present various views of the program to promote an understanding of the original source code. The experiments were intended to provide comparative data regarding the capabilities of several reverse engineering tools that cannot be evaluated without actual usage of the tool. As a secondary objective, the experiments were used to validate the information obtained about each tool in the first stage of the evaluation process.

Experiments A description of each experiment, the method used for evaluation of each, and the methods used to generate or acquire programs that meet the purpose of each experiment are described in this section. Given the previously stated objectives, the following experiments were performed:

- Capacity check--the tool was exercised with various capacity values to determine the upper bounds of tool-dependent parameters such as number of statements, number of units, level of unit dependencies, etc.

- Well-formed, known application--the tool was exercised with a small application that is easily understandable, but complex enough to use a variety of Ada constructs and include a level of interdependence among compilation units.

- Ill-formed application--the tool was exercised with a small application that had both syntactic and semantic errors to determine the tool's ability to process an ill-formed program.

- Unknown, real application--the tool was exercised with a large application by an experimenter who had limited knowledge of the application to determine if the available information views contribute to understanding the intent of the program.

- Partial system model--a portion of an application was used to see if the tool allows for import and analysis of a partially implemented application.

- Re-engineer--using the well-formed application, the program was reverse engineered. Then, using the forward engineering capabilities of the tool, if available, the program was re-engineered into source code as a way of verifying the tool's ability to capture and retain a complete model of the original source code internally.

Sources for Experiment Programs The experiments exercised the tools using actual Ada source code. The source code used for each experiment was one of the following: generated code, small programs taken from Ada texts, and code developed for an actual application.

- Generated Code--The generated code was constructed using the Benchmark Generator Tool (BGT).[16] This tool was designed to test the limits of an Ada compilation system in two areas: library capacity and recompilation efficiency based on various dependency structures. Although the BGT's orientation is toward compiler benchmarking, its generated Ada structures are useful for evaluating the CASE tools in two ways. Simple well-defined program structures can be generated for exercising the tool in a minimal way. More elaborate structures can be produced for evaluating the actual limits of the CASE tool in terms of number of compilation units, number of statements per unit, levels of nesting, total source lines, number and variety of dependencies. The BGT assists in generating a variety of Ada program structures whose representation in the CASE tool under evaluation can be easily verified since the generated program structures are known a priori and are well formed. The two sets of code used for the experiment contain 436 and 86 semicolons respectively. (The number of semicolons is a frequently used size measure of Ada projects.)

- Well-formed Applications--Given the BGT's focus, it does not generate compilation units with complex internal structures. If a context dependency is established between two packages, there is no actual usage of the exported declarations included in the generated structure. Therefore, another level of source code is needed. This level still needs to be well-formed, but should represent an actual application where unit dependencies are present and used. Various Ada texts[3,15] present such programs and their corresponding designs. Some of these textbook examples are not fully implemented, thereby providing partially implemented programs for the experiments. The availability of both source code and design provides a good validation mechanism for the reverse engineered graphical notations produced by the CASE tools. A selected set of these programs was used for a portion of the experiments. The two programs (Booch's[3] Concordance and Nielsen and Shumate's[15] Remote Temperature Sensor) used in the experiment contain 308 and 368 semicolons respectively.

- Actual Applications--Finally, actual applications were imported into the tools. These applications were used to evaluate the tools given large line of code counts and architectures produced for real-world applications. The two programs used in the experiment contain 3,058 and 53,756 semicolons respectively.

Method of Evaluation Each experiment was performed on each of the candidate tools. Comparative data was gathered for each tool's performance in each experiment. The following evaluation criteria were defined:

- Accuracy of internal model--determine if there is consistency between the internal representation and original source input into the tool. The accuracy of this internal representation can only be evaluated in terms of the support functions provided by the CASE tool for viewing the internal model. Therefore, the variety of information presented via the analysis, graphical, and reporting functions was used as the primary method of evaluation.

- Retention of information for re-engineering--determine if information is lost in the reverse engineering process. If a tool does not retain all the information from the original source code, then modification and re-engineering of the tool's internal representation will result in a new source program that does not contain the detail of the original program.

- User views--determine what subsets of information are made visible to the user. A variety of views that focus on different aspects of the program (e.g., compilation dependencies, calling trees, cross reference reports) will aid in understanding the program. The types of views and their usefulness for understandability were determined.

- Analysis support--determine if analysis functions can be executed against the internal model to derive additional program information. Such an analysis might include finding all compilation units that are dependent on a given package.

- Error handling and recovery--determine the tool's ability to reverse engineer programs with compile and/or run-time errors that are imported into the tool as well as recover from errors during normal user interaction and batch processing.

- Performance statistics--determine the times required to perform the tool's functions such as the reverse engineering process, display of a given graphics notation, etc. These times will be gathered as by-products of performing the experiments.

Table 1 provides a summary of the application of the first five of these criteria to the four tools on which the experiments were performed. Table 2 summarizes the application of the sixth criterion (performance statistics) to the four tools. In Table 2 process time is the time required to generate the internal representation for the tool. Note that the tools were weak in scaling up. As Table 2 indicates, none of the tools was successful in handling both the real applications.

TABLE 1

TOOL EVALUATION SUMMARY

Evaluation Criteria	Tool 1	Tool 2
Internal Model	• Complete and accurate compilation dependencies • Accurate overviews of program unit declarations ◦ No library support	• No abstraction takes place. No information is lost. • No library support
Re-engineering	• Incomplete-- Object, type and exception declarations omitted.	• Source code, generated PDL, and regenerated code are same.
User Views	• Compilation dependencies • Program unit declarations • Some compilation dependency diagrams overcrowed	• Program design language on a program unit, code block, and declarative region basis.
Analytic Capability	None	None
Error Handling and Recovery	• Good syntax recognition • Minimal semantic error recognition	• Very good syntax recognition • Minimal semantic error recognition

TABLE 1

TOOL EVALUATION SUMMARY

(Continued)

Evaluation Criteria	Tool 3	Tool 4
Internal Model	• Complete Ada syntax, incomplete Ada static semantics • Lack of simple mechanism to import implementation-dependent environment • No support for hierarchy of program libraries of units • Incorrect name visibility and scoping semantics implemented	• Complete Ada syntax and fairly complete Ada static semantics • Support for implementation-dependent environment for host on which tool executes • Support for hierarchies of program libraries of units • Unable to generate most forms of cross reference information • Unable to identify calling structures for nested units
Re-engineering	• Support for re-engineering of source code units via resubmission and "make" capabilities	• Support for re-engineering of source code units via resubmission and notification of obsolete dependencies
Analytic Capability	• Static set of metrics calculated and presented as part of reports	• Over 100 source code counts based on Ada structure usage

TABLE 1

TOOL EVALUATION SUMMARY

(Concluded)

Evaluation Criteria	Tool 3	Tool 4
User Views	• Support for multiple views via diagrams including compilation dependencies, call trees, declaration structures, and diagrams • Support for name cross reference reports and exception propagation • Some diagrams overcrowded, missing information • Ambiguities in name overloading and in declaration structure presentation • Limit on compilation dependency nesting • User selectable units for generation of views and reports	• Support for multiple views via reports including unit dependencies, call trees, exception propagation, name cross reference, and control flow • Tailorable standards checker • User selectable units for generation of views
Error Handling and Recovery	• Very good syntax recognition • Minimal semantic error recognition • Error reporting of syntax and some semantic errors • Inconsistent libraries caused by lack of semantic recognition • No access protection against modification of graphical information	• Very good syntax recognition • Good semantic recognition • Detailed error reporting of syntax and semantic errors • Weak error reporting at run time • Corruption of library during unit deletion • No access protection against modification of library information

TABLE 2

PERFORMANCE STATISTICS APPLIED TO TOOLS

Performance Statistics				
Experiment	Tool 1	Tool 2	Tool 3	Tool 4
Capacity Check (Minimal Sizing):				
Process Time (minutes)	35	(6)	4.5	26.3
Data Storage (Kbytes)	13		751 (1)	1609 (3)
Reports (Kbytes)	35		272	1129
Capacity Check (Minimal Dependency):				
Process Time (minutes)	6		3.4	14.3
Data Storage (Kbytes)	3	(6)	522 (1)	1052 (3)
Reports (Kbytes)	7		80	455
Well-Formed:				
Process Time (minutes)	7	18	4.75	5.5
Data Storage (Kbytes)	18	18	1042 (1)	826 (3)
Reports (Kbytes)	10	15	214	(4)
Partially Implemented:				
Process Time (minutes)	11	32	8.8	12.3
Data Storage (Kbytes)	25	25	1166 (1)	606 (3)
Reports (Kbytes)	13	39	246	(4)
Unknown Real Application 1:				
Process Time (minutes)	256 (5)	5 (6)	69.3	(2)
Data Storage (Kbytes)	342	342	8883 (1)	(2)
Reports (Kbytes)	62	11	2786	(2)
Unknown Real Application 2:				
Process Time (minutes)	1 (2)		(2)	(2)
Data Storage (Kbytes)	5064	(6)	(2)	(2)
Reports (Kbytes)	0		(2)	(2)

Notes:

(1) The storage requirements identified here represent storage for the application-specific code. In addition, storage is required for all of the predefined Ada units. These units required an additional 143 Kbytes of storage that was allocated for each experiment.

(2) Information not available since application was not successfully imported.

(3) The storage requirements identified here represent storage for the application-specific code. In addition, storage is required for all of the predefined Ada units and the run-time library. These units require an additional 16400 Kbytes of storage that was allocated once for all experiments.

(4) All reports could not be generated.

(5) Program failure before completion.

(6) Tediousness of input and predicatability of results precluded performing all experiments.

Kbytes = thousands of bytes

Summary of Findings

The hands-on experiments were extremely useful in providing information on the capabilities of the tool that could not be acquired through vendor literature or demonstration copies. Results indicate that the tools generate useful compilation dependency diagrams of an overall program system and various diagrams of individual program units. The information subsets generated are primarily graphics based and weak in scaling up. These information subsets also are primarily based on the Ada syntactical structures with very weak capture of semantic information.

We found that the tools are in their infancy. They have difficulty analyzing production-level applications. In certain cases, modest-sized applications (approximately 40,000 lines of code) could not be imported into the tool due to inefficient resource utilization. Most tools also require that the imported code be compilable. In one tool this compilation must occur on the compilation system associated with the tool's execution environment. This tool cannot process an application that makes use of implementation-dependent features that differ from the capabilities of the tool's host compilation system. Input of the application to a tool is unique per tool. In some cases it is extremely cumbersome requiring the separation of multiple compilation units in a single file into distinct physical files. This becomes burdensome for any

reasonably sized application. It also adds to the burden of using different vendor tools to examine and develop the code since each tool has its own unique form of importation. Iteration between forward and reverse engineering in a single tool is not generally supported. In one case, the reverse engineering process actually loses information that would have to be re-entered in the forward engineering step. This eliminates support for iterative development in the context of a tool.

The tools, in general, recognize and flag syntactic errors in programs. One tool even permits the interactive correction of syntactic errors with a continuation of the reverse engineering process from that point. Semantic errors, on the other hand, are not readily detected. To summarize, the tools provide limited information that would help in understanding a program.

Recommendations

We conclude with the following recommendations for Ada reverse engineering tools. The tools on the whole capture the properties of a program that can be inferred from the structure of the Ada language. The tools in general do not capture the meaning of programs which might be discernible from the dynamic or runtime semantics. The tools also should capture the error model of a program via exception recognition and understanding. Understanding the behavior of a program could be enhanced by the analysis of data structures and their use throughout a program. To provide for these capabilities, the next generation reverse engineering tools should include a query capability supported by a database that stores dynamic, structural, and context information required for understanding a program. Without such a capability ad hoc analysis support will continue to be lacking. Some commercially available products appear to provide such capabilities for other languages. The tools also should provide the capability to identify the implementation-dependent features of the Ada language in the source code. For instance, the specification of the packages STANDARD and SYSTEM should be profiled and recognized by the tool, and available to the user depending upon the compilation system used for development. The user would define an environment for an application via these packages.

REFERENCES

1. Bachman, C. July 1988. *A CASE for Reverse Engineering*, Datamation.

2. Bachman, C. 1988. *Reverse Engineering: The Key to Success*, CASE Outlook 1988, Vol. 2, No. 2.

3. Booch, G. 1986. *Software Engineering with Ada*. Menlo Park, CA: The Benjamin Cummings Publishing Company, Inc.

4. Buhr, R. J. A. 1984. *System Design with Ada*. Englewood Cliffs, NJ: Prentice Hall, Inc.

5. Bush, E. 1988. *CASE for Existing Systems*, CASE Outlook 1988, Vol. 2, No. 2.

6. Byron, D. and R. Fuss, eds. 1989. *Proceedings of the Computer-Aided Software Engineering Symposium*, 23 January 1989, MP89W00010. McLean, VA: The MITRE Corporation.

7. Campbell, K. et al. 1987. *Evaluation of Computer-Aided System Design Tools for SDI Battle Management/C3 Architecture Development*. Alexandria, VA: Institute for Defense Analysis.

8. CASE Outlook. 1988. *CASE Tools for Reverse Engineering*, CASE Outlook 1988, Vol. 2, No. 2.

9. Department of Defense. 1983. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A.

10. Firth, Robert et al. August 1987. *A Guide to the Classification and Assessment of Software Engineering Tools,"* Technical Report CMU/SEI-87-TRIO, ESD-TR-87-111. Pittsburgh, PA: Software Engineering Institute Carnegie Mellon University.

11. Kerner, J. November/December 1988. *Ada Design Language Developers Matrix*, Ada Letters, VIII: 6.

12. Lempp, P. and R. J. Torick. March 1988. *Software Reverse Engineering: An Approach to Recapturing Reliable Software*, Fourth Annual Joint Conference on Software Quality and Productivity.

13. Lempp, R. and A. Zeh. May 1988. *Interfacing a Development Support Environment to a MAPSE through Ada Code Generation and Code Feedback: A Step Towards an APSE*, National Aerospace and Electronics Conference (NAECON '88).

14. Mimno, P. R. May 1988. *James Martin Productivity Series: CASE Tools Comparison and Review*, CASE Expo Spring Infomart. Dallas, TX.

15. Nielson, K. and K. Shumate. 1988. *Designing Large Real-Time Systems with Ada*. New York, NY: McGraw-Hill Book Company.

16. Rainier, S. R. and T. P. Reagan. 1988. *User's Manual for the Ada Compilation Benchmark Generator Tool*, MTR-87W00192-01, Volume 1. McLean, VA: The MITRE Corporation.

17. Roman, D. June 1986. *Classifying Maintenance Tools*, Computer Decisions.

18. Rosenblat, G. D. and H. Fischer. 1989. *Reverse Engineering and the Ada Software Development Cycle.*

19. Scandura, J. M. 1987. *A Cognitive Approach to Software Development: The PRODOC Environment and Associated Methodology,* Journal of PASCAL, Ada, Modula-2, Vol. 6, No. 5. pp. 10-25.

20. Scarborough, K. 1988. *Tool Evaluation Report.* Reston, VA: Software Productivity Consortium.

21. Schneidewind, N. F. March 1987. *The State of Software Maintenance,* IEEE Transactions on Software Engineering, Vol. SE-13, No. 3.

22. Seviora, R. May 1987. *Knowledge-Based Program Development Systems,* IEEE Software.

23. Wilde, N. and B. Nejmeh. September 1987. *Dependency Analysis: An Aid for Software Maintenance.* Gainesville, FL: Software Engineering Research Center.

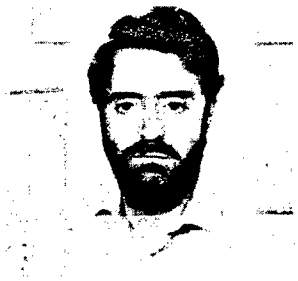
24. Wilde, N and S. Thebaut. October 1987. *The Maintenance Assistant: Work in Progress,* Post Deployment Software Support Symposium.

Authors' Address MITRE Corporation
7525 Colshire Drive
McLean, VA 22102

M. Cassandra Smith received the B.A. degree in mathematical and statistical economics from Howard University and the M.S. and Ph.D. degrees in computational linguistics from Georgetown University. Her current interests include CASE technology, software metrics, and database technology. Dr. Smith is currently employed by the MITRE Corporation as a member of the technical staff in the Washington Software Center. She is a member of ACM, ACM SIGMOD and Association for Computational Linguistics (ACL).

Diane E. Mularz received the B.S. degree in mathematics from Indiana University of Pennsylvania and the M.S. degree in computer science from Johns Hopkins University. Ms. Mularz has over 15 years experience in the field. Her current interests include software engineering environments and design quality assessment. Ms. Mularz is currently employed by the MITRE Corporation as a lead scientist in the Washington Software Center. She is a member of the ACM, ACM SIGAda and the IEEE Computer Society.

Thomas J. Smith received the B.Sc. degree in mathematics from the University of Manitoba, the Ph.D. degree in mathematics from the University of Iowa and the M.S. degree in computer science from Georgia Institute of Technology. Dr. Smith has 20 years of experience in the fields of mathematical and computer sciences. During the past 10 years he has concentrated particularly in parallel and distributed computing, programming languages, software tools, and programming environments. Dr. Smith is presently a lead scientist at the MITRE Corporation in Washington, DC. He is a member of the ACM, ACM SIGAda, and the IEEE.



THE Ada LANGUAGE SYSTEM/NAVY PROJECT

William L. Wilder

Naval Sea Systems Command PMS-412
Washington, D.C.

Abstract

The objective of the United States Navy's Ada Language System/Navy (ALS/N) development is to provide a full production Ada capability for the Navy's AN/UYK-43, AN/UYK-44, and Pre-Planned Product Improvement (PPPI) AN/AYK-14. The ALS/N implements the Ada Programming Language as specified by MIL-STD-1815A and required by Department of Defense (DOD) Directives 3405.1 and 3405.2. This Ada program generation environment will improve programmer efficiency and reduce life cycle costs; while the Ada run-time environment will satisfy Navy-specific embedded application requirements such as performance, reliability, and fault-tolerance. The ALS/N Full-Scale Engineering Development (FSED) program supports the mandate of Ada for the Navy's standard embedded computers.

Introduction

The Navy's approach to providing Ada technology has been accomplished by separate but related developments. The first development has resulted in the early availability of a pilot production Ada capability for the AN/UYK-44, called Ada/M(44). This prototype included an Ada compiler, linker, importer, exporter and run-time operating system (consisting of an executive and library) for the AN/UYK-44. This development was completed in September 1986 and was one of the fastest Ada run-time environments at that time. Subsequent developments, that is the ALS/N FSED program, have resulted in a full production capabilities for the AN/UYK-43 (Ada/L), AN/UYK-44 (Ada/M), and PPPI AN/AYK-14 (Ada/M).

The ALS/N FSED program consists of the Ada/L, Ada/M and Programming Support Environment (PSE) developments which are being performed by multiple FSED contractors in two builds. The first build consists of the "single CPU" version of the AN/UYK-44 or PPPI AN/AYK-14 and "single/dual CPU" versions of the AN/UYK-43. The second build of the ALS/N FSED program adds multiprocessing, multiprogramming, and distributed Ada capabilities to the existing ALS/N products. The schedule is:

UYK-43 & 44 Ada validated	Dec 1988
AYK-14 Ada available	Mar 1988
Ada/L & Ada/M complete	Jun 1990
PSE complete	Sep 1990
Basic ALS/N delivered	Dec 1990

According to Navy and DOD directives, systems required to use the ALS/N are any Navy applications being implemented after the December, 1988 validation (i.e., new starts and any major software upgrades). The final delivery is being revalidated, but note that multiprocessing, multiprogramming, and distributed Ada are being incorporated into the products as part of maintenance releases.

Historical Perspective

The U.S. Navy has historically standardized Mission Critical Computer Resources (MCCR). Their AN/UYK-7, AN/UYK-20, and AN/AYK-14 Standard Embedded Computers have, for years, supplied the computing power for Navy mission critical systems. Almost all of the Navy software for these machines is written in one high-order language, CMS-2, and supported by one programming environment — Machine Transportable AN/ Support Software (MTASS). The newer standard embedded computers that evolved, the AN/UYK-43, AN/UYK-44, and PPPI AN/AYK-14, were in development about the same time as the DOD Standard High Order Language Ada was being adopted.

In keeping with their practice of standardization, the Navy put together a team from the Navy laboratories that represented Navy users to specify requirements for an Ada program generation and run-time environment targeted to the new machines. The result of this team's efforts, led by Naval Sea Systems Command (NAVSEA) PMS-412, was

the ALS/N System Specification. After the ALS/N FSED phase began with the awarding of a competitive contract to the Control Data Corporation/TRW/SYSCON team and a directed task to SofTech Inc., the support from Navy laboratories continued.

The same people that developed the ALS/N System Specification became the Navy Design Review Group (DRG) which reviews all contractual items and attends all the major design reviews. In particular, the Independent Verification and Validation (IV&V) agent for the ALS/N FSED was the Fleet Combat Direction Systems Support Activity, San Diego CA (FCDSSA, SD). Science Applications International Corporation (SAIC), Comsystems was the IV&V contractor for FCDSSA, SD. Government Furnished Information (GFI) support of the ALS/N Common Ada Baseline (i.e., VAX/VMS host) is being performed by the Naval Underwater Systems Center, Newport RI (NUSC, Npt).

When the Navy DRG was preparing the ALS/N System Specification they polled the existing major programs for requirements. The emphasis was on finding performance requirements for the ALS/N Run-Time Environment that would satisfy any existing Navy program and include a factor for growth into the 1990s. Shadow programs of selected functions from some of the major programs were initiated in the laboratories using the AN/UYK-44 prototype run-time operating system to familiarize the Navy's Ada team with its use and to further prove the usefulness of the ALS/N approach.

The performance of the AN/UYK-44 prototype run-time, written in Ada, was measured and compared with the ALS/N System Specification requirements, in conjunction with existing real-time embedded Standard Executives (SDEX) for CMS-2. The results showed the prototype to be one of the fastest Ada run-time operating system in existence at that time. These measurements gave the Navy's Ada team confidence that the ALS/N FSED versions will meet the ALS/N System Specification requirements and support the Navy's mission critical embedded systems through the year 2000.

ALS/N FSED Baseline

The ALS/N FSED Baseline was initially the Kernel Ada Programming Support Environment (KAPSE) based product that was developed by the U.S. Army's Ada Language System (ALS) and transferred to the U.S. Navy in late 1986/early 1987. Since the Navy's mission has always been oriented towards developing an Ada run-time environment for the Navy standard embedded computers with a full Ada Programming Support Environment (APSE) not being required, extensive changes to what is now called the ALS/N Common Ada Baseline (CAB) were initiated.

In particular, the Navy is only providing a Minimal APSE suitable for coding and testing (i.e., implementation phase), therefore we must look to other sources to provide tools/processes for all other development phases (e.g., specification and design). Cost estimates for incorporating a tool used in the ALS/N FSED, which is PSL/PSA, indicated that this would not be cost-effective if the Navy had to bear the burden alone. However, PSL/PSA is already hosted directly on VAX/VMS, as are a large number of other tools/processes that are commercially available.

In a separate, but closely related activity, the ALS/N FSED contractors had been utilizing a VAX/VMS hosted and targeted capability in their implementation of the retargets to Navy standard embedded computers. This capability, called the AdaVAX subsystem, had been the method by which the Ada compilation system for the Army's ALS was maintained between major releases. The FSED contractors requested that the AdaVAX subsystem be utilized for acceptance testing and delivery of the Navy retargets, thus officially changing the ALS/N FSED baseline.

The Navy investigated the ramifications of changing the ALS/N FSED baseline, agreed with the ALS/N FSED contractors' request for the reasons outlined above, and publicly announced this decision in July, 1987. As such, the ALS/N CAB consists of the AdaVAX subsystem and Program Library Manager (PLM) shown in figure (1). The ALS/N FSED products now consist of the Ada/M (AN/UYK-44 and AN/AYK-14), Ada/L (AN/UYK-43), and PSE subsystems being developed by the ALS/N FSED contractors and available in conjunction with the ALS/N CAB for delivery.

The additional benefits of this FSED baseline change in the ALS/N CAB to the Navy are as follows:

- Increased maintainability and transportability based on reduction in the amount of foreign code being utilized.
- Reduced maintenance and rehost costs because a complete layer of interfaces and associated tools have been removed.
- Continued concentration of the ALS/N FSED Contractors resources on the Ada run-time environment capabilities.

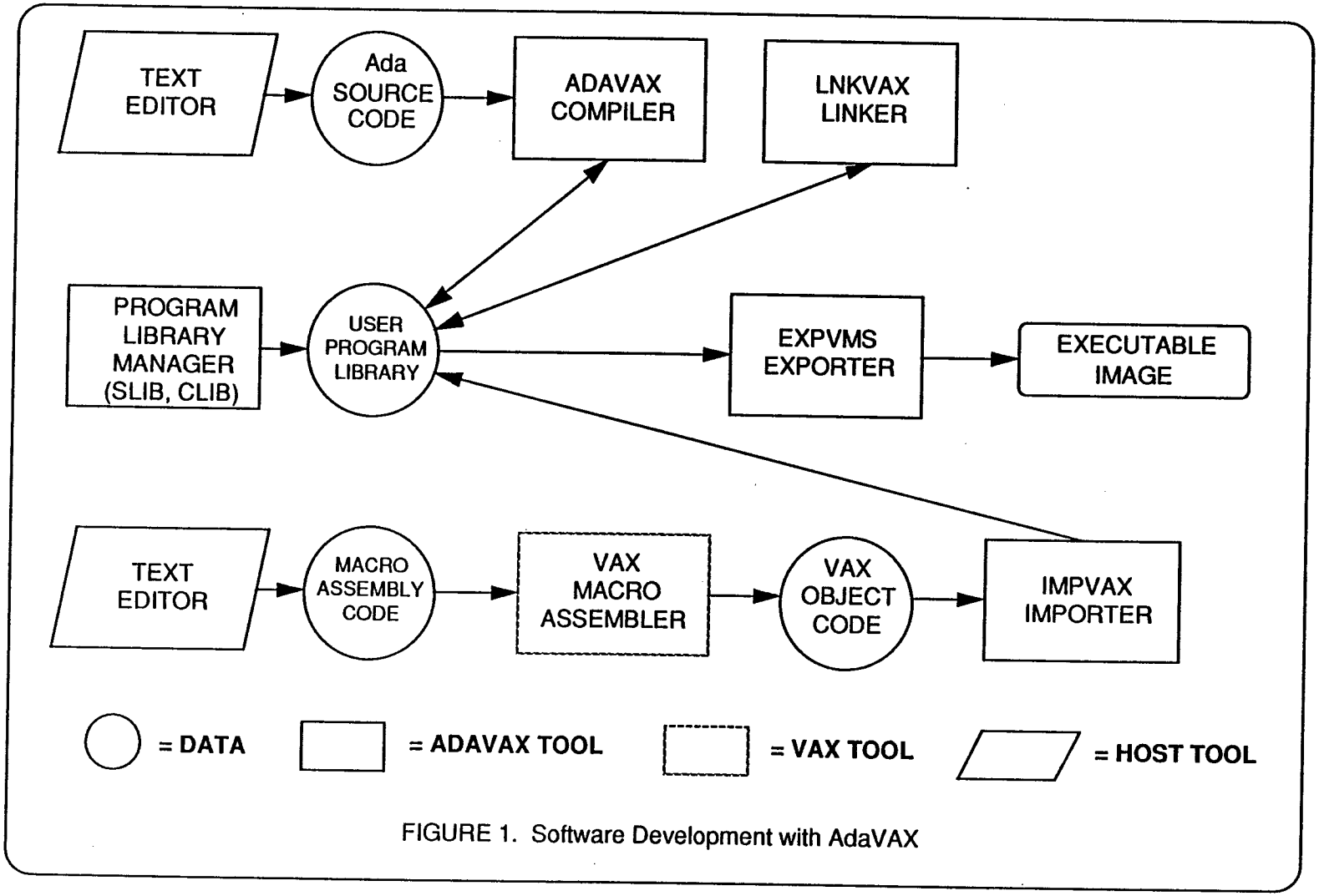


FIGURE 1. Software Development with AdaVAX

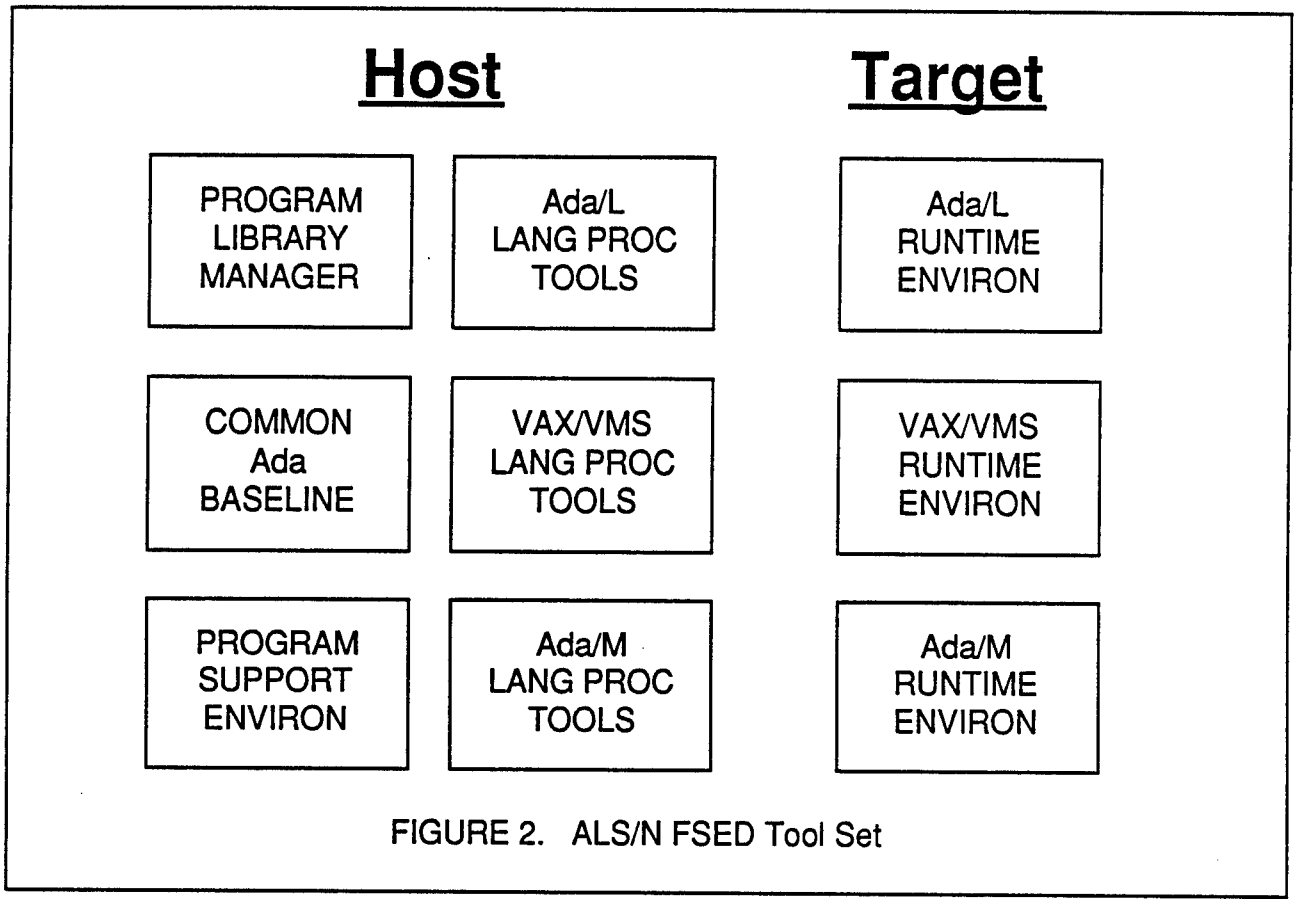


FIGURE 2. ALS/N FSED Tool Set

Functional Description

The ALS/N is basically a run-time environment for the target machines and a program generation environment currently hosted on the VAX that produces code for the target machines, plus some additional tools as shown by figure (2). The ALS/N was developed in accordance with DOD-STD-2167A, and DOD-STD-2168, designed to support large software development projects, and satisfy real-time embedded application requirements. Many features were added that are the direct result of the Navy DRG's knowledge of Navy mission critical system requirements, such as fast interrupt entries, precisely timed interrupts/delays, asynchronous Input/Output support, and fault-tolerant damage control.

Ada/L and Ada/M are the designations for the ALS/N subsystems that include Ada program generation and run-time environments for the 32-bit AN/UYK-43 and the 16-bit AN/AYK-14 and AN/UYK-44; respectively. A common specification and design approach was taken for the Ada/L and Ada/M subsystems, with the objective to implement a run-time operating system that is usable by many different Ada programs simply by tailoring the run-time executive and configuring the run-time library to the application. A standard run-time executive, in conjunction with an extensible run-time library, will reduce costs throughout the life cycle of any Navy application.

The architecture of the Navy standard computers present a challenge for the Ada language in that these machines all have a small virtual address space. The AN/UYK-44 and AN/AYK-14 can only address 64K words, and although the AN/UYK-43 can address 512K words using indirection and indexing, only 64K words can be directly addressed in a static manner. However, this problem of limited addressability was solved with two different approaches: (1) a software approach, called the "phase model," that is described below; and (2) the development of Extended Memory Reach (EMR) versions of the AN/UYK-43 and the AN/UYK-44. These EMR machines support the phase

model, but also provide specific hardware enhancements for "phasing" that includes multiple sets of memory mapping registers (called "memory groups") which are addressable in the task state (unprivileged) and, therefore, do not require executive service to be changed by the application.

Phase Model

An analytical model was developed that divides the total physical address space into a set of virtual address spaces, each called a "phase." A single phase is made of two address regions: a shared memory area, termed "phase common," and a "phase unique" portion. The Ada program can be specified and designed without regard to this "phase model". The application designer thus does not have to consider partitioning into phases as part of the design because the partitioning is performed during implementation. The Ada program is mapped onto this virtual "phased machine" as part of the linking process. The objective of the ALS/N approach is to create a program generation environment for software-first development that is independent of any underlying hardware.

Additionally, a stack model was developed to reduce overhead associated with "cross-phase" calls in the limited address space machines. Each task is assigned a separate stack, and the stack "travels" from phase to phase enabling cross-phase calls, cross-phase data reference, and cross-phase rendezvous with a minimum performance penalty. The user is able to specify which packages share phase space with other packages to optimize address space usage and efficiency of operation. This stack model is an elegant solution to the phase model implementation of Ada programs.

Input/Output Support

The Input/Output (I/O) supported within the ALS/N includes: low-level, synchronous, and asynchronous support for a variety of peripherals such as the AN/USQ-69 terminal, the AN/USH-26 and RD-358 tape drive, or the AN/UYK-3 disk drive. Support for MIL-STD-1553B, MIL-STD-1397 (NTDS) Fast/Slow Channel, and RS232 bus protocol has been provided. If the required I/O support is not available, a template device driver is provided as a model from which the user can develop application specific I/O support.

Fast Interrupt Entries

The Ada language provides for interrupt entries. An interrupt entry allows an Ada task to associate an entry with some external activity (e.g., a device) that will cause an interrupt. Interrupt entries, however, are not always capable of supporting real-time embedded systems due to the overhead imposed by Ada semantics. Fast Interrupt Entries (FIEs) allow efficient support for time critical interrupt processing and does not ignore Ada semantics. The concept of FIEs is to impose restrictions on the Ada features that are supported within the rendezvous so that the overhead associated with delivering the interrupt to the application can be significantly reduced. Also, by requiring the FIE to always be mapped (i.e., addressable by the memory mapping registers), additional savings in responding to an interrupt are possible.

Precisely Timed Interrupts/Delays

Precisely Timed Interrupts (PTIs) provide a mechanism for performing time-based scheduling, entirely within the framework of an Ada priority pre-emptive scheduler. PTIs consist of a logical collection of virtual timers that interrupt on a recurrent, nondrifting basis. These virtual timers, which are associated with underlying hardware timers, can be used to provide the functionality of cyclic schedulers, but with greater robustness and flexibility. Precisely Timed Delays (PTDs) utilize the Ada DELAY statement and also operate within the framework of an Ada priority pre-emptive scheduler. This facility depends upon delaying the task per the semantics of Ada, but immediately readying the task for execution at the expiration of the DELAY. The PTD provides for time-dependent behavior, while the PTI provides for time-critical behavior in the Ada program since tasks responding to interrupts execute at a priority higher than all other tasks.

Fault-Tolerant Damage Control

The ALS/N design makes damage control facilities available to application developers so that the Ada program can be apprised of hardware or software failures and take action that the application deems appropriate. For example, degraded mode operation after casualty reconfiguration. The ALS/N Run-Time Environment handles a hardware of software fault by isolating the event and notifying the Ada program through a Damage Control Task. It is the intent of the ALS/N implementation to expand the initial damage control facilities to perform certain types of fault-tolerant recovery actions for the user when failures occur.

Multiprogramming

The majority of the technical effort for implementing multiprogramming involves getting more than one program in execution within a single machine. Implementing the

communications mechanism between programs, be they on the same, or a distributed node, is described as part of distributed Ada. The basic features of multiprogramming include the ability to create, delete, suspend, and resume a program (and all of its associated tasks) with a single-tiered scheduler where program priority is added to individual task priority to provide an overall system priority.

Unfortunately, multiprogramming and FIEs are mutually exclusive capabilities in the baseline machines because FIEs are always placed in phase common and therefore always mapped to function. However, for the EMR processors, FIEs can still be implemented provided they are mapped in one of the memory groups.

Distributed Ada

For some time, there has been discussions about the lack of a basic "mailbox" facility in Ada. The classic paradigm of mailboxes describes a nonblocking message passing between the server and client tasks, provided that sufficient buffer space is available. Such a facility can be provided using normal Ada semantics, but there is a relatively high overhead associated with such an implementation. A mailbox facility is the form of asynchronous message passing utilized, with some limitations and using normal Ada semantics, to provide a basic communication mechanism for both distributed Ada and multiprogramming.

While the asynchronous message passing paradigm is the unifying mechanism to address communications between Ada programs whether on the same processor or distributed onto multiple processors, only the EMR processors can be supported because of addressability and performance requirements. The user is required to instantiate a generic package within each Ada program desiring to communicate in such a manner and guarantee that there is a proper correspondence between parameter types (there can be no type checking between different Ada programs). In all other respects, the communications appear to be the calling of an entry in one Ada program from another Ada program.

Minimal Ada Programming Support Environment

The ALS/N FSED phase is composed of two distinct efforts: the Minimal Ada Programming Support Environment (MAPSE) and the associated ALS/N Run-Time Environment. The MAPSE supports the development and deployment of real-time application software for the Navy's standard embedded computers. The major functions provided are: Ada compilation and linking/exporting facilities to generate applications for the target computers, interactive symbolic debugging and performance measurement facilities for execution analysis, general-purpose text editor and formatter, configuration management identification tools and report generator, a command language processor, and the associated environment database to support these functions.

The ALS/N language processing tools provide a full production capability to translate Ada source code into executable images for the AN/UYK-43, AN/UYK-44, or PPPI AN/AYK-14 targets and includes the EMR versions of these machines. These tools consist of compilers, linkers, exporters, importers, listing tools, stub generators, and the Program Library Manager (PLM) as shown by figure (3). The language processing tools support the generation of machine code to run in either task (unprivileged) mode or executive (privileged) mode as part of the Executive Option capability. The Executive Option capability allows the application developer to logically include portions of any Ada program, such as a specialized I/O driver, in the ALS/N Run-Time Environment.

Ada compilers for the ALS/N process Ada source program units by checking for correct Ada syntax/semantics and translating source code into linkable object code. These compilers support the full Ada Programming Language, MIL-STD-1815A, and their implementations include size representations, representation specifications, interrupt entry addresses, and low-level I/O support. In addition, implementation-defined PRAGMAs are recognized and processed, which provide applications with real-time capabilities. The ALS/N compilers share common baseline components for checking correctness of Ada code and for managing Ada program libraries, which contributes to a consistent and user-friendly interface to all ALS/N language processing tools.

The ALS/N linkers resolve external and internal references in the linkable object code to produce linked object code. Object code may be linked into phases, which are analogous to the target machine's virtual memory, and then into programs, which are analogous to the target machine's physical memory. The ALS/N linkers allow Ada programs to take full advantage of all physical memory resources in a target machine. The linkers also support the incremental linking of phases within Ada programs, thus reducing the overhead of rebuilding an application. As well as resolving external references, the linkers bind all required Run-Time Library support to the Ada program. Run-Time Library support may be selectively bound so that only the support that is actually needed by the application is included in an executable image.

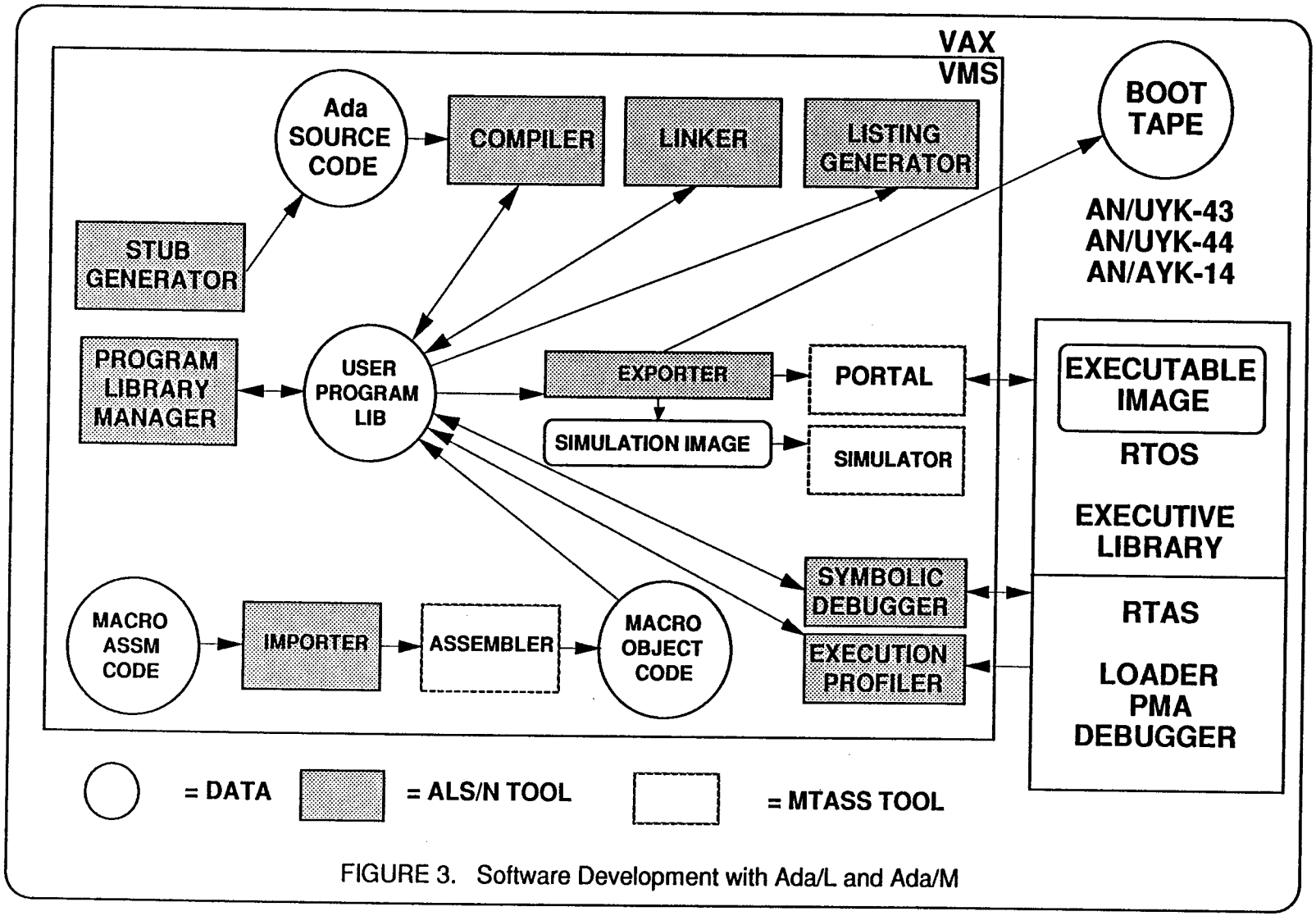


FIGURE 3. Software Development with Ada/L and Ada/M

Linked object code is processed by the ALS/N exporters to bind the Run-Time Executive with the application and perform a final layout of the object code into target memory. The exporters also implement a flexible mechanism allowing an application developer to take advantage of target machine resources (e.g., assignment of physical memory, I/O channels, etc.) for the application. The exported executable image is an Ada program that needs no further processing to run on a target machine or under a simulator. The exporters will produce a bootstrap image on tape or disk, which is acceptable to the target machine bootstrap loader; or a Target System Format file, which is acceptable to the MTASS simulators or Programmer Oriented AN/ Link (PORTAL) products.

The ALS/N importers provide the capability of implementing Ada package and procedure bodies in assembly code. The importer will format the assembly implementation in a manner acceptable to the linker. It is a user responsibility to ensure that hand-written assembly code conforms to ALS/N Generated Code Conventions prior to importation. Additionally, a foreign code importer for CMS-2 is available to interface with the Ada program. This tool supports the reuse of existing Navy application code.

Human readable listings may be produced by invoking the listing tools. Compiler listings showing the translation of Ada code to object code, linker listings showing the layout of code into phases or programs, and exporter listings showing the final binding of code to machine locations are among the listings that can be produced. Finally, the stub generators will aid Ada developers by accepting compiled Ada package or procedure specifications and automatically producing stubbed-out Ada package or procedure bodies corresponding to the specifications.

The integrated set of tools and associated support within the Programming Support Environment (PSE) consists of:

- Command Language Processor - the user need only know one command language to access all the tools and move around in the hierarchical file system of the environment database.
- Configuration Management Tools - the user has powerful tools for version and source maintenance, software problem report tracking, documentation updating, and a report generator for displaying this information.
- Document Processing Tools - the user has powerful tools for general-purpose text editing (both line and screen) and formatting (including macro expansion).
- Environment Database Manager - the associated hierarchical file system, consisting of directories, files, attributes, and associations; in support of the tools outlined above.

Run-Time Environment

The Run-Time Environment (RTE) functions support the Navy's real-time embedded mission critical requirements and has been designed utilizing state-of-the-art computing techniques to deliver a reliable and efficient capability. The RTE developments have been performed in parallel with the associated MAPSE developments. The RTE consists of a standard executive and configurable library that can be tailored to meet application requirements, stand-alone and host/target interactive debuggers, performance measurement tools for execution analysis, and dynamic program loaders. The RTE can be separated into the Run-Time Operating System and Run-Time Application Support for the AN/UYK-43, AN/UYK-44 or PPPI AN/AYK-14 and includes the EMR versions of these machines.

Run-Time Operating System

The Run-Time Operating System (RTOS) provides the Run-Time Executive, the Navy's standard executive for the Ada programming language, which can be tailored to each Navy application. The RTOS also provides the Run-Time Library, which is configurable and extendable for the Ada program. RTOS capabilities include multiprocessing, multiprogramming, distributed Ada, and provisions to have user-supplied executive code (i.e., Executive Option user routines) for specific application needs. Note that the Executive Option capability is the basis of the Run-Time Operating System implementation.

Run-Time Executive

The Run-Time Executive (RTExec) runs in the executive state (privileged and protected) and controls all of the hardware resources of the AN/UYK-43, the AN/UYK-44, and the PPPI AN/AYK-14. The RTExec responds to either Run-Time Library requests for an executive service or requests from the user-supplied Executive Option routines. These Executive Service Requests (ESRs) or direct requests from Executive Option routines are presented to the RTExec through the RTEXEC_GATEWAY by the ESR Interrupt, with a corresponding ESR code. When the requested action is completed, control is returned via the normal interrupt return mechanism of the target computer (i.e., reloading the saved general purpose registers, the program status registers, and the program counter).

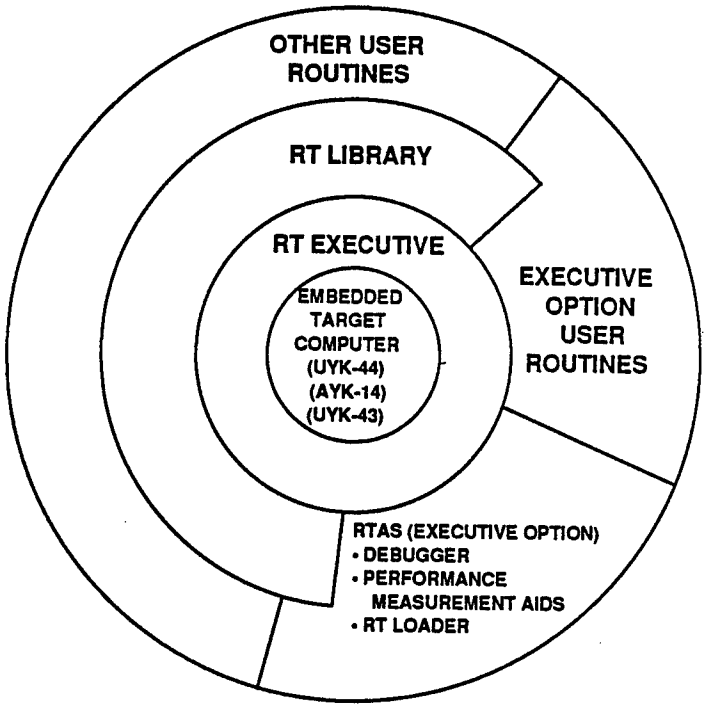


FIGURE 4. ALS/N Run-Time Environment

Run-Time Library

The Run-Time Library (RTLib) runs in the task state (unprivileged and unprotected) and responds to all Ada program requests, except for Executive Option user requests. The RTLib either acts upon the request itself or directs the request to the RTExec through the RTEXEC_GATEWAY. Note that user-supplied Executive Option routines have the capability to either utilize the features of Ada normally through the RTLib or go directly to the RTExec and RTLib interfaces when required by the application. The Executive Option capability is the foundation on which all of the Run-Time Application Support functionality is implemented.

Run-Time Application Support

The Run-Time Application Support (RTAS) augments the Run-Time Operating System with capabilities to enhance the development and deployment of application programs as shown by figure (4). The RTAS consists of a Run-Time Loader (RTLoad), Run-Time Debugger (RTDebug) that cooperates with the Embedded Target Debugger, and Run-Time Performance Measurement Aids (RTAids) that operates in conjunction with the Embedded Target Profiler.

Run-Time Loader

Dynamic, static, and overlay loading is supported by the Run-Time Loader. In conjunction with the RTOS, the RTLoad provides applications with the capability to efficiently manage the resources of the target computer, support of run-time assignment of physical memory locations to load modules, and includes a mechanism for dynamic reconfiguration of an application to accommodate changes in the computing environment.

Run-Time Debugger

Access to all hardware resources of the AN/UYK-43, AN/UYK-44 and PPPI AN/AYK-14 not restricted to the RTOS is provided by the Run-Time Debugger. Operating completely within the target computer and requiring only a terminal device, the RTDebug equips the application developer with a powerful interactive tool for Ada program checkout. Facilities provided within the RTDebug include: setting and clearing breakpoints, examining and modifying registers or memory, blocking and unblocking tasks, and tracing call stacks. In addition, the Run-Time Debugger

supports the host/target relationship with the Embedded Target Debugger.

Symbolic access to all code and data elements of an executing Ada program is available through the Embedded Target Debugger. This user-oriented symbolic debugging tool is designed to support efficient application development in a laboratory environment. Reference to source code and data through the symbols defined in the Ada program isolates users from the details of the Ada language implementation, thus allowing the developer to more productively debug applications. Furthermore, resources of the host and target are utilized in a hardware testbed to ensure an efficient and reliable operation of the Embedded Target Debugger.

Run-Time Performance Measurement Aids

Detailed run-time performance analysis of application programs is performed through use of the Run-Time Performance Measurement Aids and Embedded Target Profiler. The RTAids works in concert with the RTOS to collect data characterizing the performance of an executing program. Collected data is then processed at a later time by the Embedded Target Profiler producing histograms or "profiles" of the application's performance characteristics.

Conclusions

The ALS/N FSED program represents a multi-million dollar government investment managed by the Naval Sea Systems Command (NAVSEA) PMS-412 in a Minimal Ada Program Support Environment and ALS/N Run-Time Environment to implement Ada and meet the goals set for it by the Department of Navy. The ALS/N is designed to reduce Software Life Cycle Maintenance (SLCM) costs and to support Mission Critical Computer Resources. The requirements set for the ALS/N insure that these goals are realized and include factors for performance that will reflect the challenge of the 1990s.

The competitive ALS/N FSED Contract for Ada/L, PSE tools, and integration of Ada/M into the ALS/N was awarded to the Control Data Corporation/TRW/SYSCON team. The Ada/M FSED was developed as a directed task to SofTech, Inc. and continued development is being performed by CDC. The SLCM agent for ALS/N FSED is Fleet Combat Direction System Support Activity, San Diego (FCDSSA, SD) and Science Applications International Corp., Comsystems is providing SLCM Testing & Evaluation under a competitively awarded omnibus contract. FCDSSA, SD is responsible for distributing releases to all ALS/N users, collecting Software Problem/Change Reports as they are submitted and providing corrections (and updates) to the ALS/N users. It is the goal of NAVSEA PMS-412 to make the ALS/N FSED products available to all Navy application developers at no charge.

The ALS/N project is the current phase of Navy Ada development for the AN/UYK-43, AN/UYK-44, or Pre-Planned Product Improvement (PPPI) AN/AYK-14 and includes the EMR versions of these machines. This phase began in September 1985 and will be completed by December 1990. (Note that enhancements to the ALS/N in support of the PPPI for the AN/UYK-43 and AN/UYK-44, started in September 1989 and will be finished by December 1992.) Its evolution has been:

- The Army's ALS -- target independent portions have been retained as appropriate to the ALS/N; rights to products are co-owned by the Government and SofTech; commercial work has been retrofitted into the VAX/VMS host and target; the Government continues to improve the ALS/N CAB.
- The Ada/M(44) prototype -- retarget to AN/UYK-44 with near production quality compiler was developed; prototype for ALS/N run-time capabilities targeted to embedded systems (e.g., memory management, precisely timed delays, I/O subsystem) was developed; the prototype run-time was written in the Ada language itself and was one of the fastest Ada run-time operating systems available.
- The ALS/N FSED -- production-quality compilers for AN/UYK-43, AN/UYK-43, and PPPI AN/AYK-14; all issues raised by Ada/M(44) prototype development (i.e., compiler code quality, RTOS algorithms, linker/exporter tools, etc.), are resolved; PSE tools and Embedded Target Debuggers/Profilers are provided.

Based on the sizing and timing test results gathered by the ALS/N FSED contractor team and the IV&V contractor, all ALS/N System Specification performance requirements are achievable. Additionally, an independent assessment of the ALS/N has reported performance information comparable to the ALS/N test results and has provided many valuable insights from a user perspective to improve the ALS/N CAB, Ada/L, Ada/M, and PSE subsystems.

Major enhancements to the ALS/N will be made to support PPPI of the standard embedded computers, which are being upgraded to meet emergent Navy requirements. The improvements are the Enhanced Processor (EP) AN/UYK-44, the High Performance Processor (HPP) AN/UYK-43, and the Very High Speed Integrated Circuit

(VHSIC) AN/AYK-14. The EP and HPP instruction set architecture permits direct addressing of all memory in task state and provides for flexible protection mechanisms. These features make the EP and HPP machines better targets for execution of Ada programs within the ALS/N implementation. The EP and HPP improvements will also provide for faster execution than the current machines (5-20 MIPS versus 1-4 MIPS). The VHSIC improvement provides faster processors for the AN/AYK-14 (5-10 MIPS versus 1-2 MIPS) and can be configured as two independent processors with 1 MByte of onboard memory.

The ALS/N has been specified and designed for portability. The first scheduled rehosts will be to the Extended Memory Reach (EMR) AN/UYK-43, under SHARE/43, and VAX, under Portable Operating System for UNIX (POSIX). The rehost to the EMR AN/UYK-43, running SHARE/43, will support Navy users throughout the extended life of their applications by using the EMR AN/UYK-43 as both their host (for development) and target computer. The rehost to VAX, running POSIX, will allow integration of ALS/N products with other Government and commercial software development environments.

In summary, the ALS/N is a Navy user-driven system designed and implemented to support real-time embedded mission critical applications. The flexible ALS/N Run-Time Environment allows tailoring by diverse mission critical computer systems thus saving costs in the development phase of the life cycle. The portable Minimal Ada Programming Support Environment is available to reduce costs in the maintenance phase of the software life cycle. The requirements for the ALS/N provided by the Navy Design Review Group have taken into account future needs of the Navy's embedded Ada applications. NAVSEA and Control Data Corporation, the prime contractor for ALS/N, have developed the technology for a program generation and run-time environment that will support Ada in real-time embedded mission critical systems through the year 2000.

References

[1815A] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, Department of Defense, January, 1983.

[SPEC88] Ada Language System/Navy System Specification, NAVSEA, 29 July 1988.

[2167A] Defense System Software Development, DOD-STD-2167A, Department of Defense, 29 February 1988.

[2168] Defense System Software Quality Program, DOD-STD-2168, Department of Defense, 29 April 1988.

Biography

Mr. William L. Wilder is the Acquisition Manager for Ada Language System/Navy at the Naval Seas Systems Command, PMS-412. As Acquisition Manager, he is the Contracting Officer's Technical Representative for the ALS/N FSED Contractor, Control Data Corporation and has oversight responsibility for the ALS/N IV&V efforts.

Mr. Wilder was first introduced to Ada technology as a graduate student. He has worked at both Intermetrics and SofTech on Navy and Ada related projects before joining NAVSEA PMS-412, in support of the ALS/N effort. Since then, he has provided technical direction on the Navy's prototype Ada capabilities and was promoted into a management position for the ALS/N program. His expertise is in software engineering, Ada, and the development of large applications.

Mr. Wilder received his Bachelor and Masters of Science in Computer Science from University of Maryland. He has published papers and also represents the Navy at various Ada user meetings. He is a member of the national chapters of ACM and SIGAda.

PAGES 176-186 OMITTED

AN ADA GRAPHICS PACKAGE FOR PERSONAL COMPUTERS

Jeffrey A. Fox Verlynda S. Dobbs

Wright State University
Dayton, Ohio

ABSTRACT

This paper describes the design and implementation of a general purpose graphics package written in Ada and runs on a personal computer under the MS-DOS operating system. The virtual device interface that the graphics package is based upon is compared to the standard GraphicalKernel System (GKS).

I. INTRODUCTION

This paper describes the design and implementation of a general purpose graphics package written in Janus/Ada on a Zenith Z-386 high-speed personal computer for scientists and engineers working in the the Software Engineering Laboratory at Wright State University.

An interface was developed to link the popular commercial, general purpose graphics package DISSPLA from Computer Associates, to user procedures written in Ada. The DISSPLA package, written in FORTRAN-77, is presently running on a Digital Equipment Corporation VAXstation II/GPX using the DEC VAX/VMS operating system. The pragma interface was then used as the design specification guide for a general-purpose Ada graphics package (AGP).

Section II of this paper describes the AGP implementation: required hardware and software, AGP use, and AGP procedures. Section III compares AGP to the standardized Graphics Kernel System (GKS) that is defined by the American National Standards Institute X3.124-1985 and ISO 7942-1985. Section IV concludes.

II. ADA GRAPHICS PACKAGE IMPLEMENTATION

Required Hardware and Software

Ada's development was sponsored by the United States Department of Defense (DOD). They wanted a single high-level language that would support modern software engineering principles and could be used to develop complex military systems. Why use Ada at all? One of Ada's main design goals was reliability, which is a very important aspect of sophisticated embedded weapons systems typical of DOD. The reliability feature will make Ada grow in use in commercial environments, since reliability is just as important when monitoring patients in an intensive care unit, monitoring aircraft in an air traffic control system, or performing financial transactions.

The Ada Graphics Package (AGP) is implemented on a Zenith Z-386 32-bit microcomputer running the MS-DOS operating system and using Janus/Ada Version 2.0.2 software and Talpsit Environment I, a binding from Janus/Ada to GEM, the Graphics Environment Manager (GEM), from Digital Research.

The Zenith Z-386 features the powerful zero wait state 80386 microprocessor which runs at 16 MHz with an optional 80387 coprocessor for math-intensive applications. The standard two megabytes of RAM allows the Z-386 to handle the largest integrated software packages with ease. The standard MS-DOS V3.21 and Microsoft Windows/386 provides the Z-386 with a multitasking

environment. The Zenith's high-resolution 640 x 480 video card provides compatibility with EGA, CGA, MDA, and Hercules video. The system monitor is a Zenith flat screen color monitor running at 31.5 KHz FTM.

Ada compilers are now available for 80x86 systems running either PC-DOS or MS-DOS, which makes Ada available to a much larger group of users. Janus/Ada Version 2.0.2 is a fully validated Ada compiler which conforms to ANSI/MIL-STD-1815a as determined by the Ada Joint Program Office (AJPO) under its current testing procedures. The Janus/Ada compiler requires MS-DOS Version 3.0 or greater, 640K of memory, and about 3.2 Mbytes of disk storage space. The programming environment of Janus/Ada is somewhat limited, although it contains many extras, such as a "Make" function, syntax checker and a pretty printer. There is also a profiler package that analyzes individual programs.

The Graphics Environment Manager (GEM/3) is an operating system interface containing a set of features that determines how to use the computer and its operating system. GEM/3 is simply an interface between the user and the computer. The information the user sees in a graphic form are the features of GEM/3 that the Ada Graphics Package uses to generate the graphical output. The connection between Janus/Ada and GEM/3 is through Talpsit Environment I, a binding from Janus/Ada to GEM. This binding provides a re-usable interface to the GEM VDI (virtual device interface) graphics environment. The Talpsit Environment provides data, subprograms, and type definitions which makes graphics generation possible.

Figure 1 shows an overview of the software layers needed to produce a graph using AGP. MS-DOS is installed, followed by Janus/Ada and GEM. Then the Talpsit bindings are copied into the appropriate Janus/Ada subdirectories. The AGP packages are copied into the user's directory and compiled. Then the user's Ada procedure is compiled, linked and executed producing a graph.

AGP User's Guide

AGP is a set of high-level Ada graphics packages designed especially to aid the Ada programmer working in engineering and scientific environments. AGP allows flexible and fast development of data presentation programs to produce graphs, charts, two-dimensional plots and three-dimensional objects.

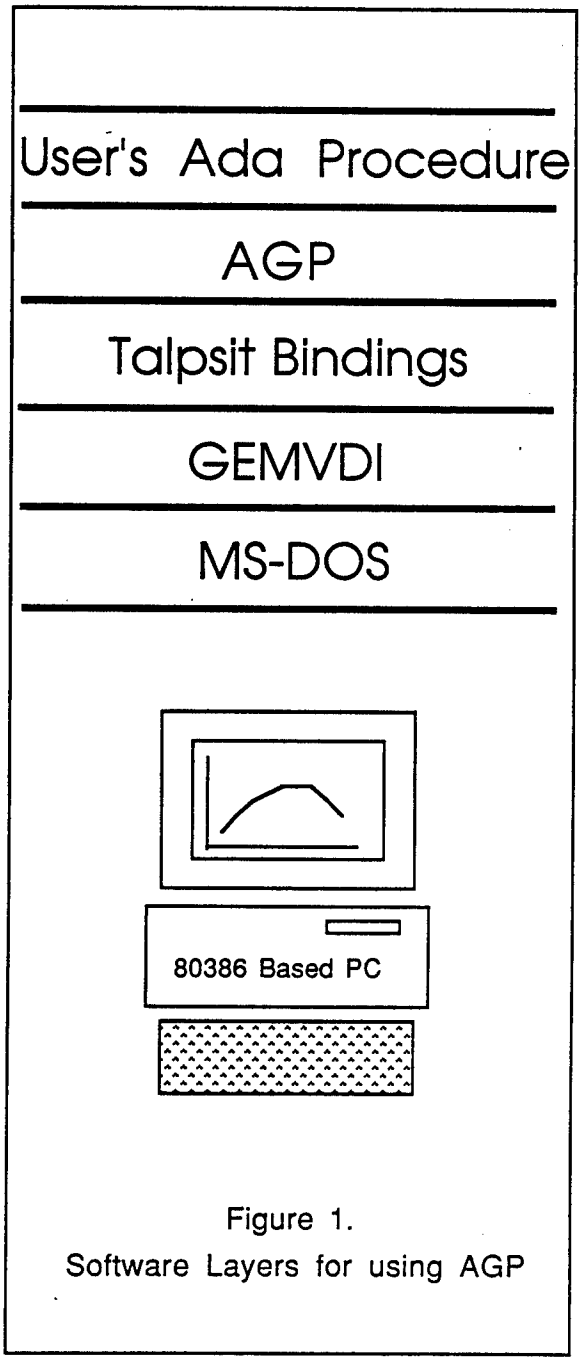


Figure 1.
Software Layers for using AGP

The purpose of visual representation is to format data so that it is easy to read and understand at a glance. The visual format must be naturally adapted to data, as well as being sharp and professional in appearance. AGP gives both maximum programming flexibility and pleasing results. AGP procedures take care of much of the mathematics, geometry, and graphical details normally involved with data representation. The AGP set of packages can remember basic structural elements requested for a plot and perform the necessary scaling and projections. The result is graphics programs that are easier to write and contain fewer lines of code.

AGP can produce simple and advanced graphics. It generates quick, simple plots with just a few lines of code; professional quality charts combining text with graphics; and specialized graphics such as three-dimensional objects. It is possible to use the same program to generate graphics with the Zenith screen or a supported printer. For example, a plot can be previewed on the Zenith screen and reproduced on a supported printer.

An AGP plot and the steps required to set up and draw a plot are presented below. After reading this section you should have a good understanding of the steps necessary to create a plot. A plot is a visual representation of data. When a plot is drawn by hand, the drawer usually takes the following steps:

1. Gather data needed for the plot and determine what kind of representation will work best to display the data.

2. Gather materials to draw the data, possibly an 8.5 x 11.0 sheet of paper and a pencil.

3. Determine the plot size and placement on the page. This amounts to sketching a box with horizontal and vertical dimensions. All data will be plotted within this box, leaving a certain amount of the page as the border.

4. Set up the horizontal and vertical (X and Y) axes along the plot area, scaling them with tick marks and labels. These axes provide a system that allows placement of data points on coordinates relative to the axes.

5. Represent the data by plotting points on coordinates relative to the axes and connecting the points with curves.

The steps in the above procedure must be performed in order, because each depends on the previous step, for example, axes cannot be drawn until data limits, plotting device, and page size is known. And curves cannot be drawn until an axis system is set up.

For comparison, the following steps demonstrate the operation of generating an AGP plot:

Step 1 - Gather the Data

The first step is to provide AGP with arrays of data. This can be done by any legal Ada method, such as reading in the data from another file, hard coding the data, or generating the data through an algorithm. Regardless of how data is passed, it should be an array represented as a variable in the AGP program. For this example, we will limit the two arrays (for X and Y coordinates) to 100 data points and generate the data for this example:

```
X : Real_array;
Y : Real_array;

For I in 1 .. 100 Loop
     X(I) := Float ( I );
     Y(I) := X(I) * 6.0;
End Loop;
```

Step 2 - Specify an Output Device

The second step corresponds to the gathering of materials. Will the plot be drawn on a screen display or a printer? In AGP, drawing materials are chosen by telling AGP what output device to use. You cannot call AGP procedures until after you have called a driver routine for the device. For this example, the Zenith screen is used:

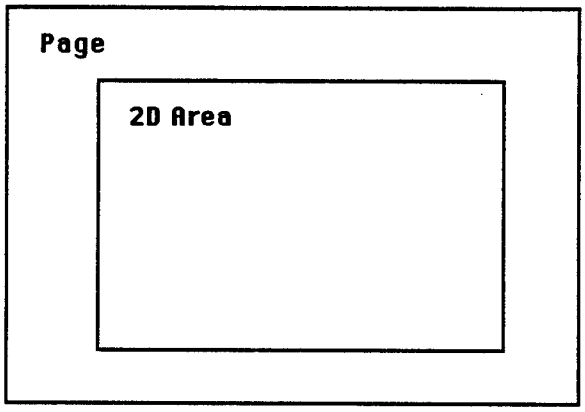
```
Set_Plot_Device_To_Monitor;
```

Step 3 - Determine Plot Size and Placement

After a device is selected, AGP procedures can be used to define a plot. The default AGP page is 8.5 x 11.0 inches. A call to the Set_Page_Size procedure will establish the area to be used by the output device. A plot area of 8.6 x 6.4 inches in the center of the page can be specified by a call to Set_2D_Area_Size:

```
Set_Page_Size    ( 10.75, 8.00 );
Set_2D_Area_Size (  8.60, 6.40 );
```

Set_2D_Area_Size defines the absolute plot dimensions in inches on the page:



AGP always works abstractly in inches and also scales a plot proportionally to fit the plotting area allowed by the chosen device. Therefore, if the plot is displayed on the system monitor, it probably won't be exactly 8.6 x 6.4 inches.

The basic text that is to be positioned relative to the plot area just defined can now be specified. For example, the user may want to center a plot title on the page and write a label for horizontal (X) and vertical (Y) axes:

```
Define_Plot_Title   ( " TITLE " );
Define_X_Axis_Label ( " X-AXIS " );
Define_Y_Axis_Label ( " Y-AXIS " );
```

Define_X_Axis_Label and Define_Y_Axis_Label writes text next to the axes, and draws the X and Y axes that will be specified in Step 4. If these routines are not called axis lines will be omitted.

Step 4 - Set up X and Y Axes

The plot area has been defined and placed on the page and text relative to the plot area has been specified. This step sets up a system for scaling data points using one of the axis-setup procedures. The call shown below specifies a linear axis setup. The X axis starts at 0.0 and ends at 100.0 with step size of 25.0. The Y axis goes from 0.0 to 600.0 with a step size of 100.0:

```
Set_Linear_X_and_Linear_Y_Axes
(0.0, 25.0, 100.0, 0.0, 100.0, 600.0);
```

Step 5 - Draw Curves and Other Plot Elements

Establishment of the axis system now allows the user to begin plotting data on the axes and drawing curves and other plot elements. One curve will be drawn in calling on the X and Y arrays defined at the beginning of the program. The curve will have 100 coordinate points with no markers at the data points:

```
Plot_Curve ( X, Y, 100, 0 );
```

Step 6 - End the Plot and Leave AGP

To end the program, the user must first end the plot and then exit AGP. End_Current_Plot will end the plot and Done_All_Plots will close communication with the output device:

```
End_Current_Plot;
Done_All_Plots;
```

There are many ways to enhance a basic plot. Some possibilities might be to add a grid, change the kind of line used, or place explanatory messages on the plot.

Shown below is a complete listing of the example procedure described in Steps 1 through 6 with an illustration of how to compile, link, and execute the procedure. Also shown is the directory where the executable file must reside. Figure 2 shows the output generated by procedure EXAMPLE on the Zenith monitor.

```
Pragma All_checks (On);
Pragma Debug(On);

With Text_io, Agpdata, Agp;

Procedure Example Is

 Use Text_io, Agpdata, Agp;

 X : Real_array;
 Y : Real_array;

 Begin

    For I in 1 .. 100 Loop
          X(I) := Float ( I );
          Y(I) := X(I) * 6.0;
    End Loop;

    Set_Plot_Device_To_Monitor;

    Set_Page_Size     (  10.75, 8.0 );
    Set_2D_Area_Size (   8.60, 6.6 );

    Define_Plot_Title   ( " TITLE " );
    Define_X_Axis_Label ( " X-AXIS " );
    Define_Y_Axis_Label ( " Y-AXIS " );

    Set_Linear_X_and_Linear_Y_Axes
    ( 0.0, 25.0, 100.0, 0.0, 100.0, 600.0 );
    Plot_Curve ( X, Y, 100, 0 );

    End_Current_Plot;
    Done_All_Plots;

End Example;
```

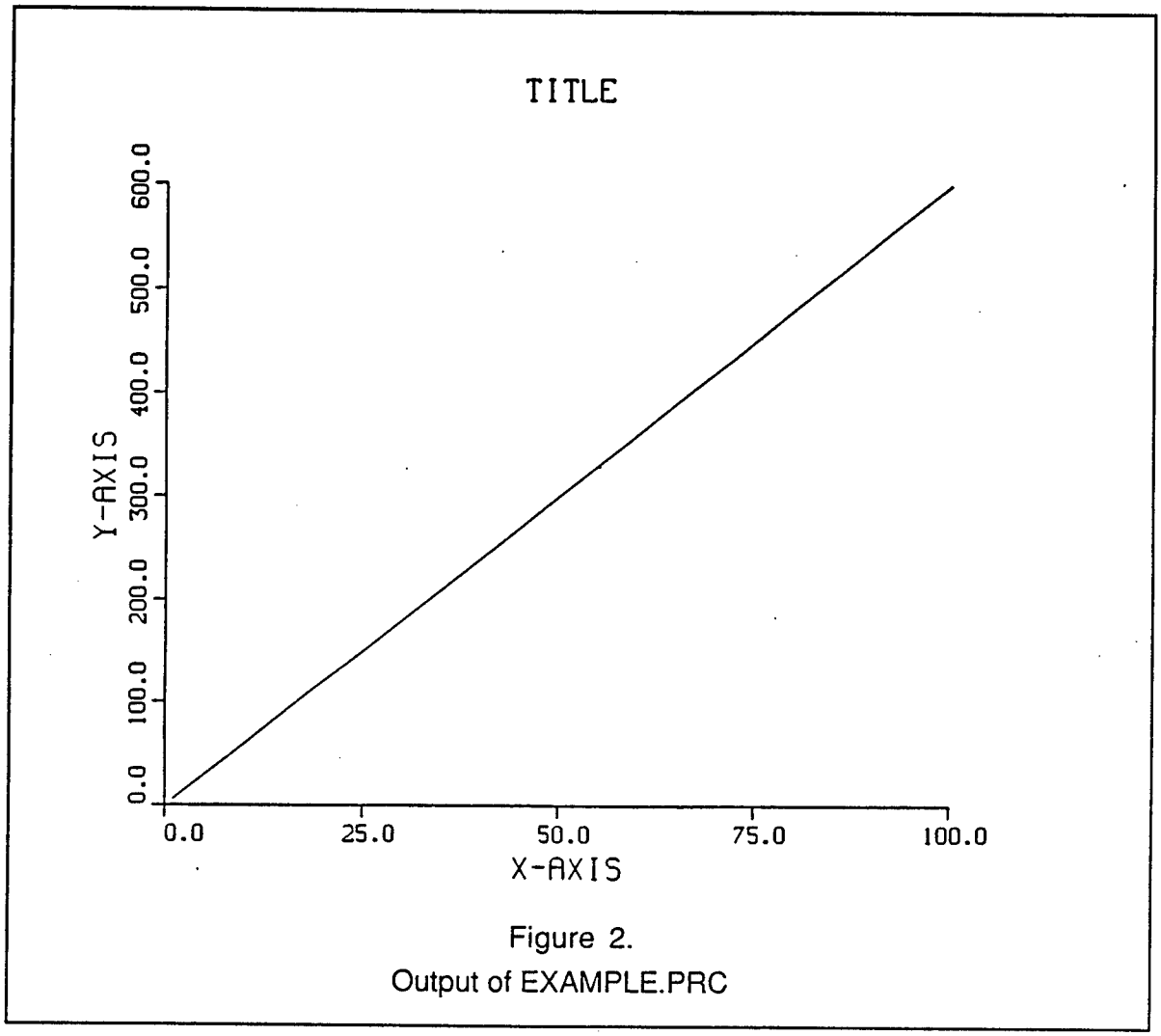


Figure 2.
Output of EXAMPLE.PRC

To compile EXAMPLE.PRC, enter

```
> JANUS EXAMPLE.PRC/O1/Z/T
```

To link EXAMPLE, enter

```
> JLINK EXAMPLE/O1/Z
```

The file EXAMPLE.EXE must reside in the directory of C:\GEMAPPS\GEMSYS before it can be executed, so enter

```
> COPY EXAMPLE.EXE C:\GEMAPPS\GEMSYS\*.*
```

Change directory to C:\GEMAPPS\GEMSYS then to execute the EXAMPLE.EXE procedure, enter

```
> GEMVDI -EXAMPLE.EXE >EXAMPLE.TXT
```

The use of the MS-DOS redirect I/O command will send normal output to the file EXAMPLE.TXT.

AGP Procedures

The three main packages of the Ada Graphics Package are AGP, AGPE and AGP3D.

Package AGP contains the basic plotting procedures which include the following procedures:

```
Set_Plot_Device_To_Monitor;
Set_Plot_Device_To_Printer;
Clear_Monitor;
End_Current_Plot;
Done_All_Plots;
Set_Page_Size;
Set_2D_Area_Size;
Set_Graph_Limits;
Plot_Curve;
Set_Marker_Type;
Define_Plot_Title;
Define_X_Axis_Label;
Define_Y_Axis_Label;
Set_Line_Color;
Set_Solid_Line;
Set_Dot_Line;
Set_Dash_Line;
Set_Chain_Dot_line;
Set_Chain_Dash_Line;
Generate_a_Grid;
Draw_a_Frame;
Delete_Border;
Set_Text_Rotation_Angle;
Output_a_Message;
Output_an_Integer_Value;
Output_a_Real_Value;
Define_X_and_Y_with_Integer_Axes;
Define_X_with_Integer_Axis;
Define_Y_with_Integer_Axis;
```

Package AGPE contains the log plot procedures and also contains the procedures to set the border, frame, and line sizes. This package includes the following procedures:

```
Set_Log_X_and_Linear_Y_Axes;
Set_Linear_X_and_Log_Y_Axes;
Set_Log_X_and_Log_Y_Axes;
Set_Border_Width;
Set_Frame_Width;
Set_Line_Width;
```

Package AGP3D contains the 3D plotting procedures which include the following procedures:

```
Define_view_point;
Draw_3D_line;
Show_All_3D_lines;
Draw_3D_object;
```

AGP creates a file called AGPERR.TXT which contains summary information about the plot being generated. AGPERR.TXT is located in directory C:\GEMAPPS\GEMSYS, where executables must reside before they can be executed. Executable files must reside in this directory so that when they are executed, the VDI part of GEM thinks they are GEM application programs.

III. COMPARISON BETWEEN AGP AND GKS

Several graphics standards have been developed or are under development. All are interface standards, in that they standardize the communication at an interface between two parts of a hardware/software system. Data interface standards specify digital representations of graphical data, as might be used to send commands from a computer to a graphical display or to store pictures in a data file. Subroutine interface standards specify the behavior of a set of subroutines that can be called to manipulate graphical data. The standards differ in the kinds of services they provide and their role in a graphics system.

Figure 3 shows how graphics standards may be used by an application program. The

Initial Graphics Exchange Standard (IGES) defines a format for engineering drawings that can be read or written by applications. Graphical Kernel System (GKS) is a subroutine interface that application programmers use to manage graphical input/output. The Virtual Device Interface (VDI) defines a standard way for a program to drive a graphics device. Virtual Device Metafile (VDM) is a data format used to record pictures on disk files.

Most of the basic concepts of GKS and VDI are the same. Both deal only with two-dimensional images. Both provide the output primitives polyline, polymarker, text, fill area, and cell array. Primitive attributes are the same: line style, marker type, color index, fill interior style, and so on. Input is provided in both cases by logical input devices of type locator, valuator, stroke, choice, and string. Inquiry functions are used liberally to determine the properties of the hardware and the state of the graphics package.

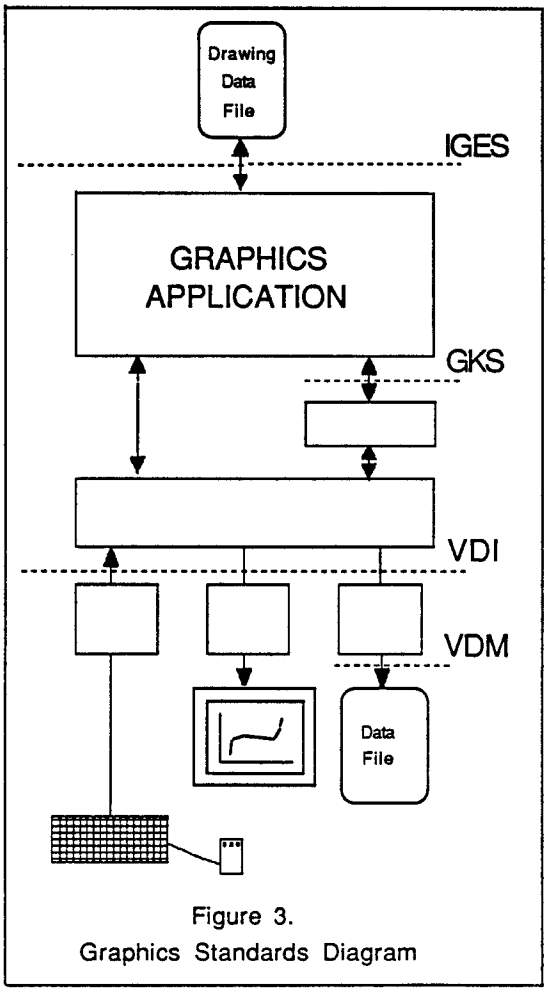


Figure 3.
Graphics Standards Diagram

Some differences between GKS and VDI are so strong that they are likely to persist as the standard VDI evolves:

- While GKS assumes that there is a single GKS package that controls all graphical devices attached to the computer, grouped into workstations, VDI functions apply to a single device. Each VDI function call is directed to a particular device.

- VDI can operate input and output devices separately, echoing the motion of an input device by making appropriate calls to an output device. In GKS, these devices are combined into a single workstation.

- VDI applies a single window/viewport transformation to coordinates; graphical objects can be clipped to the window. This mapping is simpler that the two-level mapping in GKS: world-to-NDC and NDC-to-device, where NDC are Normal Device Coordinates.

- Segments are missing from VDI; all output is direct. Since segments are not provided, neither is the pick input device.

- VDI dispenses with the generalized drawing primitive, offering instead individual output primitives that display rectangles, circles, ellipses, arcs, and polygons that may contain holes.

The VDI insists on a standard coordinate system that runs from 0 to 32767 in both x and y. This square region can be mapped in two ways to a nonsquare display; it maps to fill the screen, (0,0) is at the lower left corner, and (32767,32767) is at the upper right corner; or it maps to the smallest square that encloses the screen. The second mapping has the advantage that the aspect ratio is preserved.

One of the objectives of the VDI standard is to encourage manufactures of displays,

plotters, and other graphical devices to build VDI drivers into the hardware, thus reducing even further the size of software device drivers. Since many graphical devices require some kind of controller in any case, a standard way to control the device will offer many advantages.

IV. CONCLUSIONS

After GEM and the Talpsit bindings were installed, the test routines from Talpsit were used to verify that this interface worked. The test routines did not work without modification. The test routines that generated figures on a hardcopy printer were converted to drive the system monitor.

As the development of AGP progressed a point was reached where the AGP package would not compile due to the lack of symbol table space. Janus/Ada recommends smaller compilation units. This is why the package AGPE contains the enhanced features and the log axis procedures.

When the 3D development started a new package was created which included duplicate code contained in the basic package of AGP. This allowed the user to "with" only the AGP3D package.

Ada compilers on personal computers are a very realistic and pleasant development environment. The programming environment is growing and many tools are available to make the move to Ada a little bit easier. There is no longer any reason to avoid using Ada for software development on personal computers, even if the development is targeted for minicomputers or mainframes. Ada development with a 80386 machine is easy and the picture can only get brighter in the very near future.

REFERENCES

Computer Associates. CA-DISSPLA User's Manual Computer Associates International, 1988.

Digital Research. GEM/3 Desktop User's Guide. Digital Research, 1987.

Gilpin, Geoff. JANUS/ADA Extended Tutorial. Prentice-Hall Publishing Company, 1986.

Talpsit Technology. Talpsit Binding User's Manual. Talpsit Technology, 1988.

U.S. Department of Defense. ADA Programming Language. ANSI/MIL-STD-1815A (1983).

Jeffrey A. Fox received the M.S. degree in computer science from Wright State University in 1989. Mr. Fox has been employed by the University of Dayton Research Institute for the past 15 years as an Engineering Systems Analyst.



Verlynda Dodds received her Ph.D. in computer science from The Ohio State University in 1985. Her research interests are in the areas of software engineering, artificial intelligence, and Ada for artificial intelligence. Dr. Dobbs is currently on the faculty of the Department of Computer Science and Engineering at Wright State University, Dayton, Ohio 45435.

PROBLEMS WITH ADA NUMERIC ROUTINES

Paul D. Buck
Sherry L. Day
Daniel Gonzalez

National University
4141 Camino Del Rio South
San Diego, California 92108

Abstract

Computer programmers believe that their systems will always perform as expected. Standardization of computer languages has been a major advance towards this goal. The Ada language was intended to further advance this goal by rigorously defining the language. In addition the language was intended to improve the portability of programs. Our experience showed that the definition of the Ada language does not ensure mathematical algorithms will function consistently across multiple hardware environments. The method used by the language designers to specify the floating point data types appears to be the cause of the problem.

Introduction

There are many issues that can be addressed when discussing portability of software. We intend to concentrate on the issues associated with the portability of elementary functions and Ada. The discussion includes examples drawn from a real life project.

This paper will be a detailed account of some of the issues revealed during the implementation of "generic" elementary mathematical functions for several hardware architectures. The differences in the hardware environments limited our attempt to implement the functions. The primary focus will be to detail the pitfalls of using standard Ada code to approximate elementary math functions. The problems found, the solutions, and those problems we were not able to solve will be discussed.

The lack of strict standards for the implementation of fundamental numeric types in Ada makes it very difficult to ensure that any package will perform as expected on different hardware architectures. This means, because of hardware features, fundamental operations normally provided by a programming language should be developed specifically for the target machine. This must be done to ensure the consistency and accuracy of the elementary functions.

As the primary, and mandated language for Department of Defense applications the Ada language is intended to make writing programs easy, enhance self documentation, make programs easier to maintain, and ensure program reliability. These features were intended to make the language powerful and flexible enough so that developers would garner all the benefits of a High Order Language (HOL). Also the Ada language supports the construction of reusable packages, procedures, and functions.

Our project was to develop a portable package of elementary mathematics functions. The Ada language was selected, for the reasons stated above, as the source language for the project.

This paper will not cover other aspects of problems relating to the portability of modules developed in Ada or any other language. Furthermore, the issues involved with the implementation of math functions using fixed point or integer data types will not be covered.

Background

In April of 1989 a project was undertaken to implement a standard set of elementary mathematics functions using the Ada language. The functions to be implemented included square root, logarithm (natural and common), antilogarithms, and the fundamental trigonometric operations (sine, cosine, tangent, etc.). This library was intended to be portable over a broad range of computer architectures, requiring no changes to the code when it was moved to a new machine.

It should be noted by the reader that this is not a new project. Nor were we the first group which uncovered the problems discussed in this paper. However, it does appear that software developers keep re-inventing the wheel. This is because the literature describing problems with the implementation of numeric routines seems to be located in

in somewhat obscure places. In addition, problems with portability of numeric routines are assumed to have been "solved", and do not require any additional attention. It was certainly our impression that the implementation of numeric libraries would not be very difficult. The users of Ada need to know that the accuracy of their math calculations are not as reliable or as accurate as they might believe.

Definitions are offered in the glossary to ensure that the reader has a clear understanding of the terminology used by the authors.

Discussion

Desired Implementation

The project was to implement a standard Mathematical function library using the Ada language. This library is to be utilized by students and staff on an IBM 4381 mainframe and a standard IBM compatible personal computer. The functions to be implemented included those defined in the Cody & Waite reference [6] and some others. This includes trigonometric, exponential, logarithmic, and algebraic functions. None of the functions to be implemented are defined as part of the Ada language, though most other high level languages normally provide these functions.

Implementation Strategy

The Cody & Waite Solution Is it possible to develop portable mathematical software? This question is answered in the affirmative by Cody & Waite [6] and also in the article by Wichmann [24] which discusses some of the elementary functions. This would appear to be the solution to our project design. All that should have been required was to use the Cody & Waite implementation to fulfill our design requirements.

Cody & Waite [6] is a "cookbook" of algorithms that may be used to build approximations for the elementary functions. Unfortunately, each of these algorithms is complicated by the requirement that it must be tailored to the specific machine architecture on which it is to execute. This significantly complicates the design of a portable function library.

Each of the 16 function listed in Cody & Waite have two or three different approximation algorithms. Each algorithm is designed to ensure the best possible results from the approximation for a specific fixed or floating point number type.

What this means is that there is at least one algorithm for fixed point machines, and one or two algorithms for the different types of floating point machines. These types define the characteristics of the hardware floating point (decimal, binary, etc.). Our interest and discussion will only address the algorithms for the floating point types.

For optimum accuracy of the floating point algorithms, the developer must be able to query the machine for the characteristics of the floating point number type. This is so that the correct approximation algorithm can be selected and executed.

The two most critical parameters of the floating point system are the radix and the number of binary bits used to represent the stored mantissa. Once these parameters are determined, the correct scaling factors can be selected for the argument reduction and preliminary calculations. The Ada language does provide attributes which allow these two parameters to be easily determined.

The main polynomial expansion or rational expression must also be selected based on these two parameters. This gives the optimum approximation for the size of the mantissa and the radix of the floating point type.

For completely portable functions, the code generated is up to eight times larger than for a function coded for a specific hardware architecture. This is the result of the following:

1) Two different code sections, one for each of the two floating point systems.
2) For each of the systems in (1) above, there will be four polynomials to cover the range of permissible mantissa sizes.

Obviously if the software is adapted to a specific floating point implementation the code size problem does not exist.

Unsuccessful Software Reuse Attempts

One of the problems with the Ada language is the lack of elementary functions in the specification. As it turns out, we are not the only people who have had the desire to solve the problem of implementing these functions. Other software developers have had to develop the elementary functions. Research uncovered the source code for two of these efforts. In all cases, the results we obtained from these libraries were less than satisfactory. The cases are defined and described in the following sections.

Common Ada Missile Package The first library found was the "Common Ada Missile Packages" [3] which is a collection of several megabytes of Ada source code. The first drawback to this library is that we never did get any of the packages to compile. The basic reason was that the source code modules were too large for the compilers we were using (Table 3-1). In

addition, inspection showed that the source code did not have the degree of portability desired. The developers for this library assumed that the target hardware would have a specific radix and mantissa size. This was indicated by the use of only one algorithm and polynomial expansion for each function. This library was determined to be inadequate since it did not address the portability requirements our design.

Compiler	Version	Comments
Meridian AdaVantage	2.1	Validated against ACVC 1.8 23Jul87
Telesoft Telegen2	1.1	Copyright 1986 & 1987
Telesoft Telegen2	2.1.0	Copyright 1986 & 1989

Table 3-1

Software Manual For The Elementary Functions We uncovered two different versions of the next library reviewed. These libraries were developed by W.A. Whitaker & T.C. Eicholtz of the U.S. Air Force [23]. After modification, the first version was compiled on both machines (IBM 4381 & PC). The second version of this library had comments which indicated that it was tried on a number of different machines and compilers by it's developers and that it functioned correctly. We could not duplicate these results because the library would not compile on either of our machines. Further work was not attempted since there did not appear to be any significant differences between the two versions (with the exception of more documentation in the second version).

Both versions of this library used an internal "Mantissa Type". This mantissa type was intended to enable the user to control the accuracy of the functions. The default is to use the standard Ada float type. The two machines (IBM 4381 & IBM PC) which we used did not do very well with this library. This occurred because the "Mantissa Type" relies extensively on the accuracies of the underlying floating point type (which is all we used in our tests). The number of conversions in and out of the "Mantissa Type" appeared to be the main source of the contamination of our results. As the user is not allowed to define a higher precision floating point type easily, a new floating point type with greater accuracy was not tried. The use of extended precision fixed point types was not tried either. Our project was to develop an operational package (and not to "fix", or analyze earlier efforts) so any additional research on the defects/failings of these packages will have to wait.

The main test we used on this package was the "Savage" benchmark [19] (Table 3-2) On the personal computer with the AdaVantage compiler (Table 3-1); the tangent function reported an error, on each iteration, of "Argument out of range". On the IBM 4381 mainframe, using the Telesoft version 1.1, there was no error message. However, the results had a cumulative error of 24,000+ and an answer of about 100. This is the inverse of the correct answer of 25,000 and an error of 0. This is a clear indication that the library is not acceptable for our purposes.

Pseudo Code of the Savage Benchmark

```
A := 1
Loop 24999 times
 A := tan(atan(exp(ln(sqr(A*A)))))+1
abs(Error := 25000 - A)
Print A, Error
```

Ideal results are A=25000, Error = 0

Table 3-2

Tutorial Material on the Real-Types in Ada The last version of the elementary function libraries was found in the article by Wichmann [24]. This article implemented one elementary function module (sine/cosine) using a fixed point type. As published, the package would not compile due to errors in the source code. More than likely these were simple typographical errors in the original article. This code would not compile using the Meridian compiler (on the PC) even after these errors were corrected.

The first error message stated that the personal computer implementation of the compiler did not support fixed point types with ranges. When the range qualifier was removed from the declaration the compiler insisted that the user must specify a range with all fixed point types. Because of this deficiency, we decided that this library function did not provide the portability that we required.

Even though it was flawed, this implementation did give the team some ideas on how portable routines could be developed. This implementation did indicate how properly written elementary functions could be made more portable (at least theoretically).

Cody & Waite Shortfalls

During the first stages of the project, there were high hopes for a successful completion. The project did not appear to have significant, unsolveable issues related to it's completion. the first attempts to prototype some aspects of the project quickly brought

significant issues to light.

The first problem encountered was that programs which compiled correctly on two different machine architectures, gave different results when executed on those machines. The differences in the results are so significant that a detailed discussion of those differences is beyond the scope of this paper (see table 3-4 & 3-7).

Our belief had been that programs which compiled and executed as expected on one machine would perform in a similar manner on other machines if it compiled correctly there. Unless, of course, the compiler used on one of the machines generated bad code due to internal errors (bugs). The Ada Language Reference manual (LRM) does not give any indication that a program might not execute as expected on both machines. Our research revealed that this is a serious problem which should, and must, be addressed.

Major portability Problems

The implication in the Cody & Waite book is that the developer has the ability to develop (code) a truly portable function. Provided the developer implements all of the required steps of all of the algorithms (see earlier paragraph), and develops the code as a "generic" function. The characteristics (mentioned previously) must be used to generate internal control constants. These control constants will be of different values for all of the types used to instantiate copies of the "generic" package. The user may now use the function with any floating point number type on any hardware architecture. The selection of the correct approximation algorithm will be made automatically by the use of the control constants. One of the Cody & Waite algorithms requires the values shown in table 3-3 to be calculated. Since these values will not change during the execution of the function, they should be constants.

```
Significand_Bits : constant := real'Mantissa
  --the size of the real mantissa in bits
Type Int is range 0..4*2**(Significand_Bits
    /2);
Y_Max : constant Int := Int(Pi * 2**
   (Significan_Bits/2) + 0.5);
  --the max argument that can be processed
  --by the function
EPS : real := 2.0 ** (Significan_Bits/2);
  --The smallest recognized value in the
  --floating point system
```

Table 3-3

The compiler may "want" these derived constants to be "static". If this is the case, then the generic package we are discussing will not compile. This is a problem with the LRM not being specific enough. Or is it a problem with the implementation of the compilers we used? We were not able to determine which was the source of our problem. The problem with this approach is that not all Ada compilers will allow these types of declarations in "generic" packages. If the type "real" is a type declared in a standard declaration, there is no problem. If, however, it is a generic formal parameter; then our experience is that the compiler on the mainframe will have an error. If it is not possible to use these "generic" packages with any floating point type, then portability is reduced or eliminated.

This approach of using a "generic" package was proposed in a paper by J. Kok & G.T. Symm [16]. The paper does not describe the experiences the authors had in attempting to build their package. However, the approach does appear to be valid. Once again, our experience indicates that true portability cannot be guaranteed even with this method. Another paper [17] does state that the author's company has managed to implement a package which is truly portable.

To accomplish this portability, their paper indicates that the package uses a specially defined floating point type which is a record containing the required components. In this manner the developers have gotten around all of the limitations of the intrinsic floating point types. A floating point type of any required precision may be defined.

The difficulty with this approach is that all of the fundamental operations must be developed to support the new type including addition, subtraction, etc. An additional requirement is that the attributes for the newly developed type must also be overloaded. It is not clear if Ada will allow the attributes to be overloaded or how that may be accomplished. The paper did not indicate if this had been accomplished or not.

Minor Problems During the development, the team discovered some minor problems which caused varying degrees of difficulty. The problems are listed in no particular order of importance. A minor issue which arose during the development of the routine to approximate the exponential function. Cody-Waite [6, page 70, note 9] indicates that the output routine on non-binary (IBM 4381 uses radix 16 floating point) machines requires a different output conversion. The conversion involves two variables (not defined by the original authors) and an extra step to scale the output values. Our implementation did not include this extra conversion for the non-binary machines. The need for this extra conversion is questionable since the IBM 4381 appeared to give correct results.

The operations in chapter 2 [6] are

not clearly explained. Some of them have only one example. Then the authors throw the reader on the tender mercies of their references. The team looked for almost a month before we could find one of the references. This made it difficult to be sure that the author's definitions were correctly understood by the development team.

Another problem in the use of Cody & Waite [6] is the MACHAR routine. Each time it was run on the personal computer it halted due to a numeric exception. At first it was assumed that the routine was incorrectly implemented in Ada. Reference [7] indicates that there is a problem with the original algorithm. This algorithm will not function correctly when the internal results are calculated to a greater precision than what is stored in the memory. This is the situation which occurs when the machine uses an implementation of the IEEE 754 standard. We were not able to obtain the updated algorithm and did not verify if it corrected this problem.

To obtain any value which would normally be provided by MACHAR, we used Ada attributes. Our original plan was to cross check the two (MACHAR & the attributes) but without the updated code this was not done.

Another interesting problem occurred during the testing of the functions. The first version of the 4381 compiler caused a float type conversion of the integer value 0 to the value of -7.23E+75. This for some strange reason cause the polynomial approximation to be biased away from the expected result. To correct this problem an "IF" test was used for integer values of 0, On detection the direct assignment of a 0.0 value occurred. Otherwise a type conversion was performed normally.

Testing of the Developed Functions

Several generic test routines were developed to test the functions coded. Of these test routines the most general was titled "Table_Test". This routine could test any function which returned a floating point value, and had a single argument. For test, tables of arguments, "ideal" values, upper and lower limits are stored in an array of records. This array, the actual type of the function, and the function to be tested are used to instantiate the test procedure (table 3-4 & 3-5). Once instantiated, the arguments are sent to the function, and the returned values are compared to the "ideal" and the limits. For most of the tests we did the data came from the NBS Minimal Basic Standard [9]. This method was preferred over other methods [5] due to it's simplicity and understandibility. There is some evidence that these tables have errors in the test data for the tangent function.

Successful Software Work Arounds

As mentioned in the article by R.A. Weissensee [21], the issue of the definition of (and the deficiencies in the definitions of) the "short" and "long" types is causing problems in the ability to reuse software. This is an unfortunate outcome of the LRM's lack of a strict specification for the implementation for data types.

This problem caused unexpected difficulties in the implementation of the random number generators. The algorithm for a generator sometimes required that the range of the integer type be capable of handling values up to 2.14E+9. This range is the same as the Long_Integer type on the Meridian compiler. In contrast, the IBM mainframe compilers did not have the Long_Integer type. However, the same range is available in the Integer type (see table 3-6).

In contrast to the IBM 4381's first compiler, the PC had wider dynamic ranges for the floating point types. In the case of one random number generators [14], it gave the correct results. This same generator failed to operate correctly on the 4381 until the updated compiler was available. As it turned out, the generator requires that the floating point type have the ability to store about 12 digits of accuracy. Originally the IBM 4381 could store only about 6 digits accurately. This caused the generator to return incorrect results. With the 15 digits of accuracy both machines returned correct results (Table 3-7).

Similar problems with floating point types (long & short) also occurred on the mainframe. When an updated compiler was installed the data type characteristics changed. This type of problem can make a library useless. Unless a data type with the required range or greater is available on the machine the conversion is not possible.

We solved our problem for the two machines by defining our own type and let the compilers determine how to implement the actual type.

```
type Test_Record is
  record
    Argument : real;
    True_Value : real;
    Low_Bound  : real;
    High_Bound : real;
  end record;

generic
  type Index_type is range <>;
  type Array_Type is array(Index_type range <>) of Test_Record;
  with function Test_Function (argument : real) return real;
  procedure Table_Tests (In_Table : in Array_Type; Fail_Count : in out natural;
                         Title_String : in string);
-- These declarations allows the use of an external data structure to test any function.

Title : string(1..2) := "Ln";
Min_Index : constant := 1;      Max_Index : constant := 2;
type Index_type is rnage Min_Index..Max_Index;
type Array_Type is array (Index_Type range <> ) of Test_Record;
Fail_Count : natural := 0;
Test_Stuff : array_Type (Min_Index..Max_Index) :=
           ((0.100000000E-37, -0.874982335E+02, -0.874982936E+02, -0.874981735E+02),
            (0.100001000E-37, -0.874982235E+02, -0.874982836E+02, -0.874981635E+02));
function Ln is new Math_Package.Ln_Generic(real, real);
procedure Test_Ln is new Test_Generics.Table_Tests(Index_Type, Array_Type, Ln);
  begin
    Test_Ln(Test_Stuff, Fail_Count, Title);
  end Ln_Tests;
-- This shows the instantiation and use of the Table_Test procedure with the Ln function.
```

Table 3-4

Test	Argument	True Value	Computed Val.	Error Measure	Result
Sine Test Data	-2.300000E+0	-7457052E-1	-7.457052E-1	2.46712482E-5	Ok
Exponent Test Data	-2.222220E+0	1.083683E-1	1.083683E-1	2.52479893E-5	Ok
Cosine Test Data	-9.876540E+4	9.999484E-1	9.999484E-1	7.74582982E-8	Ok
Tangent Test Data	-5.708100E-1	7.313574E+4	7.313574E+4	3.48613769E-10	Ok
	-5.708000E-1	2.722422E+5	2.722418E+5	8.91619718E-6	** Fail **

Table 3-5

Data Types	Personal Computer	4381 Ver 1.1	4381 Ver 2.1
Short Integer	Has Byte Type	Not Available	-32768..32767
Integer	-32768..32767	-2.14E+9..2.14E+9	-2.14E+9..2.14E+9
Long Integer	-2.14E+9..2.14E+9	Not Available	Not Available
Short Float	Not Available	Not Available	Not Available
Float	-1.79E+308..1.79E+308	-7.23E+75..7.23E+75	-7.23E+75..7.23E+75
Long Float	Not Available	Not Available	-7.23E+75..7.23E+75
System Max Digits	15	6	15

Table 3-6

Expected Value	PC's Output	4381 Ver 1.1	4381 Ver 2.1
6533892	6.53389200000000E+6	4.89063E+6	6.53389200000000E+6
14220222	1.42202220000000E+7	4.92264E+6	1.42202220000000E+7
7275067	7.27506700000000E+6	7.10038E+6	7.27506700000000E+6
6172232	6.17223200000000E+6	1.51204E+7	6.17223200000000E+6
8354498	8.35449800000000E+6	9.64838E+6	8.35449800000000E+6
10633180	1.06331800000000E+7	4.27274E+6	1.06331800000000E+7

Table 3-7

Summary

Besides all that has been discussed earlier in the paper, we also have the following problems. . .

The designers of Ada did not decide to give the programmer the ability to determine (at least not easily) the actual values of the exponent or mantissa. To determine these values, which are critical for some function approximations, the programmer must use time consuming high level code.

Even worse is the statement in the Ada Language Reference Manual that: "Examples of permissible variations are: The represented values of fixed or floating numeric quantities, and the results of operations on them." [1, Page 1-2, paragraphs 1.1.1/16 & 17]. This seems to contradict the statement that the goal was to enhance portability of the language over many hardware implementations [1, Page 1-1].

The issue of portability is of significant interest to software engineers. The consensus is that Ada has successfully addressed this issue (See Table 4-1). In fact, Ada does not adequately address the issues of portability with regard to the floating point number systems and the operations on these numbers.

One specific is the attribute Machine_Rounds. This attribute does not specify which operations it applies to. In other words, the computer must return the value "False" if the machine does not round for one of the predefined operations [1, Paragraph 13.7.3 & 21], even if the machine does round for all of the other operations. Worse yet, the rounding method used by the machine cannot be determined. One paper, not referenced, indicated that there were almost too many methods to count them. The IEEE 754 standard has several methods which might have been selected. It is regrettable that the LRM does not require the use of this widely used standard.

In contrast to the quote in Table 4-2 we discovered that Ada numeric software is not portable or even transportable, to any significant extent. In other words, any software which has a requirement to perform mathematics beyond simple operations is going to require extensive tuning for different hardware architectures.

These are self imposed constraints. As was indicated earlier, it is possible to develop truly portable packages in Ada if the user is willing to develop all of the required supporting operations for the specially designed data type.

One of the interesting things we discovered is that we are not the only group which has had similar problems with the math function libraries. In another paper [12], the author indicates that he discovered the same problems that we did. His project also started out with a developed library, which he found to be incomplete and inaccurately implemented. In addition, he found it necessary to select a particular implementation of the Ada compiler to obtain the desired result. So much for portability. The application of Ada to real world problems will be severely restricted until these problems are corrected.

The first item requiring attention is the Ada LRM and it's definitions of the

"This standard specifies the form and meaning of programs written in Ada. Its purpose is to promote the portability of Ada programs to a variety of data processing systems. " [1, Page 1-1, 1.1]
"An algorithm written to rely only on the minimum numerical properties guarenteed by the type definition for model numbers will be portable without furthur precautions." [1, page 3-19, 3.6.6/8]
"However it is possible to write programs that are guaranteed to be portable." [8, page 6]
"Ada's designers have taken great pains to maximize Ada's extensibility without sacraficing portability - the ability to run on many different machines." [15]
"DOD initiated the Ada program to save taxpayers money through standardization. The savings will come from the portability and reuse of operational software..." [4]
"Ada is designed to be universal, totally portable, and exceptionally reliable. ...it must handle math as competently as ..." [18]
"The goals of such a high order language are well agreed upon...transportability allows the reusing of major portions of software and tools from previous projects ..." [22] NOTE: This reference was describing the goals of the DOD's Common High Order Language Effort, the project which resulted in the Ada Language.
"The business world is now discovering the software engineering attributes and portability features that make it (Ada) so appealing to the defense community. Reliable, maintainable, portable software is often as important for a dynamic economic model as it is for a fire control system." [10]

Table 4-1

"Although the Ada language was not designed for numeric work, it meets most of the obvious requirements for numerical computation." [13]

Table 4-2

real types. Even after a couple of months of study it is not clear what these definitions mean. These sections have been written so that virtually no one can understand what is being said. I doubt that many users of Ada truly understand how their programs are affected by the lack of definitions that are clear. The lack of strict requirements for floating point types needs to be addressed first.

We would like to propose a better way of expressing the definitions of the LRM, but since they are not at all clear we are unable to do so. As an example, try to find a clear definition of what a "Safe" number is (HINT: see [1, Paragraph 3.5.6/4]). After the definition has been found, define what is really meant by the term "Error Bounds". The subject of model numbers is also not clearly explained. We did find a number of definitions in other texts. But they differed, so none of them can be considered definitive.

To solve this and the other problems described in the paper, the simplest solution seems to be to change the Ada language specification to require compiler writers to meet a designated level of accuracy for floating point numbers. An easy solution would be to change the Ada specification to require the use of the IEEE 754 standard for binary floating point. This would ensure that programs written on one computer hardware configuration, would return similar results when executed on other hardware. This puts the burden for these matters on the compiler developers, rather than the users (see table 4-3).

"Functions included in a language standard might be implemented poorly by many compiler writers, while functions omitted from the standard might be implemented poorly by many users." [11, page 31, Du Croz paraphrasing Meissner]

Table 4-3

Next a suite of standard functions, perhaps in a package form, like Text_IO, should be added to the specification. This package should, as a minimum, include all of the functions in Cody & Waite [6]. With these changes, conforming Ada compilers would be tested by the ACVC for correct implementations of the IEEE 754 standard. In addition, the minimal acceptable accuracy for the elementary functions would be tested. This would make Ada the first language which would allow the developer to have reasonable faith that the functions were returning values which were consistant with the functions definition. This would be because both the floating point types and the elementary functions would be tested for compliance. Most language standards only specify some of the attributes of the floating point types. No other language specification (that we are aware of) mandates any level of accuracy for the elementary functions implemented. Perhaps Ada should lead the pack and be the first language specification to do so [2].

To those who would say that the IEEE 754 standard cannot be implemented on mainframe computers, we have a reply. If the microcomputer people can make software emulators which run on many PCs, similar emulators can be written for any mainframe computer. For those lucky compiler writers who have access to hardware accelerators (which comply with the IEEE 754 standard), their jobs are just a bit easier. In closing the quote in Table 4-4 is most appropriate.

"It is impossible to define a minimum quality of the hardware arithmetic and to condemn as intolerable what is below that limit. The reason is that one can overcome any shortcomings of the hardware by appropriate countermeasures in the software." [20]

Table 4-4

Glossary

Attribute A function or other operation which yields a predefined characteristic of a named entity.

Exception An unexpected, abnormal operating condition which should cause the executing program to halt.

Instantiation To represent an abstraction by a concrete instance. The process by which a "generic" procedure or function is made into an executable code module with the appropriate data types and objects.

LRM Ada Language Reference Manual, ANSI/MIL-STD-1815A, 22 January 1983

Machine epsilon The value representing the smallest spacing between the value 1.0 and the next value above. This represents the smallest interval between model numbers. As defined in the Ada Language Manual, 1.0 + Machine_Epsilon /= 1.0; 1.0 + (any number < Machine_Epsilon) = 1.0 [1, section 3.5.8]

Model Number Those numbers which can be represented exactly by the floating point representation.

Normilization This is the process of aligning the binary pattern in the floating point variable so that the leading digits have no zeros in them. This ensures that the greatest possible number of digits may be accurately represented by the floating point variable.

Overloading When an identifier can have several alternative meanings at a given point in the program text.

Portable Software Software which may be moved from one environment to another without change.

Safe Number A value which is exactly representable by the floating point representation, and is a member of a super-

set of numbers which enclose all of the model numbers (the set of model numbers). As stated in the LRM [1, 3.5.6/4] the safe numbers provide guaranteed error bounds for any operation on a range of numbers.

Transportable Software Software which may be changed by a mechanical pre-process to a form which is executable on the new target environment.

References

1. Anonymous; ANSI/MIL-STD-1815A-1983, Reference Manual for the Ada Programming Language; United States Department of Defense; February 17,1983

2. Black, Cheryl M.; Burton, Robert P. & Miller, Thomas H., The Need for an Industry Standard of Accuracy for Elementary Function Programs; ACM Transactions on Mathematical Software; Volume 10 #4; December 1984; Pages 361-366

3. Common Ada Missile Package

4. Carlson, William E & Fisher, Dr. David A.; First Complete Ada Compiler Runs on a Micro; Mini-Micro Systems; September 1982; Pages 207-219

5. Cody, William J. Software for the Elementary Functions; in Mathematical Software; Editor Rice, J. Academic Press; New York; 1971; Pages 171-186

6. Cody, William J. & Waite, William; Software Manual for the Elementary Functions; Prentice-Hall; 1980

7. Cody, William J. MACHAR: A Subroutine to Dynamically Determine Machine Parameters; ACM Transactions on Mathematical Software; Volume 14; #4; December 1988; Pages 303-311

8. Cohen, Norman H. Ada as a Second Language; McGraw-Hill; 1986

9. Cugini, J.V. Bowden, J.S. & Skall,M.W. NBS Minimal Basic Test Programs (version 2) NBS Special Publication 500-70/2; November 1980

10. Dortenzo, Megan; Ada Out of Uniform; PC Tech Journal; April 1989; Pages 86-106

11. Du Croz, J.J.; Programming Languages for Numerical Subroutine Libraries; in The Relationship Between Numerical Computation and Programming Languages; edited by Reid, J.K. North-Holland; 1982; Pages 17-32

12. Frush, James Anthony; An Implementation of the Standard Math Functions in Ada; Proceedings of the 7th Annual National Conference on Ada Technology; 1989; Pages 226-229

13. Hammarling, S.J. & Wichmann, B.A.; Numerical Packages in Ada; in The Relationship Between Numerical Computation and Programming Languages; Reid, J.K. editor North-Holland; 1982; Pages 225-244

14. Harmon, Marion G. & Baker, T.P.; An Ada Implementation of Marsaglia's "Universal" Random Number Generator; ACM Ada Letters [SIGADA]; Volume VIII #3; 1988 Pages 110-112

15. Johnson, R. Colin; Special Report: Ada the Ultimate Language?; Electronics; February 10, 1981; Pages 127-132

16. Kok, J. & Symm G.T. A Proposal for Standard Basic Functions in Ada; Ada Letters; Pages IV.3-44 to IV.3-52

17.Leavitt, Randal; Adjustable Precision Flaoting Point Arithmetic in Ada; ACM Ada Letters; Volume VII #5; Pages vii.5-63 to vii.5-78

18. Loveman, David; Ada Defines Reliability as a Basic Feature; Electronic Design; September 27, 1980; Pages 93-98

19. Plauger, P.J. Programming on Purpose: Computer Arithmetic; Computer Language; Volume 5 #2; February 1988; Pages 17-23

20. Reinsch, Christian; Some Side Effects of Striving for Portability; in Portability of Numerical Software; edited by Cowell, W. Springer-Verlag; 1977 Pages 3-19

21. Weissensee, Robert A.; An Interim Ada Based Preliminary and Detailed Program Design Language, Part II; Journal of Pascal, Ada & Modula 2; September/October 1985; Volume 4 #5; Pages 5-12

22. Whitaker, William A. LtCol. USAF; The Department of Defense Common High Order Language Effort; ACM SIGPLAN Notices; Volume 13 #2; February 1978; Pages 19-29

23. Whitaker, W.A. & Eicholtz, T.C An Ada Implementation of the Cody-Waite "Software Manual for the Elementary Functions"

24. Wichmann, B.A.; Tutorial Material on the Real Data-Types in Ada; ACM Ada Letters Volume I #2, Pages I-2.15 to I-2.33



Daniel M. Gonzalez
4241 Jutland Drive
San Diego, California 92117

Mr. Gonzalez retired after 22 years from the United States Navy as a Master Chief Electronics Warfare Specialist. He is presently employed with a Department of Defense Contracting firm in San Diego, California. Employment is in the capacity of a software engineer involved in the design, development, and validation of Navy embedded Command and Control Systems, as well as conversion of existing DoD software to Ada. Mr. Gonzalez received his Undergraduate degree in Computer Science and Graduate degree in Software Engineering from National University in San Diego, California. Previously from Los Angeles, Mr. Gonzalez now makes his home in San Diego, California.



Sherry L. Day
3957 Bob Street
San Diego, California 92110

Miss Day was previously a Data Processing Specialist for 5 years with the United States Navy. Miss Day is presently employed by a Department of Defense contracting firm in San Diego, California. Employment is in the capacity of a Configuration Manager for the development of Navy embedded Intelligence Correlation, Combat Information Dissemination and Data Fusion systems being developed in Ada. Miss Day received her Undergraduate degree in Computer Science and Graduate degree in Software Engineering from National University in San Diego, California. Previously from Minnesota, Miss Day now makes her home in San Diego, California.



Paul D. Buck
10210 Keppler Place
San Diego, California 92124

Mr. Buck is a Senior Chief Aviation Electronics Technician, United States Navy, who will be separating from active duty in March 1990 after completing a 20 year career. His educational background has been with Navy Technical schools as a student and as an instructor. Collegiate work was at National Unversity where he graduated in 1988 with a Bachelor's degree in Computer Science (Summa Cum Laude) and in 1989 with a Masters Degree in Software Engineering. He currently resides in San Diego with his wife Nancy and dog "Sam".

IAT : AN INTELLIGENT ADA TUTOR

S. PALANISAMY

COMPUTER SCIENCE DEPARTMENT
OKLAHOMA STATE UNIVERSITY

ABSTRACT

This paper describes about an Intelligent Tutoring System called Intelligent ADA Tutor (IAT), to teach ADA to students. IAT has two parts, Tutor and Expert Diagnostician. Tutor presents the ADA course material in an adaptive manner using the concept of a perfect student model. IAT assesses the student performance continuously by asking questions and presents review material, if necessary. Expert Diagnostician attempts to interpret the ADA compiler error messages like a human tutor. This gets the compiled version of a student program, interprets the error messages and gives the proper advice with a review on the relevant course material, if required.

INTRODUCTION

Computers are now being used in many ways by forward looking educators. Once regarded merely as teaching machines[16], these electronic devices are beginning to be utilized more creatively by even the most cautious educators. For example, Taylor[24], views computers as vehicles for instruction (tutor), assistance (tool), and creative problem solving (tutee).

An expert system is a computer-based system that uses knowledge, facts and reasoning techniques to solve problems that normally require the ability of humanexperts[11]. The knowledge the expert system uses is made up of either rules or experience information about the behaviour of the elements of a particular subject domain. The goals sought by expert system designers in the development of expert systems include substituting for an unavailable human experts, training new experts, providing requisite expertise on projects that do not attract (or) retain experts, and providing expertise to projects that cannot afford experts[11].

To develop an expert system in a particular domain, we need a human expert in that domain who must be willing to present his knowledge to the knowledge engineer, i.e. an expert system designer. Because, an expert system is a method of incorporating human expertise in a particular domain into the computer so that the precious human brain can be used for more important tasks.

An expert system approach to the computer aided instruction is not an idea which is developed recently. Many expert intelligent tutors are already available in the market. One typical example may be an intelligent LISP tutoring system[25]. IAT presents the course material according to the students' level of expertise. It keeps track of the students' performance dynamically during a consultation by asking questions about the material presented. Upon presenting the material, if the student's observed performance level is less than the desired performance level, IAT provides review on those chapters to whose questions the student has given an incorrect response. IAT also provides diagnostics to errors in student programs in ADA.

Intelligent Tutoring Systems (ITS) evolved from CAI, Computer-Aided Instruction. IAT has three components of "Intelligence", expert knowledge, diagnostics knowledge and the curriculum knowledge[18]. The first component requires that the subject matter should be well known to the computer so that it can solve problems in that domain. The second component requires that the system must be able to determine the students' level of knowledge acquired in that domain. The third component requires that the courseware should be arranged in an intelligent manner so as to handle the vagaries of student expertise. ITS usually take the form of computer-based 1) problem-solving monitors, 2) coaches, 3) laboratory instructors and 4) consultants[18]. The domain for ITS are usually carefully chosen so that a justification can be derived for replacing a human tutor on that field.

The Intelligent Tutoring Systems of today incorporate several features not found in CAI programs of yesterday. Moreover, the ITS's tend to address limited topics.

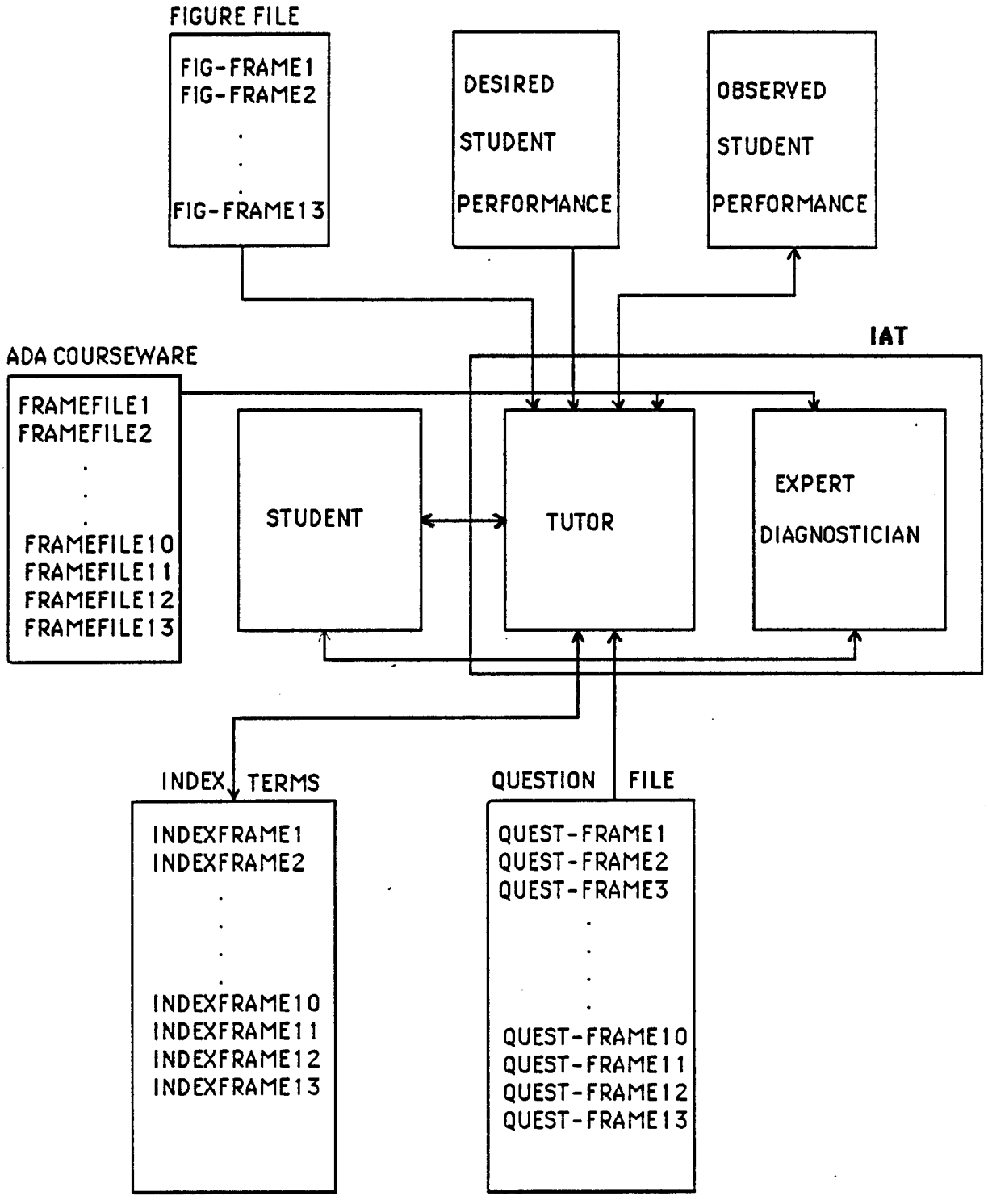
DESCRIPTION OF INTELLIGENT ADA TUTOR (IAT)

This paper presents the design of an intelligent tutoring system to teach the programming language ADA. IAT is designed using the concepts of student modeling[6]. This uses index terms for judging student's level of expertise about a construct. Index terms are terms with a predefined set of values maintained for each construct of ADA. Typically, IAT uses three values, 0, 0.5 and 1 for index terms. Each index term gets a value based on the response given by a student to a question asked about the construct it represents. Fig. 1. gives the complete block diagram of IAT.

IAT consists of two parts :

1. TUTOR
2. EXPERT DIAGNOSTICIAN

FIG 1 - COMPLETE BLOCK DIAGRAM OF INTELLIGENT ADA TUTOR



TUTOR

The Tutor is responsible for presenting the course material to students in an adaptive manner. The entire courseware is divided into 13 frames. They are shown in Fig. 2.

Each frame consists of two chapters. Chapter one provides the general explanation of a construct with appropriate illustrative diagrams and chapter two gives exact ADA syntax of that construct.

Tutor uses three windows for the tutorial. The first one is the main window (course window) that displays the course material. The second one is the illustration window, which displays the figures relevant to the construct being presented. The third one is the question window, which displays a question relevant to the construct being presented, in a multiple choice format. The student is asked to respond in the third window. Fig. 3. shows the screen display in a tutorial session of a particular construct.

Each courseware frame has a set of 15 questions. Different frame presentations use a different question and avoid repeating the same question. Tutor updates 13 variables, namely INDEXFRAME1 - INDEXFRAME13 during a tutorial session. These index terms are interpreted to indicate the level of knowledge acquired by the student from a particular frame.

Each index term can have three values, 0, 0.5, or 1. The value 0 for a term means that the student has a complete knowledge about the frame. A value '1' means that the student does not have any knowledge of the material provided by the frame, so a complete review is necessary. A value '0.5' means that the student has some knowledge about the frame, so review of only chapter 2 is necessary. Because, chapter 2 contains the syntax of the construct presented. Only one question will be asked for each frame and the answer selected by the student is used to set values for the index terms INDEXFRAME1 - INDEXFRAME13. The response from the student is also used to update the correct responses counter used for calculating actual student performance.

The two blocks, 'Desired Student Performance' and 'Observed Student Performance' are used by the tutor to present the review material to the student. The desired student performance is a perfect student model[6] used to compare against the actual student performance. The perfect student model gives a measure of the desired student performance. This is measured by the percentage of correct responses given by the student. IAT assumes the desired student performance as 60%.

The the actual performance is measured by the performance index, which is calculated using the formula :

$$\text{performance index (PI)} = \frac{\text{\#correct responses by the student}}{\text{total \#questions asked in the tutorial}} \times 100$$

PI (desired), the desired performance expected is 60, this is used as the perfect student model. Based on the student's answer for each question, a counter, namely RESPONSE COUNTER, is updated by TUTOR as follows :

For each question, three to four possible answers will be presented to the student. If the answer he selected is completely wrong (a wild guess), the response counter is incremented by -1/2. If the answer is completely correct, the response counter is incremented by 1. If the answer is partially correct, the response counter is incremented by 1/2. If the answer is "DON'T KNOW", then nothing will be added to the response counter (incremented by 0). Thus, the observed student performance index is calculated as follows :

$$\text{performance index (observed)} = \frac{\text{response counter}}{\text{total \#questions asked in the tutorial}} \times 100$$

The index term for a frame is updated as follows :

If the answer is completely wrong, then index term is set to 1. If the answer is partially correct, then the index term is set to 0.5. If the answer is completely correct, then the index term is set to 0. If the answer is "DON'T KNOW", then the index term is set to 1.

EXPERT DIAGNOSTICIAN

Expert diagnostician gets a compiled version of a student's program and interprets the error message given by the compiler. Eventhough the Ada compiler provides very good error messages, they are not completely dependable. Errors could be triggered by previous recovery. Therefore, the expert diagnostician attempts to diagnose the errors (given by the compiler) like a human tutor and gives proper diagnostic messages. In particular, the error messages given by the ADA compiler are cryptic in the sense that the error might have occurred at some point, but the compiler might have given error at some other point. Or, sometimes, the compiler might give an error as a cumulative error, i.e. there may not actually be an error at that stage, but it might have been interpreted as error due to the error in previous lines. The human tutor as a diagnostician takes care of these inconsistencies and provides advice accordingly.

OPERATION OF IAT

The flow charts in Fig. 4 to Fig. 6 describe the operation of IAT. IAT has two functions :

1. presenting the courseware frames.
2. presenting the diagnostic information.

THE COURSEWARE FRAME PRESENTATION

Before presenting each frame, IAT asks the student whether he has any prior knowledge about the construct to be presented. If the student responds as 'YES', then only the chapter containing the syntax is presented to the student. If the student responds as 'NO', then both the chapters will be presented to the student. But in either case, before presenting chapter 2, the student is asked a question related to the construct. The question may be related to any concept utilized in the frame.

FIG 2 - ADA COURSEWARE FRAMES

FRAME	DESCRIPTION
1	BASICS OF ADA - STANDARD DATA TYPES, EXPRESSIONS.
2	'IF' CONTROL STRUCTURE
3	'CASE' CONTROL STRUCTURE
4	LOOP, FOR, WHILE, EXIT CONSTRUCTS
5	TYPES - ENUMERATION, ARRAY, SUBTYPES
6	SUB PROGRAMS - PROCEDURES, FUNCTIONS
7	DATA STRUCTURES - MULTIDIMENSIONAL ARRAYS, MATRICES, CONSTRAINED ARRAYS, RECORDS
8	PACKAGES
9	INPUT AND OUTPUT
10	EXCEPTIONS
11	DYNAMIC DATA STRUCTURES
12	FILES
13	GENERICS

The questions are multiple choice questions. Based on the answer selected, IAT updates the RESPONSE COUNTER (#correct responses) and the corresponding index term. In case of an error, depending on the mistake, appropriate review material is presented. Then chapter 2 is presented to the student. The same method is adopted for presenting all the frames. After presenting these frames, observed performance index [PI(observed)] is calculated and tested with the desired value of 60. If the observed level of expertise is lower than the desired value, a further review is provided by scanning through the index terms and displaying the material based on their values. If the observed level of expertise is greater than (or) equal to the desired value, no further review is presented.

THE DIAGNOSTIC INFORMATION PRESENTATION

Upon presenting the 13 frames in the ADA courseware, the student is asked whether the diagnostician be involved. If the student responds as 'YES', then IAT gets the '---.lis' file (compiled version) from the student, interprets the compiler error messages, and gives proper diagnostic messages. Here again, based on the error found in the student program, proper review material is presented. But no question will be asked during review.

DESCRIPTION OF THE KNOWLEDGE BASE

The knowledge base is stored as frames. IAT knowledge base consists of two frames, TUTOR and EXPERT DIAGNOSTICIAN. The frame tree for IAT is given in Fig. 7.

The root frame is the TUTOR and its child frame is the EXPERT DIAGNOSTICIAN. EXPERT DIAGNOSTICIAN can access all the parameters in the parent as well as its own. In other words, EXPERT DIAGNOSTICIAN is the subframe of TUTOR.

CONCLUSION & FUTURE WORK

The following features make IAT distinct from a simple CAI program:

1. The idea of presenting course material in an adaptive manner.
2. The review material is being presented immediately following the error with the usage of a particular construct.
3. This expert system presents to the user a debugging environment unlike other CAI packages available in the market, thus making it more suitable for a student environment. Other packages offer a developing environment for the user. In the developing environment, the user will be given a standard programming assignment at the end of each chapter. The various ways of arriving at a solution are stored in the knowledge base so as to analyze the student's code against errors. But in IAT, students can submit the compiled version of any program and expect to receive informed diagnostic messages.

DEVELOPMENT ENVIRONMENT

IAT is being implemented using PCPLUS, a rule-based expert system shell developed by Texas Instruments Inc., Texas. In the future, IAT will be having more advanced features than the one presented in this paper. One of the features incorporated in the future will be an enhanced version of the perfect and actual student models. The student models are now constructed based only on the student responses to questions asked during consultation. But in the future, the models will be constructed from the student's conceptions/misconceptions about a construct[16].

At present, the index term for a courseware frame is set based only on one question. In the future, more than one question will be asked to set the value for an index term. Similarly, if the diagnostic messages given by expert diagnostician need to be changed, only the rules in the corresponding frame need to be changed. IAT is implemented in such a manner that updates will require changes only at the expert system level. A further look into the future scope of this project leads us to the possibility of extracting an ITS shell out of this IAT. This shell will provide a skeleton for an Intelligent Tutoring System. Any subject can be taught using this shell. The ITS shell will require the courseware frames, corresponding illustrative diagrams, questions to be of a common format designated by the shell.

FIG 5 - FLOW CHART FOR PRESENTING A FRAME

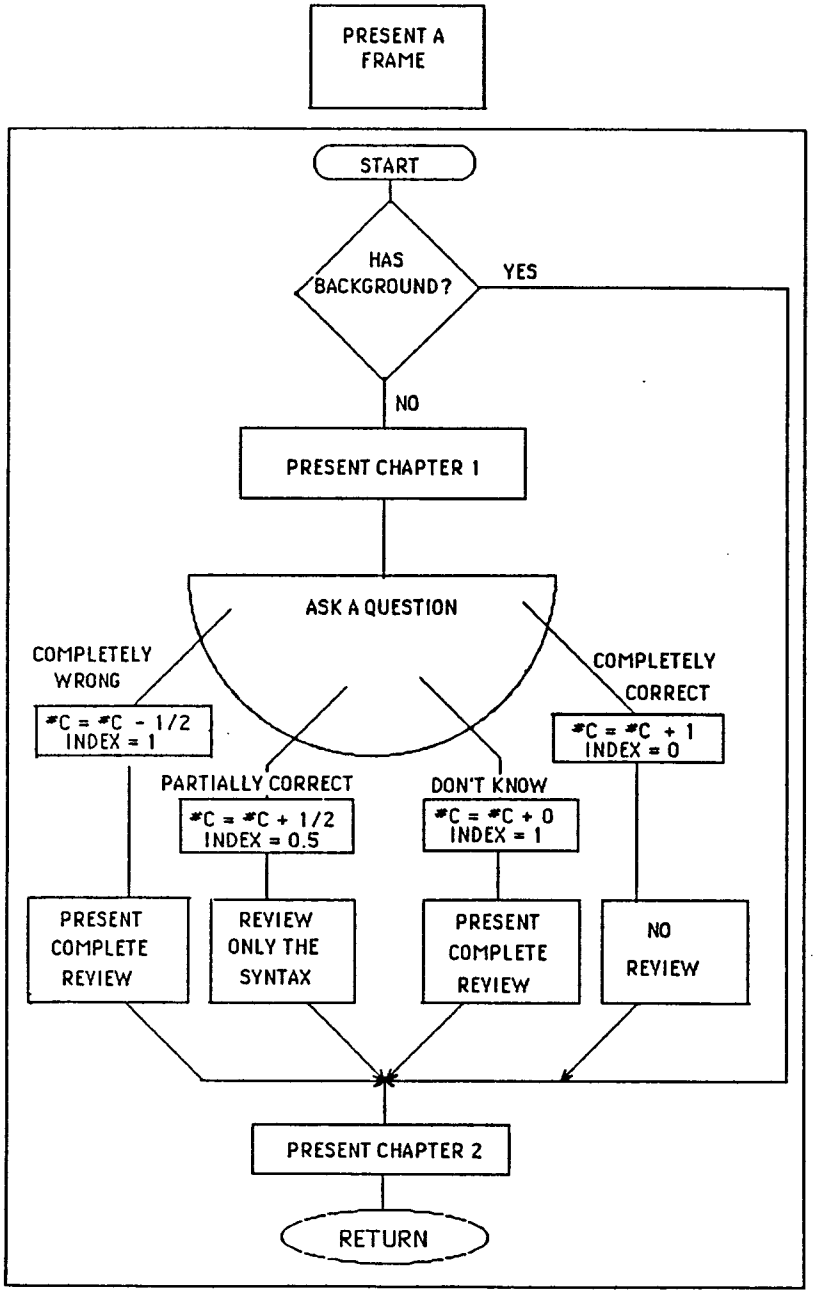


FIG 6 - FLOW CHART FOR COMPARING INDEX TERMS AND PRESENTING REVIEW

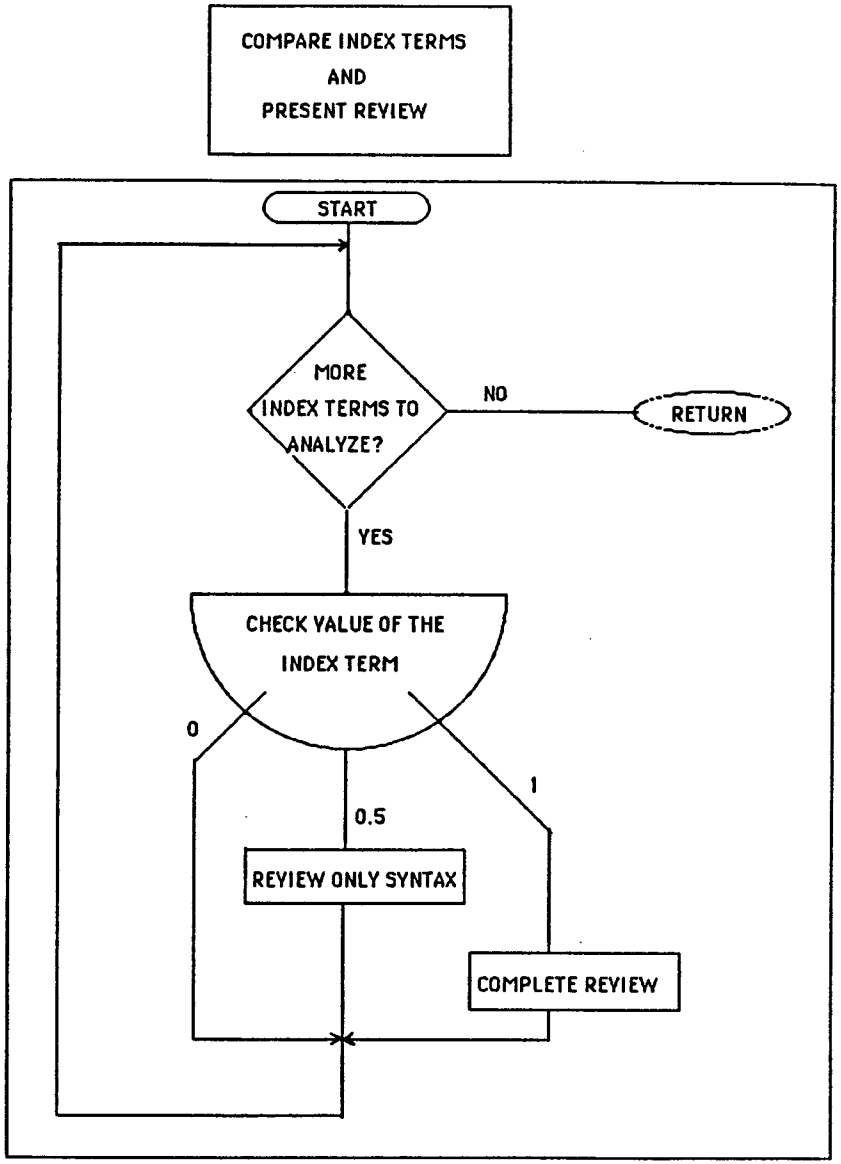


FIG 3 - STRUCTURE OF THE SCREEN

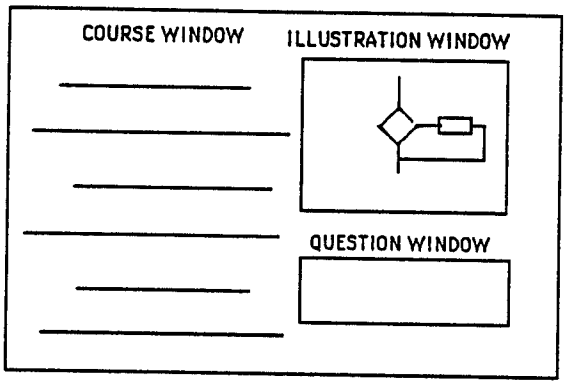


FIG 4 - MAIN FLOW CHART

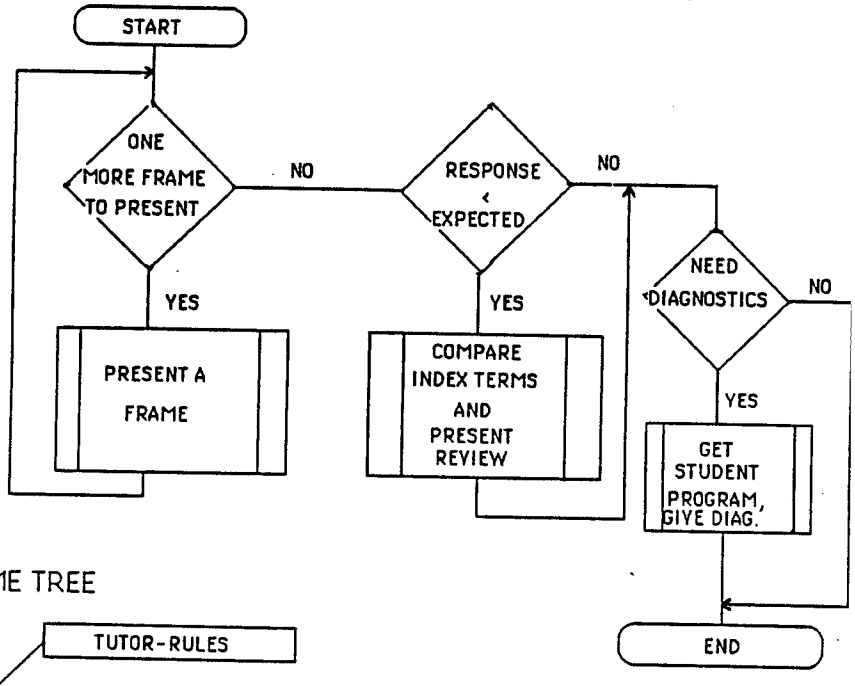
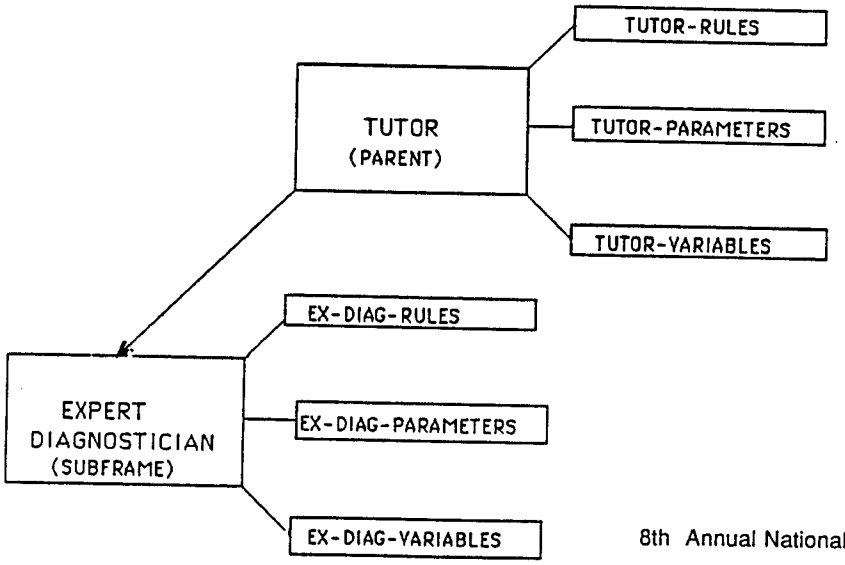


FIG 7 - STRUCTURE OF THE FRAME TREE



REFERENCES

1. Alty, J. L., Coombs, M. J. *Expert Systems, Concepts and Examples*, NCC Publications, Manchester, England (1984).

2. Ann Hill Duin. Computer-aided Instructional Displays : Effects on Students' Computing Behaviors, Prewriting, and Attitudes. *Journal of Computer-based Instruction*, Spring 1988, 15, No : 2, 48-6.

3. Beverly Woolf, Patricia A. Cunningham. Multiple Knowledge Sources in Intelligent Teaching Systems. *IEEE EXPERT*, Summer 1986, 41-54.

4. Beverly Woolf, David A. McDonald. Context-dependent Transitions in Tutoring Discourse. *International Journal on Man-machine Studies (1986)11*, 355-361.

5. Brian L. Dear. AI and the Authoring Process. *IEEE EXPERT*, Summer 1986, 17-24.

6. W. J. Clancy. *Knowledge Based Tutoring : The Guidon Program*, M. I. T. Press, Cambridge, Massachusetts, England (1987).

7. M. David Merrill. An Expert System for Instructional Design. *IEEE EXPERT*, Summer 1986, 25-37.

8. David Warman, Kenneth L. Modesitt. Learning in an Introductory Expert System Course. *IEEE EXPERT*, Fall 1989, 45-49.

9. P. Ercoli, R. Lewis. *Artificial Intelligence Tools in Education*, Elsevier Science Publishers B. V., The Netherlands (1988).

10. Gosta Grahne. Adaptive Features of CAL System Based on Information Retrieval. *Computers & Education*, 6, 1982, 99-104.

11. James Martin, Steven Oxman. *Building Expert Systems - A Tutorial*, Prentice Hall, Englewood Cliffs, New Jersey (1988).

12. Jan Skansholm. *Ada From The Beginning*. International Computer Science Series (1988).

13. John Self. *Artificial Intelligence and Human Learning*, Chapman And Hall, New York (1988).

14. A. Jones , F. D. Tuggle. Inducing Explanations for Errors in Computer-assisted Instruction. *International Journal on Man-machine Studies (1979)11*, 355-361.

15. Joseph Psotka, L. Dan Massey, Sharon A. Mutter. *Intelligent Tutoring Systems - Lessons Learned*, Lawrence Erlbaum Associates Publishers, Hillsdale, New Jersey (1988).

16. W. Lewis Johnson, Elliot Soloway. Intention-based Diagnosis of Programming Errors. *Proceedings, AAAI*, Austin, Texas, 1984, August.

17. Mark L. Miller. A structured Planning and Debugging Environment for Elementary Programming. *International Journal on Man-machine Studies (1978)11*, 79-95.

18. Martha C. Polson, J. Jeffrey Richardson. *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Associates Publishers, Hillsdale, New Jersey (1988).

19. Masanobu Watanabe, Toru Yamanouchi, Masahiko Iwamoto, Yuriko Ushoda. CL : A Flexible and Efficient Tool for Constructing Knowledge-based Expert Systems. *IEEE EXPERT*, Fall 1989, 41-50.

20. Merrill P. F.,Salisbury, D. Instructional Design Flaws in Computer-assisted Instruction. *Journal of Computer-based Instruction*, 15, Spring 1988, 76-84.

21. *Personal Consultant Plus Reference Guide*. Texas Instruments Incorporated, Austin, Texas.

22. *PC Scheme - A Simple, Modern LISP*. Texas Instruments Incorporated, Austin, Texas.

23. Putnam P. Texel. *Introductory ADA - Packages for programming*. Wadsworth Publishing Company, 1986.

24. Richard Sinatra. Holistic Applications in Computer-based Reading and Language Arts Programs. *Computers in Schools*, 4, Spring 1987.

25. Robert G. Farrell, John R. Anderson, Brian J. Reiser. An Interactive Computer-based Tutor for LISP. *Advanced Computer Tutoring Projects*, Department of Psychology, Carnegie Melon University, Pittsburgh, 1983.

26. Robert D. Tennyson, Dean L. Christensen. The Minnesota Adaptive Instructional System - An Intelligent CBI System. *Journal of Computer-based Instruction*, 11, Winter 1984, No : 1,2-13.

27. Roy S. Freedman, Jeffrey P. Rosenking. Designing Computer-based Training Systems : OBIE-1:KNOBE. *IEEE EXPERT*, Summer 1986, 31-38.

28. D. Sleeman, J. S. Brown. *Intelligent Tutoring Systems*, Academic Press, New York (1983).

29. Susane M. Humphrey. A knowledge-based Expert System for Computer-assisted Indexing. *IEEE EXPERT*, Fall 1989, 25-38.



BIOGRAPHICAL SKETCH

S. Palanisamy is currently a Graduate student in the Computer Science Department, Oklahoma State University, Stillwater. He is expecting his Master's Degree in May 1990. He holds a Bachelor's Degree in Electronics and Communication Engineering from P. S. G. College Of Technology, India. He obtained his Bachelors Degree in December 1985. He is presently working as a Graduate Teaching Assistant with the Computer Science Department, Oklahoma State University.

Questions or comments about this article may be addressed to
S. Palanisamy
Computer Science Department
219 Math Sciences Building
Oklahoma State University
Stillwater, Ok 74078

A SIMPLE IMPLEMENTATION OF A PETRI NET FOR AN AUTOMATED STATIC ANALYSIS OF ADA TASK INTERCOMMUNICATION

David Divine and Michael Fowles

National University, San Diego, California

This paper describes the research associated with, and the development of, a software system that uses an Adjacency Matrix implementation of a Petri Net graph to perform a static analysis of Ada source code involving "tasking". In addition to providing several useful task intercommunication metrics, the system is capable of identifying a potential "deadlock" condition. The system was developed as a graduate project in the Master of Science in Software Engineering program at National University, San Diego, California.

Introduction

This manuscript documents how we selected a topic to explore and how the scope of the project was defined. Understanding the project requires some knowledge of Petri Nets. Since many people are not familiar with Petri Nets, a brief discussion of their structure and uses is provided. We then describe how Petri Nets were applied to the project and what conclusions we drew from our work.

Petri Nets

The theory and application of Petri Nets have their grounding in "Kommunikation mit Automaten", the 1962 doctoral dissertation of Carl Adam Petri [3]. In the United States, much of the early work with Petri Nets was carried out by the Information System Theory Project of Applied Data Research and by Project MAC at the Massachusetts Institute of Technology [3]. Petri Nets are particularly well suited for modeling systems where many things or events can happen at once but which require events to occasionally get together and interact [1,2].

(note: In this informal description, what we call a Petri Net is really a form of Petri Net graph [3]).

A Petri Net is a directed graph with two types of nodes. One type is called a "place", the other a "transition". Places represent conditions, while transitions represent movement from one state to another. A place can be an input place to one or more transitions or the output place of one or more transitions, or both. Graphically, the places are represented as circles and the transitions as bars (Ref. figure 1). The places and transitions are connected by arcs (arrows). The arcs point from an input place to its transition(s) and from a transition to its output place(s). Places are never directly linked place to place and transitions are never directly linked transition to transition. An input place is related to its transition by an input function, a transition to its output place by an output function. If a place contains a "token", that place is said to be "marked". Pictorially a token is represented as a dot inside of a place. A place may be marked with more than one token. A place need not be marked. If all of the input places to a particular transition are marked then that transition is said to be "enabled". An enabled transition "fires". When it fires it removes one token from each of its input places and puts one token into each of its output places. This, of course, may cause that transition to become "disabled" (unable to fire) and may, in turn, enable other transitions. Any given distribution of tokens represents a state of the system. A Petri Net is "executed" by firing all of its enabled transitions until no transition remains enabled [1].

Petri Nets can be used to model a wide variety of situations and systems, particularly systems involving simultaneous events (e.g. concurrent processing). An example of the many different types of systems which can be modeled using Petri Nets is given by G. Hura: [2]

1. Queuing networks -- the number of requests are denoted by the number of tokens in the queues where queues are denoted by places and arcs connecting queues by respective transitions.

2. Weather systems -- the type of weather can be represented by places and the change from one weather front to another by tokens, where tokens represent the condition for the next weather transition. Other constraints can also be included in the model.

3. Education system -- For the transfer of knowledge from a Professor to students through teaching. Furthermore, knowledge can be transferred by the students in various directions.

4. Communication protocols can be modeled and analyzed by Petri nets.

The preceding is an incomplete description of Petri Nets, but we hope that it will provide the reader who is unfamiliar with Petri Nets enough background and terminology to understand how we applied and implemented this modeling tool in our project.

Defining the Problem

We began this project knowing only that we wanted to do "something" involving Ada metrics and Ada tasking. Early research on these topics led us to the article "Towards Complexity Metrics for Ada Tasking" by Sol M. Shatz. In this article Shatz proposes that the total complexity of a distributed program is the sum of the complexities of the

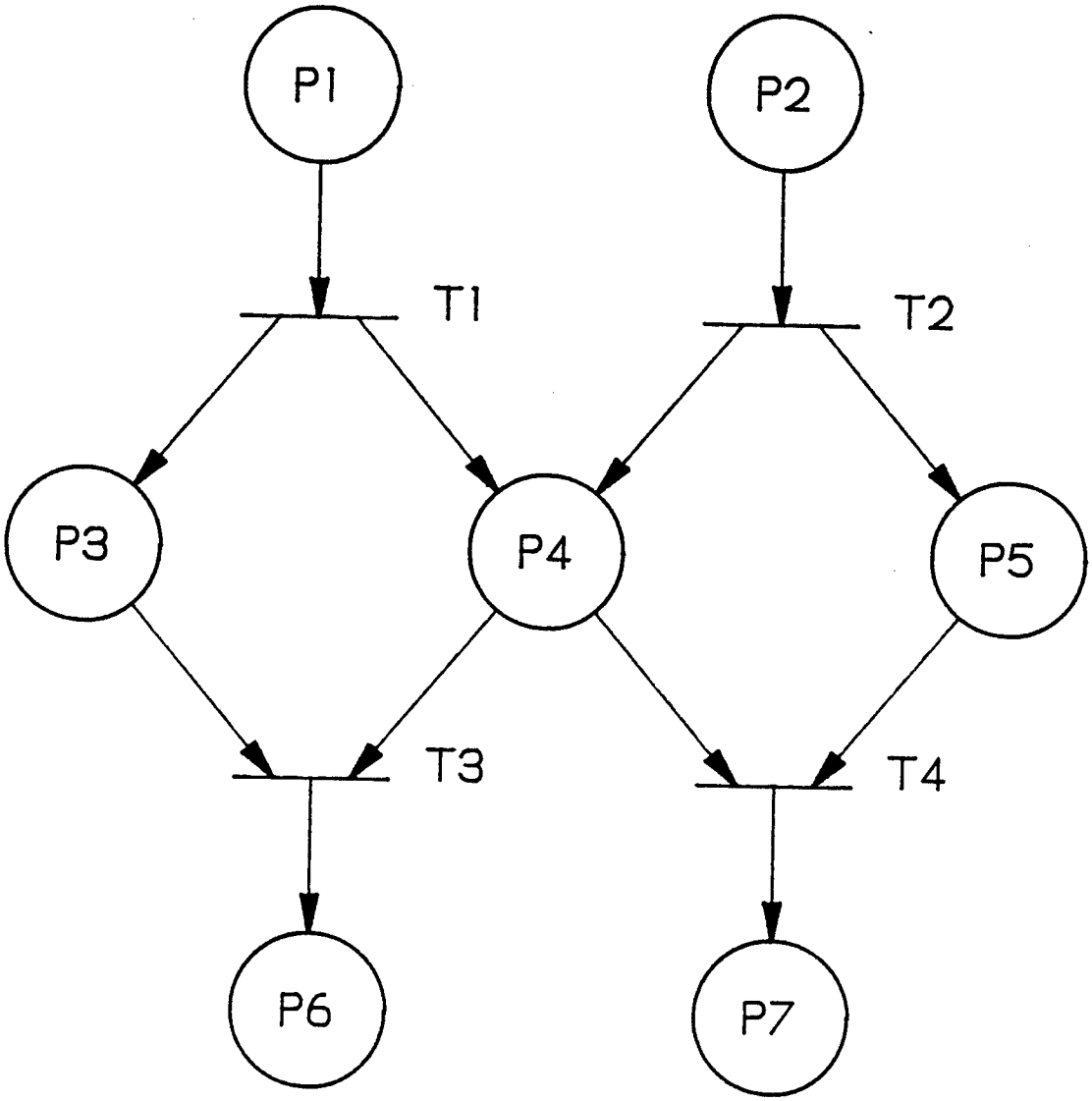


Figure 1: A generic Petri Net graph.

individual tasks (local complexity) added to the complexity of the intercommunication among the tasks (communication complexity). Our main interest in this article was that it introduced us to the idea of Petri Nets and their usefulness in modeling (among other things) Ada task communication. With our interest piqued we decided to implement a Petri Net based "metrification" of Ada source code using techniques developed from the aforementioned article and from another article: "A Petri Net Framework for Automated Static Analysis of Ada Tasking Behavior" by Shatz and W. K. Cheng which became the foundation for our project.

In this article Shatz and Cheng describe an abstract grammar which is equivalent to a Petri Net model of Ada source statements. They provide a table of templates of the form:

(input place names) --> (transition name) (output place names)

Each type of source statement (entry call, accept statement, etc.) has a template associated with it. In translation, an Ada source statement type would be applied to its appropriate template which would be customized by uniquely identifying it with the proper input place name(s), transition name, and output place name(s) for that particular source code statement. Additionally, if more statements must be read in order to complete the template then a marker ("S") is used in place of the output place names to indicate that those actual places are yet to be determined. Unique identification is accomplished by combining the place or transition type identifier with a task name and/or source code line number(s).

The following describes the Shatz and Cheng grammar associated with an "entry" call [5]:

p(entry) --> t(entry) p(wait_ack) p(ack_entry)

p(ack_accept) p(wait_ack) --> t(end_entry) S

Since an "entry" call involves communication between two tasks, there are two associated productions (templates): the first applies primarily to the "calling" task and the second primarily to the "called" task. In the Shatz and Cheng grammar, a production contains, at most, one transition (t) and one or more input and/or output places (p). The grammar identifies the place and transition types associated with a particular production. The first production pertaining to an entry call identifies a place of type "entry" that indicates the condition (if marked) that an entry call can be made. That place is input to a transition of type "entry" that represents the action of making the call. The output of that transition is a place of type "wait_ack" (waiting for acknowledgment of the call by the called task) and a place of type "ack_entry" (acknowledgment of the entry call by the called task). The second template identifies a place of type "ack_accept" (acknowledgment of the accept execution by the called task to the calling task) and a place of type "wait_ack" which is the same place as one of the output places of the first production. These two places are input to a transition of type "end_entry" that represents the action of the calling task completing the entry call. The identifier "S" indicates an output place that is yet unknown. That place indicator will be replaced with the first input place of the next production. A graphic representation of an entry call is provided through a sample program described later.

These articles, along with some other sources on Petri Nets, convinced us that we could do a meaningful project involving a static, Petri Net based analysis of task related Ada source code. In order to have a successful project we had to scope it properly. We decided, based on our curriculum-imposed time and manpower constraints, to develop a system which represents the Petri Net Analysis Subsystem and the Back-End Information Display Subsystem of Shatz and Cheng's Analytic Framework. We wanted to write this system in a manner which would allow it to be easily expanded or incorporated into other systems. What we really invisioned was implementing the subsystems described above but doing it in such a way that it could stand as the communication complexity portion of the total complexity metric proposed by Shatz [4,5].

In order to implement the Analysis Subsystem without a Front-End Translator Subsystem we decided to proceed under the assumption that we would have available to us a file of source tokens in the format required by Shatz and Cheng's translation template table. We realized that for testing purposes we would have to develop a facility for building such a file "by hand" in the absence of a real Front-End Translator. The following is a sample program and the resulting token file that would serve as input to our Analysis Subsystem. (For verification purposes, we used the same sample programs as Shatz and Cheng [5]).

Sample Program:

```
task body TASK1 is          -- line 1
begin
   TASK2.E;
end TASK1;

task body TASK2 is          -- line 5
begin
   accept E do
   ....
   end E;
end TASK2;
```

```
task body TASK3 is          -- line 10
begin
   TASK2.E;
end TASK3;
```

The Resulting Token File Is:

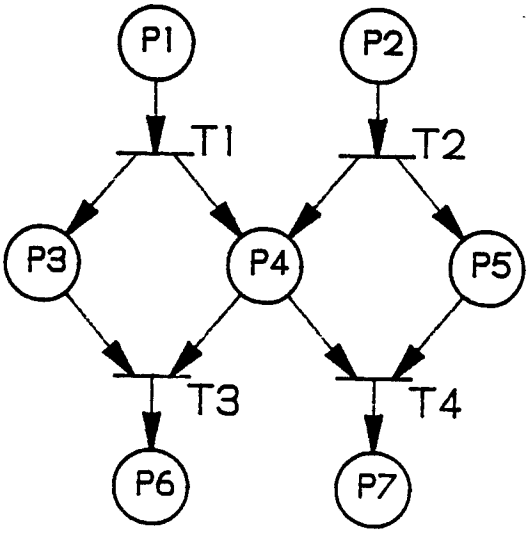
```
TASK-BODY TASK1          -- line 1
BEGIN
ENTRY TASK2 E
END
TASK-BODY TASK2          -- line 5
BEGIN
ACCEPT-DO E
END
END
TASK-BODY TASK3          -- line 10
BEGIN
ENTRY TASK2 E
END
```

Application of Petri Nets to the Project

Having decided that we were going to use a Petri Net to model Ada task intercommunication, we next had to decide how to implement the net in Ada code. After considering several options we decided to implement the net as a Petri Net graph using an adjacency matrix. The advantage of this implementation is that it is fairly straightforward and reduces both design and coding complexity significantly.

The adjacency matrix itself was implemented as a two dimensional array with one dimension representing places and the other dimension representing transitions. Each "cell" in the array (i.e. a place/transition intersection) is a record composed of two Boolean fields. One field indicates an input place, the other an output place. This record is used as follows: for each place/transition combination in the array, if the output field is set to TRUE then that place is an output place of the corresponding transition; if the input field is TRUE then that place is an input place to the corresponding transition (note: it is possible for a place to be both input to and output of the same transition). Using an array in this manner provided a simple means for representing places and transitions and the relationship (input and/or output) between them.

The next concern was to keep track of the net's initial and subsequent markings. We did this by using a one dimensional array dimensioned the same as the place dimension in the adjacency matrix array. Each element in this "token count" array is a non-negative numeric used to hold the token count of the place with the corresponding index in the adjacency matrix array. For example: If the third element in the token count array contains the numeric value 2, that means that the place represented by the third position in the place dimension of the adjacency matrix array has a token count of 2. If an element in the token count array has the value 0 then the corresponding place is unmarked. This technique provided us with a simple way to track the net's markings. Figure 2 shows a Petri Net graph and corresponding Adjacency Matrix with initial markings.



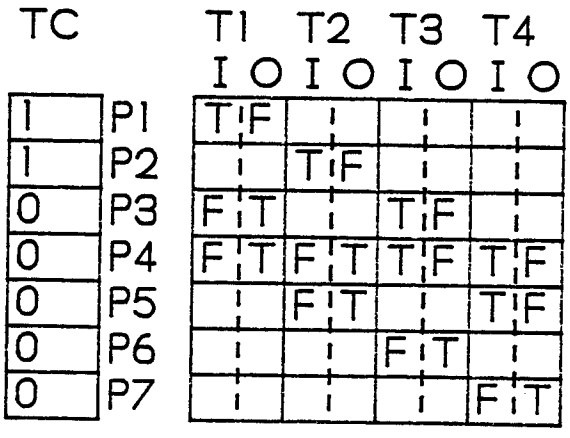


Figure 2: A Petri Net graph and corresponding Adjacency Matrix.

In order to relate place names and transition names to the indices of the adjacency matrix array we used two lists, one for place names and one for transition names. Each element in either list is a record with two fields: one to store a place name or transition name (as the case may be), the other to store the index number that tells which row or column (either transition or place dimension) in the adjacency matrix array is to be associated with that place or transition name. These lists are implemented as binary search trees.

As previously mentioned, Shatz and Cheng use an abstract grammar in order to translate Ada source statements into a logical Petri Net model of the task intercommunication. They provide a translation table giving a translation template for each Ada language construct that could be contained within a task body [5]. We implemented these productions as separate Ada procedures, one for each template. These templates provide the capability of identifying, by name, which place and transition types must be created and how they should be linked in order to model the Ada concurrency. The place and transition names are made unique by combining the type identifier with the task name and/or source code line number. All of our procedures work together to build a usable Petri Net based on the rules established by the abstract grammar. Place and transition locations in the adjacency matrix array are found by searching, by name, either the place list or the transition list (as the case may be). If the name searched for is found then the index into the appropriate dimension of the adjacency matrix array for that item is returned. If the item is not found then the item's name is entered into the list along with the next available index in its dimension. This index is then returned. Once the locations of the places and transition which a procedure needs have been determined, the procedure can then access the adjacency matrix array for each place/transition intersection it is using and set the place's Boolean fields. If the production being processed is one which cannot be completed until more statements are translated a flag indicating this condition is set and the transition name associated with the uncompleted production is saved. When the next production is being processed that flag would cause the first input place to the current production to be used as the output place to the saved transition in order to complete the previous production before the current production was actually processed. Stacks are used to save incomplete conditions (such as statements contained within compound statements) until the information required to complete them is acquired.

The initial marking of the net is accomplished by setting to 1 those elements in the token count array which correspond to places which are initial task references (eg. the "begin" statement of a task body). All other elements in the token count array are initialized to zero.

When every statement within each task body derived from the original Ada source code has been processed, the adjacency matrix array and its associated token count list represent an initially marked Petri Net built according to Shatz and Cheng's abstract grammar.

After the Petri Net is built it must be analyzed. This is done by traversing the adjacency matrix in the following manner. Each transition is checked to see if it is enabled. This is easily accomplished by visiting each place/transition node associated with that transition. If a place is an input place to that transition then its matching location in the token count list is checked, if its counter is greater than zero the place is known to be marked. If all the input places to a transition are marked then the transition is enabled. If a transition is enabled then it is fired. Firing is done by revisiting each place for that transition. If it is an input place then its counter (in the token count array) is decremented. If it is an output place then its token counter is incremented. Decrementing the counters of input places and incrementing the counters of output places corresponds to removing a token from each of a transition's input places and adding a token to each of its output places. This entire analysis process is repeated for each transition in the matrix until no transition remains enabled, at which point the Petri Net has been executed.

As this traversal is progressing various data (such as maximum number of concurrently enabled transitions, total token counts, etc.) can be accumulated for use in producing complexity (and other) metrics. The possible metrics are many.

The final part of analyzing the net is determining and locating a deadlock condition. A deadlock can be found by checking the completely executed matrix to see whether there are any places which are input places to some transition(s) and are still marked. This condition implies a deadlock because there is something waiting to happen (the marked place) that is input to a transition which will never fire because its other input places are not marked. Since this analysis is done only after it has been determined that there are no more enabled transitions, we know that this final state of the net will never change to allow the enabling of that transition. The name of any transition having marked input place(s) at this point can be reported to indicate where the deadlock occurred.

Once analysis is complete, the results are reported to the user by displaying them on the screen and writing them to a file for later reference. The following is the output of the analysis of the source code of the sample program.

PRODUCTIONS:

trans: TRANS_BEGIN- 2
input: BEGIN- 2

trans: TRANS_BEGIN- 2
output: ENTRY-TASK2- 3

trans: TRANS_ENTRY- 3
input: ENTRY-TASK2- 3
output: WAIT_ACK-TASK2- 3
output: ACK_ENTRY-TASK2- 3

trans: TRANS_END_ENTRY- 3
input: ACK_ACCEPT-TASK2- 3
input: WAIT_ACK-TASK2- 3

trans: TRANS_END_ENTRY- 3
output: END-TASK1

trans: TRANS_BEGIN- 6
input: BEGIN- 6

trans: TRANS_BEGIN- 6
output: ACCEPT- 7

trans: TRANS_ACCEPT- 7- 12
input: ACCEPT- 7
input: ACK_ENTRY-TASK2- 12
output: ENTRY_EX- 7- 12

trans: TRANS_ACCEPT- 7- 3
input: ACCEPT- 7
input: ACK_ENTRY-TASK2- 3
output: ENTRY_EX- 7- 3

trans: TRANS_ACCEPT- 7- 3
output: END_ACCEPT- 8

trans: TRANS_ACCEPT- 7- 12
output: END_ACCEPT- 8

trans: TRANS_END_ACCEPT- 7- 3
input: END_ACCEPT- 8
input: ENTRY_EX- 7- 3
output: ACK_ACCEPT-TASK2- 3

trans: TRANS_END_ACCEPT- 7- 12
input: END_ACCEPT- 8
input: ENTRY_EX- 7- 12
output: ACK_ACCEPT-TASK2- 12

trans: TRANS_END_ACCEPT- 7- 12
output: END-TASK2

trans: TRANS_END_ACCEPT- 7- 3
output: END-TASK2

trans: TRANS_BEGIN- 11
input: BEGIN- 11

trans: TRANS_BEGIN- 11
output: ENTRY-TASK2- 12

trans: TRANS_ENTRY- 12
input: ENTRY-TASK2- 12
output: WAIT_ACK-TASK2- 12
output: ACK_ENTRY-TASK2- 12

trans: TRANS_END_ENTRY- 12
input: ACK_ACCEPT-TASK2- 12
input: WAIT_ACK-TASK2- 12

trans: TRANS_END_ENTRY- 12
output: END-TASK3

A DEADLOCK HAS OCCURRED:

place: ACK_ENTRY-TASK2- 12
trans: TRANS_ACCEPT- 7- 12

place: WAIT_ACK-TASK2- 12
trans: TRANS_END_ENTRY- 12

A DEADLOCK CONDITION WAS DETECTED

TOTAL NUMBER OF PLACES: 18
TOTAL NUMBER OF TRANSITIONS: 11
TOTAL NUMBER OF STATES: N/A
MAX. NUMBER OF TRANSITIONS ENABLED: 3
TOTAL NUMBER OF TERMINAL PLACES: N/A

The output contains the customized productions with the uniquely identified transitions and places as well as the results of the deadlock check and other complexity measures. As indicated, the sample program represents a deadlock condition and the input places and transitions associated with the deadlock are identified. If a deadlock condition is detected, the "total number of states" and "number of terminal places" are misleading, as the net has not been completely traversed. Figure 3 depicts the Petri Net graph of the sample program with the customized places and transitions identified.

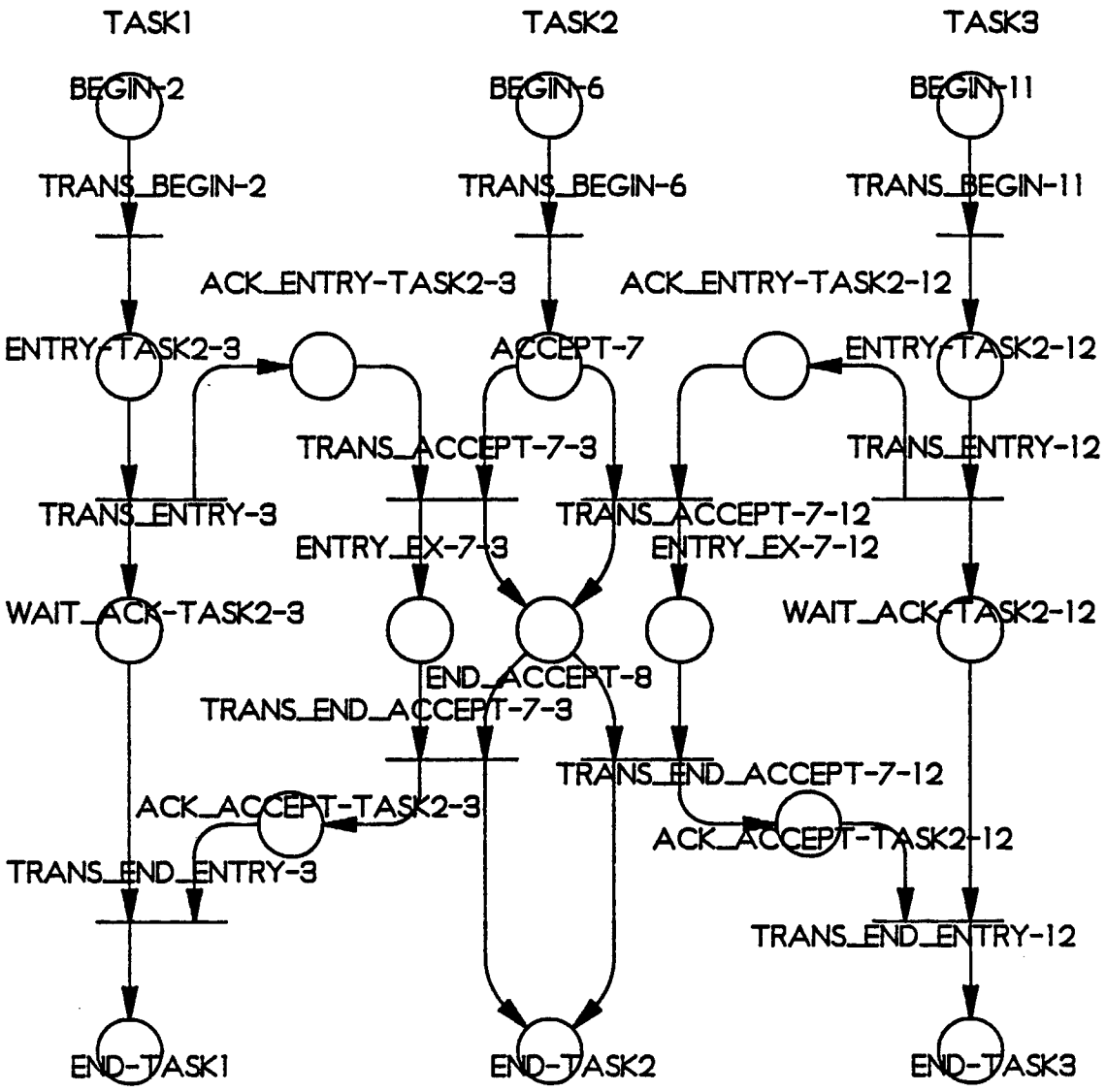


Figure 3: Petri Net graph of the sample program with customized places and transitions.

Conclusions

We found that there was quite a bit of information on Petri Nets and how to apply them to modeling program code (including Ada). The biggest drawback we found was in the area of actual implementation of Petri Nets on computers, especially straightforward implementations for everyday computer users as well as software developers. Of course we realize that Petri Nets are not something needed or even wanted by the everyday user and that this has an impact on their availability. But it's kind of like the chicken and the egg -- if more people want and use Petri Nets then more simple implementations will be forthcoming, but having more simple models already available will enable and encourage more people to make use of these powerful and versatile modeling tools.

Shatz and Cheng said that one of the main reasons for choosing their abstract grammar representation was that it was close to the input requirements of some software tools they were using [5]. We didn't have any Petri Net tools available and ended up having to make our own. In fact, what our project really did was take Shatz and Cheng's Petri Net representation as an abstract grammar and remodel it as a Petri Net graph so that we could analyze it using programming techniques with which we were familiar. In so doing, we believe that we have developed a simple, understandable implementation of the Petri Net graphic

structure which could be generalized into a basic multipurpose Petri Net structure. Of course, the production rules and analysis procedures for any particular application would have to be customized to build and interpret a Petri Net which accurately modeled the system under consideration.

In our implementation we used static data structures (arrays) for the main structures representing the net and its markings. This was done for the sake of simplicity in building and maintaining the net and in analyzing it. We think that this simple approach enhances the "understandability" of how we relate our structures to the Petri Net itself. Other equivalent implementations are possible. But the overhead associated with each implementation should be carefully assessed. For example, a dynamically allocated adjacency matrix would require at least two pointer fields for each place/transition node as well as the two data fields. Assuming that each of the internal pointers in a node required four bytes and that the compiler only allocated the two bits actually required for the two Boolean data fields (that may be naive), that would mean a pointer overhead of 64 bits for every 2 bits of data! Of course, other memory management schemes are possible.

For the metrics produced by our net analysis we chose: maximum number of enabled transitions, total number of places, total number of transitions, and number of terminal places as well as the deadlock check. Our main objective with the "number of whatever" measures was to show that such measures could be derived from our Petri Net representation. Another reason for choosing these is that they were readily verifiable manually and so could be used to check the accuracy of our analyses. We make no claims one way or the other about the meaningfulness of those measures regarding the actual complexity or quality of an Ada program. We do think that our model does provide much data about tasking programs which could be used to produce meaningful metrics. However, more experimentation is needed in this area. The deadlock check is a different matter. We feel that this is a very useful and significant piece of information. Moreover, it is accomplished through a static rather than run-time analysis of the source code. This means that a deadlock condition could be detected, located, and (hopefully corrected before a program was ever run. Deadlock determination by itself is sufficient to make our project worthwhile.

References

[1] Gudareep Hura, *Petri Nets*, IEEE Potentials, Oct. 1987.

[2] Gudareep Hura, *Petri Net Applications*, IEEE Potentials, Oct. 1987.

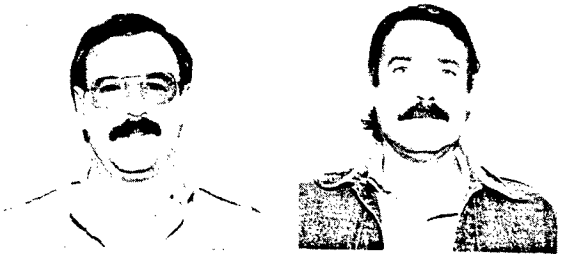
[3] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

[4] S.M. Shatz, *Towards Complexity Metrics for Ada Tasking*, IEEE Trans. Software Eng., Aug. 1988.

[5] S.M. Shatz and W.K. Cheng, *A Petri Net Framework for Automated Static Analysis of Ada Tasking Behavior*, J. Syst. Software, Dec. 1988.

About the Authors:

David Divine and Michael Fowles received their Master of Science in Software Engineering degrees from National University. Both are programmer/analysts in the Software Development Department, National University, 4141 Camino del Rio South, San Diego, Ca. 92108.



David Divine Michael Fowles

Problems in Maintaining and Porting Ada Software

Shenqian Shen

Department of System and Computer Science, School of Engineering
Howard University, Washington, DC 20059

1. INTRODUCTION

Software maintenance and porting play a very important role in the life cycle of a system. The procedure of maintenance and porting a system may contain almost all aspects of software development stages. There are many factors have to be considered when a system will be maintained and ported. These factors include programming environment, the size of the system, hardware environment for the system, complexity of the software, and so on. The requirement of maintaining and porting a system involve improvement of effectiveness of the system performance, correction of the errors in the system, extension of system processing ability, adding new functionality to the system, and so on. There are many approaches to deal with these problems. The paper will discuss the problems encountered and the solutions found during a period of maintaining and porting a software system called VOTER. The system was originally designed on an AT&T-PC6310 and was maintained and ported to an AT&T/3B2 and then was ported to an AT&T-PC6310 and also to a SUN 3/60. The final ports were to an Intel HYPERCUBE.

The paper has the following organization. The VOTER system is discussed in section 2. The performance of the original system is given in section 3. Sections 4 to 7 contain a discussion of the performance of maintenance operations and the porting of the VOTER system to AT&T3B/2, AT&T-PC6310, SUN 3/60 and Intel HYPERCUBE respectively; section 8 is the summary of the paper.

2. DESCRIPTION OF VOTER SYSTEM

The VOTER system is a multi-tasking software system that was written in the Ada programming language by a team of programmers.

The system is part of an experimental system for measuring software fault tolerance [1]. The idea used in design the system was to perform several versions of an algorithm with collecting and comparing the results produced by some operations in the algorithm at several predetermined breakpoints. The fault-tolerant aspect of the VOTER software is that a "correct" version of the state of the system is determined by means of a majority vote. The algorithms were implementations of an infinite precision desk calculator with input data in Reverse Polish notation format. In the VOTER system, every version of the desk calculator algorithm was performed as an independent task; each of the tasks has several pre-determined common points called breakpoints. At each of these breakpoints, the result of every push or pop operation on the top of stack in the version which is a task, was passed to another independent task called VOTER by means of the Ada language rendezvous mechanism; then the VOTER would compare these results to determine correctness by means of a majority vote at the breakpoint.

The problem that each of the versions attempted to solve was the creation of an infinite precision desk calculator program using the same algorithm. The original programming team then had to write a main program named VOTER to compute a majority vote at the breakpoints. The different versions of the algorithm were written by different programmers and the entire system was designed to run multiple tasks:

version 1 of algorithm -------- task1

version 2 of algorithm -------- task2

version 3 of algorithm -------- task3

.
.
.

version N of algorithm -------- taskN .

comparison of result at breakpoint)

A user can select the number of versions to run in the system from the range 1 to N. Initially, there were four version tasks and a VOTER task. Each of the versions was written by a different programmer.

3. Performance of Original System on an AT&T-PC6310

The original system was designed on an AT&T-PC6310 . An AT&T-PC6310 is similar to an IBM PC-AT; it uses an Intel 80286 CPU with 1 MB RAM and ran MERIDIAN Ada compiler v2.1. The original system had 5500 lines of Ada source code and consisted of 36 modules. The main task VOTER and each version of desk calculator task was tested and worked separately as a component of a multi-tasking system when running as a single task. However, when the entire system was combined and executed, there were some logical and tasking errors were created. The way these tasks communicate is that the main-task VOTER issues a number of entry calls to each desk calculator task. The input file name for each desk calculator task was passed from VOTER through a rendezvous between VOTER and a desk calculator task. Then every desk calculator task read data in reverse Polish notation from the file. After every operation of either push or pop was performed, the result of the operation was stored on the top of a stack and was sent to VOTER by means of a rendezvous between the task and VOTER.

4. Maintaining and porting on AT&T/3B2

Maintenance of the original system was performed on an AT&T-3B2/500 with 4MB RAM and AT&T Ada compiler (preliminary version based on Verdix Ada). System maintenance is a complex project. One have to understand the functions and structure of the system and component of unit of the system. The strategy of maintenance of the system was a bottom-up approach to compose the system into the separate tasks and to check each task as a single program. After each task was working correctly, the task modules were linked together. The maintenance procedure can be divided into three parts: source code reading; module testing and modification; and designing of new modules. The first part of maintenance was to follow the ideas of system designer, understand the system. This procedure involves review the system specification, reading the source code of the system, talking to the member of programming team of original system. Because the system did not use any compiler dependent function or statement and the system was fairly small, we considered the compiler on AT&T/3B2 and on AT&T/PC6310 to be similar.

The second part of the work was to test and fix the errors in the system. The way to test the system is to examine each module from bottom to up at the system; the errors were fixed at lower level modules first. In this way, the work of the maintenance was organized in a hierarchical manner. There were quite a large number of global variables in the system modules; those variables can decrease the maintainability of the system and increase the time of maintaining the system.

The third part of work is to design new modules to put in the system. Because some modules in the system did not have a proper algorithm or have a serious logical errors, so it is necessary to construct some new modules to make the system work correctly. The problems in the system are as follows:

improper interface between modules that were developed by different programmers, and

tasking errors in the system.

The first problem was that a parameter in one module which was passed to other modules was outside the scope of the parameter defined in the other module. The problem is showed in following example.

```
procedure A( x : in string(1 .. 10);

procedure B( .... ) is

y : string(1 .. 12)
  ...

 BEGIN

    ...

    procedure A( y );

    ....

 END procedure B;
```

The second problem was more complex. In each version, the selective wait statements were executed sequentially. That is, the problem was of tasking errors in task synchronization. Whether the rendezvous of tasks was successful depended on the time of execution of sequential statements. Task communication did not work and the relationship between the tasks was complex.

```
task body VOTER is
 ...

 begin
  ...

  entry_call_A( ... );

  ...

  loop

  ...

  entry_call_B( ... );

  ...


 end loop;

...

end VOTER ;
```

```
task body DESK_CALCULATOR is

begin

...

loop1:
 loop
 ...
 select
    accept entry_call_A( .. ) do
    ...
    end entry_call_A;
 or
    terminate;
 end select;
 ...
 loop2:
  loop
 ...
select
 accept entry_call_B( ... ) do
       ...
     end entry_call_B;
 or
 terminate;
end select;
     ...

    end loop2;
    ...

 end loop1;

end DESK_CALCULATOR;
```

The way we handled the problem was to simplify the relationship between the tasks by creating a new task named driver-task which provided an interface between the desk calculator tasks and the main task. The driver task collects the data from each desk calculator task and sends an entire data set to VOTER by means of a rendezvous between driver-task and VOTER.

The method of communication between the tasks is that each desk calculator task makes an entry call to driver task to pass a complete set of votes to driver-task when an operation of either push or pop is finished. After a complete set of votes is collected, the driver-task accepts an entry call from VOTER to pass the complete set of votes to VOTER. At this time VOTER computes the majority vote. After the newly designed system worked correctly, two new versions of the desk calculator algorithm were added to the system. These versions were created by combining portions of other working tasks. Thus there was a total of six versions or tasks that were desk calculators; in addition, one driver-task and one VOTER task were run concurrently. Thus there were total eight tasks in the system. The problem observed on the AT&T/3B2 is that when the whole system ran, if a task read data from an input file longer than certain a limit, some task would be terminated automatically, and an error of tasking deadlock was created. The point at which the program stopped depended on the length of the input file. Because of the termination of a calculator task during system execution, the driver-task was not finished and so the driver-task must wait the forever for the task completion. That is the reason that an error of tasking deadlock was created. There would be only one process to run the eight tasks of the VOTER system instead of eight processes in the UNIX operating system processing table. The implementation of the system behavior is compiler dependent. On the AT&T/3B2, the file size must be quite small; generally about 24 lines in each input file.

5. Porting to SUN3

Another experience in porting was obtained by moving the system to the SUN-3. There were no changes when the system was moving from AT&T/3B2 to SUN-3. When we ran the system on a SUN-3 system, with 8 MB RAM and Berkeley UNIX 4.3 operating system and the Verdix 5.5 compiler. The problem was that when we ran a big file (15000 lines), a storage error was created. If the system ran on the SUN-3 in single user mode, the problem dispeared. This means that the same system running in different environments may get different results.

6. Porting New System Back to AT&T-PC6310

In the AT&T-PC6310, when six tasks were running concurrently, some tasks would be terminated automatically before the task completed its task and without an error message resulting from the task. The termination point of the task in the program was not the end of program, The result of this problem was to make some other tasks wait forever; a tasking error was produced. However, if five calculator tasks ran, there were no problems.

7. Maintaining and Porting to Hypercube

The Intel Hypercube iPSC/2 is a parallel computer in which the individual processors do not share any common memory. Our configuration is an eight node machine. The Ada compiler for this computer allows multiple tasks to be run on the same processor and each processor can run multiple tasks. However, this compiler allows no task to execute on more than one processor. Different tasks running on the same processor communicate using the rendezvous mechanism; tasks running on different processors communicate by means of message passing. Further details will be reported in [2].

8. Conclusions

Maintaining and porting a software is a complex procedure. There are many principles and rules of software engineering one should follow. Maintaining and porting a system is not just fixing the some errors in the system; it is work. The work involves the designing of modules, reorganizing system structure, and system function enhancement. One suggestion to remember when you design or maintain or port a system is

"leave a clear clue to the person who will maintain your system"

There are many way to deal with the same problem . Some points one must consider are:

First, the system developers have to aware of the programming environment, which includes restrictions of the hardware and limits of the programming language's compiler.

Second, interfaces between modules must be clear before the modules are developed.

Third, any systems programmer should follow the same standards for the interfaces between the modules.

Finally, the system structure must be clear and the use of global variables used in system must be limited.

A system that is organized in a hierarchical form and designed using good software engineering principles will save a lot of the time and effort that might otherwise be spent on maintaining and porting it.

REFERENCES

1. Coleman, D. M., and Leach, R. J., "Performance Issues in C Language Fault-Tolerant Software." Comput. Lang., Vol. 14, No.1, pp.1-9.(1989).
2. Coleman, D. M., and Leach, R. J., "N-Version Programming Using the Ada Tasking Model", to appear.

Design of Integrity Kernels for Distributed Systems

S. Ramanna and J.F. Peters
Department of Computing & Information Sciences
Kansas State University, Manhattan, KS 66506

Abstract --This paper reports work on a language called Ada/ISL which is intended to be used for the specification of integrity kernels for distributed systems. Ada/ISL is based on a form of interval temporal logic and is an extension of the Ada package construct which provides a framework for a logic of knowledge and belief about data integrity. The extensions of the Ada package construct include (1) VDM style IN/OUT assertions about parameters in Kernel operations, (2) interval temporal assertions to express well-formed presumptions about data integrity and to express global and local package invariants and (3) abstract types (sets and sequences). Package specifications facilitate the design of trusted integrity kernels implementable in a distributed Ada environment. The Clark and Wilson integrity model designed to prevent fraudulent and erroneous data modification is subsumed. This paper also extends the notion of data integrity captured in an Integrity Characteristics Tuple (ICT).

Index Terms -- Ada, design, belief, distributed systems, knowledge, office document retrieval system, specification language, interval temporal logic.

1. INTRODUCTION

Integrity constraints are requirements formulated about data. These requirements are in the form of assertions which single out the constrained data items and state the properties which the data must possess. It is commonplace to view integrity constraints as not varying over time [11]. However, time and integrity are interwined. This suggests a need to introduce a new approach to the specification of integrity constraints which are time-varying to keep pace with the changing nature of the data values themselves. A variety of integrity models have been introduced to enforce integrity constraints [9, 10]. Recent work on temporal databases has shifted the emphasis to establishing a computational framework for organizing temporal facts [12]. This paper introduces an Integrity Specification Language (Ada/ISL) used in the design of an integrity kernel presented in the context of a distributed office model similar to MULTOS [5, 6, 7, 8]. This office application environment is built atop the CONDUCTOR model [25]. The CONDUCTOR model guarantees reliable group communication and atomic message delivery in a distributed environment. Ada/ISL has also been used to design an integrity kernel for a distributed database application in the context of SDD-1 [3, 4, 27] Ada/ISL utilizes interval temporal logic [1, 21, 30] to formulate integrity constraints. Ada/ISL also provides a formal means of expressing both knowledge based on facts and beliefs based on intuition about the integrity of data items. The Ada/ISL logic is a variation of the logic for reasoning about computer security introduced by Moser [23].

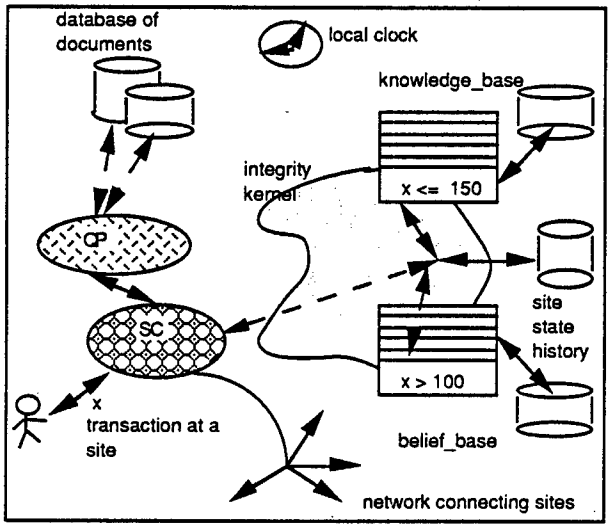
Integrity constraints are written as temporal assertions which are allowed to vary over time. The changing values of data are controlled by evaluating integrity constraints with respect to a knowledge base. A knowledge base is viewed as a set of first order sentences [28]. An Ada/ISL knowledge base consists of first order sentences about the world (the database being modelled). Ada/ISL also utilizes a belief base which contains first order sentences about the expected values of the data in the database. The Ada/ISL kernel monitors the "realism" of existing integrity constraints vis-a-vis our current knowledge about the world. A principal task of an Ada/ISL kernel is automating the maintenance of its integrity constraints (belief base) based on knowledge it obtains from kernel site state information. In this paper, a generalized integrity kernel using a variation of the Clark and Wilson integrity model is introduced [10]. In the Ada/ISL model, data in the database are partitioned into constrained and unconstrained data items. Associated with each constrained data item is an Integrity Characteristics Tuple (ICT). This tuple expresses characteristics of data integrity such as timeliness, quality, completeness, and confidence.

The architecture of a Distributed Office Site (DOS) is introduced in section 2. An overview of the underlying CONDUCTOR distributed system model supporting reliable interprocess communication between DOSs is given in section

3. The model for data integrity which is an extension to the Clark and Wilson integrity model, is presented in section 4. An ov erview of constrained and unconstrained document components, Mdoc integrity triples (MITs), and the notion of knowledge and belief in the context of ICTs, is given in this section. The formulation of presumptions with interval temporal logic is presented in section 5. The syntax of Ada/ISL is introduced in section 6 and Appendix A. A specification of a kernel is given in section 6.1 and Appendix B. A sample query session is elaborated in section 6.2.

2. FRAMEWORK FOR DISTRIBUTED OFFICE SITE

The Distributed Office Site (DOS) structure shown in Figure 1, is based on the client-server paradigm [31]. The Query Processor (QP) gives remote centralized storage of documents to work stations connected to a Server Controller (SC) via a communication network.



Legend: SC = server controller, QP = query processor

Fig. 1. Office site with integrity kernel

The SC parses client requests, performs authorization as well as version control on documents, and keeps clients informed about currently supported media. The SC also interfaces with its local integrity kernel, with other SCs via the network and with its local query processor. A query processor (QP) checks syntactic correctness of queries, performs decomposition, optimization and execution of queries. In addition, the QP manages processes that perform type-checking and storage-handling of documents. The functions of the SC and QP have been abstracted into single processes. In reality, these functions could be distributed over multiple processes.

The SC responds to a query by parsing the client request(s) and performing authorization checks. It then hands over control to the query processor to evaluate the query. After the QP has the optimal set of documents for processing a query, it makes a call to the SC for integrity checks. The SC responds to a QP call by interfacing with its integrity kernel. It is the responsibility of the integrity kernel to determine if document components (constrained vs. unconstrained) must be checked for integrity. If an integrity check on a document component fails, the query processing stops.

When an integrity kernel processes a client request, it utilizes information available to it from the current state of the site where the server is located. This state information (current timestamp, currency of the document components, and so on) available to a server, provides input to a set of evaluations of presumptions about the data items referenced in a client request. The server provides not only values of site parameters derived from the current state of a site at the time of a client request but also a history of past site states. It is the combination of the information from the current site state as well as the site state history which allows an integrity kernel to make informed judgements about the integrity of data items. All client requests need not be preprocessed by the integrity kernel, if a distinction is made between non-data-sensitive and data-sensitive client queries. This would ease the burden on the integrity kernel, making it less of a bottleneck.

The DOS architecture relies on the CONDUCTOR model to provide reliable interprocess communication in a distributed environment.

3. CONDUCTOR MODEL DISTRIBUTED ARCHITECTURE

The CONDUCTOR model formulates a hierarchy of administrator processes. Administrators hide worker processes as well as the number of instances of worker processes. Administrators can be document managers, time servers, and so on. Workers manage resources. This model introduces administrator groups to provide increased concurrency. The administrator groups are "workers" which are administered by a conductor process. The conductor process serves as a sequencer which provides partial ordering of all client messages. When clients require reliable communication of messages, the conductor and administrator

groups join forces to guarantee atomic message delivery. This accomplished with a form of atomic multicasting from the conductor to an administrator group which does synchronization. The architecture of the CONDUCTOR model is robust inasmuch as it is process failure tolerant. This is made possible by utilizing watchdog processes whose sole purpose is detection of process failures and failure recovery [24]. This simplifies the client view of the world and offers increased concurrency and reliable message interprocess communication.

The CONDUCTOR model provides reliable communication between processes by enforcing atomicity and ordering of message delivery [25]. An overview of this model is given in Figure 2:

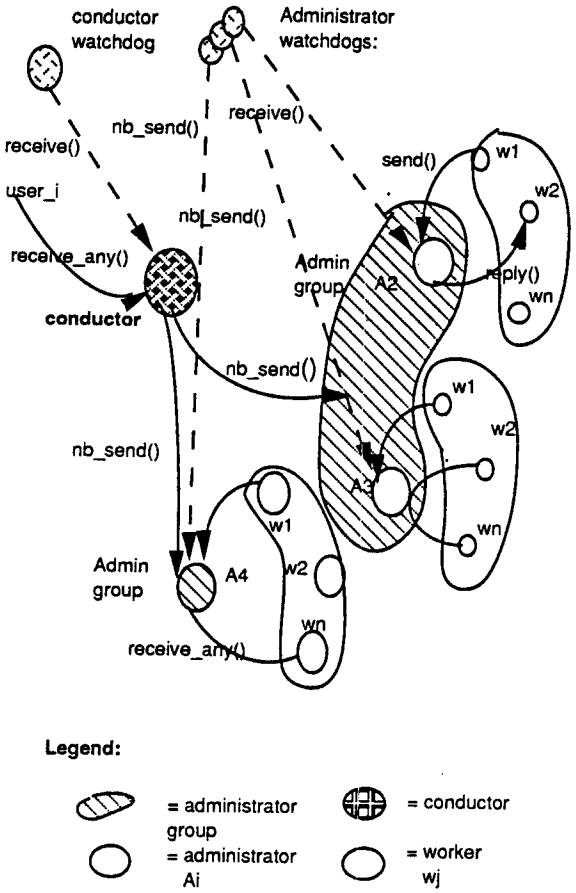


FIG. 2 CONDUCTOR model architecture

4. INTEGRITY SYSTEM TERMINOLOGY

In a DOS environment, the conceptual components of a document are partitioned into two disjoint sets similar to the constrained and unconstrained data items as in the Clark and Wilson (C & W) model [10]: unconstrained document component (UDC) and constrained document component (CDC). A CDC is a conceptual component of a multimedia document (Mdoc) which has constraints placed on it by the site administrator. An UDC is a conceptual component of a multimedia document with no associated integrity constraints. At any time a UDC can be converted to a CDC This modified C&W model relies on Integrity Verification Procedures (IVPs) and Query Procedures (QPs) to guarantee document component integrity. An IVP confirms that all CDCs for a system conform to the document integrity specifications at the time an IVP is executed. The purpose of a QP is to change the set of CDCs from one valid state to another valid state. A QP is defined with respect to a set of CDCs which the QP manipulates. This relationship between a QP and a corresponding set of CDCs is expressed in terms of an Mdoc Integrity Triple (MIT) similar to the C&W integrity triple. An MIT is given by (Userid, QP_i, $\{CDC_1,...,CDC_K\}$) which identifies a user who is authorized to use the ith query procedure QP_i on the set of Constrained Document Components $\{CDC_1,...,CDC_K\}$.

The integrity kernel introduced in this article formalizes the notion of an IVP in terms of knowledge and belief about constrained document components. Briefly each CDC includes a set of beliefs (BeliefSet) and a knowledge set (KnowledgeSet). A *belief* is a presumption about a document component and this presumption may be refuted by factual information (knowledge). All beliefs in the BeliefSet are presumed to be true *unless* they are refuted by presumptions in the KnowledgeSet. The KnowledgeSet for a CDC consists of presumptions based on factual information which constitute the knowledge base. A knowledge base may or may not have all the information about some relevant aspect of the world of interest [22]. The ith CDC in an MIT will have the following structure in the new model:

CDC_i = (document component, BeliefSet, KnowledgeSet, timestamp)

The evaluation of beliefs in the BeliefSet is performed in conjunction with corresponding presumptions in the KnowledgeSet. These sets consisting of assertions are written in interval temporal logic [1, 13, 30].

5. SPECIFICATIONS WITH INTERVAL TEMPORAL LOGIC

To evaluate integrity constraints over intervals of time, there is a need to introduce temporal operators with a time interval interpretation. This form of logic is called interval temporal logic [21, 30], which provides a concise means of expressing presumptions about knowledge and belief.

In temporal logic, the temporal interval specified extends over the sequence of states from the present state to the end of the computation. This sequence is what is known as a tail sequence. The temporal operators are interpreted over the entire tail sequence[26]. For this reason, temporal operators like **eventually** and **unless** cannot be used to specify eventually properties in bounded intervals. An assertion P (knowledge, belief) which evaluates to true in the interval from ith to the jth site state, is expressed by writing $K_{<i,j>}(P)$ [19, 20]. For example, in a distributed database for an airline reservation system, the following can be asserted:

$K_{<i,j>}$(B) = $K_{<2,10>}$('The number of 727 first class seats for airline X = 20')

This says that for the temporal interval beginning with the timestamp 2 and ending with the timestamp 10, it is presumed that 727s belonging airline X have 20 first class seats. The semantics of a modified unless operator (a variation of the operator given in [20]) is given below:

$K_{<i,j>}$ (Assert_B **unless** Assert_K) ==
(exists x: i <= x <= j: $K_{<X,X>}$ (Assert_K)
and all y: i <= y < x: $K_{<Y,X>}$ (Assert_B)
and $K_{<X,X>}$ (not Assert_B))

Presumptions in this logic can be verified by evaluating assertions written with the *unless* operator as $K_{<i,j>}$ (B *unless* K). This says that we believe B is true over the entire interval unless the presumption expressed by K refutes B over all or some portion of this interval. For example, in terms of the presumption about airline reservations, we can formulate the following check on this assumption:

<2,10> Belief B **unless** Knowledge K

where Q = '727 first class seats for airline drops below 20.' This says we continue to believe Belief B as long as assertion K is not true over the temporal interval defined by <2,10>.

For simplicity, assertions with the *unless* operator will be written without the K prefix. Temporal operators without an interval prefix are defined over an unbounded tail sequence. For the sake of clarity, an unbounded tail sequence is given by <i, inf> where i the current state and inf stands for "infinity." The following table introduces the semantics of the Ada/ISL temporal operators used in this paper:

Operator	Semantics
always	$K_{<i,j>}$ (always P) == (all x: i <= x <= j: $K_{<x,x>}$ (P))
eventually	$K_{<i,j>}$ (eventually P) == (exists x: i <= x <= j: $K_{<x,x>}$ (P))
until	$K_{<i,j>}$ (P until Q) == (exists x: i < x <= j: $K_{<x,x>}$ (Q) and (all y: i <=y < x: $K_{<y,y>}$ (P)))
seq	$K_{<i,j>}$ (seq (P)) == $K_{<i,j>}$ (eventually P) $K_{<i,j>}$ (seq (P1, ... , Pn)) == $K_{<i,j>}$ (P1 until seq (P2, ... , Pn))
unless	$K_{<i,j>}$ (Assert_B **unless** Assert_K) == (exists x: i <= x <= j: $K_{<X,X>}$ (Assert_K) and all y: i <= y < x: $K_{<Y,X>}$ (Assert_B) and $K_{<X,X>}$ (not Assert_B))

The seq operator introduced in this paper is a variation of the seq operator found in [17,18].

The temporal intervals in Ada/ISL assertions can be mapped to corresponding DOS timestamps. That is, the temporal interval <i, j> in an assertion

<i, j> (P) == <ts(i), ts(j)> (P)

where

ts(i) = DOS timestamp for ith state

ts(j) = DOS timestamp for jth state

Then

$K_{<i,j>}$ (P) == (all timestamps t:
ts(i) <= t <= ts(j): P(t) is true)

6. Ada/ISL: INTEGRITY SPECIFICATION LANGUAGE SYNTAX

The Integrity Specification Language (Ada/ISL) provides a formal means of specifying and characterizing the behavior of an integrity kernel. An essential element of an Ada/ISL specification is the package construct [2, 16]. This language provides syntax for well-formed formulas which express constraints on the required behavior of an integrity kernel. Predicates written in Ada/ISL will utilize an extension of modal logic found in Moser [23]. This modal logic subsumes the usual predicate logic. An integrity kernel specified in Ada/ISL has syntax indicated by the following

grammar fragment:

```
Ada/ISLspecification ::=      [generic
      ParameterPart {; ParameterPart}]
    package KernelName is
          Package_Declarative_Part
      end KernelName;
      Package_Declarative_Part
            ::=     {Basic_Declarative_Item}
                    {WFP}
                    {Specific_subprogram_specification}
                    {Definition}
                    {Package_Property}
      Specific_subprogram_specification
            ::= subprogram_specification
            [--IN: expr]
            [--OUT: expr]
            [Local_Property]
      Local_Property
            ::= LocalProperty WFP
      Package_Property
            ::= PackageProperty WFP
      WFP   ::= /* well-formed presumption */
```

The undefined nonterminals (Basic_Declarative_Item, for example) in this grammar have exactly the same syntax as that found in the Ada Language Reference Manual [2]. A more complete grammar for Ada/ISL is given in Appendix A. A variation of ISL called Ada/TL is given in [17]. A partial Ada/ISL kernel specification together with a some skeletal operations is presented in section 6.1. A sample query processing session is given in section 6.2.

6.1 SPECIFICATION OF A SIMPLIFIED INTEGRITY KERNEL

In this section a partial specification for a simplified integrity kernel written in the Ada/ISL language is given. A more complete specification of this kernel is presented in Apppendix B. This specification suggests the broad outline for the design of a multimedia document integrity kernel.

```
--Ada/ISL specification for a multimedia document server
-- integrity kernel

generic
  type      ComponentType is pending;
  type      LogicalClockType is pending;
  type      FirstOrderSentType is pending;

package Serverkernel is

  type MdocBaseType is seq of CDCrec;
   --defines entire document base

 --def   EvalUnless(Belief, Knowledge) ==
       if  Belief unless Knowledge then return true
       else return false endif;
  --evaluate "Belief unless Knowledge" in the current
  --state over the same interval.

 --def   AddBelief(i, b, characteristic) ==
      (seq (atomic_broadcast("check belief",
       SitesId, i, b, characteristic; ack : OUT Boolean),
        ( all ack: site'RANGE: ack = "belief is verified" and
         atomic_broadcast( "insert belief",
              SitesId, i, b, characteristic;  ack : OUT Boolean)
        or
        atomic_broadcast("cancel installation",
            SitesId, i, b, characteristic;  ack : OUT Boolean)
        )
       )
      )
     --initiates network-wide addition of valid beliefs

--def  CheckBelief(c, b, characteristic; OUT ack) ==
        (seq( FindComponent(c),
            (CancelInstallation()
             or
            (FindKnowledge(ord(c) = i, characteristic;
                          K : OUT ICTrec),
              VerifyBeliefwithKnowledgeSet(i, K, b),
              FindBeliefSet(ord(c) = i, b, characteristic;
                          B : OUT ICTrec),
              VerifyBeliefwithBeliefSet(i, b, B)
             )
            )
           )
        )
     --verifies belief b with respec to belief & knowledge
     --sets of component c

     --def   atomic_broadcast(IN: attribute,
                        SitesId, ValueSet; OUT: ack) ==
     --define in terms of CONDUCTOR reliable
     --blocking_send( )  primitive

     procedure InstallBelief( c: INOUT CDCrec;
                  b: IN PresumptionRec;
                  characteristic: CharacteristicType);
     --Add belief b about CDC.component to the belief set
     -- of CDC
     --OUT: (all MdocBase: MdocBase'RANGE:
      MdocBase'OUT = MdocBase'IN (0..FindComponent(c)-
            1) + (FindComponent(c).characteristic'OUT
          =   FindComponent(c).characteristic'IN + {b})   +
       MdocBase'IN (i+1...last)

     LocalProperty
         seq(  CheckBelief(c, b, characteristic; ack : OUT
                          Boolean),
           (ack = f and CancelInstallation())
            --belief refuted at local site

           or (AddBelief(ord(c) = i, b, characteristic),
               (ack = f and CancelInstallation())
            --belief refuted at non-local site
           or
                InsertBelief(c, b, characteristic)
               )
              )
            )
           )

     end InstallBelief;
```

```
PackageProperty
 <CurrentState, Inf> always (all b in
                          SetOfBeliefs'RANGE and
   all k in SetOfKnowledge'RANGE and
     all components: ComponentType and
       all i, j: CharacteristicType: b.i = k.j and i = j
             and
         EvalUnless(components.b(i),
                    components.k(j)) = false)
     )
--all beliefs in a belief base exist on the condition
--that they are not refuted by presumptions in the
--component knowledge base.

end ServerKernel;
```

An Ada/ISL kernel specification is a generic package, which is a crucial prerequisite for software reuseability (a package can be created by instantiated its generic unit). In addition, some package parameters are left unspecified. For example, ComponentType is left pending. The primitive type "pending" is borrowed from Gypsy [14] and denotes generic parameters which are identified but not specified otherwise. Individual package procedure parameters are accompanied by IN and OUT assertions which specify procedure pre- and post-conditions. A LocalProperty of a procedure describes the required temporal behavior of that procedure in execution. A PackageProperty specifies the required behavior of the package as a whole. Local and package properties are expressed as interval temporal assertions.

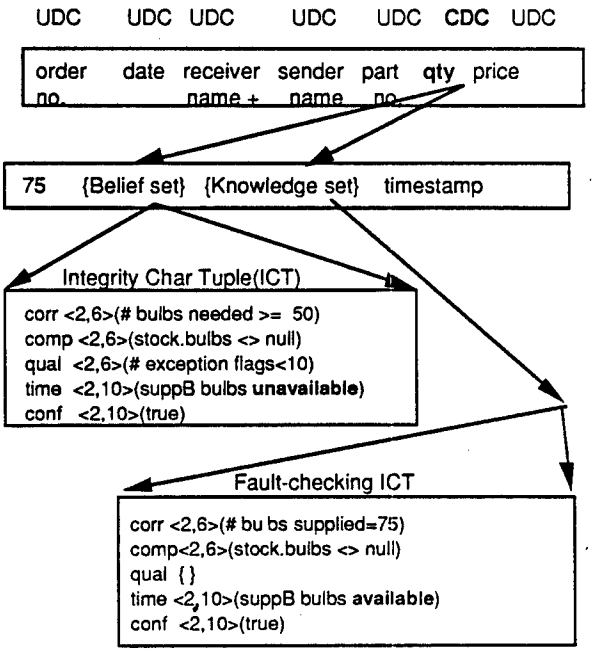
Each temporal interval is mapped to timestamps derived from a local clock. Each DOS clock is assumed to be synchronized with the other clocks in the CONDUCTOR distributed system. This synchronization of DOS clocks is made possible with the help of a distributed clock synchronizer such as TEMPO [15]. The availability of synchronized distributed clocks provides uniform timestamps used in the writing and evaluation of beliefs. This is crucial to the specification of the AddBelief procedure, which broadcasts beliefs to be added to all belief bases across the network. Each site receiving a belief in an AddBelief broadcast must be able to correlate the timestamps associated with the received belief with its own clock for evaluation purposes. In addition, it is assumed that the EvalUnless operation is performed on a pair of presumptions having a common temporal interval.

The document base in the DOS architecture is assumed to be fully replicated across all network sites. Included in the sample integrity kernel is an atomic_broadcast operation. Whenever a client wants to install a new belief in the belief base using the InstallBelief operation, the issuing DOS site emerges as the leader in initiating the installation of the belief. The leader perfoms an atomic broadcast and blocks itself waiting for replies from all other sites.

A summary of the Ada/ISL extensions of package specification given in the ALRM is as follows:

extension	meaning
IN-, OUT-asserts	IN-assert is a precondition on procedure parameters before procedure execution. OUT-assert specifies relationships between the input and output parameters which must hold true as a result of a procedure execution.
abstract types definitions	sequences and sets (as in VDM). define functions used in specifying OUT-asserts and properties.
local property	Temporal assertion describing the required execution behavior of a procedure.
package property	Temporal assertion describing the package behavior as a whole (i.e. interactions between package procedures and preservation of desired properties package resources).

6.2 SAMPLE QUERY PROCESSING SESSION



In the above example, only one belief in the belief set is refuted by a fact in the knowledge set. This refuted belief is the *timeliness* aspect of integrity as represented in the integrity characteristic tuple for the particular data item, namely, bulbs. When a client requests the document component **qty** in the order form, the integrity kernel will validate the query before allowing it to be executed. This evaluation is based on information available to the integrity kernel from the current site state as well as information such site state history, partial information available in the current state and system administrator's intuition expressed in both the belief sets.

Some of these document components have integrity restrictions imposed on them and will be henceforth referred to as CDCs (e.g., **qty**). The rest of them will be known as UDCs. Each CDC will carry with it timestamp information which indicates the most recent time that the belief set was verified. The belief set represents presumptions about each characteristic of the ICT record. The presumptions in the knowledge set are designed to either support or refute the presumptions in the belief set. There will be instances when the knowledge set is empty. In such a case, a belief is believed to be true. This belief could be refuted by a presumption in the knowledge set arriving at a later time.

6. CONCLUSION

Site administrators are constantly faced with only partial knowledge about the integrity of data items over time. Typically, site administrators and some users also have an intuition concerning the integrity of the data. Ada/ISL provides a vehicle for capturing this idea within a logic of knowledge in terms of facts and belief in terms of intuition. With predicates over temporal intervals defined by timestamps, this logic takes into account real-world concerns about the duration over which a presumption about data integrity holds true. In this paper, the concept of data integrity has been extended to include document integrity in an office environment. A generalized kernel has been introduced to implement this concept. The focus of an Ada/ISL integrity kernel is the expression of the full range of integrity constraints placed on document components in a precise way.

At the heart of an Ada/ISL specification is the correctness of package properties. A package property must be consistent with package local properties. Not shown in this article is the proof system for Ada/ISL, which is currently being developed.

ACKNOWLEDGEMENT

We wish to thank Profs. Elizabeth Unger, William Hankley, S.B.V. Ramanna for encouraging us to pursue research in specification theory which led to the development of Ada/ISL. We also wish to thank Prof. K. Ravindran for his helpful comments in the development of the CONDUCTOR model.

REFERENCES

[1] Aaby, A.A. *Specification of Distributed Programs Using Using Using Temporal Interval Logic.* Ph.D. Dissertation, Penn State U, 1988.

[2] Defense, U.S. Department of *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD-1815A-1983. NY: Springer-Verlag, 1983.

[3] Berstein, P.A. Shipman, D.W. "The Correctness of Concurrency Control Mechanisms in a System for Distributed (SDD-1)". ACM TODS, vol. 5, no. 1 Mar. 1980), 18-51, 52-68.

[4] Bernstein, P.A. Goodman, N. "Concurrency Control in Distributed Database Systems". ACM Computing Surveys, vol. 13, no. 2 (June 1981), 185-122.

[5] Bertino, E. Rabitti, F. "Query Processing Based on Complex Object Types". IEEE Database Engineering (Dec. 1986), 22-29.

[6] Bertino, E. et al. "A multimedia document server. In Proceedings of Advanced Database Symposium" (Tokyo, Japan, Aug. 29-30), Information Processing Society of Japan, 1986, 123-133.

[7] Bertino, E. Rabitti, F. Gibbs, S. "Query Processing in a Multimedia Document System" ACM Transactions on Office Information Systems, vol. 6, no. 1 (Jan. 1988), 1-42.

[8] Rabitti, F. "A Model for Multimedia Documents". In Office Automation ed by D. Tsichritzis. NY: Springer-Verlag, 1985, 227-250.

[9] Biba, K.J. "Integrity Considerations for Secure Computer Systems". USAF Electronic Systems Division, Bedford, MA: EDS-TR-76-372, April, 1977.

[10] Clark, D.D. Wilson, D.R. "A Comparison of Commercial and Military/Computer Security Policies". IEEE Proceedings of the 1987 Symposium on Security and Privacy, April, 1987.

[11] Codd, E.F. "Extending the Database Relational Model to Capure More Meaning". ACM TODS vol. 4, no. 4 (Dec. 1979), 397-434.

[12] Dean, T. "Using Temporal Hierarchies to Efficiently Maintain Large Temporal Databases". Journal of the ACM, vol. 36, No. 4 (October 1989), 687-718.

[13] Diaz-Gonzales, J.P. *The Requirements Engineering of Real-Time Systems: A Temporal Logic Approach.* Ph.D. Dissertation, U of SW Louisiana, 1987.

[14] Good, D. "Revised Report on Gypsy 2.1." Tech Report, Institute of Computing Sciences, UT at Austin, TX, July 1985.

[15] Gusella, R. and S. Zatti. "An Election Algorithm for a Distributed Clock Synchronization Program". 6th International Conference on Distributed Computing Systems. IEEE CS (May 1986), 364-371.

[16] Habermann, A.N. Perry, D.E. *Ada for Experienced Programmers.* Reading, MA: Addison-Wesley Publishing Company, 1983.

[17] Hankley, W. and Peters, J. "A Proof Method for Ada/TL Specifications". Proceedings of 8th Annual National Conference on Ada Technology (March 5-8, 1990).

[18] Hankley, W. Peters, J. "Temporal Specification of Ada Tasking," Proceedings of the 23rd Hawaii International Conference on System Sciences (Jan. 1990).

[19] Kripke, S.A. "Semantic Analysis of Modal Logic". Z. Math. Logik Grundlagen. vol. 9, 1963, 67-96.

[20] Kroeger, F. *Temporal Logic of Programs.* NY: Springer-Verlag, 1987.

[21] Lamport, L. "Timesets: A New Method for Temporal Reasoning About Programs". Logics of Programs Conference, vol. 131 (Sept. 1981), 177-196.

[22] Levesque, H.J. *A Formal Treatment of Incomplete Knowledge Bases.* Ph.D. dissertation, University of Toronto, 1981

[23] Moser, L.E. "A Logic of Knowledge and Belief for Reasoning about Computer Security". TRCS89-16. U of California, Santa Barbara, May, 1989.

[24] Natarajan, N. "Communication and Synchronization Primitives for Distributed Programs". IEEE Trans. on Software Engineering, vol. SE-11, no. 4 (April 1985), 396-416.

[25] Peters, J.F. and Ramanna, S. "CONDUCTOR: A Reliable Model for Distributed Communicating Processes". Technical Report, Department of Computing & Information Sciences, Kansas State University, 1990.

[26] Pneuli, A. "The Temporal Logic of Programs," 18th Annual Symposium on Foundations of Computer Science (Nov. 1977), 46-57.

[27] Ramanna, S. Peters, J.F. "Logic of Knowledge and Belief in the Design of a Distibuted Integrity Kernels ." Proceedings of PARBASE-90:International Conference on Database, Parallel Architectures, and their Applications, Miami Beach, FL (March 7-9, 1990).

[28] Reiter, R. "On Integrity Constraints". Proc. of the Second Conf. on Theoretical Aspects of Reasoning and Knowledge (Mar. 7-9, 1988), 97-112.

[29] Rothnie, J.B. et al "Introduction to a System for Distributed Databases (SDD-1)". ACM TODS, vol. 5, no. 1 (Mar. 1980), 1-17.

[30] Schwartz, R.L. Melliar-Smith, P.M. Vogt, F.H. "An Interval Logic for Higher Level Temporal Reasoning". ACM PODC, 1983.

[31] Svobodova, L. "File Servers for Network-based Distributed Systems. ACM Computing Surveys, vol. 16, no. 4 (Dec. 1984), 350-398.

Appendix A Ada/ISL Syntax

Notation: { } means 0 or more occurrences;
[ ] means 0 or 1 occurrence.

```
Ada/ISLspecification ::=    [generic
      ParameterPart {; ParameterPart}]
      package KernelName is
            Package_Declarative_Part
      end KernelName;
      Package_Declarative_Part
            ::=    {Basic_Declarative_Item}
                   {WFP}
                   {Specific_subprogram_specification}
                   {Definition}
                   {Package_Property}

      WFP   ::= [K] [<time_interval>] ( assert )
      assert ::= true | false | identifier
            |      not assert
            |      quant_assert
            |      assert BinOp assert
            |      temporal_expr
      quant_assert
            ::=(quant id {, id}: id_constraint: quant_assert)
            |    temporal_expr
      id_constraint
            ::= id_const [ logical_operator id_const]
      id_const
            ::= type_mark | expression
      quant  ::= all | exists
      id     ::= identifier
      BinOp  ::= logical_operator | unless | until
      time_interval ::= const , const
      Specific_subprogram_specification
            ::= subprogram_specification
            [--IN: expr]
            [--OUT: expr]
            [Local_Property]
      Local_Property
            ::= LocalProperty WFP
      Package_Property
            ::= PackageProperty WFP
      temporal_expr
            ::= [TempOp] expr [BinOp temporal_expr]
      TempOp
            ::= always | eventually | seq
      expr   ::=  expression
            | seq( temporal_expr { , temporal_expr } )
      Definition  ::=
      --Def {DefName (ParameterList) == Boolean_expr}
```

Appendix B Specification of an Mdoc Integrity Kernel

```
--Ada/ISL specification for a multimedia document server
--integrity  kernel

generic
  type       ComponentType is pending;
             --document conceptual component type
  type       LogicalClockType is pending;
  type       FirstOrderSentType is pending;
  type             IndexType is integer;

package Serverkernel is

  type PresumptionRec is
    record
     start_time, ending_time: LogicalClockType;
        --temporal interval
       presumption: FirstOrderSentType;
    end record;

  type  ICTrec is
--integrity  characteristics  tuple
   record
      corr, comp, qual, time, conf: set of PresumptionRec;
   end record;

type CDCrec is
--constrained document component
   record
      Component                  : ComponentType;
      BeliefSet, KnowledgeSet    : ICTrec;
```

```
    TimeStamp                    : LogicalClockType;
      --current time
  end record;

type ChacteristicType is (corr, comp, qual, time, conf);

type MdocBaseType is seq of CDCrec;
--defines entire document base = sequence of CDC records

 --exceptions:
    reject_belief          : exception;
                           --when belief is contradicted
    expired_belief         : exception;
                           --when current time > end_time
                           --of a belief

var c                      : CDC := ();
    MdocBase               : MdocBaseType := seq( );
    attribute              : CharacteristicType;
    SitesId                : integer;    --unique DOS site id
    ValueSet               : pending;
    attribute              : string;
    SetofBeliefs,
    SetofKnowledge         : set of ICTrec;

--def   EvalUnless(Belief, Knowledge) ==
      if Belief unless Knowledge then return true
      else return false endif;
--evaluate "Belief unless Knowledge" in the current
--state over the same interval.

--def FindComponent(c: CDCrec; i : OUT IndexType) ==
       (let (exists i in MdocBase:
         MdocBase(i).Component = c.Component): i)
--identifies ith component of Mdoc document base.

--def    VerifyBeliefwithKnowledgeSet(i, K, b) ==
        (all k in K:K'RANGE: EvalUnless(b, k)
 --verify belief b with respect to every presumption k
 --in  KnowledgeSet K

 --def   VerifyBeliefwithBeliefSet(i, b, B) ==
         (all belief in B:B'RANGE: EvalUnless(b, belief)
        )
--verify belief b with respect to every presumption
--belief in BeliefSet B

--def   FindKnowledge(i, characteristic;
                      K : OUT ICTrec) ==
    Mdoc(i).KnowledgeSet.characteristic
--returns knowledge set associated with characteristic
--of data item

--def  InsertBelief(c :INOUT CDCrec; b, characteristic) ==
    (OUT assert of InstallBelief(c,b,characteristic))
--Inserts belief b associated with the characteristic of
-- a component c into the belief base.

--def   AddBelief(i, b, characteristic) ==
    (seq (atomic_broadcast("check belief",
          SitesId, i, b, characteristic; ack : OUT Boolean),
      ( all ack: site'RANGE: ack = "belief is verified" and
       atomic_broadcast("insert belief",
          SitesId, i, b, characteristic; ack : OUT Boolean)
      or
       atomic_broadcast("cancel installation",
          SitesId, i, b, characteristic; ack : OUT Boolean)
      )
     )
    )

--performs the following sequence over the
--CONDUCTOR distributed system:
--(1) broadcasts instruction to execute  CheckBelief
- -     operation,
--(2) broadcasts instruction to execute InsertBelief
- -     if all ack's indicate non-refutation of belief b
- -     or
--       broadcasts instruction to cancel installation
- -     of belief.
--def   CheckBelief(c, b, characteristic;
                        ack : OUT Boolean) ==
     --ack = t or f
    (seq( FindComponent(c),
          --returns index i of doc component
         (CancelInstallation()
           --component not found
          or
          (FindKnowledge(ord(c) = i, characteristic;
                         K : OUT ICTrec),
           --returns knowledge set K
            VerifyBeliefwithKnowledgeSet(i, K, b),
            FindBeliefSet(ord(c) = i, b, characteristic;
                          B : OUT ICTrec),
           --returns BeliefSet B
            VerifyBeliefwithBeliefSet(i, b, B) ))))
--(1) verify belief b not contradicted by beliefs in
--belief set B of component c,
--(2) verify belief b not refuted by knowledge in
--knowledge set K of component c.

--def    AddKnowledge(i, k, characteristic) ==
       (atomic_broadcast("insert knowledge",
                  SitesId, i, k, characteristic),
        and
         InsertKnowledge(i, k, characteristic))
--initiates network-wide execution of
--InsertKnowledge operation

--def    InsertKnowledge(i, k, characteristic)
    ( seq( OUT assert of InstallKnowledge(chr(i),
                          k, characteristic),
       FindBelief(i, characteristic; B : OUT ICTrec),
       --returns belief set B
       (all b in c.BeliefSet:
         ack = EvalUnless (c.b(characteristic),
                           c.k(characteristic)),
        (ack = true and
          Delet eBelief(i, b, characteristic)
        --belief refuted by knowledge
        or
         skip()
         --do nothing ) ))
--(1) adds knowledge k to knowledge set for
-- component c,
--(2) deletes all beliefs in belief set refuted
-- by knowledge k

--def     CheckIntegrity(c, CurrentTime;
                         ack : OUT Boolean) ==
(OUT assert of VerifyIntegrity(c, CurrentTime, ack))
--verifies integrity of all beliefs in belief base with
--respect to all presumptions of the knowledge base of
--component c.

--def   atomic_broadcast(attribute, SitesId, ValueSet;
                         ack : OUT Boolean) ==
--define in terms of CONDUCTOR reliable
--blocking_send( ) primitive
```

```
procedure InstallBelief( c: INOUT CDCrec;
                         b: IN PresumptionRec;
                 characteristic: CharacteristicType);
--Add belief b about CDC.component to the belief set
-- of CDC
--OUT: (all MdocBase: MdocBase'RANGE:
 MdocBase'OUT = MdocBase'IN (0..FindComponent(c)-
      1) + (FindComponent(c).characteristic'OUT
   =    FindComponent(c).characteristic'IN + {b})   +
      MdocBase'IN  (i+1...last))

LocalProperty
    ( seq(  CheckBelief(c, b, characteristic;
                    ack : OUT Boolean),
      ((ack = f and CancelInstallation())
       --belief refuted at local site

      or (AddBelief(ord(c) = i, b, characteristic),
          (ack = f and CancelInstallation())
       --belief refuted at non-local site
      or
          InsertBelief(c, b, characteristic) ) ) )

end InstallBelief;

procedure InstallKnowledge(c: INOUT CDCrec;
              --constrained doc component
              k: IN PresumptionRec;
              --knowledge about component
              characteristic: CharacteristicType);
--Add knowledge to the Knowledge set of the CDC
--OUT:  symmetric with OUT assert of InstallBelief

LocalProperty
    ( seq(FindComponent(c),
       --returns index i of doc component
       (CancelInstallation()
        --component not found
       or
        AddKnowledge(ord(c) = i, k, characteristic) ) )

end InstallKnowledge;

procedure VerifyIntegrity( c: IN CDCrec;

       CurrentTime  : IN logicalClockType;
       ack                  : OUT Boolean);
--Verify the integrity of CDC when an update to a
--document component is attempted
--OUT:
   if  (all i in attribute, j in ICTrec'range:
       c.BeliefSet(i).ending_time(j)  <  CurrentTime)
   then
       raise  expired_belief;
        --a belief is no longer current
   else  --evaluate CDC
     if (all i in attribute, and
              all j,k in ICTrec'range:
          EvalUnless( c.BeliefSet(i).presumption(j),
             c.KnowledgeSet(i).presumption(k))
     then
        ack = true;
     else
        ack = false;
        raise  reject_belief;
        --1 or more beliefs are faulty
     endif ;
   endif;

LocalProperty
    (seq( CheckIntegrity(c, CurrentTime, ack),
        (ack = false and
           CancelInstallation()
         ) or
         atomic_broadcast(IN: "check integrity",
              SitesId, c, CurrentTime; OUT: ack),
         (all ack: site'RANGE: ack = "true" and
              ack'OUT = true)
         or
         ack'OUT = false
         )
      )
   )

end VerifyIntegrity;

PackageProperty
 <CurrentState, Inf> always (all b in
                          SetOfBeliefs'RANGE and
   all k in SetOfKnowledge'RANGE and
     all components: ComponentType and
       all i, j: CharacteristicType: b.i = k.j and i = j
             and
         EvalUnless(components.b(i),
                  components.k(j)) = false)
      )
--all beliefs associated with a component in a belief
-- base exist on the condition
   --that they are not refuted by presumptions in the
   --component knowledge base.

 end ServerKernel;
```



Sheela Ramanna received the B.E. and M.S. in Electrical Engineering and Computer Science at from Osmania Univ., India and is currently a Ph. D. candidate in Computer Science at Kansas State University. She has previously worked as a scientist at the Defense Research and Development Laboratory, Hyderbad, India. Her current research interests include specification theory, distributed systems, database management systems and data integrity theory, and artificial intelligence. She has co-authored other articles with James Peters. She is currently a member of ACM and Upsilon Pi Epsilon.

James F. Peters III received the B.A. and M.S. in Logic and Mathematics from the University of Santa Clara, SC, CA and is currently a Ph.D. candidate in Computer Science at Kansas State University.
He has previously taught Mathematics and Computer Science at St. John's University, MN. He worked as a systems analyst for the Micro City Project, St. Cloud, MN and as editor of Inside Dust Jacket, St. Cloud Daily Times, St. Cloud, MN. He is an author of several books and has coauthored a number of other specification theory articles in Computer Science. His research interests include temporal logic, specification theory, distributed systems, Ada, and artificial intelligence. He is currently a member of the ACM, IEEE, Sigma Xi and Upsilon Pi Epsilon.

A Clean Solution to the Readers-Writers Problem Without the COUNT Attribute[1]

David Levin — University of Wisconsin-Parkside

Daniel Nohl — Illinois Benedictine College

Tzilla Elrad — Illinois Institute of Technology

Abstract

The classical readers-writers problem in all its various forms presents challenges for the correct implementation in any concurrent language. In Ada, most solutions to the form of the problem that gives preference to writers over readers incorporate the use of the COUNT attribute. A current report [Schiper 89] indicates an overlooked trap in these solutions and presents another solution that eliminates the trap but still necessitates the use of COUNT. The COUNT attribute should be avoided whenever possible according to the Ada LRM. We present a solution to the readers-writers problem with preference given to writers using a preference control construct. The solution is clear and straight forward, naturally expressing the problem specification. Ada is a language suitable for expressing a solution throughout the entire life cycle of a software project [Booch 83]. Adding preference control to Ada conforms to this design philosophy.

Introduction

The readers-writers problem is one of the benchmark problems in the evaluation of concurrent programming languages. In this problem there is data shared by a number of tasks, some trying to read the data and others trying to write to the data. The problem is to ensure that one writer and no readers have access to the data or that a number of readers (but no writer) have access. There are several algorithms for solving this problem which employ different standards of fairness (e.g. readers having preference over writers or vice-versa). In Ada, most solutions use a task to manage the shared resource. Implementations of some of the more sophisticated algorithms in Ada have used the COUNT attribute in the guard to select alternatives within the manager task. However, it is well known that such use of the COUNT attribute may cause problems. The Ada Reference Manual points out that this use of the COUNT attribute should be avoided.

[1] This work is partially supported by the U.S. Army Research Office CECOM and Battelle Scientific Services Program #1263.

In this paper we identify the reason the use of the COUNT attribute causes the problem. As is well known, there is an inconsistency in the use of COUNT in guards on select alternatives in Ada and that such use of COUNT should be avoided wherever possible. We propose a solution to the readers-writers problem with writer preference which avoids use of the COUNT attribute. Our solution employs explicitly stated preference control.

Readers-writers with writer preference

In this paper we will consider the reader-writer problem with preference given to the writer. According to this algorithm the manager should not accept any further requests to read once a request to write has been acknowledged. We refer to this policy as readers-writers with writer preference. Figure 1 illustrates a widely accepted solution to the readers-writers problem with writer preference [Gehani 84].

```
task RW_MANAGER is
  entry START_WRITE;
  entry START_READ;
  entry END_WRITE;
  entry END_READ;
end RW_MANAGER;

task body RW_MANAGER is
  NB_READERS:NATURAL := 0;
  WRITER_PRESENT:BOOLEAN := FALSE;
begin
  loop
    select
      when not (WRITER_PRESENT) and
           NB_READERS = 0 =>
           accept START_WRITE;
           WRITER_PRESENT := TRUE;
    or
      when not (WRITER_PRESENT) and
           START_WRITE'COUNT = 0 =>
           accept START_READ;
           NB_READERS := NB_READERS + 1;
    or
                 accept END_WRITE;
                 WRITER_PRESENT := FALSE;
    or
                 accept END_READ;
                 NB_READERS := NB_READERS - 1;
    end select;
  end loop;
end RW_MANAGER;
```

Figure 1

The stowaway reader and a problem with COUNT in guards

Schiper and Simon [Schiper 89] point out that in the solution given in Figure 1 it is possible for a task to successfully call START_READ after a task has called START_WRITE violating the problem specification. They refer to this as the problem of the stowaway reader. This inconsistency occurs because the guards are evaluated at the beginning of the select statement and are only re-evaluated after the body of the select statement is executed. This may cause the START_READ guard to be open, when START_WRITE is, in fact, greater than 0, since the guard has not been re-evaluated after the call to START_WRITE has been made.

This method of evaluation of guards is reasonable and should be safe because the guard, containing only local variables, provides a form of control private to the task executing the select statement. Since the task executing the select statement is waiting for a call to one of the select alternatives, the values of these variables should not change. Thus, it is reasonable to expect that a closed guard should remain closed and an open guard remain open until the task performs some action. The use of the COUNT attribute in a guard contradicts this expectation. Syntactically, START_WRITE'COUNT is local to RW_MANAGER and can, therefore, be used as a means of private control within a guard. Semantically, on the other hand, START_WRITE'COUNT is not private since its value is dependent upon the actions of other tasks.

Here we are employing a distinction between two means of controlling nondeterminism in the Ada select statement. Conditions which are determined solely by the task executing the select statement provide private control. It is this form of control which the boolean guard is intended to provide. Conditions which depend upon events outside the task, such as another task calling a particular entry, provide forms of consensus control. Semantically, the use of the START_WRITE'COUNT attribute provides a means of consensus control, rather than private control, over the availability of the START_READ alternative. That is, the availability of the alternative is determined by RW_MANAGER and other tasks which might call START_WRITE. In distinguishing private from consensus controls we are using a classification scheme developed by Elrad and Maymir-Ducharme [Elrad 86].

Schiper and Simon propose the solution in Figure 2 which eliminates the possibility of a stowaway reader, but which does not avoid the use of the COUNT attribute in guards.

This solution solves the problem of the stowaway reader by evaluating the guard condition, START_WRITE'COUNT /= 0, in the if-statement executed immediately after the call to START_READ is accepted. While this solution avoids a stowaway reader, we wish to point out three problems with it.

```
task RW_MANAGER is
  entry START_WRITE;
  entry START_READ;
  entry END_WRITE;
  entry END_READ;
end RW_MANAGER;

task body RW_MANAGER is
  NB_READERS:NATURAL := 0;
  NB_WRITERS:NATURAL := 0;
begin
  loop
    select
      when NB_WRITERS + START_WRITE'COUNT = 0 =>
          accept START_READ do
            -- check now to see if calls have been
            -- made to START_WRITE since the last
            -- evaluation of the guards
            if START_WRITE'COUNT /= 0 then
              -- first let readers finish
              while NB_READERS /= 0 loop
                 accept END_READ;
                 NB_READERS := NB_READERS - 1;
              end loop;
              -- second allow writers to write
              while START_WRITE'COUNT /= 0 loop
                 accept START_WRITE;
                 accept END_WRITE;
              end loop;
            end if;
          -- calling task reads
          end START_READ;
          NB_READERS := NB_READERS + 1;
      or
          when NB_WRITERS + NB_READERS = 0 =>
               accept START_WRITE;
               NB_WRITERS := 1;
      or
               accept END_WRITE;
              NB_WRITERS := 0;
      or
              accept END_READ;
              NB_READERS := NB_READERS - 1;
      end select;
    end loop;
end RW_MANAGER;
```

Figure 2

— It offers a very unclear solution to a problem with clear specifications. The solution is very difficult to follow. A condition appears both in the guard to a select alternative and in an if-statement executed as soon as the alternative is selected. Accept statements for START_WRITE, END_WRITE, and END_READ each appear twice in the select statement. For these reasons we believe the solution violates principles of good software design.

— The solution still employs the use of the COUNT attribute. Instead, the solution works around the problem with the use of COUNT by re-evaluating the condition within the guard depending upon a COUNT attribute after the alternative has been selected.

```
-- Rendezvous should be kept as short as
   possible.  However, in this case the
   START_READ rendezvous could be
   indefinitely long, if writes continue
   to call START_WRITE.  While the
   indefinite waiting by readers is an
   unavoidable product of the any
   solution to the readers-writers
   problem with preference given to
   readers, with other solutions readers
   wait suspended on the START_READ
   queue.  With this solution a reader
   can be forced to wait during a
   rendezvous.
```

We conclude that avoiding the stowaway reader with the tools currently available in Ada is not worth its costs.

We would like to investigate solutions which avoid the troubling use of the COUNT attribute in guards altogether rather than working around the inconsistency in such use. Furthermore, embedding the loops in the reader rendezvous as this solution does inordinately lengthens the rendezvous time which may have drawbacks of its own.

Solutions avoiding the use of COUNT

We will present a clean solution which is readable, avoids the use of the COUNT attribute, and clearly follows the problem specification. As a first attempt at a solution we consider adding a REQUEST_WRITE alternative to the select. This solution is presented in Figure 3.

A task wishing to write would have to rendezvous twice with RW_MANAGER in order to get access; once calling WRITING_REQUEST and a second time calling START_WRITE. In this solution a count of tasks wishing to write is explicitly kept by accepting a WRITING_REQUEST without condition and recording an increase in NB_WRITER_RQS. Although this avoids the stowaway reader without using COUNT, the solution is flawed. Suppose WRITER_PRESENT = FALSE, the guards to both the START_READ and WRITING_REQUEST alternatives are open. If calls are pending at both these alternatives, then at the beginning of the select statement both these alternatives are available for selection by RW_MANAGER. Since writers are to be given preference over readers, we want WRITING_REQUEST to be selected in case both are available. However, given the semantics of select, RW_MANAGER will nondeterministically make a selection between these available alternatives. Moreover, Ada does not have a means of explicitly controlling such a race between available alternatives. Solutions employing the COUNT attribute resolve this race condition by rendering the START_READ alternative unavailable when a call to START_WRITE has been acknowledged.

The issue of preference controls in Ada has been investigated by Elrad and Maymir-Ducharme [Elrad 86]. Elrad and Maymir-Ducharme propose a theoretical construct **prefer** which controls the race between available alternatives within a

```
task RW_MANAGER is
    entry WRITING_REQUEST;
    entry START_WRITE;
    entry START_READ;
    entry END_WRITE;
    entry END_READ;
end RW_MANAGER;

task body RW_MANAGER is
    NB_READERS:NATURAL := 0;
    NB_WRITER_RQS:NATURAL := 0;
    WRITER_PRESENT:BOOLEAN := FALSE;
begin
    loop
        select
                accept WRITING_REQUEST;
                NB_WRITER_RQS := NB_WRITER_RQS + 1;
        or
            when NB_READERS = 0 and
                    not (WRITER_PRESENT) =>
                accept START_WRITE;
                NB_WRITER_RQS := NB_WRITER_RQS - 1;
                WRITER_PRESENT := TRUE;
        or
            when NB_WRITER_RQS = 0 and
                    not (WRITER_PRESENT) =>
                accept START_READ;
                NB_READERS := NB_READERS + 1;
        or
                accept END_WRITE;
                WRITER_PRESENT := FALSE;
        or
                accept END_READ;
                NB_READERS := NB_READERS - 1;
        end select;
    end loop;
end RW_MANAGER;
```

Figure 3

select statement by forcing the choice of alternatives with the highest preference. Given such a construct, the solution presented in Figure 4 is clear and simple.

Although this solution uses the **prefer** construct not available in Ada, it shows how a clean solution would be possible if we could explicitly control the race between available alternatives. This solution gives preference to writers while avoiding the use of the COUNT attribute. The means of controlling the choice of a writer over a reader is one of controlling a race between available alternatives rather than rendering all but one alternative unavailable. In this way the use of preference more truly reflects the nature of the readers-writers problem than do solutions employing COUNT. Preferences have not been assigned to the START_WRITE, END_WRITE, and END_READ alternatives. The reader is challenged to consider different ways of assigning preferences to these alternatives.

This solution might be criticized because it requires a double rendezvous for successful access to writing. Such added communication should, of course, be avoided wherever possible. The solution in Figure 5 avoids the double rendezvous.

```
type prefer is (HIGH,MEDIUM);

task RW_MANAGER is
     prefer HIGH :   entry WRITING_REQUEST;
                     entry START_WRITE;
     prefer MEDIUM : entry START_READ;
                     entry END_WRITE;
                     entry END_READ;
end RW_MANAGER;

task body RW_MANAGER is
     NB_READERS:NATURAL := 0;
     NB_WRITER_RQS:NATURAL := 0;
     WRITER_PRESENT:BOOLEAN := FALSE;
begin
     loop
          select
            prefer HIGH :
                  accept WRITING_REQUEST;
                  NB_WRITER_RQS := NB_WRITER_RQS + 1;
          or
              when NB_READERS = 0 and
                       not (WRITER_PRESENT) =>
                  accept START_WRITE;
                  NB_WRITER_RQS := NB_WRITER_RQS - 1;
                  WRITER_PRESENT := TRUE;
          or
            prefer MEDIUM :
              when NB_WRITER_RQS = 0 and
                       not (WRITER_PRESENT) =>
                  accept START_READ;
                  NB_READERS := NB_READERS + 1;
          or
                  accept END_WRITE;
                  WRITER_PRESENT := FALSE;
          or
                  accept END_READ;
                  NB_READERS := NB_READERS - 1;
          end select;
     end loop;
end RW_MANAGER;
```

Figure 4

Although this solution avoids the double rendezvous, it does so at the cost of extending the rendezvous time of the writer in case there are readers present. We are not certain whether the cost of the double rendezvous or that of the extended rendezvous is greater. Perhaps the choice of which cost to endure is best left up to the programmer.

The solution in Figure 6 which also uses preference control is essentially equivalent to the one presented in Figure 5, but without any guards to evaluate.

In their paper Schiper and Simon also discuss a problem arising with the classical solution of Figure 1 if a writer cancels a call to START_WRITE. While a full discussion of that issue is beyond the scope of this paper, this problem can be solved by adding a cancellation entry with highest preference.

Conclusion

The classification of the controls over the nondeterministic choices help us understand the

```
task RW_MANAGER is
     prefer HIGH   : entry START_WRITE;
     prefer MEDIUM : entry START_READ;
                     entry END_WRITE;
                     entry END_READ;
end RW_MANAGER;

task body RW_MANAGER is
     NB_READERS:NATURAL := 0;
     WRITER_PRESENT:BOOLEAN := FALSE;
begin
     loop
          select
            prefer HIGH :
              when not (WRITER_PRESENT) =>
                  accept START_WRITE do
                    while NB_READERS /= 0 loop
                       accept END_READ;
                       NB_READERS := NB_READERS - 1;
                    end loop;
                  end START_WRITE;
                  WRITER_PRESENT := TRUE;
          or
            prefer MEDIUM :
              when not (WRITER_PRESENT) =>
                  accept START_READ;
                  NB_READERS := NB_READERS + 1;
          or
                  accept END_WRITE;
                  WRITER_PRESENT := FALSE;

          or
                  accept END_READ;
                  NB_READERS := NB_READERS - 1;
          end select;
     end loop;
end RW_MANAGER;
```

Figure 5

source of the problem with the use of COUNT in guards to select alternatives. In order to remain consistent with the assumption that private control is purely private to the task, the boolean guard should not contain any expressions with consensus semantics; that is, an expression which depends upon events originating outside the task. The COUNT attribute is only one example of an expression which is syntactically private control, but semantically consensus control. The use of any such expression should be avoided as a guard condition.

In this paper we have proposed solutions to the readers-writers problem which avoid the problematic use of the COUNT attribute in guards to select alternatives. Our solutions employ explicit preference control. While Ada does not presently offer a means of such control, our solutions show that with such control clear and readable solutions to the readers-writers problem are attainable. We believe that clarity and consistency afforded by such preference control is in line with the philosophy common among users of Ada that implementations of programming problems should be consistent, clear, and straight forward. For these reasons we believe the benefits of implementing preference control are well worth its cost in efficiency.

```
task RW_MANAGER is
     prefer HIGH   : entry START_WRITE;
     prefer MEDIUM : entry START_READ;
                     entry END_WRITE;
                     entry END_READ;
end RW_MANAGER;

task body RW_MANAGER is
     NB_READERS:NATURAL := 0;
begin
     loop
          select
               prefer HIGH :
                    accept START_WRITE do
                      while NB_READERS /= 0 loop
                         accept END_READ;
                         NB_READERS := NB_READERS - 1;
                      end loop;
                    end START_WRITE;
                    accept END_WRITE;
          or
               prefer MEDIUM :
                    accept START_READ;
                    NB_READERS := NB_READERS + 1;

          or
                    accept END_READ;
                    NB_READERS := NB_READERS - 1;
          end select;
     end loop;
end RW_MANAGER;
```

Figure 6

The classical solution to the readers-writers problem simulates preference control by using the COUNT attribute. We show that a solution using an explicit preference control is more reliable and provides a natural expression of its implementation in Ada. In this solution we have used static preference only. An additional characteristic of the algorithm discussed here is a possible starvation of readers. The reader is challenged to simulate algorithms which avoid reader starvation by using a capability to modify preference values at run-time.

References

[Boosh 83] Booch, Grady, "Software Engineering with Ada", The Benjamin/Cummings Publishing Company, Inc., 1983.

[Elrad 86] Elrad, T. and F. Maymir-Ducharme, "Introducing the Preference Control Primitive: Experience with Controlling Nondeterminism in Ada", Proceedings of 1986 Washington Ada Symposium, Laurel, Maryland, March 24-26, 1986.

[Gehani 84] Gehani, N., "Ada Concurrent Programming", Prentice-Hall, 1984.

[Schiper 89] Schiper, A. and R. Simon, "Traps Using the COUNT Attribute in the Readers-Writers Problem", ADA LETTERS IX:5, July/August 1989.



David S. Levin, Philosophy Department, University of Wisconsin - Parkside, Kenosha, WI 53141.

Levin is an Associate Professor of Philosophy at the University of Wisconsin - Parkside where he teaches courses in Computer Science as well as Philosophy. Levin received a B.S. from the State University of New York at Buffalo and the M.A. and Ph.D. in Philosophy from Cornell University and is currently working on a M.S. in Computer Science at Illinois Institute of Technology. Besides concurrent programming Levin's research interests are in natural language processing, expert systems, and logic.



Daniel E. Nohl, Department of Mathematics and Computer Science, Illinois Benedictine College, Lisle, IL 60532.

Nohl is Assistant Professor of Computer Science at Illinois Benedictine College. Nohl received his B.S. (Mathematics) in 1973 and M.S. (Computer Science) in 1977 from the University of Illinois. He is currently working on a Ph.D. at Illinois Institute of Technology. Nohl's research interests include the control of scheduling in concurrent programming systems. He has been teaching in the field of computer science for 13 years.



Tzilla Elrad, Computer Science Department, Illinois Institute of Technology, Chicago, IL 60616.

Elrad is an Assistant Professor of Computer Science at Illinois Institute of Technology. Elrad received the BS in Mathematics, Physics, and Education from the Hebrew University, the MS in Computer Science from Syracuse University, and the PhD in Computer Science from the Technion. Elrad's current research interests include concurrent programming, real-time applications, and formal specification and verification.

USE OF ADA FOR EXPRESSING REQUIREMENTS AND SPECIFICATIONS FOR A SYSTEM FOR THE AUTOMATION OF THE AIR TRAFFIC CONTROLLER'S ASSISTANT'S ASSISTANT

Lee Lahti, Amy Lipscomb
Terence Every, and William Longshore

Computer Science Department, Embry-Riddle Aeronautical University
Daytona Beach, Florida 32114

As an effort to reduce the workload of Air Traffic Controller Assistants, there is a need to develop an automated system to carry out some of the administrative tasks of Air Traffic Control units at small to medium size airports. It is important that such an automated system be flexible enough for use by various airports with minimal change. The system must be capable of keeping a record of all Air Traffic Movements (Arrivals/Departures), calculating and issuing Landing and Parking fee invoices, and producing statistics about the traffic flow at the airports. We felt that using the Ada language would help us to follow the processes, princilples, and goals (PPG) [1] of Software Engineering in developing a prototype software system to automate the tasks of Air Traffic Controller Assistants. Our paper presents a method to create such a system by stating the requirments and specifications needed to successfully perform the task.

1. INTRODUCTION.

We became aware of the above mentioned problem when Mr. Hassan Al-Mousawi, a graduate student at our university (former graduate of the Aviation Computer Science Degree program at Embry-Riddle Aaeronautical University), suggested it for our large project requirement in our CS331 course, An Introduction to Software Engineering. We discussed the acceptability of the problem described above with our instructor, Dr. Jagdish Agrawal. He suggested that we produce a Requirements Definition Document after interviewing the customer (Mr. Hassan Al-Mousawi, who is an experienced Air Traffic Controller at Doha International Airport at Doha, Quatar). We were to make a solution flexible enough to make the software usable at airports comparable in size to Doha International Airport, Quatar. Once the Requirements Definition Document was approved and accepted, we were to produce a Requirements Specification Document using Ada for expressing the specifications, which was to be followed by a Design Document.

1.1 IDENTIFICATION.

This system has been given the approved identification number CS331-01-4-003. It has been titled the "Air Traffic Controller Assistant's Assistant." It has been abriviated ATCAA for the purpose of reference.

We have made a proposal to our university to allow us to implement the design for the system using the Meridian Ada Compiler on an IBM AT compatible system. This implementation will take place under a special topics course, CS399, under the supervision of Dr. Thomas B. Hilburn and Dr. Surendra Kumar, in consultation with Dr. Jagdish Agrawal in the Spring 1990 semester. The implemented system will be treated as a first prototype of the ATCAA. We believe that the prototype will demonstrate the feasibility of doing fair sized software development in Computer Science courses like Software Engineering, as well as demonstrate the feasiblity of automating the maual desk job of Air Ttraffic Controller Assistants.

1.2 SYSTEM OVERVIEW. The system overview has already been provided in the first two paragraphs of the SUMMARY AND INTRODUCTION.

1.3 DOCUMENT OVERVIEW. The System/Segment Specification (SSS) specifies the requirements for the ATCAA. The Software Requirements Specification (SRS) details the specifications for the ATCAA. Following review meetings and upon customer approval and authentication, the SSS and SRS for the ATCAA became the Functional Baseline for the system and its design. These documents provide a general overview of the system, and for a completeness in understanding their context, portions of the documents have been reproduced and combined below and made a part of this paper.

2. APPLICABLE DOCUMENTS.

The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superseding requirement.

Below is a summary of documents that were applicable to this project.

2.1 GOVERNMENT DOCUMENTS.

The following are the governement documents referenced in this document:

1.) DOD-STD-2167A Military Standard on Defense System Software Development.

2.) DOD-STA-1815A Military Standard on the ADA Language.

3.) Designators for Aircraft Operating Agencies - A government provided document listing all companies which operate aircraft.

4.) Location Indicatiors Document - A government provided document listing all federally approved locations for aircraft arrival and departure.

2.2 NON-GOVERNMENT DOCUMENTS.

The following non-government documents have been referenced in this document:

1.) ATC movement report - A summary of one aircraft movement used by Air Traffic Controller units.

2.) ATC landing fee voucher - A summary of landing and parking fees for a particular aircraft or a particular company.

3.) ATC company/aircraft report - A summary of the number of movements for a particular company and its aircraft.

4.) Landing and Parking Fee List - A list of all landing and parking fees for companies and aircraft and those immune from fees for a particular airport.

3. SYSTEM REQUIREMENTS.

System requirements below are categorized in accordance with DOD-STD-2167A to the highest possible degree.

3.1 DEFINITION.

As an effort to reduce the workload of Air Traffic Controller Assistants, there is a need to develop an automated system to carry out some of the administrative tasks of Air Traffic Control units at small to medium size airports. It is important that such an automated system be flexible enough for use by various airports with minimal change. The system must be capable of keeping a record of all Air Traffic Movements (Arrivals/Departures), calculating and issuing Landing and Parking fee invoices, and producing statistics about the traffic flow at the airport. Air Traffic Controller Assistants input data from the ATC's strip and information gained by visual check. After the information has been entered, the system will store the information. Fig 3.1,1 states the information which will be stored. Once stored, the information will then be capable of being configured in a report.

Fig. 3.1,1

MOVEMENT : a record consisting of:

DATE
MOVEMENT NUMBER
REGISTRATION
COMPANY
FLIGHT NUMBER
AIRCRAFT TYPE
A.T.A.
ORIGIN
A.T.D.
DESTINATION
FLIGHT TYPE
FLIGHT RULES
LANDING FEE AMOUNT
USER IDENTIFICATION
REMARKS

3.1.1 DATE. The DATE is the date of the movement. The date shall be in the form DDMMYY.

3.1.2 MOVEMENT NUMBER. The MOVEMENT NUMBER is a computer generated number given to each new movement as it is entered into the computer. The Movement Number shall be in the form YYXXXXXX.

3.1.3 REGISTRATION. The RESISTRATION is the registration number off of the aircraft. The registration shall be entered directly as it is found on the aircraft in a 7 letter code in the form XXXXXXX. If the movement is a military formation, the aircraft type shall be 'FORMATN'.

3.1.4 COMPANY. The COMPANY is the designator of the company that owns the aircraft. The company shall be a 3 letter initial in the form XXX as found in the DESIGNATORS for AIRCRAFT OPERATING AGENCIES.

3.1.5 FLIGHT NUMBER. The FLIGHT NUMBER is the flight number of the aircraft in the movement. The flight number shall be a 5 letter code in the form XXXXX as it is found on the aircraft.

3.1.6 AIRCRAFT TYPE. The AIRCRAFT TYPE is the type of aircraft in the movement. The aircraft shall be computer generated from a database if the registration number is found in the database. If it is not found, type AIRCRAFT TYPE shall be entered as a 7 letter code in the form XXXXXXX. If the movement is a military formation, the aircraft type shall be an 'X' and a 2 letter initial as a summary of the types and numbers of each style of plane in the formation shall form the rest of the code.

3.1.7 A.T.A. The A.T.A. (ACTUAL TIME OF ARRIVAL) is the actual time that the aircraft in the movement arrives at the airport. The A.T.A. shall be in the form DDHHMMSS in military time.

3.1.8 ORIGIN. The ORIGIN is the location of the airport where the arriving aircraft began its trip. The origin shall be a 4 letter initial in the form XXXX as found in the LOCATION INDICATORS DOCUMENT.

3.1.9 A.T.D. The A.T.D. (ACTUAL TIME OF DEPARTURE) is the actual time that the aircraft in the movement departs from the airport. The A.T.D. shall be in the form DDHHMMSS in military time.

3.1.10 DESTINATION. The DESTINATION is the location of the airport where the departing aircraft is traveling to. The destination shall be a 4 letter initial in the form XXXX as found in the LOCATION INICATORS DOCUMENT.

3.1.11 FLIGHT TYPE. The FLIGHT TYPE is the type of flight for the aircraft in the movement. The flight type shall be SCHEDULED, NON-SCHEDULED, GENERAL avaiation, military FIXED wing, or military ROTARY wing and shall be in the form of a 1 letter initial from (S, N, G, F, R) respectively.

3.1.12 FLIGHT RULES. The FLIGHT RULES are the rules by which an aircraft flies. The flight rules shall be INSTRUMENT flight rules, VISUAL flight rules, or SPECIAL visual flight rules; and shall be in the form of a 1 letter initial from (I, V, S) respectively.

3.1.13 LANDING FEE AMOUNT. The LANDING FEE AMOUNT is the sum of the landing fee and parking fee for the aircraft in the movement. The LANDING FEE AMOUNT is generated from a database containing the COMPANY, REGISTRATION, and flight type, but is subject to change from the user and shall be stored as an integer.

3.1.14 USER INDENTIFICATION. The USER IDENTIFICATION is the access name of the person entering the information for the movement. Each person has an individual access code and it is an 8 letter code in the form XXXXXXXX.

3.1.15 REMARKS. The REMARKS are a small summary of events that happen to a movement, if something unusual occurs to the movement. The remarks shall be entered by the user and be 25 characters in length stored in the form XXXXXXXXXXXXXXXXXXXXXXXXX.

3.2 CHARACTERISTICS. Below is a list of the characteristics of the system.

3.2.1 PERFORMANCE CHARACTERISTICS.

In all situations following, with the exception of the main menu and system logon, if the user presses the <ESC> key, the system will take the user back to the previous menu. However, If the <ESC> key is pressed in the main menu or at the system logon prompt, the system will ignore the key.

3.2.1.1 LOGON TO THE SYSTEM. Before any function of the system can be used, a user will be required to logon to the system. The user will be prompted to enter their USER IDENTIFICATION and password. If the system recognizes the user id as an active user and the password is correct, the user will be given access to the system at the main menu. If the user id is invalid, or the wrong password for the user id is given, the system will remain at the logon prompt, but an error message shall be placed in the error message log file.

3.2.1.2 MAIN MENU. When the program is first booted, the main menu will be displayed to the user. It will allow the user to do one of the following things:

1.) Backup/Restore data.
2.) System Execution menu.
3.) System Manager's menu.
4.) Logoff the system.

3.2.1.2.1 BACKUP/RESTORE DATA. If this fucntion of the program is selected, the user is presented with the BACKUP/RESTORE menu. This menu has the

following functions:

1.) Backup a month's data currently on the hard-drive.
2.) Restore a previous month's data from floppy disk.

3.2.1.2.1.1 BACKUP A CURRENT MONTH'S DATA.

If this function of the program is selected, the user will be prompted to enter a month and year for the information that is to be backed-up. The system will check to see if the information for the month requested is currently on the hard-drive. If it is not, a warning will be issued and the system will return to the BACKUP/RESTORE menu. If the information is currently on the hard-drive, the user will be propmpted to insert a disk in the A: drive and press <ENTER> when ready. The system will also display a note which will read "Make sure that the disk you are inserting does not have any important information on it or the information will be destroyed." After the user hits <ENTER>, the system will then copy all current data for that month from the hard-drive and store it on the floppy disk. If the floppy disk fills up before the hard drive is finished copying the data, the program will ask the user to insert another disk to continue backing up the data until it is completed. Upon completion, the system will return to the main menu.

NOTE: This system will automaticaly format a disk after it has been put in the A: drive and <ENTER> has been hit. If the user selects this option and does not put a blank disk in the A: drive when prompted, the system will display a message stating there is no disk in the A: drive and return to the main menu.

ADDITIONAL NOTE: This function may be entered from the ENTER INFORMATION ON NEW MOVEMENT function if the new movement to be entered is the first movement of a new month. If this is the case, the program will check a flag to see if the oldest month of data on the hard-drive has been backed-up to a floppy disk with all of its current information. If it has, the system will write over this file on the hard-drive as a full backup has been made. If the month has not been backed-up, this section shall be entered, the month's information backed-up to floppy disk, the old information shall then be erased from the hard-drive, and a new file shall be created. The system shall then continue with the ENTER INFORMATION ON NEW MOVEMENT function.

3.2.1.2.1.2 RESTORE A PREVIOUS MONTH'S DATA. If this function of the system is selected, the user will be prompted to enter a month and year of a file whose information is to be restored from floppy disk back-ups. The system will then prompt the user to enter a month and year for the file of data on the hard-drive to be temporarily destroyed. A flag shall be checked to see if the month of information on the hard-drive to be destroyed has been backed-up with all of its current data. If it has, the file shall be destroyed from the hard-drive and the user shall be prompted to enter the back-up disk(s) for the month to be restored. If the information has not been backed-up, the user will be warned that the month has not been backed-up and the program will return to the BACKUP/RESTORE CURRENT DATA menu.

3.2.1.2.2 SYSTEM EXECUTION MENU. If this section of the program is chosen then the user is presented with the regular execution menu. This menu will consist of the following choices:

1.) Enter information on a new movement.
2.) Update information on an existing movement.
3.) Print out a report.

3.2.1.2.2.1 ENTER INFORMATION ON A NEW MOVEMENT.

In this section the user is presented with a display which lists all the variables required from the user to complete the record for that movement. The system will use a format similar to that of Fig. 3.2.1.2.2.1,2. The user will be required to enter the information. Once all the information has been entered, the user will be prompted to say the information is correct. If the information is incorrect, the the user will be allowed to update it. If the information for the movement is correct, it will be stored on the hard drive using the format in Fig. 3.1,1. At the conclusion of this, the system will return to the system execution menu.

Some of the information needed will be generated from other information entered into the system. Fig. 3.2.1.2.2.1,1 shows the information which will be generated by the system.

Fig. 3.2.1.2.2.1,1 Computer Generated Input

INPUT	GENERATED OR GENERATES
MOVEMENT NUMBER	: (IS GENERATED GIVEN THE DATE AND THE PREVIOUS MOVEMENT NUMBER)
REGISTRATION	: (GENERATES THE COMPANY AND AIRCRAFT TYPE [FROM DATABASE])
LANDING FEE	: (IS GENERATED BY THE AIRCRAFT TYPE AND COMPANY)

The movement number will be generated with every new movement created. The system shall keep a counter to control this. The registration and landing fee generate information from databases stored on the hard drive with all other information. For information about the databases, refer to 3.3.

Fig. 3.2.1.2.2.1,2 Input System for ATCAA (CG = Computer Generated)

REGISTRATION	: (Visual Check)
COMPANY	: (Computer Generated)
FLIGHTNUMBER	: (ATC Strip)
AIRCRAFT TYPE	: (Computer Generated)
A.T.A.	: (ATC Strip & Clock)
ORIGIN	: (ATC Strip)
A.T.D.	: (ATC Strip & Clock)
DESTINATION	: (ATC Strip)
FLIGHT TYPE	: (ATC Strip)
FLIGHT RULES	: (ATC Strip)
LANDING FEE AMOUNT	: (Computer Generated)
REMARKS	: (As Necessary)

All computer generated information is created from the database stored on the hard-dirve. If the system cannot find the information on the hard-drive, the user will be prompted to enter the information. All computer generated information can be corrected if it is incorrect.

3.2.1.2.2.2 UPDATE INFORMATION ON AN EXISTING MOVEMENT.

If a movement has occurred, but the information on it is not complete, the user will make this selection to update it with current information. The user will then be asked to enter the registration number and the most current incomplete movement for that registration will be brought to the screen. The user will then update the movement with new information. Omce the information is entered, the user will be prompted to say it is correct. If the information is incorrect, the user should update it.. If the information is correct, it will be stored on the hard-drive. The system will then return to the system execution menu.

NOTE: If this option is selected and the registration is not found on the hard drive in the file containing incomplete information for movements, the system will display a message stating that there is not a movement containing incomplete information for that registration on the hard-drive and that the user should go to the "ENTER INFORMATION ON NEW MOVEMENT" section to enter the information for that movement.

3.2.1.2.2.3 PRINT OUT A REPORT. If the user selects this option in the regular execution menu the user will be given a list of reports to choose from. These include :

1.) Movement report
2.) Landing and Parking fee voucher
3.) Departure/Arrivals for Company/Aircraft.

3.2.1.2.2.3.1 MOVEMENT REPORT. If this option is selected, the user will be asked if the system should generate a daily, monthly, or yearly movement report. Corresponding to the users request the system would then generate that report for the user. However, due to the size of the hard-drive, the user may be required to acquire the backup floppy disks for previous months if the user desires a report printed out for any length of time longer than that which the hard-drive is capable of holding. At the conclusion of this function the system will return to the Print out report section menu.

3.2.1.2.2.3.2 LANDING AND PARKING FEE VOUCHER. If this option is selected, the system will prompt the user to learn if a cash or credit voucher is being generated. If a cash voucher is being created, the system will ask the user which registration the voucher is being generated for. If a credit voucher is being created, the system will ask the user which company the voucher is being generated for. According to the input the system will print out the corresponding voucher. Upon completion of the vouvher generation, the system will return to the print out report section menu.

3.2.1.2.2.3.3 DEPARTURES/ARRIVALS FOR COMPANY/AIRCRAFT. If this option is selected, the system will ask the user which company and which aircraft the user would like the report for, and for what length of time the report should be genereated for. The system would then gener-

ate a listing of all movements for the particular aircraft over the specified length of time. At the conclusion of this function the system will return to the print out report section menu.

3.2.1.2.3 SYSTEM MANAGER'S MENU.

The SYSTEM MANAGER'S MENU is a hidden option from the main menu which is only accessable from the system manager's account. It will require an additional password before the options are displayed. Only the system manager shall know this second password. If the wrong password is entered, a message shall be placed in the error log file and the system shall automatically log the user out. If the correct password is entered, it will allow the system manager to define a password for a new user, change a password for an existing user, or delete a user account. The system manager will also be able to change the password required to enter the system manager's menu from this function.

3.2.1.2.4 LOGOFF THE SYSTEM. If this function of the program is selected, the system will return to the logon screen. No further system function will be available until a user logs onto the system.

3.2.2 SYSTEM CAPABILITY RELATIONSHIPS. This proposed system is being designed to help carry out administrative tasks of ATC units. It is capable of handling at least 1000 movements a day with an estimated average of 30,000 movements a month . It is capable of updating each individual movement. It is capable of generating reports for a movement or a set of movements (a set being defined by the user). A single movement is defined as one take-off and one landing.

3.2.3 EXTERNAL INTERFACE REQUIREMENTS. As there is no current system in this field, all future systems will be designed to interface with this system and any updated version subsequently created.

3.2.4 PHYSICAL CHARACTERISTICS.

This software will run on an IBM compatable computer with minimums of a 80 megabyte hard-drive and 4 Megabytes of memory, and a printer. The two proposed systems are both acceptable for running the software system. The proposed systems will range in price from $2500-3500.

Proposed system #1	Proposed system #2
20 Mhz AT	23 Mhz 386SX
80286 Processor	80386SX Processor
1.44 Mb 3.5" drive	1.44 Mb 3.5" drive
Monochrome Monitor	Monochrome Monitor
80 Mb Hard-Drive	80 Mb Hard-Drive
4 Mb RAM	4 Mb RAM
Printer	Printer

The proposed systems could be upgraded if more money is willing to be spent on items such as more powerful disk-drives, faster processors, or color monitors if the program is to be color-coated.

The system is required to have an ADA compiler. The recommended ones are an IBM A370 ADA compiler or the Meridian ADA Compiler. Since Ada strictly adheres to the DOD-STD 1815A, the source code can be ported between ADA compilers. It then need only be recompiled to create executable code for the new system. The system shall be run under DOS 3.3 or DOS 4.0.

3.3 INTERNAL INTERFACES.

The package Movement_Fucntions contains the fucntions and procedures used for entering and updating individual movements, The functions from Movement_Functions (New_Movement, Update_Movement, Display_Movement, Info_Correct) all use the same record, One_Move. This record is used while any one record is being processed. This record is passed as needed while it is created, modified, or displayed. New_Movement is called when information on a new movement is being entered. Update_Movement is called when more information is needed for a movement or the information in a movement is being updated. Info_Correct is called when the information in the movement is being checked for accuracy. Display_Movement is called when a movement needs to be displayed.

The package Database_Control contains the functions and procedures used for manipulation of the database. The database is a summary file of information used by the system to help eliminate the assistant's repeated search of tables for generating specific information for a movement. The functions of Database_Control are Hash, Search_File, and Transfer_Info. Hash is called when the computer is searching the database for information on a particular registration number. Search_File is used to determine if information for the registration is in the database. Transfer_Info is called when a registration is found in the database and the information about it must be transfered.

The package File_Search contains the functions and procedures used for locating an individual movement in the transaction files of the system. The transaction files contain the movements with incomplete data or those which have been completed on the current day. The functions from File_Search (Hash, Search_for_Movement, Transfer_Movement, Save_Movement) use the record One_Move. Function Hash is used to determine the location of a movement in the transaction files if it exists. Search_for_Movement determines if the movement is actually in the file. Transfer_Movement is called to manipulate the movement between package File_Search and package Movement_Functions. Save_Movement saves movements to the transaction files.

The package Get_File_Name is used to convert a date into the name of the file containing information for the time desired. Get_File_Name uses the functions Get_Date and Convert_To_File_Name. The function Get_Date is used to get the month and year of the file of information which is to be handled. The values the function returns are stored in a record of type DATE. This information is passed into the function Convert_to_File_Name which passes back the file name in the variable File_Name.

The package Backup_Restore contains the functions used for backing up and restoring files of movements from the hard-drive. The package Get_File_Name is used to determine the file to be handled. The variable File_Name is passed into the function Restore_File or the function Backup_file, depeding upon which option was selected.

The package Movement_Report contains the functions used to generate reports about the traffic flow of the airport. The package Get_File_Name is used to determine the files used in the report. The variable File_Name is passed into the function Print_Report which generated a report to the printer.

The package Fee_Vouchers contains the functions used in generating the landing fee and parking fee vouchers. The function Cash_or_Credit is used to determine if the voucher is for a single aircraft or for a company. If Cash_or_Credit returns cash, a movement number is entered by the user and is passed into the Print_Voucher function. If Cash_or_Credit returns credit, a company must be entered by the user and it is passed into the Print_Voucher function. Print_Voucher will then generate a voucher containing the landing and parking fees for the aircraft(s).

The package MENUS contains the functions used for the menus of the system. Each different menu has a seperate function. Logon is used to logon to the system. Main_Menu generate the main menu. Fucntion B_R_Menu is used for the Backup/Restore menu. Execute_Menu is used for the execution menu.

3.4 DATA ELEMENT REQUIREMENTS

A MOVEMENT data structure is used for the assignment of movements for incoming and outgoing aircraft.

```
package Movement_Functions is

  type MOVEMENT is limited private;

  procedure New_Movement (One_Move :
                          in out MOVEMENT);
  -- This procedure creates a new movement

  procedure Update_Movement (One_Move :
                          in out MOVEMENT);
  -- This procedure allows a movement to
  -- be updated

  function Info_Correct (Ok : out Boolean;
             One_Move : in out MOVEMENT);
  -- This function determines if the
  -- information in the movement is OK

  procedure display_movement(new_move :
                                in MOVEMENT);
  -- This procedure displays a record of
  -- type MOVEMENT
  limited private
    type MOVEMENT is
      record
        Date          : STRING (1..6);
        Move_Num      : FLOAT;
        Registration  : STRING (1..7);
        Company       : STRING (1..3);
        Flight_Num    : STRING (1..5);
        Type          : STRING (1..7);
        ATA           : STRING (1..8);
        Origin        : STRING (1..4);
        ATD           : STRING (1..8);
        Destination   : STRING (1..4);
        Flight_Type   : (S, N, G, F, R);
        Flight_Rules  : (I, V, S);
        Landing_Fee   : FLOAT;
        Users_ID      : STRING (1..8);
        Remarks       : STRING (1..25);
      end record;

end Movement_Functions;

package Database_Control is

  type MOVEMENT is limited private;

  function Hash (Position : out INTEGER
             One_Move : in out MOVEMENT);
  -- This function determines where to
  -- search in the database for
  -- information on an aircraft
```

```
   procedure Search_File (Position : in
     INTEGER;  One_Move : in out MOVEMENT);
   -- This procedure searchs for
   -- information in the databases

   procedure Transfer_Info (Position : in
     INTEGER;  One_Move : in out MOVEMENT);
   -- This procedure transfers information
   -- about a movement if it is found in
   -- the database

   limited private
     type MOVEMENT is
       record
         Date          : STRING (1..6);
         Move_Num      : FLOAT;
         Registration  : STRING (1..7);
         Company       : STRING (1..3);
         Flight_Num    : STRING (1..5);
         Type          : STRING (1..7);
         ATA           : STRING (1..8);
         Origin        : STRING (1..4);
         ATD           : STRING (1..8);
         Destination   : STRING (1..4);
         Flight_Type   : (S, N, G, F, R);
         Flight_Rules  : (I, V, S);
         Landing_Fee   : FLOAT;
         Users_ID      : STRING (1..8);
         Remarks       : STRING (1..25);
       end record;

end DataBase_Control;

package FILE_SEARCH is

   type MOVEMENT is limited private;

   function Hash (Position : out INTEGER
                 One_Move : in out MOVEMENT);
   -- This function determines where to
   -- search in the transaction fileds for
   -- information on a movement

   procedure Search_for_Movement (Position
     : in INTEGER;  One_Move :
                      in out MOVEMENT);
   -- This procedure searchs for movements
   -- in the transaction files

   procedure Transfer_Movement (Position :
      in INTEGER; One_Move :
                     in out MOVEMENT);
   -- This procedure transfers information
   -- about a movement form the transaction
   -- file to the main system

   procedure Save_Movement (One_Move :
                            in out MOVEMENT);
   -- This procedure saves movements to the
   -- transaction files

   limited private
     type MOVEMENT is
       record
         Date          : STRING (1..6);
         Move_Num      : FLOAT;
         Registration  : STRING (1..7);
         Company       : STRING (1..3);
         Flight_Num    : STRING (1..5);
         Type          : STRING (1..7);
         ATA           : STRING (1..8);
         Origin        : STRING (1..4);
         ATD           : STRING (1..8);
         Destination   : STRING (1..4);
         Flight_Type   : (S, N, G, F, R);
         Flight_Rules  : (I, V, S);
         Landing_Fee   : FLOAT;
         Users_ID      : STRING (1..8);
         Remarks       : STRING (1..25);
       end record;

end FILE_SEARCH;

package Get_File_Name is

   type Move_file is private;

   function Get_Date (FileDate :
                      out STRING(1..8) );
   -- This function asks the user which
   -- month and year to manipulate

   function Convert_to_File_Name
     (FileDate : in STRING (1..8);
      File_Name : out Move_file);
   -- This function converts the date given
   -- by user to a file name

   private
     type Move_File is FILE;

end Get_File_Name;

package Backup_Restore is

   with Get_File_Name;

   type Move_file is private;

   function Restore (File_Name :
                     in Move_File );
   -- This function restores the file named
   -- by user to the hard-drive

   function Backup (File_Name :
                    in Move_File);
   -- This function backs up a file from
   -- the hard-drive to floppy

   private
     Move_file is FILE;

end Backup_Restore;
```

```
package MOVEMENT_REPORT is

  with File_Search;Get_File_Name;

  type Move_file is private;

  function Print_Report (File_Name :
                         in Move_File);
  -- This function prints a movement
  -- summary using search function from
  -- package File_Search

  private
    type Move_file is FILE;

end MOVEMENT_REPORT;

package FEE_VOUCHERS is

  with File_Search;

  type COMPANY is private;
  type MOVE_NUMBER is private;

  function Cash_or_Credit (Cash : out
                           BOOLEAN);
  -- Asks the user if the voucher is to be
  -- cash or credit.  True means Cash.
  -- False means Credit.

  function Get_Move_Number(Move_No : out
                           MOVE_NUMBER);
  -- This function will ask the user which
  -- movement number to print out the
  -- landing fee voucher is for

  function Get_Company (Co_Name :
                        out COMPANY);
  -- This function prompts the user for
  -- the company name that a credit
  -- voucher is desired for

  function Print_Voucher (File_Name :
                          in Move_File);
  -- This function prints a Landing Fee
  -- Voucher using search function from
  -- package File_Search

  private
    type MOVE_NUMBER is STRING(1..8);
    type COMPANY is STRING(1..3);
    type Move_File is FILE;

end FEE_VOUCHER;


package MENUS is

  type SELECTION is limited private;

  procedure Logon (Users : in Employees);
  -- This procedure controls the logon
  -- function of the system

  function Main_Menu (Main_Select :
                      out SELECTION);
  -- This function displays and gets the
  -- selection from the main menu

  function B_R_Menu (B_R_Select :
                     out SELECTION);
  -- This function displays and gets the
  -- selection of backup and restore menu

  function Execute_Menu (Exec_Select :
                         out SELECTION);
  -- This function display and gets the
  -- selection of the execution menu

  function Report_Menu (Report_Select :
                        out SELECTION);
  -- This function displays and gets the
  -- selection of the report menu

  limited private
    type SELECTION is INTEGER;

end MENU;
```

4. NOTES.

4.1 REFERENCES.

1. D. T. Ross, J. B. Goodenough, and C. A. Irving, "Software Engineering: Processes, Principles, and Goals" Computer, May 1975.

4.2 BIOGRAPHIES.

Lee Lahti is a Sophomore at Embry-Riddle Aeronautical University in the Aviation Computer Science Degree program. He will graduate in the Spring of 1992. Upon graduation, Lee will be entering the Air Force as a Second Lieutenant.

Amy Lipscomb is a Junior at Embry-Riddle Aeronautical University in the Aviation Computer Science Degree program. She will graduate in the Spring of 1991. Upon graduation, Amy plans to enter the work force.

Terence Every is a Senior at Embry-Riddle Aeronautical University. He will be graduating in the Fall of 1989. Upon graduation, Terence will receive a commission as a Second Lieutenant in the Air Force.

William Longshore is a Senior at Embry-Riddle Aeronautical University. He will be graduating in the Fall of 1989. Upon graduation, William will be persuing his Master's Degree at Embry-Riddle.

All of the author's can be reached through the following address:

Computer Science Department
Embry-Riddle Aeronautical University
Daytona Beach, FL 32114

4.3 GRATUITIES.

We would like to thank Dr. Jagdish C. Agrawal for all of the time and effort he has placed into our work. Without his guidance, this paper would not have been possible.

AUTOMATED ADA VERSUS HAND WRITTEN ADA CODE

Mallela Uma Devi

The University of Mississippi

Abstract

This paper examines ADA code which is generated automatically from user specifications using the CASE tool EPOS. It then compares that code with hand written code generated from the same set of specifications.

The EPOS system was developed to aid software engineering like requirement analysis, system design and implementation, coding, testing and debugging can be done by EPOS.

This paper describes the transition from a detailed software specification to automated ADA code. It would be done using the "single-source-multi-step process". That is, all the design specifications of a system would be created using EPOS. This would then be translated into ADA code in a finite number of steps. This objective would be achieved by using the information available from the design specifications. Again using the same set of specifications, hand written code is generated. Then a comparison of the two types of code is done.

Introduction :

Engineered Software and the principles of Software Engineering software engineering is, in the 1990's a necessary condition for the development of large software projects. More and more new methodologies are being developed to aid software engineering. And now CASE has come into picture. Many CASE tools have been developed and some are in the process of being developed.

Ada has always been an aide to the software engineering development process, mainly because of its strong support of the software engineering principles of modularity, data abstraction and information hiding. This paper compares Ada code which had been generated from a code generator and hand written Ada code generated from the same set of design specifications.

The very first section is an introductory example showing the process of generating automated code using the CASE tool EPOS[1].

The next section deals with the specifications and the generation of automated code using these specifications.

The third section compares both the codes and finally the tradeoffs are discussed.

Introductory example :

Suppose our problem is to sum up the first 100 natural numbers. Since no input is needed there is no input part. We need a module to sum up, a module to check if 100 numbers have been summed up and finally a module to output the result to the user. To output we need to use the Ada package TEXT_IO. This module would only refer to TEXT_IO so we can import it rather than describing it in the module.

The following EPOS specifications would generate the required code :

```
#new
ACTION MODULE MA.
DECOMPOSITION : (/ SUM/).
IMPORT-ACTION: TEXT_IO.
EXPORT-ACTION: SUM.
ACTIONEND
###
DECOMPOSITION :
WHILE NUMBER-LESS DO
ADD-UP
OD;
OUTPUT-IT.
ACTIONEND
###
#new
CONDITION NUMBER-LESS.
DECOMPOSITION : NUM <= 100.
CONDITIONEND
###
#new
ACTION ADD-UP.
CODE : "
SUM := SUM + NUM;
NUM := NUM + 1;
".
INPUT : SUM,NUM.
OUTPUT : SUM.
ACTIONEND
###
#new
ACTION OUTPUT-IT.
CODE: "
put_line( Sum is );
put(SUM);
new_line;
".
INPUT : INT-IO.
ACTIONEND
```

```
###
#new
DATA NUM.
TYPE : FIXED.
INITIAL : 1.
DATAEND
###
#CHANGE
ACTION PROCEDURE SUM .
DECOMPOSITION :
   WHILE NUMBER-LESS DO
      ADD-UP
   OD ;
   OUTPUT-IT .
ACTIONEND
#CHANGE
ACTION ADD-UP .
CODE :
"
SUM1:= SUM1+ NUM;
NUM := NUM + 1;
" .


INPUT : SUM1,
        NUM .
OUTPUT : SUM1.
ACTIONEND
#CHANGE
ACTION OUTPUT-IT .
CODE :
"
put_line( Sum is );
put(SUM1);
new_line;
" .
INPUT : INT-IO .
ACTIONEND
#new
DATA SUM1.
TYPE : FIXED.
INITIAL :0.
DATAEND
###
#new
DATA INT-IO.
CODE :"
package INT_IO is new
TEXT_IO.INTEGER_IO(INTEGER);
use INT_IO;
".
DATAEND
###
```

When fed to the automatic code generation package of EPOS the following code would be generated :

```
-- CODE-TRANSFORMATION FILE: EXA.ADA
--
-- 2 ->
-- TARGET-LANGUAGE: ADA
-- PROJECT: PAPER


with TEXT_IO; use TEXT_IO;
procedure SUM is
  NUM : INTEGER := 1;
  SUM1 : INTEGER := 0;
  package INT_IO is new
  TEXT_IO.INTEGER_IO(INTEGER);
  use INT_IO;
begin  -- SUM
  while
    (NUM <= 100)
  loop
    SUM1:= SUM1+ NUM;
    NUM := NUM + 1;
  end loop;
  put_line( Sum is );
  put(SUM1);
  new_line;
end SUM;
```

System specification and code generation :

For one of the software engineering courses at the University of Mississippi we had to design a system which will simulate the operation of a MC68000 chip. The system would accept a 23 instruction subset of the instruction set of MC68000. Instructions like MOVE, ADD, CMP, CLR, BRANCH, JMP, JSR, RTS were in the subset. At the end of sequence of instructions the contents of all the data registers and address registers (MC68000 calls them as D0-D7 and A0-A7) are dumped out. Also a memory dump is executed. This shows how the set of instructions were executed.

Handwritten Ada code was developed by the students (including myself) from a set of carefully written specifications as a project for the above mentioned software engineering course.

To generate the automated Ada code the following steps were taken.
The main program SIMULATOR was divided into modules and each module was further divided into submodules. Fig.1 broadly shows the hierarchial chart used for the module division. Now, each of the terminal modules shown in the figure is further divided into sub modules. This is done until a stage which I call atomic module is

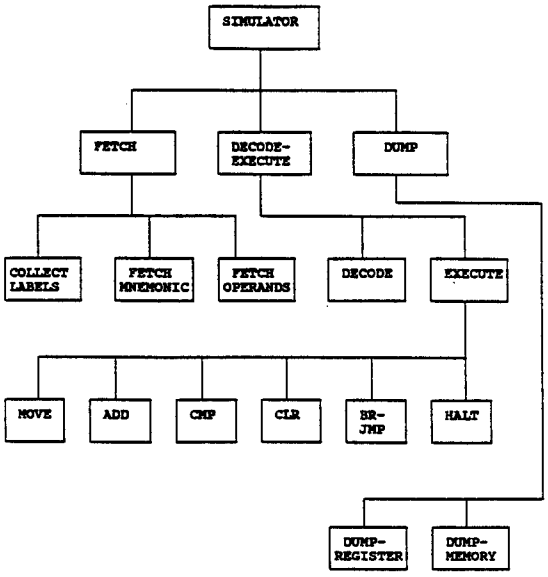


Figure 1.

reached. At this stage we do not wish to further divide the module. We can put in simple pieces of direct code. In the introductory example ADD-UP would be a atomic module. Now all these modules are entered into the EPOS-specification data base according to the syntactical rules. Once this database is created, the automatic code generation is done using the code generation routine present in EPOS.

Comparison of handwritten versus automated Ada code :

As every Ada programmer would know and have experienced, producing handwritten Ada code is quite a tedious and time consuming process. Further more it can be error prone. The following errors were the most that we encountered : syntax errors, errors due to visibility rules of Ada, that is which unit would be compiled after which and a few logical errors.

For the above specifications it took nearly four weeks and the hard work of five students to do the coding, compilation and debugging and finally execution of the code. Out of this about 50% of the time spent in coding, 30% in compilation and debugging compilation errors, 10% debugging run time errors and execution and finally 10% in documentation of the whole project.

To produce automated code it took at least 5% less time than handwritten code. On an average this project had about 500 lines of code. To save 5% time on a 500 line code is certainly a considerable amount of time. Also this CASE tool does program documentation automatically, thus the 10% of the time spent on documentation is totally saved. Furthermore, out of the 20% spent on compilation and debugging nearly 50% was spent on debugging syntax errors. By using automated code the probability of syntax errors is nearly zero. Also the modules are made in a top down approach and EPOS takes care of all the visibility rules, so compilation sequence errors are also greatly reduced. With more experience, more time could be saved in coding.

Tradeoffs :

The advantages of automated code are many the disadvantages but a few. Of course saving time is the most important advantage. Also a lot of man power is saved. The above project took the man power of 5 students for the hand written code but the automated code took much less than that. The saving of compilation and debugging time would mean the saving of computer usage, in other words CPU time. Also automatic documentation would lead to a documentation which could be used further by anyone to further develop the project.

Well every thing has a disadvantage too. There is nothing automatic about creating the specification set. It must be done by hand very carefully. So, whether the code is automatically generated or hand generated the same amount of brain and man power is required for either method at the specification stage. Also we cannot completely forget the language Ada and its rules. To produce automated Ada code we still need to know the grammar of the Ada language. For a person not knowing Ada at all it would make no difference if he generated hand written or automated code. A certain amount of Ada knowledge is also required. Also EPOS is also a rather large and complicated software package which must be implemented, understood and mastered prior to its effective use. It has a certain set of rules to be observed and a certain grammar to be followed. But learning these rules and grammar is not as difficult as learning Ada.

Conclusion :

In this age of developing faster computers we all want to have things which would be done faster and better, saving us both time and money. Automated code generation would certainly save a lot of this especially on large projects which require years to complete.

EPOS requires about 20M of hard drive and the Ada compiler about 15M of the hard disk. Also EPOS requires an expensive graphical interface and a math co-processor. The Ada compiler needs about 4M of memory and EPOS requires 640K. Certainly all these are high on costs.

Though CASE tools are very expensive and high on resources but their use is justified by the amount of saving in time.

[1] EPOS is a registered product of SPS Software Products and Services, Inc.



Ms.Mallela Uma devi is currently a graduate student working towards a degree in Computer Science at the University of Mississippi. Earlier she graduated at Delhi University, India with a Bachelors in Mathematical Statistics and pursued a Masters degree in Mathematics at the Indian Institute of Technology. She has always been in the top 1% in all classes. She is also a member of the American Mathematical Society.
Mailing address : P.O.Box 9405, Univ of Mississippi, MS 38677

ADA AND THE SOFTWARE ENGINEERING LIFE CYCLE

Audrey Lynette Pope and Yvonne Denise Couevas

The University of Mississippi

Abstract

This paper outlines the efforts of a software engineering student team using the Hierarchy Input Process Output (HIPO) model and the Ada programming language. The team's project was a simulation of the working environment of the Motorola 68000 microprocessor. The rationale behind the choice of HIPO and structured design methodology is explained. Ada's support of modularity is featured in the discussion of the overall conception, design, and implementation of the simulation project. Highlighted are the efforts to achieve reusability through the use of packages in Ada. The use of packages allowed for the reuse of code throughout the life cycle of development; thus, allowing individual team members the capability to work on interdependent code segments simultaneously. Also included are the pitfalls of both Ada and the HIPO model.

Introduction

Software engineering provides a process that insures the orderly development and implementation of large computer programs. A software engineering student team (Group 1) at the University of Mississippi used this process in the development of a large software project which was a simulation of the operation of the Motorola 68000 microprocessor. This project was developed using the Hierarchy plus Input-Process-Output (HIPO) model and written in the Ada programming language on an AMDAHL 470/V8 using the ALSYS IBM 370 version 4.1.2 compiler. An explanation of the project, the rationale behind the design decisions, the advantages and disadvantages of using Ada, and the software engineering life cycle are the topics. Highlighted are the group's efforts to achieve reusability through the use of Ada packages.

First, a description of the project. The assignment was given to an undergraduate senior software engineering class to both teach the students about software engineering and give them the opportunity to work on a large project as a group. The goal of the project was to write a program in Ada to simulate the operation of the Motorola 68000 microprocessor. The simulator would accept as input an assembly language program consisting of a subset of the Motorola 68000 instruction set. The students had to develop a program which would convert this assembly language program into object code, execute that object code, and perform a memory dump after execution.

As preparation for the project, the students spent several classes learning about Ada and the Software Engineering Life Cycle. Several team structures and design methodologies were presented to the students. The students also participated in group dynamic development activities. Finally, the class was divided into groups. This paper reports on the efforts of Group 1.

Team Structure

After much consideration, Group 1 chose as its team structure the Chief Programmer Team approach. One reason for this choice was that the responsibilities for each member were well defined; thus, each member knew explicitly what his responsibilities were. Another reason for the decision was that each member could concentrate his time and efforts on one area rather than spending a small amount of time in each area. Since much of the material was new, a lot of research had to be done in each area. As a general rule, members were responsible for researching

their individual areas and reporting their findings to the group.

The positions of the group's structure can be divided into two categories: management and programming. The management category consisted of the chief programmer, assistant to the chief, and secretary. The programming category consisted of the head programmer, programmer, and tester. Due to the small size of the group, some members of Group 1 held multiple positions. For instance, the chief programmer was also the tester, and the assistant to the chief was also the secretary. Since extensive coding was required, there were two programmers.

As stated earlier, each member was given specific responsibilities. The chief programmer was responsible for scheduling the group's meetings, formulating the agendas, and leading the meetings. Other responsibilities included keeping the program on schedule, making final decisions in controversial situations, and acting as the user liaison. The assistant to the chief programmer was responsible for helping research the available design methodologies, testing procedures, and documentation formats, and assisting with the testing procedures. In addition to keeping minutes, the secretary was responsible for maintaining the group journal, reserving the meeting room, and handling the typing tasks of the group.

In the programming division, the head programmer guided the coding of the system. He assigned tasks to the individual programmers, contributed to the writing of the code, and coordinated research of Ada and other program related issues.
The two programmers coded the modules assigned to them by the head programmer, and they were responsible for the debugging of the program and the return of completed modules to the head programmer. The tester designed the testing procedure to be used by the group,and with the help of the assistant to chief, she tested the program.

Software Engineering Life Cycle

In attempting such a large task, some type of direction is needed. The software life cycle provided direction for Group 1 in the development of the Motorola 68000 simulator. Due to time restrictions, every step of the life cycle could not be performed by the group. Therefore, the group used only the steps deemed essential to the completion of the project. These steps were System Requirements, Detailed Design, Code and Debug, and Test and Preoperations. A formal review of each step in the life cycle is a major part of the life cycle process; however, due to time constraints, formal reviews were conducted only after the System Requirements and Detailed Design steps. The group conducted informal reviews throughout the life of the project.

System Requirements Specification

After consulting the user, the team prepared a System Requirements Specification document. In this document, the group described the scope of the project as a system that simulates the Motorola 68000 environment. The system accepts an assembly language program consisting of a subset of the Motorola 68000 instruction set and converts this program into object code. (A subset of the Motorola 68000 instruction set was chosen due to the time constraint of one academic semester.) This object code is then executed. At the termination of the program, the simulator performs a memory dump.

A lot of time and deliberation was spent on the Interfaces section of the System Requirements Specification document. In this section, the students selected four divisions of interfaces to be defined for the simulator. The hardware interface defined the type of computer system on which the simulator would be implemented. The simulator was designed for but not bound to the IBM 370 architecture as implemented on an AMDAHL 470/V8. The simulator interfaced with a disk drive and supported four megabytes (4M) of memory and a 32-bit word size. The team decided that due to developmental time constraints and the particular subset of instructions to be implemented, interrupts would not be specifically handled by the simulator. The software interface defined the programming language, operating system, and compiler used in the implementation of the simulator. As stated earlier, it was

written in the Ada programming language. It operated under the IBM VM/CMS operating system and was compiled using the ALSYS IBM 370 Ada compiler, version 4.1.2. The human interface was of course the user. The user must know the subset of the Motorola 68000 assembly language employed by the simulator. The user must know how to construct a syntactically correct 68000 assembly language program. He must be able to use an editor to create a text file. The packaging interface defined the documents which would follow the System Requirements Specification. A design specification, test design, and user's manual were scheduled to follow. Normally, the packaging interface only refers to those documents to be delivered to the user. However, in this case, the user was also the instructor who required that the extra documentation be given to the user.

In the System Requirements document Group 1 decided to divide the project into two separate programs, a conversion program and a load and execute program. The conversion program accepts as input an assembly program from a text file stored on disk. This program is converted to object code, and this object code along with supervisory instructions are stored in a separate intermediate file on disk. The load and execute program accepts this intermediate file from disk, interprets the supervisory instructions, and executes the object code by simulating the operations of a 68000 chip. After execution, it produces a memory dump of the program and data areas of the simulated memory.

Detailed Design

In development of the detailed design, the group used the Hierarchy plus Input-Process-Output (HIPO) design methodology. HIPO is "a technique for use in the top-down design of systems."[1] The HIPO model consists of two parts: hierarchy charts and input-process-output charts. The hierarchy charts are graphical representations of the functions and their respective subfunctions used in the program. The input-process-output (IPO) charts are used to describe the input, process, and output of each function.[1]

The group's initial reasons for choosing HIPO was that there was ample information and complete documentation available on the methodology, and it's highly structured methodology made it easy to learn in the limited time available. Another reason was that it was not only useful for the design documentation, but also for expressing the requirements and specifications of the system. HIPO has many advantages which make it a beneficial tool in the design of software. One advantage of input-process-output charts is that they provide for easy transition from the detailed design to actual code. In a good design, at the lowest level of the hierarchy, the process section of the IPO charts contain actual pseudocode. Another advantage is that since the hierarchy charts and the IPO charts can be developed concurrently, the subfunctions introduced in the IPO charts can be added to the hierarchy charts at the time of their development. Thus the hierarchy chart is a dynamic entity which directly reflects the current functions and subfunctions of the system. A current hierarchy chart provides a means of visualizing the progress of the developing system at any time. The third advantage is that the division of the functions into input, process, and output allows the designers to know the exact input and output of each function at a very early stage. This leads to an overall understanding of the flow of the data long before implementation.

Coding and Debugging

The Coding and Debugging phase of the life cycle was intended to consist of coding the individual modules of the system as developed in the IPO charts. To accomplish this, the modules at the lower level of the hierarchy chart were distributed to the programmers. Knowing only the given input and desired output, each programmer was responsible for making his module perform its appropriate function. The group developed a plan to guide the coding, debugging and testing of the individual modules. In this plan, it was specified that after a programmer coded his module, he was to test it with some sample input. Once he verified the

correctness of the module, it was given to a second programmer. The second programmer was responsible for testing the module with his own sample input. If the second programmer found any errors, he returned the module to the original programmer for correction. If no errors were found, the original programmer submitted the module to the head programmer for incorporation into the overall system. The head programmer was responsible for writing the code for the integrating main module, binding the individual modules, and making sure the flow of control of the overall project was correct. Once all the modules were tested and bound, integration testing could be performed. Due to time constraints imposed on the group, one academic semester, only a few modules in the conversion process were completed.

Testing

In the Testing step of the life cycle, Group 1 developed a test design for the simulator. In this design, the group outlined the procedures that would be followed in the testing stage. The first phase was the testing of the individual modules as explained above. The second phase involved checking the flow of control of the system and the functionality of each module. The flow of control was tested by implementing the system using stub modules. Once this was completed, the modules were tested by adding the body of a single module to the stub and retesting the stub until all modules were incorporated into the system. Some of the checkpoints in this phase were conversion of individual mnemonics to their correct opcode, conversion of operands, conversion of data, and correct output of loader module. The third phase of testing was to check how well the system handled errors in the assembly language program. It was assumed that the assembly language program was syntactically correct, but there was a possibility for logic errors. Testing also included checking for problems which might occur in the system when the assembly language program included a mnemonic which was not in the original subset, invalid operands, or a JMP statement with a label, but no corresponding label. All the phases of testing were not performed on the simulator because some of the modules were not completed.

Modularity and Reusability

In addition to the guidance provided by the software life cycle, modularity was a tool which provided assistance in the development of this large and complex software project. In the design of the project, the HIPO methodology was useful for insuring modularity of the system. The HIPO methodology insures modularity through its definition, in that, in the hierarchy chart, each function is broken into subfunctions. Then, through an iterative process, each subfunction can be divided further until the subfunctions, as represented by pseudocode in the IPO charts, are small enough to be a manageable piece of code. As a programming language, Ada in its packages, generics, and separate compilation directly supports modularity. In the hierarchy chart each function at the lowest level corresponds directly to an Ada module; thus, the modules in the Ada program can be written directly from the IPO charts for the functions. (See figures 1 - 3.)

The abstraction mechanism which Ada provides directly supports modularity through separate compilation. This allows interdependent modules to be coded and tested separately with sample input to assure that they are functionally correct. Because the dependency relationships revealed in the HIPO decomposition, programmers were able to code any function at the same level on the hierarchy tree simultaneously. The ability of Ada to support this simultaneous coding, testing, and debugging of same-level functions enhanced programmer productivity. It also strengthened the relationship between the design methodology and the programming process.

As part of the assignment, the students were to consider reusability of the code throughout the design. One way in which Group 1 allowed for reusability was by using packages which by definition can be accessed by more than one module. One example of this was the package which converted decimal numbers to binary numbers. In order to support 4M of memory, the programmers decided the system should

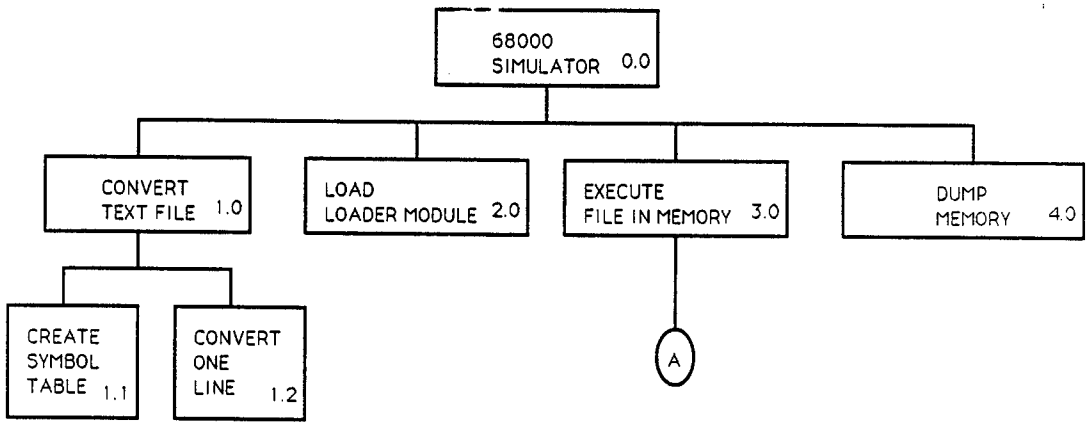


Figure 1. First level Hierarchy chart for the Motorola 68000 Simulator

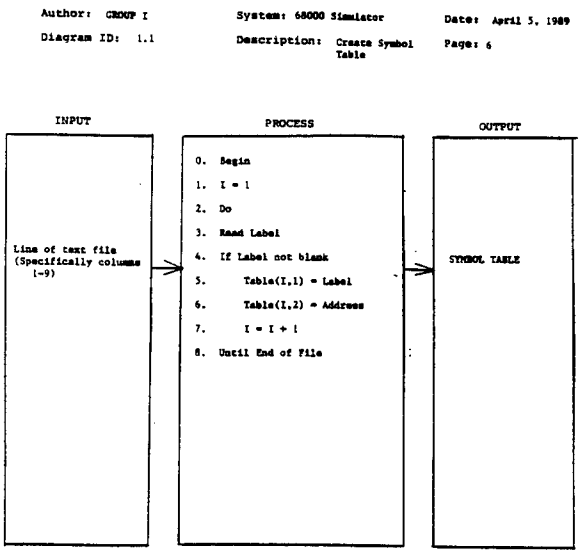


Figure 2. Input-Process-Output Chart corresponding to block 1.1 (create symbol table) in figure 1.

```
with TEXT_IO;
separate (ASSEMBLE)
procedure CREATE_SYMBOL_TABLE(TEXT:  in FILE_TYPE) is

use TEXT_IO;
MAX_SYMBOL : constant INTEGER := 50;
type S_TABLE is
  record
    SYMBOL  : STRING(1..10);
    ADDRESS : INTEGER;
  end record;

LINE_OF_TEXT : STRING(1..40);
LBL : STRING(1..10);
OPR : STRING(1..20);
BLANK :constant STRING(1..10) := "          ";
I : INTEGER := 0;
MEMORY_POS : INTEGER := 0;
PLACE : INTEGER;

package INT_IO is new INTEGER_IO(INTEGER);

begin
  while not END_OF_FILE(TEXT)
    loop
      GET(TEXT,LINE_OF_TEXT);
      LBL := LINE_OF_TEXT(1..10);
      OPR := LINE_OF_TEXT(21..40);
      if LBL /= BLANK then
        I := I + 1;
        SYMBOL_TABLE(I).SYMBOL := LBL;
        SYMBOL_TABLE(I).ADDRESS := MEMORY_POS;
      end if;
      MEMORY_POS := MEMORY_POS + 2 +
                    (2 * DATA_LENGTH(LINE OF TEXT));
      SKIP_LINE(TEXT);
    end loop;
end CREATE_SYMBOL_TABLE;
```

Figure 3. Ada code written to preform the function shown in the Input-Process-Output chart shown in figure 2.

convert the users assembly language program to its decimal equivalent. Thus it was the decimal equivalent of the original input on which th system operated. Therefore, the package to convert from one base to another was called by several modules of the program. Another means of providing for reusability was that the group designed the project with the idea that the subset of the Motorola 68000 instruction set it accepts could later be expanded. With this in mind, the group developed the system in such a way that the addition of more instructions would lead to changes in only a few of the modules. Thus most of the code could be reused with no changes being made. For example, in this program the subset of instructions could be divided into four groups according to the first two bits of the instruction. The four groups were MOVE, AND/ADD, COMPARE, and "OTHERS". Each group corresponded to a function (block) in the hierarchy chart. In order to add a new instruction, using the first two bits of the instruction, one would determine to which instruction group the new instruction corresponds. The new instruction would result in changes being made to only the group instruction function and those below it in the hierarchy chart. Specifically, if another MOVE instruction was added to the subset of instructions, the only module which would have to be modified would be the "Execute_move" module. There is also the possibility that some of the packages could be used by different programs. Thus allowing reusability on an even broader scale.

Disadvantages

Problems can be expected in the life cycle of any large software project. Group 1 faced several problems in the development of the Motorola 68000 simulator. A significant problem was that the students were unfamiliar with the Motorola 68000 architecture, the Ada programming language, and the Hierarchy plus Input-Process-Output methodology. Another problem was that the artificial time constraints of one academic semester was in fact a real time constraint. The students also had to overcome the obstacle of learning to work in a group. Each member had to develop a trust in the other members of the group in order to make the project successful. Being able to communicate one's ideas clearly to the other members of the group also provided a challenge to the students; it was important that each student do his assigned tasks and enlighten the other members of the group on his work.

A disadvantage of using Ada was that the language was so large that it was difficult to learn the subtleties of the language in such a short period of time. Although separate compilation aided modularity, because of the new concept of "compilation dependencies", it provided difficulties for the students; the Ada library management procedures were a new concept which the students had trouble grasping. There were also disadvantages associated with HIPO. Although the HIPO methodology partitioned the system into manageable pieces of code, it generated a substantial amount of paperwork which required the full-time efforts of two group members.

Summary

Using Ada as opposed to another language such as C or Pascal proved to be a good choice because it forced the group to be more organized. The group had to understand how each module interacted with the other modules in order to get them compiled correctly. Having little knowledge of Ada prevented the programmers from violating the design of the project by using private shortcuts. Thus, the integrity of the code was maintained. Ada provided a means of writing readable code; therefore, someone else should be able to understand the code with minimal effort.

In summary, the Software Engineering Life Cycle provides a good process for assisting in the development of large software projects. In retrospect, the members of Group 1 feel that the decision to use ADA and HIPO was a valid one. Both the language and the design methodology had disadvantages; however, the students gained experience in working as a group, developing a large project, and utilizing new techniques and a new language.

Reference

1. J. F. Stay, "HIPO and integrated program design," IBM Systems Journal, Vol. 15, No. 2, 1976.



Audrey Lynette Pope will be a May 1990 graduate of the University of Mississippi with a Bachelor of Science in Computer Science. A senior from Fulton, Mississippi, she is a national ACM member, and currently serves as vice-chairman of the local student chapter. She has completed cooperative education internship with Federal Express Corporation in Memphis, Tennessee. P. O. Box 1037, University, MS 38677.



Yvonne Denise Couevas from Biloxi, Mississippi is a senior at the University of Mississippi. She will receive her Bachelor of Science in Computer Science in May 1990. She is a member of the national ACM and currently serves as Treasurer of the local student chapter. She currently holds a cooperative education position with the USDA, Southern Experiment Station, Forest Hydrology Laboratory in Oxford, Mississippi. P. O. Box 2947, University, MS 38677.

AN ENVIRONMENT FOR TESTING ADA PROGRAMS IMPLEMENTED IN ADA

Orville R. Weyrich, Jr.

Department of Computer Science and Engineering
Auburn University, Auburn AL 36849

Abstract

ReQUEST is written in Ada and targeted to support the testing of programs in Ada. It is a reimplementation of the Query Utility Environment for Software Testing of Fortran Programs (QUEST/F77). In this paper we discuss the lessons learned from QUEST/F77 and the impact which these lessons and the additional features of Ada have on the design of reQUEST.

History of QUEST

QUEST is an acronym which stands for Query Utility Environment for Software Testing. There are three distinctly different implementations of QUEST, which are denoted as QUEST/F77, QUEST/Ada, and reQUEST. Each are described in the following paragraphs.

QUEST/F77 is the first prototype system, which was developed at Auburn University under the sponsorship of the U.S. Army. It runs on VAX/VMS and is implemented primarily in VAX Pascal. This first prototype was designed to test FORTRAN-77 program units. Previous papers describe QUEST/F77[1, 2, 3].

ReQUEST is a reimplementation of QUEST/F77 which is presently being developed by the author on his personal computer and is unsponsored. It runs on an 80386-based computer under MS-DOS and is implemented in Meridian Ada and Turbo Pascal. It uses the same general algorithms and structure as QUEST/F77, but with improvements made based on the experience with QUEST/F77. ReQUEST is designed to test Ada programs. It is the eventual goal to apply reQUEST to the testing of reQUEST. This paper represents the first published report of reQUEST.

QUEST/Ada is a completely new system which is presently being developed at Auburn University under the sponsorship of NASA. It runs on VAX/VMS and is implemented primarily in C. The algorithms used in QUEST/Ada for test case generation are completely different from those used in QUEST/F77 and reQUEST. This version of QUEST is designed to test programs written in Ada.

The discussion in this paper refers to QUEST/F77 and/or reQUEST, but not QUEST/Ada. Except where specifically noted, the discussion in this paper applies to both QUEST/F77 and reQUEST.

Software Testing

Since it is not feasible to exhaustively test a real-world system, we must be content with testing some subset of the system's input domain. We must exercise each relevant aspect of the system at least once. We must, however, define what aspects are relevant. Various criteria may be used for determining when we have tested all the relevant aspects of the program under test. We call such a criterion a test adequacy criterion.

Some test adequacy criteria suffer from a significant problem: it is possible that some chosen aspect of a program under test might not be executable. In this case it is impossible to achieve complete coverage of the adequacy criterion. This problem can be partially addressed by modifying the adequacy criterion. However, as a result of the halting problem, we may never be sure whether our failure to achieve complete test coverage is due to an inadequacy in our test case generation, or is due to an inherent infeasibility of some aspect of the program under test.

The selection of adequacy criterion, paths and supplementary conditions have been discussed extensively elsewhere, and are not considered further here[2, 4, 5, 6, 7]. The types of adequacy criteria with which we are concerned all have a common characteristic: they lead to a search of the program graph for paths whose execution (when possibly subjected to additional constraints) will fulfill the test adequacy criterion chosen. The discussion in this paper is independent of the particular test adequacy criterion which is used. ReQUEST will be used as a test-bed for additional coverage metrics.

Ada Program Graphs

QUEST/F77 and reQUEST are primarily white-box testing systems, and depend on finding paths through the program under test which satisfy the chosen test adequacy criterion. Processing program graphs is thus an important aspect of the QUEST/F77 and reQUEST algorithms.

A program can be thought of as a directed graph in which there is a 1:1 mapping between the nodes of the graph and the atomic actions of the program. An atomic action is defined as a program component which cannot be meaningfully broken down into smaller constructs. In many cases, but not all, an atomic action corresponds to a "statement" as defined by the programming language syntax.

We consider all statements to be executable. [In FORTRAN, declaration and FORMAT statements are considered to be non-executable statements which provide information to the compiler but do not cause actions at run-time. For our purposes, we treat such statements as atomic actions which do nothing.]

A directed edge in the program graph connects each pair of nodes (n_i, n_j) for which execution of the corresponding atomic action s_i may be followed directly by execution of the corresponding atomic action s_j.

Usually, if atomic action s_j immediately textually follows atomic action s_i, then the program graph has a corresponding edge (n_i, n_j). An exception to this rule is when atomic action s_i is an unconditional transfer (for example GOTO, END LOOP, RAISE, EXIT, or RETURN. Another exception to this rule is when s_j implies a transfer to an END IF, END CASE, or END.

In addition to edges in the program graph due to s_j textually following s_i, certain atomic actions may give rise to branches to atomic actions which do not textually follow. Included in this category are IF, ELSIF, ELSE, WHILE, FOR, END LOOP, WHEN, FORTRAN computed GOTO, FORTRAN I/O statements with END= or ERR= keywords, and any atomic action in Ada which can raise an exception.

It is frequently necessary to be able to assert that all portions of a program component are executed whenever any portion of the program component is executed. This clearly is not true if the program component contains AND THEN or OR ELSE, or if short-circuit Boolean evaluation is used [as required by C and as is allowed as an optional optimization in many other languages].

When AND THEN, OR ELSE, or implicit short-circuit Boolean forms occur, each must be treated as introducing a new conditional transfer atomic action. This is because the intended behavior of such a short-circuit form is to suppress execution of portions of the containing statement. However, for the purposes of forming path predicates, we may treat AND THEN and ELSE IF constructs as normal AND and OR operators, as long as we construct our system to ignore runtime errors raised during the evaluation of irrelevant Boolean conditions. This is not an additional requirement, since runtime errors frequently arise during the search for test data anyway.

In Ada, an exception may be raised at any point during the evaluation of an expression. This means that in effect there is a conditional transfer implicit in the evaluation of each operator. If we assume that the evaluation of the operators proceeds without side-effects and that only one kind of exception can be raised (E.g. NUMERIC_ERROR), then we do not need to concern ourselves with exactly when during the evaluation of an expression an exception arises. However, if function calls may have side effects, then the point at which the exception arises (before or after a given function call) becomes important in defining the state of the program after an exception has been raised. Similarly, if more than one exception could be raised, then the ambiguity in the order of evaluation leads to ambiguity in the exception which is raised.

The simplest answer to the problem of a function with side-effects is to treat each such function call as the start of a new atomic action. Unfortunately, in Ada, any user-defined function may have a side effect, and all of the Ada operators (AND, "+", MOD, etc.) can be overloaded by functions with side-effects. Furthermore, Ada does not completely specify the order of evaluation of the operators in an expression, so even if we chose to start a new atomic action for each operator encountered in an expression, then the proper sequence for the atomic actions would remain ambiguous.

Because of the above problems, we treat functions as side-effect-free in our further discussions, relying on some hypothetical global static analyzer to warn us of any violations of our assumption.

The only answer to the problem of multiple exception alternatives is to require a determinate evaluation order. This can be done either by modifying the language definition (and compilers etc.) or by assuring that the sequencing algorithm used by the compiler of the field release version of the program under test is the same as the sequencing algorithm of the test generation software.

Unfortunately, if each operator in a program is considered as introducing a new program component which may raise one or more exceptions, the branch complexity of the program graph may become intractable. We do not have a complete solution to this problem.

Path Predicates

Associated with each path is a basic path predicate, which is formed by conjoining all the condition nodes traversed along the path, taking into account the side-effects exerted by the processing arcs.

In addition to requiring the path to traverse a specific sequence of nodes and arcs, we may also insist that various other supplementary conditions also be met. This is done, for example, when the condition, decision-condition, or multiple-condition coverage test adequacy criterion is being used. We thus speak of the augmented path predicate, which is formed by conjoining to the basic path predicate any supplementary conditions which must also be met.

Subsequent discussion will not distinguish between the basic path predicate and the augmented path predicate.

We are not concerned with how path predicates are generated in this paper. This is discussed elsewhere[2, 4, 5, 9].

White box testing is complicated by the fact that many of the paths chosen may be infeasible. Some infeasible paths may represent errors, while other infeasible paths may represent redundancies intended to improve the robustness of the program. Still other infeasible paths may be artifacts of the computational properties of the computer hardware, as discussed below.

Paths Including Procedure Calls

The testing of large systems poses special problems: the number of possible paths and the length of the paths requires the solution of more predicates of greater size. The result is that the available computing resources are frequently inadequate. Current software engineering practice discourages large monolithic programs, but this does not solve the problem of testing large systems, since the introduction of subordinate procedures simply adds another dimension to the problem.

The problem of testing programs which contain procedure calls has been previously addressed in two ways: (1) inline expansion, and (2) symbolic substitution.

Inline expansion has the problem that it enlarges the size of the path predicates and increases the number of possible paths to be taken[8]. QUEST/F77 uses inline expansion.

Ramamoorthy describes symbolic substitution (also called procedure substitution) as "generate path constraints bottom-up by first finding constraints for the called routine"[8]. The lowest level routines are tested first, and all the path constraints are stored for later retrieval when a higher-level routine calls the lower level routine. Woods has pointed out a number of problems with symbolic substitution, including that it cannot handle functions passed as parameters and it cannot handle variably dimensioned arrays[4]. In addition, it is not efficient if many input variables need to be fed to the symbolic simplifier (an equality constraint is generated between each formal and actual input parameter; each element of an array parameter leads to a separate equality constraint). Another problem is that many procedures have many more feasible paths through them than are included in the set which adequately tests the procedure. If only the paths used during testing of the procedure are included in the set of paths which may be symbolically substituted, some conditions which follow the procedure call may be erroneously found to be inconsistent.

Weyrich introduced a new method, called procedure elision, for dealing with procedures[2]. A possible criticism of procedure elision is that if there are many different paths through the elided procedure, then the number of discontinuities may require many different starting points. The response to this criticism is that if inline expansion or symbolic substitution are used, many different paths might likewise need to be tried before one is found which gives the desired response.

Solving Path Predicates

After a path and supplementary conditions have been selected, the augmented path predicate must be solved. Solving a path predicate requires that input data be discovered which causes the augmented path predicate to be true.

Not all path predicates can be solved. In some cases, the path may be infeasible, which means that no possible input data can cause the path to be traversed. When a path is infeasible, its corresponding path predicate is said to be inconsistent. Unfortunately, a consequence of the halting problem is that some path predicates may neither be solved nor shown to be inconsistent. Such paths are referred to as indeterminate paths and their corresponding predicates are referred to as indeterminate predicates.

The solving of a path predicate serves two purposes: (1) it demonstrates that the path is feasible, and (2) it provides specific program input data which will actually cause the path to be executed.

Most previous workers have used linear programming to solve the path predicates. This has limited applicability. Clarke has used a simplifier to convert constraints into cannonical form. Clarke suggests that "the conjugate gradient method ... can be applied to solve the [nonlinear] inequalities. The conjugate gradient method requires human interaction ..."[10]. This paper discusses a new method for solving predicates containing nonlinear constraints and complex variables.

Front End Processing

The front end of QUEST contains most of the language-dependent features, and are only briefly discussed in this paper. The QUEST/F77 front end recognizes and processes FORTRAN-77 in a rather ad-hoc manner, striving to pick out selected features in a manner reminiscent of preprocessors such as RATFOR. This results in a number of limitations on the range of FORTRAN language constructs which can be processed by QUEST/F77, and negatively impacts robustness and maintainability. ReQUEST represents a major improvement in this area. In the case of reQUEST, standard compiler techniques [i.e. recursive descent] are used to completely understand the source code of the program under test. The front-end produces symbol tables and a program graph which is annotated with sufficient information to be used by a subsequent [unimplemented] pass to generate compiled Ada code. The annotations are in the form of a stack-language similar to P-code or FORTH. This front end is currently almost completely implemented, using Ada as the implementation language.

Path Predicate Construction

QUEST uses a modification of forward substitution in which each variable is assigned a version number. The combination of version number and

variable name uniquely identifies each symbolic value along a path, in effect acting as a pointer to each symbolic value.

All variables involved in the path are tagged with version numbers during a forward traversal of the path as follows:

[1] Initially assign a version number of 0 to each variable.

[2] Traverse the path from top to bottom. For each decision encountered along the path, tag each variable with the current version number of that variable. For each assignment statement encountered along the path, tag each variable on the right hand side with the current version number of that variable, then increment the version number of the target variable and tag the target variable with the new version number.

Once the variables along the control path to be driven have been tagged, all assignment statements and conditions become terms of the predicate. If any terms of the predicate can be shown to be contradictory, then the path is infeasible. The above algorithm is similar to that used by optimizing compilers during the formation of normal blocks and is closely related to that used by Cheatham[12, 13].

This algorithm has several advantages over the method used by Clarke[9]:

[1] Common subexpressions are not duplicated, thus considerably reducing the size of the abstract syntax tree formed to represent the path predicate.

[2] The understandability of the path predicate is likewise improved, which facilitates manual solution of the path predicates when automatic approaches fail.

[3] The interpretive evaluation of the predicate during automatic predicate solution is sped up.

Consider the following program segment:

```
1:      C := 0
2:      I := ( X / 2 ) * ( 2 / X )
3:      for J in I .. K loop
4:           if Z(J) = Y then
5:                C := C + 1;
6:           end if;
7:           if Z(J) /= Y then
8:                C := C - 1;
9:           end if;
10:          Y := Y + 1;
11:     end loop;
12:     I := C;
```

Note that separate reference labels are assigned to the condition and action portions of an IF statement. This is useful for the purposes of documentation, since the condition may be executed without the execution of the action. In more complicated conditional statements, it may be useful to separate the condition into conjunctive normal form and assign a separate label to each term of the conditional expression.

We might want to execute the path (1,2,3,4,5,6,7,9,10,11,3,12). We therefore have the predicates:

```
1:      C.1 = 0
2:      I.1 = (X.0 / 2) * (2 / X.0)
3:      (J.1 = I.1) and (J.1 <= K.0)
4:      Z(J.1) = Y.0
5:      C.2 = C.1 + 1
6:      true
7:      not ( Z.0(J.1) /=  Y.0 )
9:      true
10:     Y.1 = Y.0 + 1
11:     true
3:      (J.2 = J.1 + 1) and (J.2 > K.0)
12:     I.2 = C.2
```

where I.n denotes the identifier I after n occurrences on the left hand side of an assignment and where the labels refer to the atomic action which generated the predicate.

Detecting Infeasible Predicates

A path predicate may be placed into one of four categories:

Always false (inconsistent),
Always true (tautology),
Sometimes true and sometimes false,
Indeterminate.

Tautological path predicates are not particularly interesting, since they are rare and have trivial solutions. On the other hand, we are particularly interested in detecting inconsistent path predicates, since it is a waste of effort to try to find a solution for them. In this section we discuss the methods used to detect infeasible path predicates. The next section discusses approaches to attempting to solve path predicates which are not known to be infeasible.

Interval Analysis

Barrett gives examples where commutivity, associativity, and distributivity may fail in computer arithmetic[12]. One way to handle the mathematical characteristics of computer arithmetic is to use interval arithmetic in the analysis, in which the result of a computations is not considered to be a single number, but an interval which contains both the true mathematical result expected and the machine result obtained[14]. When further calculations are carried out, this interval is propagated (and tends to expand).

QUEST combines interval analysis with information derived from subrange declarations (Ada), FORMAT field declarations (FORTRAN), user-specified input domain, and known characteristics of functions such as sines and cosines to propagate worst case bounds on all intermediate results. This method is quite effective in detecting many inconsistent path predicates.

For example: Suppose A and B are two integer variables which each have an input domain -9 to +99. We can immediately determine that the expression A+B has an output range of -18 to +198, and

can establish that (A+B > 200) is a contradiction (always false).

Symbolic Simplifications

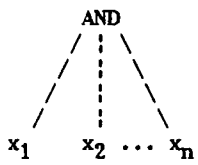
Another approach used by QUEST to detect inconsistent path predicates is to apply rewrite rules to attempt to simplify the path predicate to "false"[13].

We use the following symbols to describe the simplification rules:

&	"and"
¦	"or"
~	"not"
->	"implies"
<->	"equivalent"
@	"for all"
%	"there exists"
(x)	"x is true"
#X	"cardinality of bag X"
[x]	"bag containing the element x"

The most important rules are stated below as theorems. The proofs will be presented elsewhere.

Definition 1: Let AND(X) denote an n-ary abstract syntax tree with root node "AND" and the bag of children X = [x1, x2, ..., xn], represented diagrammatically as



and defined recursively as:

AND([]) = TRUE
AND(X) = AND(X - [x]) & x if #X > 0,

where x is an arbitrary member of X. This is equivalent to

AND(X) = @x member X (x).

Definition 2: Let OR(X) be defined similarly as

OR([]) = FALSE
OR(X) = OR(X - [x]) ¦ x if #X > 0,

which is equivalent to

OR(X) = %x member X (x).

Theorem 1: if AND(X) -> Q
then (AND(X) & Q) <-> AND(X).

Theorem 2: if P -> OR(X)
then (P ¦ OR(X)) <-> OR(X).

Theorem 3: if (AND(X) -> ~Q)
then (AND(X) & (Q ¦ OR(Y)))
<-> (AND(X) & OR(Y)).

Theorem 4: if (~P -> OR(Y))
then (OR(Y) ¦ (AND(X) & P))
<-> (OR(Y) ¦ AND(X)).

Theorem 5: if %x member X (x -> Q)
then AND(X) -> Q.

Theorem 6: if %x member X (P -> x)
then P -> OR(X).

Theorem 7: if %x member X (x -> ~Q)
then AND(X) -> ~Q.

Theorem 8: if %x member X (~P -> x)
then ~P -> OR(X).

Theorem 9: if @y member Y (y -> ~Q)
then OR(Y) -> ~Q.

Theorem 10: if @y member Y (~P -> y)
then ~P -> AND(Y).

Theorem 11: if @x member X %y member Y
(x -> y) then OR(X) -> OR(Y).

Theorem 12 if @y member Y %x member X
(x -> y) then AND(X) -> AND(Y).

Theorem 13: if #X = 1 then AND(X) <-> x.

Theorem 14: if #X = 1 then OR(X) <-> x.

Theorem 15: if @y member Y (P -> y)
then P -> AND(Y).

Theorem 16: if @x member X (x -> Q)
then OR(X) -> Q.

Theorem 17: if @x member X (~Q -> x)
then ~AND(X) -> Q.

Theorem 18: if @y member Y (~y -> P)
then ~P -> AND(Y).

Theorem 19: if %x member X (~Q -> x)
then ~OR(X) -> Q.

Theorem 20: if %y member Y (~y -> P)
then ~P -> OR(Y).

Application of Theorems

Space does not permit an extended discussion of the theorems, but as an example, consider

AND([A=B, C<D, C<=D]).

This can be simplified as follows. By Definition 1, we obtain

AND([A=B, C<D]) & (C <= D).

Since (C<D) -> (C <=D), application of Theorem 5 yields

AND([A=B, C<D]) -> (C <= D).

Finally, by Theorem 1 we obtain

AND([A=B, C<D]).

Limitations of Transformations

Clarke shows that the constraint I / J <= 7 can be simplified to I - 7 * J <= 0. [10] However, this transformation is not actually correct if I and J are integer variables in FORTRAN, as can be seen if I = 22 and J = 3. In this case, according to the FORTRAN rules of integer division, I / J = 7, which evaluates to be true. However, I - 7 * J = 22 - 21 = 1, so that I - 7 * J <= 0 evaluates to be false.

Care must also be taken in the manipulations of real variables, due to the characteristics of machine arithmetic. For example, many programs use scaling to avoid overflow and/or underflow during a series of computations, so that the expression X * SCALE * X gives a useful value whereas the algebraically equivalent expression X * X * SCALE might give overflow (if ABS(SCALE) < 1.0 and ABS(X) is large) or underflow (if ABS(SCALE) > 1.0 and ABS(X) is small). Computations of this type may be found in the LINPACK routine SNRM2. [11]

QUEST presently avoids doing transformations in which machine arithmetic can differ from the algebra of continuous variables.

Predicate Solution

Special Features of Path Predicates

Certain aspects of path predicates appear to cause difficulty for the predicate solver, while other aspects may be advantageous. These features are discussed below.

One beneficial feature of path predicates is that we are only interested in finding a feasible solution, rather than finding any kind of optimal solution. This means that if the problem is formulated as a linear programming problem, as soon as all artificial variables have been driven to zero we may stop. Thus we are only interested in Phase I of the Simplex linear programming method.

A feature of path predicates which is not beneficial is that they are general mixed integer problems, in which some variables may be continuous, while others are discrete, and in general the discrete variables are not of the 0/1 variety. This type of problem tends to be much harder than standard linear programming problems. However, again, since we need only find a feasible solution, rather than an optimal one, the standard algorithms may be simplified. For example, if the branch and bound algorithm for implicit enumeration is applied, the "bound" portion of the algorithm may be ignored, since we would stop as soon as a feasible solution is obtained which satisfies the integrality constraints.

Another problem with path predicates is that they may include nonlinear constraints which involve products of variables, transcendental functions, quotients, MOD functions, etc.

QUEST Predicate Solver

No simplification of path predicates is needed in order for the QUEST solver to work; however, in the case of predicates containing many disjunctions, it may be advantageous to convert the predicate to disjunctive normal form and attempt to solve each minterm in turn until a solution is found or all minterms have been considered. Since the formation of disjunctive normal form is computationally expensive, this is done only as a last resort.

A randomly generated starting point is used to generate an initial trial point within the user-specified input domain. The path predicate is then interpretively executed, replacing evaluation of Boolean operators and relational operators by evaluation of appropriate penalty functions.

A non-parametric optimization algorithm which is an extension of Nelder's Simplex method is applied[15]. Nelder's Simplex method is not to be confused with the Simplex method of linear programming. This approach works even when the penalty function contains non-linearities and discontinuities. This allows the use of "procedure elision" in which the execution path through procedure and function calls need not be elaborated. This is especially important in languages such as Ada which depend heavily on the use of packages with separate definition and implementation components.

If the starting point does not converge to a solution, a new starting point is randomly selected.

Results

The implementation of QUEST/F77 is complete and it runs on VAX/VMS systems. However, it's capabilities are limited by the range of program constructs allowed by its front end. Attempts to port the entire QUEST/F77 to Turbo Pascal failed due to the link edit phase aborting because of relocation table overflow. This is a consequence of the monolithic design of QUEST/F77. However, the back-end portion of QUEST/F77 has been ported to Turbo Pascal. The front end of reQUEST is partially implemented, and will be integrated with the back-end (predicate solver portion) of QUEST/F77 to yield a system which can process a large subset of Ada. Initial results show that QUEST/F77 can generate most test cases for programs including arrays, non-linear calculations including transcendental functions, and can in most cases detect infeasible path predicates.

Preliminary results indicate that the predicate solver is robust and widely applicable.

Conclusions

Additional work is needed to couple the front-end processor for Ada to the backend predicate feasibility checker/solver. While results are presently incomplete, they are encouraging.

Acknowledgements

This paper is based on work toward fulfilling the requirements of a Ph.D. degree in Computer Science and Engineering at Auburn University.

I would like to thank Professor David Brown and Mr. Mark Lanus for useful comments on drafts of this paper. Any remaining errors are mine, however.

This work was supported in part by the Army Strategic Defense Command Terminal Imaging Radar program under contract DAAH01-84-D-A030/0006 to the Department of Computer Science and Engineering of Auburn University.

The views, opinions, and/or findings contained in this paper are those of the author and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

References

1. Brown, D.B.; Haga, K.D.; and Weyrich Jr., O.R., "QUEST -- Query Utility Environment for Software Testing." International Test and Evaluation Association 1986 Symposium Proceedings: Challenges of Testing Space Systems, Huntsville, Alabama, September 30-October 2, 1986, pp 38-43.

2. Weyrich, Orville R., Jr., Cepeda, Sandra L., and Brown, David B. "Glass Box Testing Without Explicit Path Predicate Formation." Proceedings of the 26th Annual Southeast Regional Conference of the ACM, Mobile AL, 1988, pp 596-601.

3. Weyrich, Orville R., Jr., Brown, David B., and Miller, John A. "The Use of Simulation and Prototypes in Software Testing," Paper presented at the 1988 Eastern Simulation Conference: Tools for the Simulation Profession. Orlando, FL, April, 1988.

4. Woods, John Lowe. Path Selection For Symbolic Execution Systems, An Arbor, MI: UMI Research Press, 1982.

5. Cepeda, Sandra Liliana. Automated Generation of Program Paths to Achieve Logic Coverage. Master's Thesis, Auburn University, December 12, 1986.

6. Weyuker, Elaine J. "Axiomatizing Software Test Data Adequacy." IEEE Trans. Software Eng. SE-12, No. 12 (December), 1986, pp 1128-1138.

7. White, Lee J., and Cohen, Edward I. "A Domain Strategy for Computer Program Testing," IEEE Trans. Soft. Eng., Vol. SE-6, No. 3, 1980, pp 247-257.

8. Ramamorthy, C.V., Ho, S.F., and Chen, W.T. "On the Automated Generation of Program Test Data." IEEE Trans. Soft. Eng., Vol. SE-2, (Dec), 1976, pp 293-300.

9. Clarke, Lori.A., and Richardson, Debra.J. "Symbolic Evaluation -- Implementations and Applications," in Computer Program Testing, edited by B. Chandrasekaran and S. Radicchi. Amsterdam: North Holland, 1981.

10. Clarke, Lori A. "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. Soft. Eng., Vol. SE-2, 1976, pp 215-222.

11. Dongarra, J.J, Moler, C.B., Bunch, J.R., and Stewart, G.W. LINPAK User's Guide. Philadelphia: SIAM (1979).

12. Barrett, William A. and Couch, John D. Compiler Construction: Theory and Practice. Chicago: Science Research Associates, 1979, pp 554, 576.

13. Cheatham, Thomas E. Jr., Holloway, Glenn H., and Townley, Judy A. "Symbolic Evaluation and the Analysis of Programs." IEEE Trans. Soft. Eng., Vol. SE-5, No. 4 (July), 1979, pp 402-417.

14. Moore, Ramon E. Methods and Applications of Interval Analysis, Philadelphia: SIAM, 1979.

15. Nelder, J.A., and Mead, R. "A Simplex Method for Function Minimization," Computer J., Vol. 7, 1965, pp 308-313.

Biographical



The author received a B.A. degree in chemistry, mathematics, and physics from Union College of Kentucky in 1973, an A.M. degree in Chemistry from Duke University in 1976, and a Ph.D. degree in Chemistry from the University of Tennessee at Knoxville in 1982. He is in the final stages of completing a Ph.D. degree in Computer Science and Engineering at Auburn University. He is currently employed by Transform Logic Corporation of Scottsdale, AZ. His research interests include software testing, formal software development techniques, software reverse engineering, and applications of artificial intelligence to health care delivery. His mailing address is 9600 N. 96th Street #140, Scottsdale, AZ 85258.

Reusability and Extensibility in Ada[1][2]

Fuyau Lin, Saad Harous

Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

Abstract

We study the facilities found in Ada for **reusability** *and* **extensibility** *with those of OOPLs such as Smalltalk, C++, and Object-C. Reusability and extensibility are both facilitated by inheritance in OOPLs. Reusing an existing object that is exactly what is needed is not any different from ordinary languages, e.g. Ada, where one can use an existing procedure or package whose specifications meet the requirements. The different feature of inheritance for reusability in OOPLs is that a programmer can reuse an object (class) for purposes different than what it was originally created for by modifying it through subclassing. Though the Ada package concept provides a good basis for the development of families of software components, Ada requires more improvements to keep up with the mechanisms that OOPLs offered.*

1 Introduction

We study the facilities found in Ada for **reusability** and **extensibility** with those of OOPLs such as Smalltalk [4], C++ [11], and Objective-C [2]. Reusability and extensibility are both facilitated by inheritance. The reason for selecting Ada for this comparison is that among languages that are not object-oriented, it is one of the most advanced in its support for these features. The **control abstraction** which delivers a sequence of actions as a single one. The package provides **data abstraction**: type definitions for a user-defined data type can be grouped together with the subprograms that operate on the data type in one package, and the entire structure can then be treated as a built-in type.

Ada has **generic** subprograms and packages, which relax the "**as is**" condition on reusability, while the reusability and extensibility provided by generic unit in Ada is not as general as provided by **classes** and **inheritance** in OOPLs. The notion of objects was first proposed in the language Simula [3], designed in the late 60's. Reusability is the ability of a system to be reused, in whole or in parts, for the construction of new system. Extensibility is the ease with which a software system maybe changed to account for modification of its requirements.

The high cost of building large software systems can be attributed to the fact that most software development efforts are done from scratch. In an ideal software development environment, new system are built by "ordering components from the catalogs of software modules and combining them, rather than reinventing the wheel every time" [7]. Component reuse is not restricted to the initial construction phase of a software system. It continues to pay an important part in the maintenance of the system. Maintenance is a process of reusing modules across time, rather than across application. Enhancing the system results in modifications to the system components in unforeseen ways, just as during system design reused components need to be tailored to fit the new application. Therefore, reusability and extensibility are interrelated; this is the reason for study them together in this paper.

The rest of this paper is organized as follows: in section 2 we introduce the basic terminology used. Reusability and extensibility are discussed in section 3 and 4, respectively. Comparison to reusability and extensibility in Ada is given in section 5. We conclude or paper by summarizing our investigation.

2 Background and Terminology

This section describes some terminology used in this paper. The discussion will use the terms in Smalltalk; the terminology is somehow specific to this language.

Objects

An object is well-defined data structure coupled with a set of operations that manipulate the data. This is the same as the definition of *type*. The behavior of an object is characterized by the operations defined on it, i.e. only the operations can manipulate the object. We can send a *message* to an object, called *receiver*, to invoke one of the object's operation. The message receiver has *methods* which describe how the operations should happen. A method is akin to a procedure in that it is comprised of a sequence of executable statements.

[1] Ada is the trademark of DoD.

[2] This work is supported by DoD under grant MDA904-89-C-6018.

Classes

A description of the common features of these objects is provided in a *class* description; individual objects are known as *instance* of the class. The description includes the form of the instances' private memory, and the methods that manipulate the instances. Classes are objects themselves, and as such they are themselves instance of class, called *metaclass*. The metaclass is useful for defining the behavior of the class as a whole; its most common use is to provide methods to create and initialize instance of the class. Every class inherits from the class *Object*. This is a class provided by the language which described rudimentary behavior common to all objects in the system.

Reuse

Reuse is the activity of using an existing component in defining a new component. Reuse has been recognized as an important activity in software engineering. Reusability improves reliability since new software components can be built on existing components which have been extensively tested. Reuse during program development is also vital in providing significant leverage in productivity. This view is supported by the fact that software engineering research is mainly concerned about the impact of reuse on software productivity.

Inheritance

The inheritance mechanism is closely tied with the notion of a *class*. A class provides encapsulation to its instance. A class captures static properties of objects such as *attributes* and *methods* in an explicit form. A new class can be defined as an *extension* of existing classes with the support of inheritance. Class inheritance provides a natural classification of components and enhances modularized modification. Inheritance allows new classes to be built on top of older, less specialized classes rather than written from scratch. The new class is the *subclass*; the old one is the *superclass*. The subclass inherits the instance variables and methods of the superclass. The subclass can add new instance variables and methods of its own. It can also define a method with the same selector as one of the superclass's methods; this is know as overriding a method.

The terminology used here has some synonyms in other languages. Instead of message sending, many OOPLs use a variant of function call to invoke an operation on an object. Method also known as operation, procedure, or function. Some language use the word *type* instead of class. Subtype and supertype correspond to subclass and superclass respectively. Some language do not view classes as objects, thus they use other mechanisms for creating initialized instances.

The simplest kind of reusability is to use an existing component **as is**. This is possible in most languages through the existence of subroutine libraries. In Ada, packages containing primitives routines specific to some domain, such as mathematical functions package, IO package, and so on. This kind of reusability is limited, because the functionality provided by a package subroutine is fixed with the exceptions of parameters. Hence it can only be reused in the construction of a new program if there is a need for exactly the same behavior as that provided by the subroutine. Although this is useful in certain situations, it does not solve the reusability problem in general. The reason is that the ad hoc packages do not allow the routine to be modified in ways unforeseen by its designer.

This is exactly what defines a truly reusable component: it can be combined, adapted and modified to fit in a new application in ways unpredicted by the implementor of the component. It must be clear what the component does in order for a programmer to determine if it is useful for reuse in his application. A programmer will not reuse a piece of code if it takes him longer to find out that the code exists, and where the code is, than it would take him to rewrite the code.

The class construct in OOPLs provides a data abstraction mechanism with a well-defined interface specifying what the class does. Thus, the class is the starting point for the creation of reusable components in OOPLs. Classes can be combined and modified to fit a new application by means of *inheritance* and *polymorphism* respectively. We discuss these two issues in the next two sections, and close the discussion with a comparison of reusability and extensibility in Ada.

Ada was designed to support the large scale software development with reusable components. Some of the features of Ada that support such reusability are listed below:

- program units: includes subprograms, packages, and tasks as the basic building block.
- information hiding: allows systematic separation between visible syntactic interface specification and hidden bodies.
- strong type checking: requires the consistency between formal parameters of module definitions and actual parameters of module invocations to be enforced at compile time.
- generic program unit: offers parameterized templates for generating software components.
- program libraries: retains separately-compiled reusable program units.

The facilities provided by Ada for reusable software components are richer than most of the conventional programming languages. The term *object* is used in different ways by many computer communities. All uses of this term have in common the idea that an object has an interface with attributes, operations or capabilities that determines its behavior, and a hidden internal state manipulated by the operations. The first object-oriented programming language with facilities for defining classes, subclasses, and instances

was Simula [3]. Smalltalk is an experimental object-oriented language developed at Xerox PARC which derives from Simula.

Ada support object-oriented programming through its package mechanism. Packages define a collection of operations on an internal state which may persist between calls on operations. Abstract data types are realized by package specification containing a type with a hidden data specification from which *object* may be created by the package user. The generic facility is an alternative abstraction mechanism which allows multiple instances of a package to be created by generic instantiation. Ada does not allow packages to be passed as parameters, be components of structures, or be assigned as values of variables. Thus package provide a facility for object-oriented programming, but are not themselves first-class objects [12].

3 Inheritance Issue

When an existing code is not exactly what is required for a new application, the designer can make use of the existing class by customizing it in some way to fit its new purposes. There are two ways of achieving such customization:

1. modify the original code definition with or without copy,
2. modify the original code by augmentation.

When the original abstraction becomes more and more complicated there are problems for tailoring in various applications, for example, it is difficult to understand and to extend. Modifying a copy has the update problem with duplicated data, for example, changes to the original code are not automatically made to the copy.

The second approach is performed in OOPLs by means of inheritance. A new abstraction is defined by specifying in a new class the difference between the new abstraction and a pre-existing class, and appending the new class to the old one by making the former a subclass of the latter. The important property of defining a new abstraction using inheritance is that it does not affect existing classes, and no code is replicated. Inheritance is therefore a major tool for reusing software.

3.1 Single Inheritance

A new class can be built from a primitive one in different ways, depending on how the new class differs from the old one. We can identify three different ways of derivation:

- extension: the subclass adds instance variables and methods of its own, thus refining the base class abstraction.
- variation: the subclass redefines some of the features inherited from the base class.
- specialization: this is a combination of extension and variation.

Extension is the simplest way of inheritance. It is often used in conjunction with a base class that defined primitives common to many other related classes.

Sometimes, the base class provides the operations needed by the subclass, but the operations need to be implemented differently in the subclass. Variation is most often accomplished by *overriding* the inherited method, i.e. defining the same operation with the same name in the subclass. In Ada there are constraints on how the operations can be redefined. C++ and Simula use virtual functions for run-time type resolution. In C++, a function declared as virtual in a base class can be redefined in each derived class. In some cases, neither extension nor variation alone is powerful enough to construct a new class from an old one.

3.2 Multiple Inheritance

The main difference among OOPLs with multiple inheritance is what they do when an operation is inherited from more than one ancestor. This is similar to reuse some components from different piece of code. There are two different approaches to perform multiple inheritance:

- class precedence list: The approach taken by Flavor [8] and Loops is to compute a class precedence list by a depth-first traversal of the inheritance subgraph rooted at the class, and choose the method from the class with the highest precedence.

  CommonLoops considers the precedence list should not be built into the language, but should be under the control of the programmer. The metaclass of a class determines the algorithm for computing the class precedence list from the local precedence of the class being defined.

- explicit precedence: Extended Smalltalk, etc., take the position that no simple precedence relationship for multiple inheritance will work for all the cases, so none should be assumed at all. Whenever a method is provided by more than one superclass, the programmer of the subclass must explicitly indicate which one should be used.

Sometimes, a method needs to be defined as a combination of other methods. The simplest form of method combination is through the use of **super**, or similar constructs. Sending a message to **super** indicates that the search for the method should start in the superclass of the class containing the method in which **super** was used. Loops extends this notion to allow more than two methods to be combined.

One problem with inheritance and method-combination as mentioned above is that the behavior of an instance is fixed at instance creation time. This is undesirable in long-running system, where run-time events can cause the behavior of the objects in the system to change over time.

4 Polymorphism Issue

Polymorphism is an important feature of some programming languages that allows the definition of flexible software elements amenable to extension and reuse. The polymorphism operation is one that has multiple meaning depending on the type of its arguments [1]. In Smalltalk, a variable or an expression representing the receiver of a message may dynamically vary in type. The method that is executed, as specified by the message selector, is directly associated with the type of the receiver. Therefore, different results will be produced depending on the type of the object receiving the message. This is called *simple polymorphism*: the operation invoked is dependent on the type of only one argument. *Multiple polymorphism* is the ability to execute different functions based on the types of more than one of the function's arguments.

In essence, a polymorphic function has an implicit or explicit type parameter which determines which function is to be invoked. The simple polymorphism is not sufficiently powerful to handle all the extensions that may be needed in the evolution of a software system. For example, extending the functionality of a graphic viewer to allow graphical objects to be displayed on various devices, leads to problems if only simple polymorphism is available. The reason is that we now have two polymorphic variables, the one holding the graphical object and the other holding the display device. This gives rise to the doubly polymorphic interaction.

Languages with only simple polymorphism facilities extension in only one dimension. This kind of polymorphism can be simulated in conventional language such as Ada in a similar fashion. It is expected to have some unacceptable results.

CommonLoops handles multiple polymorphism. A method can be specified in terms of the types of any number of arguments. Ingalls [5] suggests to use message transmission to reduce a polymorphic variable to a monomorphic one by the type dispatch inherent in method lookup. Therefore, for the case of two polymorphic variables, we need to send two messages to reduce the double polymorphism.

5 Ada Generic for Reusability and Extensibility

In this section we will compare the facilities supported in Ada **reusability** and **extensibility** with those of OOPLs such as Smalltalk, C++, and Object-C. Reusability and extensibility are both facilitated by inheritance. The reason for selecting Ada for this comparison is that among languages that are not object-oriented, it is one of the most advanced in its support for these features. The **control abstraction** which delivers a sequence of actions as a single one. The package provides **data abstraction**: type definitions for a user-defined data type can be grouped together with the subprograms that operate on the data type in one package, and the entire structure can then be treated as a built-in type.

Ada has two constructs for structuring a program: the subprogram and the package. The former provides control abstraction which treats a sequence of actions as a single one. The package provides data abstraction: type definition for a user-defined type with the subprograms that operate on that data type in one package. Thus the entire structure can be treated as a built-in type.

Both subprograms and packages can be reused "as is"; for example, if a new application needs to use a data type that was defined as an abstract data type in a previous program, then the old abstraction can be reused if it is exactly what is needed in the new application. However, this accounts for only a small percentage of the times existing program fragments could be used, because of the requirement that the old piece and the new use match exactly.

Ada has *generic* subprograms and packages, which relax the "as is" condition on reusability. generic unit allow types to be parameterized, thus factoring out the dependency of the units on types. For example, a generic procedure to swap two items can be written as:

```
generic
   item is private;
procedure swap(x, y : in out item);

procedure swap(x, y : in out item) is
   temp : item := x;
begin
   x := y; y := temp;
end swap;
```

In this example, `item` is declared private; this means that the only operations allowed on `item` in the unit are assignments and tests for equality and inequality. If other operations were needed, additional generic parameters would need to be supplied so that an actual operator for the type used for `item` would be given when the procedure is instantiated.

Generic units are templates; the `swap` procedure given above cannot be invoked as it is. It first must be instantiated and particular type specified for `item`. For instance, to swap two integers, we could instantiated the generic `swap` procedure by `procedure swap_ints is new swap(integer);`

This results in a distinct procedure being generated at compile-time with the type `item` replaced with the type `integer`. Thus, the generic facility in Ada is similar to a macro expansion feature. The reusability and extensibility provided by generic units in Ada is not as general as provided by classes and inheritance in OOPLs for two reasons:

- There are only two levels of reusable units: the generic module which must be instantiated before its use, and the fully instantiated module which cannot be further refined.
- Compile-time instantiation of generic units: it is not possible to specify an operation on a generic object

that will execute a different operation depending on the type of the object at run-time.

We now describe why the polymorphism found in Ada is not as powerful as that in OOPLs for reusability and extensibility [6]. Ada supports two kindsof polymorphism: overloading of subprogram names and generic units. Using overloading, we can define a number of `draw` procedure that take different type of objects as parameters, for example:

```
procedure draw(object : in LINE);
procedure draw(object : in CIRCLE);
...
```

Since the subprograms are distinguished by the type of at least one operand, no ambiguity arises. However, this solution falls short of providing true polymorphic entities as in object-oriented programming languages. This reason is that overloading resolution occurs at compile time. Note that this is not an essential restriction of statically-type languages. In fact, in C++, a statically-type object-oriented language, this is not a problem because a variable declared of type `object` can denote any object of this type or of any of its subclass, thus achieving the same kind of polymorphism discussed for Smalltalk.

Ada's generic procedures do not work for the same reason. Generic instantiation is performed at compile time with actual type values that must be determinable at compile time. The only feature of Ada could be used to eliminate the polymorphism of graphic object example has nothing to do with overloading or genericity; it is the record with variant type.

```
type GraphicObject(object : ObjectType) is
   record
      -- common parts to all object types;
      when LINE =>     -- for LINE object;
      when CIRCLE =>   -- for CIRCLE object;
      ... other objects
   end record;
```

`ObjectType` is an enumeration type with elements `LINE`, `CIRCLE`, and so on. There would be a single version of each procedure on graphical objects, e.g. `draw`, each containing a case discrimination of the form:

```
case object is
   when LINE =>     -- action for LINE object;
   when CIRCLE =>   -- action for CIRCLE object;
   ... other objects
end case;
```

This solution creates "closed" software systems, i.e. systems that are hard to extend. Addition of another kind of graphic objects is difficult because it would involve changing the variant type and all discriminations that are scattered throughout the program. Another weakness of this solution is that it depends on the programmer manipulating types in a way that cannot be checked by the compiler, a disadvantage for statically-typed languages such as Ada.

6 Summary

Reusability and extensibility are both facilitated by inheritance in OOPLs. Reusing an existing object that is exactly what is needed is not any different from ordinary languages, e.g. Ada, where one can use an existing procedure or package whose specifications meet the requirements. The different feature of inheritance for reusability in OOPLs is that a programmer can reuse an object (class) for purposes different than what it was originally created for by modifying it through subclassing. This allows a class in an object-oriented system to be reused and extended in ways that the original designer of the class not have precipitated, This is different from reusable units in languages that do not have inheritance, for instance Ada generic procedure, where the designer of the generic procedure must specify in advance the kind of types the procedure can be instantiated with.

Besides being able to extend the functionality of a system by subclassing, extensibility of a system is facilitated by the "message-passing" semantics. This allows the object that receives the message determines the method that is executed. Thus, the same message-sending operation result in different executions based on the kind of object the receiver is.

The execution of a system to handle a new kind of object is easily done by adding a class defining the behavior of the new object along with methods for handling messages. In statically-typed languages, the message receiver is declared to be a specific class; at run-time, it can be an instance of that class, or of any of that class's descendents. An interesting issue to be explored in future work is whether this restricts extensibility, and if so what constructs are needed in order to retain the benefits of static typechecking while allowing the development of extensibility systems.

The Ada package concept provides a good basis for the development of families of software components. It should be possible to develop application libraries containing Ada package for reuse. Many researchers consider Ada is a good object-oriented design language. The lesson we learned from this study, Ada still has a long way to go in order to keep up with the mechanisms that OOPLs offered.

References

[1] L. Cardelli, P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism," ACM Computing Surveys, December 1985.

[2] B. J. Cox, "Object Oriented Programming: An Evolutionary Approach," Add'son Wesley, 1986.

[3] O. Dahl, B. Byhrhaug, K. Nygaard, Common Base Language, technical Report Publication No. S-22, Norwegian Computing Center, October 1970.

[4] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison Wesley, 1983.

[5] D. H. H. Ingalls, "A Simple technique for handling Multiple Polymorphism," in Proceeding of OOPLSA 89, September 1986.

[6] B. Meyer, "Genericity versus Inheritance," in Proceeding of OOPLSA 86, September 1986.

[7] B. Meyer, "Reusability: The Case for Object-Oriented Design," IEEE Software, March 1987.

[8] D. A. Moon, "Object-Oriented Programming with Flavors," in Proceeding of OOPLSA 86, September 1986.

[9] A, Snyder, "CommonObjects: An Overview," ACM SIGPLAN Notices, 21(10) October 1986.

[10] G. L. Steele, Jr., Common Lisp the Language, Digital Press, Burlington, MA, 1984.

[11] B. Stroustrup, The C++ Programming Language, Addison Wesley, 1986.

[12] P. Wegner, On the Unification of Data and Program Abstraction in Ada, in Proceeding of Programming Conference, January 1983.

SOME GENERIC PACKAGES FOR TASK COMMUNICATION IN ADA

Chin-Yun Hsieh

School of Electrical Engineering and Computer Science
University of Oklahoma
Norman, Oklahoma 73019

Abstract

The specification of process interaction is an important part in the design of a distributed computing system. Since the Ada language [1] adopted direct and assymmetric naming scheme for the communication channel, it is well suited for specifying process interaction of type single-client/single-server or multiple-client/single-server , but not for that of single-client/multiple-server or multiple-client/multiple-server. This paper presents the design of a number of generic packages for task communication in the Ada language. These communication packages provide Ada programmers a simple and unified way to easily establish a task communication channel for each type of the above mentioned process interaction. In addition, the effect of *asynchronous* communication, either *buffering* or *non-buffering*, can also be achieved which may improve the system efficiency by preventing a sending process from being delayed unnecessarily.

Introduction

A distributed computing system is ususlly composed of a set of communicating sequential processes [2]. These processes run concurrently and communicate each other when necessary. Since processes are termed tasks in the Ada language, the terms process and task are used interchangeably throughout this paper.

The communication between the sending and receiving processes can be characterized by two attributes: 1) the *communication model*; and 2) the *naming scheme* for the communication channel. In a concurrent programming language, the communication model can be either *synchronous* or *asynchronous*, while the naming scheme for the communication channel can be either *direct* or *indirect* in nature. The direct naming scheme can further be divided into two types: 1) *symmetric*; and 2) *assymmetric* [3,4].

The communication model and the naming scheme are two important factors to the classes of problems a concurrent programming language is suitable to apply. As for example in the important programming paradigm of *client-server* relationship, the direct symmetric naming scheme adopted by the Ada language makes it well suited for specifying process interaction of type *single-client/single-server* or *multiple-client/single-server*, but not for that of *single-client/multiple-server* or *multiple-client/multiple-server*.

The design of a number of generic packages for task communication is presented in this paper. These communication packages provide Ada programmers a simple and unified way to establish a communiation channel for each type of the above mentioned process interaction of the client-server relationship. In addition, the effect of asynchronous communication model which is not directly provided by the Ada language's communication primitives can also be achieved.

Message-passing Communication Models

Message passing is a simple mechanism well suited for programming systems in which physically distributed processes cooperate and interact in order to perform a single task [3,5,6]. A message can be used to convey data and, if so desired, to implement synchronization between a sending and a receiving process. Message-based communication nodels can be either synchronous or asynchronous in nature. In the synchronous communication model, the sending and receiving processes are required to synchronize for a message passing to occur, i.e., they should meet at a rendezvous and whichever reaches the rendezvous first is forced to wait for the other. In the asynchronous communication model, on the other hand, the sending process simply sends the message and proceeds no matter whether the receiving process is ready to receive the message. However, it should be noted that the receiving process does wait for the message if it has not arrived when the receiving process needs it. More specifically, we have a *blocking sender* and a *blocking receiver* for synchronous communication model and a *non-blocking sender* and a *blocking receiver* for asynchronous communication model.

Typical Programming Paradigm for Process Interaction

In the real environment, there are two important programming paradigm for process interaction. One is usually called *pipelining* in which the output of each process is used as the input of the other. The other is called *client-server* relationship in which the server provides service for the client. As far as the number of processes involved is concerned, the process interaction in the client-server relationship can be divided into four types, they are *single-client/single-server*, *single-client/multiple-server*, *multiple-client/single-server* and *multiple-client/multiple-server*.

A communication channel can be viewed as a virtual connection line with one end for message input and the other for message output. The input end is connected to the sending process and the output end to the receiving process. In addition, an end is generally called a *point* if it is dedicated to a single process or a *port* if it is connected to more than one process. From this point of view, it is more than appropriate that each of the previously mentioned process interaction types also be termed as *point-to-point* or *one-to-one*, *point-to-port* or *one-to-many*, *port-to-point* or *many-to-one*, and *port-to-port* or *many-to-many*. It is easy to see that the pipelining paradigm is an extension of the single-client/single-server type of the client-server paradigm.

The Naming Scheme for the Communication Channel

The communication model and the naming scheme for the communication channel both play an important role to the classes of problems a concurrent programming language is suitable to apply. The naming for the communication channel between the sending and receiving processes can be either *direct* or *indirect* in nature. The direct naming scheme can be further divided into two subtypes: *symmetric* or *assymmetric*. In the direct symmetric naming scheme, the names of the sending and receiving processes are used as the source and destination respectively for the message passing. In the direct assymmetric naming scheme, however, only the sending process is required to name the receiving process. In indirect naming model, process names are not used and communication is performed through a mailbox accessible to both the sending and receiving processes.

Among the well known concurrent programming languages (or language proposals), Communicating Sequential Processes (CSP) [2] adopted direct symmetric naming scheme while the Ada language [1], Distributed Processes [7], and Synchronizing Resources [8] all adopted direct assymmetric naming scheme. On the other hand, both PARLANCE [9] and Extended-CLU [10] adopted indirect naming scheme.

The direct symmetric naming scheme along the lines of CSP is suitable for process interaction of type single-client/single-server only. The direct assymmetric naming scheme in the Ada language, where a called task is unaware of the identity of the caller has contributed to problems with multiple-client/single-server interaction, other than the trivial case of single-client/single-server interaction. However, it is still not easy to gracefully program a set of processes that feature single-client/multiple-server or multiple-client/multiple-server interaction in the Ada language.

In addition, since the Ada language adopted synchronous communication model, it is not well suited for applications that feature aynchronous communication, i.e., applications in which the sending process is not required to wait for the message to be received in order to proceed. Basically, if a message arrives when the receiving process is not ready to receive it then the message should be stored in the communication channel if it is considered critical or simply be ignored if not.

Generic Packages for Message Buffers

The difficulties in programming mentioned earlier can be gracefully handled in the Ada language if an intermediate message buffer is introduced between the sending and receiving processes. The major function of the message buffer is to provide a temporary storage for pending messages. The capacity of the message buffer may be of unity, bounded, or effectively unbounded depending on the requirement in each application.

It is noted that we can take advantage of the features of a library unit to implement the message buffer. The role of the message buffer makes it more appropriate to be implemented as a task than a subprogram in the Ada language. However, since a task is not a compilation unit, it is thus embeded inside a package to achieve the effect of a library unit. In addition, the package is designed as a generic one to achieve flexibility and reusability which are among the Ada language's major goals.

It is easy to see that the desired application types mentioned previously can be classified into three models : *asynchronous and non-buffering*, *asynchronous and buffering*, and *synchronous*. In this section, an example of the implementation of the generic package is provided for each of the three application models. However, these examples are only used to demonstrate the construct as well as the fundamental features that should be incorpo-

rated in designing the message buffer and should not be considered as complete ones.

In each of these implementation examples, the message is represented as a single unit. It is reasonable to design it this way since the contents of a message is based on the application and its structure and associated attributes, i.e., the number of elements, the type of each element, and the way these elements are put together should be available at the program design phase.

It is easy to see that if a message is a scalar value without components then it can be represented by the corresponding scalar type. On the other hand, if a message is logically composed of several different components, a composite type is used. A composite type can be either an array type or record type. If the components in the message are of the same type then the array type is used, otherwise, a record type is used.

Asynchronous and Non-buffering Model

The generic package shown in Figure 1 provides asynchronous and non-buffering communication for each of the four types of interaction between the sending and receiving processes. The Boolean variable *filled* is used to prevent a single message from being used for more than once. However, since the first alternative of the select statement is not guarded by *filled*, the sending process can keep sending messages to the message buffer no matter whether the previous ones have been received by the receiving process or not.

Asynchronous and Buffering Model

Figure 2 shows a generic package that provides asynchronous and buffering communication for each of the four types of interaction between the sending and receiving processes. As stated previously, it is suitable for applications where each single message is critical while the sending and receiving processes are not required to synchronize for a message passing to occur. Since the size required for the buffer varies from problem to problem, it is designed as a generic parameter to be specified by the user. Also notice that a size of one will force the instantiated message buffer to function as a synchronous model.

Synchronous Model

Figure 3 shows a generic package that provides synchronous communication between the sending and receiving processes. It is suitable for the problem space in which every message is critical and thus may not be ignored. In addition, the sending process can not proceed before a message sent is received by a receiving process. Notice that since the accept statement for the entry *receive* is

```
- - This generic package can be used for asynchronous
- - communication. No buffering capacity is provided
- - and the sending process can not proceed before the
- - message is received by a receiver.
generic
    type message_type is private;
package buffers is
  task buffering is
    entry send(m : in message_type);
    entry receive(m : out message_type);
  end buffering;
end buffers;
package body buffers is
  cell : message_type;
  filled : boolean := false;
  task body buffering is
    begin
      loop
        select
          accept send(m : in message_type);
          cell := m;
          filled := true;
        or
          when filled =>
            accept receive(m : out message_type) do
              m := cell;
            end receive;
            filled := false;
        end select;
      end loop;
  end buffering;
end buffers;
```

Fig. 1: A generic package for asynchronous and non-buffering model

nested in the accept statement for the entry *send*, the synchronous communication between the sending and receiving processes is assured. This generic package can be used for each type of the client/server interaction, or multiple-client/multiple-server, however, it may not be desirable for the type of single-client/single-server for the sake of system efficiency.

It is noted that if the accept statements are given in the following form

```
accept send(m : in message_type);
cell := m;
accept receive(m : out message_type) do
  m := cell;
end receive;
```

instead then the sending process may proceed as soon as the message is received by the message buffer and before it is passed to the receiver . However, any sender trying to send another message will still be held waiting until

the one residing in the message buffer is received by the receiver. In the real environment, it is easy to find a problem space that features this type of process interaction.

All-in-one Message Buffer

The packages given above can be combined into a single one for the sake of convenience. In doing so, two more generic parameters should be added so that the user is able to indicate the type of message buffer desired. An example of the generic package for this purpose is shown in Figure 4. Again, it is used to demonstrate the construct and fundamental features only.

```
- - This generic package can be used for asynchronous
- - communication. A buffer is created with a size
- - specified by the user in the instantiation.
generic
   type message_type is private;
   bound : in natural := 0;
package buffers is
   task buffering is
     entry send(m : in message_type);
     entry receive(m : out message_type);
   end buffering;
private
   buffer : array (0..bound) of message_type;
end buffers;
package body buffers is
   cell : message_type;
   i,o : integer := 0;
   size : positive := bound+1;
   task body buffering is
     begin
       loop
         select
           when i< o+size =>
              accept send(m: in message_type);
              buffer(i mod size) := m;
              i := i+1;
         or
           when o < i =>
              accept receive(m : out message_type) do
                m := buffer(o mod size);
              end;
              o := o+1;
         end select;
       end loop;
   end buffering;
end buffers;
```

Fig. 2: A generic package for asynchronous and buffering model

```
- - This generic package can be used for synchronous
- - communication.
generic
   type message_type is private;
package buffers is
   task buffering is
     entry send(m: in message_type);
     entry receive(m: out message_type);
   end buffering;
end buffers;
package body buffers is
   cell : message_type;
   task body buffering is
     begin
       loop
         accept send(m : in message_type) do
           cell := m;
           accept receive(m : out message_type) do
              m := cell;
           end receive;
         end send;
       end loop;
   end buffering;
end buffers;
```

Fig. 3: A generic package for synchronous model

```
- - This generic package can be used for each type of
- - process interaction mentioned in this paper. Both
- - type and size of the buffer are included in the
- - generic parameter specification.
generic
   type message_type is private;
   synchronous : in boolean;
   bound : in natural := 0;
package buffers is
   task buffering is
     entry send(m : in message_type);
     entry receive(m : out message_type);
   end buffering;
private
   buffer : array (0..bound) of message_type;
end buffers;
package body buffers is
   cell : message_type;
   i,o : integer := 0;
   filled : boolean := false;
   size : positive ;
   initialization_error : exception;
   task body buffering is
     begin
       if synchronous then
         loop
           accept send(m : in message_type) do
              cell := m;
```

```
            accept receive(m : out message_type) do
              m := cell;
            end receive;
          end send;
        end loop;
      elseif bound = 0 then
        loop
          select
            accept send(m : in message_type);
              cell := m;
              filled := true;
          or
            when filled =>
              accept receive(m : out message_type) do
                m := cell;
                filled := false;
              end receive;
          end select;
        end loop;
      elseif bound > 0 then
        size := bound+1;
        loop
          select
            when i < o+size =>
              accept send(m : in message_type);
              buffer(i mod size) := m;
              i := i+1;
          or
             when o < i =>
               accept receive(m : out message_type) do
                 m := buffer(o mod size);
               end;
               o := o+1;
          end select;
        end loop;
      else
        raise
          initialization_error;
      end if;
  end buffering;
  exception
    when initialization_error =>
      return_error_message;
end buffers;
```

Fig. 4: All-in-one Generic Package

The Use of the Message Buffers

Having these generic packages as library units (Only the specification part of a package is actually treated as a library unit), the process of the application of a message buffer is simply to create an actual package by a generic instantiation associated with the actual generic parameters. For example, the following instantiation

```
package asynchronous_nonbuffering_package is
  new buffers(massage_type=>integer);
```

corresponding to the generic package in Figure 1 would create a package called *asynchronous_nonbuffering_package* which is capable of handling messages of type integer without buffering facility.

As a second example, given

```
type person is
  record
    age : positive;
    name : nametype;
    sex : gender;
  end person;
```

then the generic instantiation

```
package student_record is
  new buffers(person,10);
```

corresponding to the generic package in Figure 2 would create a package called *student_record* which is capable of handling messages of type *person* with a buffer capacity of 11 messages.

The generic actual parameters corresponding to the generic formal parameters may be provided in two ways. Firstly, they can be declared in the program unit which contains the corresponding generic instantiation. Secondly, they can be declared in a library unit and then imported to the generic package by a with clause. While the first way is suitable for cases where the message type is only local to a set of communicating tasks, the second way is more desirable in cases where different sets of communicating tasks would share the same message type. It is common and good practice to declare data types to be used systemwidely in a library package unit. Thereby, any unit in the system can import the desired data types by including the corresponding library unit in its *with* clause.

Conclusion

Process communication specification is crucial to the design of distributed computing systems. The communication model and naming scheme for the communication channel of a concurrent programming language is an important factor to the problem space the language is suitable to apply. The use of the proposed intermediate message buffer is not only able to ease the communication specification, but also able to provide various communication needs that may not be handled gracefully simply by using the communiation primitives in the Ada language.

In addition, the overall efficiency of a distributed system may be improved in that it prevents the sending

process from being delayed unnecessarily in the case of asynchronous communication. Notice that the use of the message buffer also complies with the principles of distributed software design, e.g., modifiability, modularity, uniformity, and reusability - to name a few.

Even though the design of the message buffer proposed in this paper is based on the Ada language, the approach can be applied to another language which has the same facilities as provided by the Ada language.

References

[1] U.S. Dep. Defence, "Reference Manual for the Ada Programming Language," MIL-STD 1815A, Feb. 1983.

[2] Hoare C. A. R., "Communicating Sequential Processes," *Comm. ACM,* vol. 21, pp. 666-677, Aug. 1978.

[3] Andrews, G. R. and Schneider, F. "Concepts and Notations for Concurrent Programming," *ACM Comput. Surveys,* vol. 15, pp. 3-44, Mar. 1983.

[4] Mohan C., "A Perspective of Distributed Computing Models, Languages, Issues and Applications," Working Paper, DSG-8001, University of Texas at Austin, Feb. 1980.

[5] Stankovic J. A., "Software Communication Mechanism: Procedure Calls versus Messages," *IEEE Computer*, vol. 15, Apr. 1982.

[6] Stotts, P. D., "A Comparative Survey of Concurrent Programming Languages," *SIGPLAN Notices,* vol. 17, pp. 76-87, 1982.

[7] Brinch Henson, P., "Distributed Processes: A Concurrent Programming Concept," *Comm. ACM,* vol. 21, pp. 934-941, 1978.

[8] Andrews, G. R., "Synchronizing Resources," *ACM TOPLAS,* vol. 3, pp. 405-430, Oct. 1981.

[9] Reynolds, P., "Parallel Processing Structures: Languages, Schedules, and Performance Results," Ph.D. Thesis, University of Texas at Austin, 1979.

[10] Liskov, B., "Primitives for Distributed Computing," *Proc. 7th Symp. of OS Primitives,* Dec. 1979.

Chin-Yun, Hsieh

address: 143A, W. Constitution
Norman, OK, 73072
Area of Specialization: Software Engineering; distributed Systems
Research adviser: Dr. P. S. Chang
Degrees obtained: B.S.E.E. equivalent, NTIT, Taiwan, R.O.C., June 1979;
M.S., Computer Science, U. of Mississippi, August 1984

PREPARING NON-ADA PERSONNEL FOR TRANSITIONING INTO THE ADA WORLD

Sharon Sutherlin

TELOS Systems
Lawton, Oklahoma

ABSTRACT

This paper presents how TELOS Systems has addressed the issue of preparing personnel deeply ensconced in developing maintenance versions of one computer system to be ready to quickly and effectively assume development or maintenance of a replacement Ada based system. TELOS provides technical and engineering support to the U.S. Army Communications and Electronics Command (CECOM) for U.S. Army Battlefield Automated Systems comprising the Fire Support segment of the Army Tactical Command and Control Systems.

INTRODUCTION

The computerized command and control functionality currently provided to the U.S. Army by the TACFIRE Brigade/Corps/DivArty and Battalion systems will be replaced at some point in the future by the Advanced Field Artillery Tactical Data Systems (AFATDS). While meeting current Post Deployment Software Support (PDSS) schedules for TACFIRE maintenance versions, we are actively preparing our personnel for stepping into the Ada world to support the AFATDS system. While this paper is relative to the TACFIRE/AFATDS transition, the concepts presented can be adapted to other systems evolving into Ada technology in the 1990's. The following paragraphs discuss the methods we are implementing to prepare our employees for the future. To accomplish this, we have emphasized the following areas:

- Motivating software personnel
- Software/hardware/engineering methodology training
- DOD-STD-2167A familiarity
- AFATDS functional knowledge
- Ada coding experience through TACFIRE tool development

MOTIVATING SOFTWARE PERSONNEL

In the past two years, the TACFIRE software development section has increased its number of Ada trained personnel to one hundred percent. Motivating the programmers, software engineers, and system engineers to spend primarily their own time to learn a language they were not using on their job was accomplished in a variety of ways.

Promoting awareness of the Government's role in developing and supporting Ada was the first step to motivating the TACFIRE personnel. Combined with that, our company's active corporate and local emphasis on training promoted individuals' decisions to enroll in Ada classes.

Additionally, the knowledge that TACFIRE is scheduled to be replaced and will be faced with a decreasing level of effort, has been an impetus to cause many to seek Ada training. With the on-going discussions and decisions to re-write current PDSS systems in Ada and re-host them in Army Command and Control (ACCS) common hardware, more options are becoming available for individuals prepared to enter into Ada development. Ada-trained employees recognize the opportunity to contribute to current Ada tasks. Those include the Forward Entry Device, under development for fielding in the Handheld Terminal Unit, and the re-host of the Multiple Launch Rocket System in the ACCS hardware.

However, we have found that with Ada, the driving factor for our programmers', software engineers', and system engineers' involvement has been an interest in the language itself. Discussions of a language which enhances, encourages, and, at times, enforces modern software engineering concepts such as abstraction, information hiding, reusability and portability have caught the attention of technicians working in other languages. A desire to understand the benefits of features such as packages, private and generic types, and tasking has motivated our personnel to study and use Ada.

Basically, our employees' motivation to prepare for future systems evolved on its own course. With that motivation, they were ready and willing to obtain training.

TRAINING

TELOS' commitment to provide up-to-date training in software, hardware, and software engineering methods has resulted in a work force which is easily adaptable to various tasks. The TACFIRE personnel's formal Ada training consists of upper division courses at Cameron University and comprehensive Ada and Advanced Ada classes taught in-house.

To supplement the Ada language with knowledge of the target AFATDS hardware, select TACFIRE programmers and software and system engineers attended the ACCS Common Hardware and Software (CHS) courses conducted by Hewlett-Packard. Programmers attended Programming Support Environment (PSE) training, gaining knowledge necessary for developing AFATDS software. TACFIRE software and system engineers attended Battlefield Automated Systems (BAS) training, acquiring target system hardware and configuration information. That knowledge is critical to making long-range plans for the installation of the AFATDS software development and test-bed facility at Fort Sill.

In addition to obtaining software and hardware training, our personnel have attended classes in Structured Analysis and Structured Design (SA/SD), Object Oriented Structured

Analysis (OOSA), and in Object Oriented Design (OOD). We have incorporated SA/SD techniques into our TACFIRE analysis and design activities and have used SA/SD, OOSA, and OOD in the development of software tools.

Enthusiasm for gaining pertinent training has carried over into a desire to become familiar with documentation associated with the DOD-STD-2167A life cycle.

DOD-STD-2167A FAMILIARITY

An important aspect of transitioning into Ada systems is adapting to the difference in the type and amount of documentation required by DOD-STD-2167A. Our engineers and programmers in non-Ada tactical systems are experienced in developing in-depth requirement definition, and, throughout the life cycle, developing system level, preliminary, and detailed designs to support the approved requirement definition.

The recognition that the basic process of definition and system and software design remains the same minimizes the inhibitions many newcomers to DOD-STD-2167A face. That recognition occurs as familiarity of the documentation associated with the DOD-STD-2167A life cycle is gained.

In order to gain that familiarity, our non-Ada system personnel have participated in working groups which developed comparison studies between the old and new standards, addressed tailoring documentation, and developed coding standards. Additionally, reviewing and providing comments to documentation produced for Ada systems has educated the personnel targeted for transitioning into Ada systems.

AFATDS FUNCTIONAL KNOWLEDGE

System knowledge is a value inherent to personnel developing functionality which lies at the heart of software and hardware being placed on the "state-of-the-art" bandwagon. TACFIRE personnel have the in-depth expertise necessary to translate field artillery operational requirements into software requirements. The TACFIRE knowledge has been utilized and enhanced by reviewing AFATDS documentation and by preparing matrices tracing TACFIRE functions into AFATDS documentation. Again, providing personnel the opportunity to become familiar with the future system will ease the transition.

ADA CODING EXPERIENCE

Recognizing that you either "use it or lose it", TACFIRE software personnel decided to "use it". That decision came along with a commitment to use their own time to enforce and enhance their Ada training. A brainstorming session identified tools which could be written in Ada and which would actually benefit their TACFIRE work. The following tools, consisting of several thousand lines of code, were chosen for implementation.

- SUPER MEG is a tool used to create TACFIRE message format skeletons for prototyping tactical message changes and for building automated test procedures. The tool reads the object TACFIRE format library message files and provides the user with legal entry prompts based on the syntax requirements of the targeted data base. The tool is executed interactively, accessing a configured baseline system or a development system. It replaces a tool which is extremely limited and takes approximately 20 hours to run. Assessment of the code has determined that it is highly portable. The code may be reused as a part of an automated test system.

- The Effects Reporter is a tool which extracts effects data from extensive test files and formats a concise report of the data. The reusability of this code is high. It can be easily modified to capture various types of data from test files.

- Fire Support Element (FSE) Effects Model, which is currently under development, will test modifications to the nuclear effects data.

The experience gained from a team effort of designing and implementing tools which are then used by the team is invaluable. Through their own initiative and their willingness to use their own time, our team gained a depth of knowledge which is much greater than that which was gained by training alone.

CONCLUSION

Several benefits have been realized from implementing the methods described in this paper. Our software personnel have a much more flexible future than they felt they had two years ago. Learning and using Ada and various analysis and design techniques has not only prepared them for future work, but has given them a fresh perspective toward the implementation of TACFIRE requirements. Their confidence level is high - they know they have improved their marketability. They also know they have improved the productivity of their jobs by developing useful tools.

In addition to the individual and team benefits, our approach has developed a group of professionals ready and willing to transition into the Ada world of software development.



SHARON SUTHERLIN

TELOS Systems
P. O. Box 33099
Ft. Sill, Oklahoma 73503-0099

SHARON SUTHERLIN is the TACFIRE Software Task Manager in the Command and Control Branch of TELOS Systems in Lawton, Oklahoma. She has nine years of experience in scientific computing, and seven years of experience in PDSS, working on TACFIRE and MLRS Ada projects. Sharon is also an adjunct faculty member at Cameron University, teaching classes at Cameron, Fort Sill, and Altus Air Force Base. She has a Bachelor's of Science and an Associate's degree in Technology, and is active in the Southwest chapter of SIGADA.

MANAGING ADA SOFTWARE DEVELOPMENT IN THE REAL WORLD

Sandra J. Fee

Vitro Corporation
Silver Spring, Maryland

Abstract

Managing the development of software in a dynamic environment where specifications are in flux is always challenging. Developing this software in any new language, and especially, in a new language as robust as Ada will only be successful if project managers are knowledgeable and resourceful. There are ways to utilize Ada and the software engineering methodology it demands to achieve quality, cost and schedule goals. The software development project described in this article is an example of a management approach that resulted in the successful utilization of Ada. Some practical management lessons learned have emerged and are described herein.

The Real World, a Dynamic Environment

While the standard waterfall software life cycle model is common throughout the industry, in most cases, the exact segmentation of software development into phases implied by the model is actually an ongoing, iterative process. It is not unusual to find that software requirements specifications must be revised late in the software development life cycle. Perturbations of specifications cause software engineers to attempt to meet unrealistic deadlines, while customers complain about unachieved goals. There are two basic solutions to this problem. One is to assume that the specifications are definitive and unchangeable; the other is to design for change. Several initiatives, such as rapid prototyping and mathematical formalism, are being implemented to improve the definition of the specifications. However, in many applications involving unprecedented systems, it is nearly impossible at the beginning of the life cycle to define accurate specifications which will remain invariant throughout the development. Therefore, "designing for change" is an option that must receive serious consideration. Several features of Ada, such as the separation of package specifications and bodies, plus support for separate compilations, provide an environment that supports designing for change.

Project Description

The Ada project described herein consisted of the development of software to support a shipyard testing program for large-scale naval submarine weapons systems. It was developed in an environment where submarine systems designers and contractors are developing their respective hardware systems, the testing program itself is in a state of flux, and the pass/fail criteria used to judge the tests are under development. Nothing is finalized until the tests themselves are less than 6 months away. Developing software used to analyze the various submarine systems under these conditions has always been a challenge. Developing it in Ada was more of a challenge, but as the project progressed, it became apparent that Ada was an asset. The project was the first Test Department Ada project, and it was completed within budget and on time. During a period spanning 42 months, which included the Requirements Phase, more than 360,000 Ada source lines of code were produced.

To support the shipyard testing program, both real-time and post-mission analysis software were developed. Furthermore, the post-mission analysis software was partitioned into two steps, data translation and scientific analysis. To satisfy the different software functions and to meet schedule goals, three different target computers were

selected, the Rolm HAWK/32, the Data General MV/10000, and the VAX 11/785.

Software Development Plan

This software project began with the drafting of the Software Development Plan and other software development standards documents in 1984. This Plan called for a standard waterfall life cycle development process with appropriate documentation, formal reviews, and certification of documents and software. Relevant milestones were defined to measure progress and to delineate the phases, top-down structured modular design[6] was designated, and general guidelines for configuration management and quality assurance were established.

Language/Environment Solution

At the initiation of the project, the languages being considered were FORTRAN 77 (which most of our software engineers knew) and Ada. With the selection of Ada came the challenge to meet the same development schedule that had been projected for development using FORTRAN 77. In the process of determining that Ada would be the language of choice, with assembly language used only if necessary, several Ada compilers and environments were investigated.

The Rolm HAWK/32 was the designated target computer on board the submarine for the real-time software. Since a self-hosted Ada compiler for the HAWK/32 was not available at that time from Rolm, and since the HAWK/32 is a ruggedized version of a Data General computer, we elected to use a Data General Ada compiler and environment on a DG MV/10000 for development of the real-time software.

The VAX 8650, along with the DEC Ada compiler and environment, were chosen for the development of the post-mission scientific analysis software after various Ada compilers and environments were investigated. The post-mission data translation software was developed on the DG MV/10000 because of its availability to process non-standard formatted data tapes.

Training and Tools

At this time, Ada was very new and we recognized the need to emphasize training and hands-on compiler experience. Our people were not only dealing with a new Ada environment, but also with new design and development concepts. We quickly learned that a person who claims Ada experience, because he has written Ada pseudocode or has had minimal training, is not the same as an experienced Ada software engineer. Those who have only written Ada pseudocode lack the knowledge of Ada that comes from "doing battle" with the compiler and environment of choice. There is a natural learning progression - learning Ada syntax, semantics, and related software engineering concepts; designing and implementing Ada; redesigning and implementing Ada as knowledge of Ada and of the application increases. This learning progression is needed not because of a deficiency in Ada as a language, but, rather, because its robust nature is only fully understood after experimentation. Some problems can arise from the immaturity of Ada compilers and environments. However, our problems in this area were minimized due to our efforts to evaluate Ada compilers and environments beforehand.

Software Engineering Program

During the entire development of this project, the Vitro Software Engineering Program (SEP) was providing new tools to facilitate the execution of activities associated with each phase of the software life cycle, and the issue of when to insert new software technology into an ongoing project had to be addressed. The Software Engineering Program is a corporate initiative that established a dedicated group of software engineers who develop corporate software development guidelines, tools, and training for in-house use. The corporate training program developed by SEP personnel includes courses in software project planning, software engineering and Ada for managers, plus structured programming in Ada and modern software engineering disciplines for practitioners. The software engineering group developing the submarine weapon systems analysis software used these courses as basic training for all personnel. Select personnel also attended Ada-specific courses presented by various national organizations. As training and the project progressed, a small nucleus of the weapon systems analysis software engineers began to experiment with newly developed tools recommended by SEP in order to evaluate the feasibility of inserting their use in the software life cycle. A Technology Transfer Committee

was formed to interface with SEP and to recommend methods for transferring the most recent state-of-the-art software engineering practices to the weapon system analysis software engineers.

Tasking/Generics

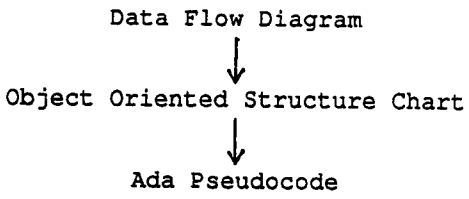
The transition to Ada was planned so as to minimize the impact on schedules. Initially, the use of generics was limited because the implementation of generics in the Ada development environment was not as mature as desired. Also, the use of Ada tasking was limited because it was perceived as being a high-risk and unnecessary use of the language for some applications. However, as expertise grew, tasking was inserted to facilitate efficient data handling for post-mission data processing and tasking became a key element of the real-time executive used to control data and device handling for different real-time systems analysis programs. Also, generics were inserted as soon as their specifications could be solidified, and they are now being utilized more often for new, related projects. When dealing with a robust language such as Ada, realistic choices regarding its use are important.

Software Development Strategy

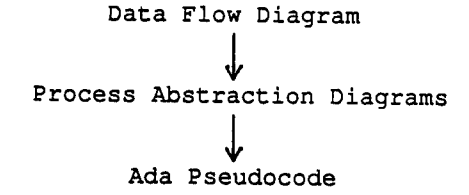
As part of the plan of designing for change, a modular development scheme with emphasis on loose coupling and reusability was chosen[5]. Analyst/Programmer teams consisting of one analyst assigned to coordinate and define software requirements, one real-time software engineer, and one post-mission software engineer were established. These A/P teams were instructed to design modules which could be used by both major software disciplines and to isolate in modules those parts of the software that were thought to be high risk items. A predictable high risk item was software used to analyze any new system that had not had a successful prototype or predecessor developed. Another high risk area was defined to exist in situations where the pass/fail criteria were still debatable. In some cases test procedures were in a state of flux, so top-level logic was designed to allow for every scenario being discussed. The A/P teams were an excellent vehicle for opening lines of communication between analysts specifying requirements and software designers. A/P teams could respond quickly to revisions and coordinate the definition of modules.

Data flow diagrams were used for the requirements analysis and were presented as part of the requirements review. For post-mission analysis software, the design process began with the use of Booch diagrams to create object oriented structure charts[1,2,3]. However, the real-time software engineers utilized G. W. Cherry's Process Abstraction Method[4] to begin their design process. They supported the Beta testing of AdaGraph, a tool which produces the relevant diagrams, and found these diagrams to be an extremely useful aid in reaching the next step of our design process, production of an Ada pseudocode design. Thus, the primary design processes consisted of the following steps:

Post Mission:



Real Time:



The Ada pseudocode and other design representation techniques, such as structured English outlines, assisted the analysts in understanding the total design and enabled them to contribute meaningfully to the design reviews.

Software Design Notebooks

Another means of addressing the dynamic software development environment was to adopt a Software Development Plan that included the use of Software Design Notebooks, based on the established practice of using Hardware Design Notebooks that track the entire

development process until an item is placed in production. This plan enabled us to develop a different documentation strategy which facilitated our ability to meet delivery schedules.

A policy was established that the software requirements document and the design document would never officially be revised once they were approved. Any necessary changes after approval were documented in the Software Design Notebooks and these changes were subjected to a separate review and approval process. Software Design Notebooks also contained a history of software development in terms of meetings held, action items accomplished, and milestones achieved. Any documents that related to the software while it was in the development process were kept here. These Software Design Notebooks were particularly useful when a software engineer had to be replaced in the middle of the software design process.

Improved Software Development Process

The first delivery of software was made in 1988, and similar software is now being developed for different ships' systems and also for a U.S. ally. The software development process continues to improve and the department Technology Transfer Committee has become an important management aid in improving the software development process. Ada and the design concepts needed to use Ada effectively are part of the improved process. Another part of the improved process is the availability of an integrated environment which supports the automatic generation of data flow diagrams, mini-specifications, data dictionaries, object-oriented structure charts, and Ada pseudocode. We are currently in the design phase of the development of software for analysis of different ships' systems. This new project is following the same guidelines described herein, but the tools mentioned above are being used. The result is more uniformity and clarity in design documents, plus an increased level of productivity.

Automated Structured Testing

The software engineering environment now also includes the Vitro Automated Structured Testing Tool (VASTT). One of the major purposes of this tool is to provide automated support for structured testing based on the methodology defined by McCabe. However, its utility extends to several different phases of the software development cycle. VASTT analyzes source code and structured representation of the design and produces a number of reports including:

- Unit Metrics Report
- Call Tree Report
- Test Paths Report
- Unit Re-entrancy Report
- Test Classes Report
- Data Cross Reference Report
- Integration Test Paths Report

This tool is being used to analyze the design of the new ships' system analysis software and also to analyze the software completed in 1988 that will be reused to support the U.S. ally contract. The latter is only in the preliminary design phase. The reports generated by VASTT help management decide whether or not software proposed for reuse needs revision and, if needed, where the most critical need exists. Of course, the primary use of the VASTT tool will be during unit and integration testing for both of the new projects.

Schedules and Metrics

During the first Ada project, while Gantt schedule charts and periodic software status reviews helped us maintain schedule goals, the collection of error metrics was relegated to production-ready software. Error forms that address errors discovered during all the various software development phases have now been developed and placed in the primary software engineering environment. The forms are available as a "window" that a software engineer can easily invoke while using workstations, and the use of these forms is a mandatory part of the process for new projects. Also, the updating of Gantt charts is now available to us as an automated tool, and data on hours expended and estimated percentage of subtasks completed is being inserted weekly to track the progress of the software development.

Automated Testing

During the first Ada project, validation, verification and testing folders were created for the software modules and integrated tasks/programs. When that software was revised, the folders were revised to include output

from tests that illustrated correct revisions and non-regression. After the beginning of the original Ada project, commercial test managing and performance coverage tools were investigated by the Technology Transfer Committee. Their insertion in the process was judged to be disruptive at that time, but these tools are being used in the new software development projects. These tools improve both quality and tracking of initial testing and regression testing by both software developers, independent test personnel, and the configuration management personnel.

Lessons Learned

Specific Ada usage lessons were learned and certain general guidelines emerged from managing the first Ada project. As new Ada projects progress, it is apparent that several practical management rules that occasionally become obscured during the development process, need to be reenforced.

- A mature Ada development environment is essential. Procedures for controlling the insertion of new tools and technology in the software life cycle must be established. For first time Ada projects, Ada, its various constructs, and Ada development environments must be treated as new technology.

- Sound cost/schedule estimation techniques must accommodate longer requirements and design phases for Ada software development. Investing time in the first phases of the process to produce stable designs is important. The reward for early, stable designs will be reduced resource utilization (recompilation issues), significantly shortened software integration and integration testing cycles, and the production of high-quality, easily understood, reusable, and easily maintained software.

- It is important that software managers understand the implications of using Ada in developing high quality, efficient, reusable and easily maintainable code. Good training, provided in a timely manner, is essential for both managers and the technical staff. Formal training must be supplemented by on-the-job training and exposure to expert Ada software engineers.

- It is advantageous to put in place the concept of maintaining a Software Design Notebook to facilitate passing software development responsibility to a new person, a normal occurrence during any project. The small investment made by the software engineer in maintaining a notebook can significantly improve the chances of completing the project on schedule.

- Informal verbal communication links between the experts in requirements definition, design, implementation and testing are needed to supplement documentation and formal reviews. For our projects, the concept behind Analyst/Programmer teams worked well in promoting verbal communications.

- Ada support for reuse is strong, but designing for Ada reuse requires an extra time investment during the initial phases of the project. Trade-offs will need to be evaluated.

- Tailoring the entire development process, and particularly the documentation process, to the project and the needs of the customer can control cost/schedule deviations.

Concluding Remarks

Our experience has shown that Ada can be positively managed to produce a successful product. In fact, in our case, Ada proved to be more useful and versatile than expected. Of the more than 360,000 lines of code, fewer than 100 lines had to be written in assembly language. We initially anticipated problems with Ada - Fortran and Ada - database utility interfaces, but the problems were resolved and the interfaces are viable. Ada portability issues were easily surmounted after the first attempt to port common modules from one computer to another. Even the bit manipulation needed to make data produced by Data General equipment compatible with the representation of that data on VAX equipment was written in Ada.

Ada slowed progress in the beginning because personnel were just learning the language and the principles behind it.

Also, Ada environments and compilers were moderately immature. However, even in the worst cases, Ada provided the means for developing flexible software that was easily integrated and easily revised without excessive recompilations. The resulting software is understandable, reliable, and maintainable. Ada can help the software community "design for change", and creative management can utilize Ada to meet delivery schedules.

References:

1. Booch, Grady. *Software Engineering With Ada*. Benjamin Cummings Publishing Company, 1987.

2. Buhr, R. J. A. *System Design with Ada*. Prentice-Hall, Inc., 1984.

3. Bulman, David. *Ada Software Engineering Workshop Workbook*. Pragmatics, Inc., 1984.

4. Cherry, George W. *Parallel Programming in ANSI Standard Ada*. Reston Publishing Company, Inc. (Prentice-Hall), 1984.

5. Houpis, Constantine H., and Gary B. Lamont. *Digital Control Systems Theory, Hardware, Software*. McGraw-Hill Book Company, 1985.

6. McGouan, Clement L., and John R. Kelly. *Top Down Structured Programming Techniques*. Petrocelli/Charter, March 1975.



Sandra J. Fee is Assistant Group Supervisor of the Simulation and Advanced Software Engineering Group in the Test Department of Vitro Corporation. She has been a software engineer with Vitro Corporation for 11 years. She was previously employed by Autonetics (Anaheim) and Argonne National Laboratory. Ms. Fee graduated *summa cum laude* from Avila College (Kansas City) with a B.A. in Mathematics and holds an M.A. in Mathematics from the University of Kansas (Lawrence), where she was elected Phi Beta Kappa.

The mailing address for Ms. Fee is:

Sandra J. Fee
Vitro Corporation
TDD, Bldg 12-2009
14000 Georgia Ave
Silver Spring, MD 20906-2972

TRANSITION TO AdaBIS

Claton J. Hornung

Computer Sciences Corporation - ATD
Indianapolis, IN 46216

ABSTRACT

The emergence of Ada Business Information Systems (AdaBIS) gives software engineers the control and technology needed to facilitate the development and maintenance of cost effective Management Information Systems. Standard Army Financial System-Redesign (STANFINS-R) is the first large Business Information System in the United States using the Ada Technology. The completed Ada software system consists of approximately 2 million lines of code. It operates on large IBM compatible mainframe systems at Regional Data Centers using CICS and a relational database. The system provides worldwide accounting processing functions through the use of front-end smart terminals with interactive on-line processing. This paper will provide the software engineer with an understanding of potential AdaBIS problem areas and the methods used to overcome software development obstacles.

AdaBIS Transition

The first AdaBIS software development effort has provided the experience needed to develop lessons learned and recommended approaches to solutions in the AdaBIS Environment. The lessons will include the following factors that positively effect the development effort; The knowledge that Total Quality Management can be a positive influence on the AdaBIS environment and that the development must be a complete team effort. Planning using an Integration Build concept is critical to the success of a large AdaBIS software development project. AdaBIS project planning should consider using a longer period of time for design and shorter periods for coding and integration. All members of the AdaBIS project staff do not need to be super technicians, but a blend of skills must be used to make a successful development team. If training is effective, COBOL software developers can be trained to become quality Ada Software Engineers. In the design area, a mix of Structured Top Down Design at the higher level and Object Oriented Design for the detailed level makes an effective design method for AdaBIS projects. Also it was proven that compiler maturity and the Ada Programming Support Environment (APSE) must be positioned early in the life cycle phase to avoid negative impacts on productivity.

CSC's experiences and understanding proves that software for Management Information Systems (MIS) can effectively be developed in the Ada environment. Lessons learned in AdaBIS management, planning, training, design, and development are provided as a means of supporting the transition to an AdaBIS environment.

Effective Management

Management must maintain control of the total life cycle software development process in order to deliver an on-time quality product. A Total Quality Management effort includes emphasis on the team development approach with management control and reviews ensuring that a quality product is developed on time and within cost.

Total Quality Management (TQM)

Ada Business Information Systems can be successfully developed by using a Total Quality Management strategy. TQM consists of a continuous process that includes all members of the development team in an integrated effort of improving performance at every organizational level. An AdaBIS TQM effort is effective when it includes the following factors: requirements for user satisfaction, a focus on problem prevention instead of problem occurrence, quality as a continuous productivity

improvement process, innovation in technology and system process development, and insuring that the TQM effort involves the total development team approach. Each factor is discussed below.

User Requirements

A quality monitoring process will insure that user requirements have been designed and developed into the AdaBIS and that the software programs are traceable back to the functional requirements.

Problem Prevention Process

Management must cause quality to materialize as a problem prevention process instead of appraising the developed system in a mode of post development problem determination. Standards development should take place at project initiation as a cooperative effort with the development staff taking the lead. The early identification and preparation of risk mitigation plans will assist in the prevention of problems that may later impact the AdaBIS development. An understanding of the potential risk factors will guide the use of project controls in monitoring critical tasks, thus providing for effective problem prevention.

Productivity Improvement

Quality, with defect free software, can be achieved though a continuous process of productivity improvement. Project leadership should strive to meet the goal of quality on-time deliverables through increased productivity. The use of software productivity metrics will provide the manager with project evaluation information and techniques to help gauge the TQM productivity effort.

Technology Influences on TQM

The goals of TQM are to increase quality and productivity. A way to achieve this is through innovation and the resulting improved technology or the use of new technology. New and improved technology can take many forms, such as, new productivity oriented tools, methods that facilitate effective design and development, and new quality procedures such as the implementation of software quality metrics.

Team Development Approach

In an AdaBIS development project it is effective to use a team approach where the staff has specialized expertise, with the design and development lead changing hands within the team, but the total team taking the project all the way from initial design to implementation. Through use of this concept the learning curve is reduced and communication is facilitated. With development being the top priority of the total staff, integration problems are reduced. And since the team is seamless, synchronization problems are limited as the team members move from one development stage to the next. Leadership must insure that all members of the team know and understand the project's goals. This tends to keep all members productive and focused towards developing a quality system.

Management Control and Review

The AdaBIS Software Development Manager must have control of the development processes. Before control can take place an understanding of the work effort, the tools and methods used, the work status, and the problem areas must be obtained. Most of these items will materialize only with detail involvement in the project. As a mechanism of understanding and continued assessment, the manager should, on a regular basis, perform a technical management review of the project. This review will allow direct interaction between the manager and the software engineers. The Project Manager should also spend a portion of his time each day examining the impact of future events. Current results should be measured against both planned and projected milestones. The review of the project's current status against planned and projected performance and milestones will improve management control and insure the project is headed in the right direction.

Planning For AdaBIS

Planning is a very important phase of the AdaBIS software development life cycle. Planning should include establishment of the training program, selection of the development environment, risk evaluation and mitigation, comparison of actual performance against planned and projected performance, and the initial planning for the integration build approach. Adequate planning will facilitate a smooth

development process that will capitalize on lessons learned from earlier AdaBIS software developments.

Planning the AdaBIS Training Program

It is a fact that the majority of COBOL programmers can become effective Ada Software Engineers. An AdaBIS Training Program should be developed based upon the "Train the Trainer" concept. The program should include the use of a variety of learning tools and be tailored to the specific Ada development and application environment. All staff members, including managers, should be trained in the Ada Programming Language with the classes containint a high degree of practical exercises tailor to the application being developed. The training program should also provide information on the design approach and methodology to insure that everyone can capitalize on the known methodology which will result in a more efficient quality product.

Planning for Risk Evaluation and Mitigation

Assisted by the use of the AdaBIS Manager's Review Guidelines, located at the end of this article, the Project Manager should identify all potential risks during the project's life cycle. Each risk should be evaluated regarding the degree of risk, i.e. high, medium, or low and associated with its corresponding life cycle phase and milestone. The impact of each risk should be assessed and the proposed mitigation action identified and included in the project's planning. Planning should allow for the assignment of responsibility for the risk and its mitigating actions. As with project goals, members of the project team should be aware of risk areas and the mitigating actions expected to minimize or neutralize the impact.

Planning for the Measurement of Project Performance

The impact of AdaBIS training requirements causes progress measurement to take on more meaning. Until developing skills become more practiced, time-lines may not hold to the traditional measurements. Although the typical AdaBIS Project Manager may be buried in the details of day to day activities, he should plan to divide his time so that a portion of each day is spent looking at future events. Current results should always be compared to the plan. Just as importantly, the current status of the project should be measured against future aspects of the plan. This process gives the manager the ability to measure current performance against planned and projected performance. With this ability the manager may employ a technique similar to management by exception, allowing him to correct deficient performance and to exploit superior performance, thus maximizing his control over the project and in meeting future milestones.

AdaBIS Productivity Dependencies

During the planning process, before the AdaBIS schedule and milestones are established, the project productivity dependencies should be analyzed and evaluated. Dependencies include size/complexity of the project, the degree of reusability of available AdaBIS code, the volatility of the functional requirements, the documentation requirements, the availability of an Ada experienced staff and the maturity of the development environment.

Integration Build Approach

It is important in any development effort to avoid "the big bang" or single development approach, and adopt the build approach during software development. The build approach is a concept where smaller pieces of the system are developed in sequential modules and then integrated into units called builds. The process allows the system to be built and tested on an integral basis so that major design and implementation flaws can be detected and resolved early. The build approach focuses on selecting subsets of requirements with a separate development effort for each subset. During integration and testing a building block approach is used with the programs being included in all successor builds with the last build containing the complete product. This concept provides a good environment for the team approach to software development and also provides flexibility in staffing and project management.

The build strategy should start with the first build as simple as possible and using it to familiarize project staff with standards, coding models and the environment. The first builds should test structural design issues, database and operating system interfaces, test and configuration control tools and software

that will control major system functions which will be included in later builds. The next to last build should include the last part of the clients operation requirements, with the last build being small and simple. Frequently the first build includes a menu system with the last build being non critical items that were not in a readiness state during the scheduled build.

Ada Development Consideration Factors

Use of the Ada programming language will provide for a longer analysis and design period (55 % of the life cycle) but shorter code (25 %) and integration (20%) phases. Also Ada development efforts should include consideration for Ada start-up cost and time and the need for training all staff members for the initial Ada project. The common code (code generation and the ability to compile units separately) features of Ada should be considered when estimating the initial Ada development project. But the reusability features of Ada will not provide a positive impact to development planning until successive Ada development efforts.

Establishment of the Ada Development Environment

One of the initial planning tasks is the definition and selection of the development environment. A review of Ada Programming Support Environment reference material will provide the planner with an idea of the magnitude of the environment required for software development. Experience proved that the Rational Development Environment provided an effective APSE. The AdaBIS planner should be very careful to provide ample storage and to ensure that adequate computer power will be available. Add 25 % to the initial estimate of the project's computer power to avoid a short fall in total requirements.

Use of CASE Tools to Increase AdaBIS Productivity

Through the use of CASE, approximately 60 percent of the code on the first AdaBIS project was generated. Without CASE this productivity would not have been possible. In order to provide total life cycle capability for the CASE tool, a database that serves as the common storage area for all functional and physical requirements was used to perform the process of code generation. Also, the common database is key in providing traceability from functional requirements to actual code and back. Important to the effectiveness of the integrated CASE in the AdaBIS environment is the compatibility among the tools, such as the requirements database and individual tools within the Ada Programming Support Environment and the ability of the CASE tool set to satisfy the total life cycle software development requirements.

Ada Resource Leveling

A common myth among Ada developers, is that every staff member must be an Ada "SuperTechi" or super-proficient in the Ada technology. This first AdaBIS development is proof that all staff members do not have to meet that expectation. In fact, one's professional background and experience is a function of super-proficiency in applying the technology. And, while individual performance is important, collective efforts are much more so, since the most effective and productive staff would have a combination of talent levels. SuperTechi's are needed to build the specification, code generators and models, with mid or junior level programmers to do the majority of the application development coding. Also it is important that the AdaBIS development staff have a blend of experience that include a strong business systems analysis background and some staff members having extensive database experience. In staffing an AdaBIS project, consider a blend of software development expertise which is based on a foundation of background and experience in structured programming and design technologies.

Importance of Ada Consultants

Because the first AdaBIS was at the "bleeding edge" of technology, it was important to use the services of an experienced Ada consultant. One who had a background in the COBOL language and in business systems in order to verify design strategies and provide Ada training support. Training support was in the form of advanced subjects and working "one on one" to maximize any prior COBOL experience. Without Ada consultants such as Dick Bolz, the learning curve would have been more of a challenge to overcome. Consultants have played a critical role in the start-up of the first AdaBIS project and should be considered during the

planning phase of an AdaBIS start-up effort. This first AdaBIS project has provided CSC with a wealth of exportable AdaBIS expertise. This expertise has a bedrock of hands-on Ada programming experience in a structured environment, the lessons learned in transitioning from a COBOL environment, and comprehensive knowledge in business systems applications.

AdaBIS Development Standards

Tried and proven standards are important in any development effort. In an AdaBIS development it is equally important that the standards be selected by the project team responsible for the software design, development, and implementation. The QA staff should be involved in assuring that the most recent version of the necessary standards are obtained. The basis for selecting the design standards, coding standards, documentation standards, and testing standards is the project plan. The importance in providing each member of the project team with a thorough understanding of the selected standards and procedures cannot be over emphasized. The project manager that takes the time to do this is paid back in terms of better or complete project control and enhanced productivity.

AdaBIS Training

Training is always important but for the first-time AdaBIS project it is especially important to have an effective Training program. The program should be developed under the concept of initially "train the Trainer", provide a variety of learning tools, tailor the training program to the Ada development environment and the application, have extensive practical exercises and experiences and train all staff members including management.

AdaBIS Training Courses

Training can be separated into three learning patterns; management, Staff, and APSE training. Management training should provide all management with an understanding of Ada, it's impact on design strategy, the Ada Programming Support Environment and software engineering principles. Staff training should consist of an introduction and intermediate course in the Ada program language and selected advanced Ada topics. The APSE training program should cover the fundamentals and advanced techniques of project management, system operator, and system administrator in the APSE environment.

Fundamental AdaBIS Training

The training staff should provide an entry level training program that includes software engineering principles, object oriented design, APSE training, and Ada programming syntax. Teaching methods should include classroom lectures based upon actual experience and a large degree of practical exercises. Computer Aided Instruction (CAI) should be used to enhance the training program. AdaSoft CAI was used because of the ability to insert AdaBIS programming experiences in the course presentation. Ada Videos were determined to be of little value due to their inflexibility and cost. CAI strengthens the Ada training program and was especially important to those desiring or requiring additional training or more time.

Mind-set Transition

It was determined that COBOL programmers who had never worked with a structured programming language such as Ada, PASCAL, C, etc., had preconceived notions about Ada and a strong religious devotion to the COBOL way of design and programming. To overcome these notions, similarities were used to change this focus. As an example, data typing was introduced by relating it to COBOL picture clauses. The objective was to provide experiences to the programmers to transition their mind set to the state that would make them successful AdaBIS Software Engineers. The training demonstrated that concepts discussed in terms of a COBOL programmer facilitated the mind set transition. After this experience the programmers became more receptive to new ideas and appreciated the new found power of Ada and associated software engineering concepts.

Fundamental Computer Science Concepts

Many students demonstrated a lack of fundamental understanding in the basic concepts of data structures and basic design. Not only did training have to focus on these concepts, but it became necessary to focus on software engineering principles and goals as well. Building these fundamental concepts into the Ada based training program produced a more firm foundation which can be effectively

used with Ada design and programming and most other third and fourth generation languages.

AdaBIS Training Lessons Learned

Experiences with the first AdaBIS Training program demonstrated that it was important to tailor the material to reflect the project functionality. Separate classes should be provided to those having a different language background or level of skill. Extra reading assignments should be required to expedite the learning process. However, time in the classroom should not be crammed. There must be provisions for extensive practical applications. The use of CAI should be carefully planned and used sparingly as a supplement to the training program.

AdaBIS Design

It is particularly important that the design be approached from the physical aspect and that it be readily traceable back to the functional requirements. The AdaBIS design used a combination of both structured top down and object oriented design. The physical approach to design laid the foundation for the use of generic's in producing common module design easily and efficiently.

Requirements Design Database

The use of a database to store all functional and physical design facilitated productivity gains due to the ease of on-line access and the ability to use the database as a source of all documentation and code generation. The Requirements Design Database also facilitated the tracing of all process and design data back to the functional requirements.

Maintaining Functional Integrity

It is important for the AdaBIS developer to have the ability to track from the functional baseline to the production system, and back, to insure that all functionality has been considered and implemented. AdaBIS uses modern development models that stress the tracking of functional and physical processes and maintain functional integrity. In the process area this traceability is from functional procedure to physical program units and in the data area the traceability is from logical to physical records and data items.

Design Approach

Most frequently, the design of a system has common functions and data in both the logical and physical designs maintaining a functional decomposition look. Since the physical design of the system should be from the point of view "how to effectively design from the standpoint of the computer system's physical capabilities", it would be most productive to design the system based upon the physical concept of how the data can most effectively be processed by the computer system. In developing an AdaBIS the most effective physical design process is to break the system out by major physical processes to form subsystems. The physical subsystems include the following: Interactive Data Store Maintenance, Interactive Information Generation, Batch Data Store Maintenance, Batch Information Generation, and Auxiliary Support. This is the way common physical functions or programs are grouped together into physical subsystems. This physical breakout of processes facilitates the use of common code and generic's in the APSE environment. The physical design is only visible to the designer and developer. The user will view the system from an access control menu. The menu still provides a functional decomposition of the processes to be performed by the user.

Software Engineering Principles Improve System Productivity

The quality of the AdaBIS design, as with any other design, is determined by the effective implementation of sound engineering principles. The AdaBIS development environment supports a focus on software maintainability which will enhance system productivity during operations and maintenance The structured environment of AdaBIS actually facilitates adherence to sound software engineering principles which improves the quality of the developed programs. The quality programs of the AdaBIS design, in turn, raises the level of attainment towards the software engineering goals of modifiability, efficiency, reliability, and understandability. The software engineering principles that facilitate development of a sound AdaBIS design include, abstraction, information hiding, modularity, localization, uniformity, completeness, and confirmability.

Effective AdaBIS Design Mix

Total object oriented design has been used in most Ada real time or embedded computer system developments. CSC found it most effective to use a structured top down approach in the design of the major physical subsystems and then to develop the design for each program within a particular physical subsystem using object oriented design. This is where the data store or the I/O object is the focal point with the related processes supporting the particular object program unit. This blend of structured top down and object oriented design techniques provide a strong base for effective system performance and high productivity during software development.

Reusability in the AdaBIS Development

The development of reusable components is maximized in an AdaBIS project. This is achieved through Ada's ability to develop program units (packages, task units, generic units, and subprograms) and compile them separately. The program may be designed, written, and tested as a set of independent software components. Reusability in the design approach should not be the driving force in the development effort. The first emphasis should be placed on developing an effective system within time and cost constraints. When time permits, the reusable software should be catalogued according to acceptable standards.

Development Lessons Learned

The AdaBIS project manager must be aware of high impact areas in planning for an AdaBIS development. CSC has a wealth of early experience in AdaBIS in such areas as: the need for mature compilers, ineffective database and transaction processor monitor, a need for strong emphasis on configuration management; and many other areas, such as, problems related to porting from the development to the target environment. These are some of the AdaBIS problem areas that must be overcome during the early stages of the development process.

Compiler Maturity

It is important that the AdaBIS development project use a mature Ada compiler that has mature database and transaction processor interfaces. A certified compiler does not mean a 100 % implementable compiler as it relates to the planned AdaBIS application. For example, most compilers have problems related to the implementation of the generic's features of Ada even with certification. The selected Ada compiler must support a production quality teleprocessing monitor for it to be acceptable in the AdaBIS environment.

Database Interface

It has been observed during the AdaBIS project that an Ada call to an assembler pragma is an effective means for an Ada implementation of relational databases. The integration of the Ada-SQL binding tends to complicate the effective use of the interface. Currently, many DBMS do not have the Ada/SQL interface capability. Also it is important to note that Ada's own I/O handing is not adequate for normal database transaction handling. Performance evaluation and tuning is necessary for an effective Ada interface with CICS.

Transaction Processor Monitor

Ada can effectively be interfaced with CICS, but performance evaluation and tuning will be needed to make the services an effective use of computer resources. The selected Ada compiler must support a production quality teleprocessing monitor for it to be acceptable in the AdaBIS environment.

AdaBIS Development Foundation

The production of Ada programming packages in the Ada Programming Support Environment must be based on a strong foundation of lowest to highest level packages. There is less dependency between programs as we ascend the inverted pyramid of AdaBIS program generation. The lowest level consists of data items common throughout the entire application and is accomplished by completing the base data packages. Upon this foundation, the product packages (consisting of screens, I/O interfaces, and reports), the application data type, and database view packages are layered. Last are the process application packages. Programming changes to the body of the program unit will not cause automatic recompiling of developed code. But specification changes on lower level units will cause a recompilation ripple impact to related programs throughout the lower level programs. It is important to stabilize the lower and middle level packages as soon as possible, in order to

limit unnecessary recompilation of developed code.

Configuration Management Importance

The use of Ada requires a great emphasis on configuration management. The extensive amount of program units that may be "withed" by other units and the need to develop detail compilation scripts are a few of the many reasons why an integrated configuration management system is important to the success of an AdaBIS software development project.

Target Environment Concerns

When code is ported from the development to the target system, a new series of problems usually appear. In the target environment it is very important to have an effective source level debugger. Along this same line, the debugger should be mature, effective, easy to use, and take up only limited space on the system. Performance of the system in the target environment must be monitored closely with an effective on-line performance monitoring system. AdaBIS executable size is unusually higher than expected, with the causes being nested generic's, private types, and excessive packages. On the positive side, execution speed is within acceptable standards and faster than expected.

Testing Lessons Learned

It is important to start software development testing as early as possible during the first build. Since the first build should include the first cut of the menu system, the testing should be performed in a top down structured manner. Pretest activity should be comprised of two phases, APSE testing and environment target testing. APSE testing will permit problem corrections before porting to the target environment, where continued problem correction will facilitate the SDT. It is important to conduct pretest activity as a means of increasing test efficiency. If parts of the system are difficult to test, the system may also have other software engineering related problems that might later impact operation and maintenance effectiveness.

Independent Testers

Coordination with independent testing organizations should begin no later then during pretest activity. The independent tester's knowledge of the Ada software development features should not be assumed. Successful implementation of an AdaBIS means advanced planning which includes insuring that the independent tester is prepared to do the testing. Hands-on experience will provide the independent tester with a real appreciation of the rippling affect of code changes to the specifications of an Ada program package.

SUMMARY

Lessons Learned from the first AdaBIS software development are many and penetrate every phase of the software development life cycle. CSC's experience proves software development staffs, regardless of programming language background, will receive a high degree of job satisfaction from working in an AdaBIS software development environment. It was also proven that Ada can be used effectively in the Management Information Systems environment to provide efficient and cost effective systems on time. The AdaBIS development is enhanced through a quality support system of CASE tools, effective project management and leadership, and a total team effort.

Total Quality Management has a positive influence on the AdaBIS environment when the development is a total team effort. An Integration Build concept should be considered as critical to the success of an AdaBIS software development project. During project planning a longer design phase and shorter periods for coding and integration should be implemented. In planning for the AdaBIS staff, a blend of skills will create a successful development team. The team should see the AdaBIS project through to completion. COBOL software developers can be trained to become effective Ada software engineers. Structured top down design at the higher level and object oriented design for the detailed design, is an effective method for AdaBIS projects. Compiler maturity and Ada Programming Support Environment (APSE) must be considered early and positioned in the life cycle phase before they can have a negative impact on productivity.

CSC's experiences have proven that Ada software development can be effective in the Business Information Systems environment. Since AdaBIS development of commercial systems will become a common occurrence during the next decade, it is

important and necessary that software engineers take advantage of these first AdaBIS lessons learned.

Acknowledgement: Extra words of appreciation must be provided to Mr. John S. Medley for his assistance in the preparation of "Transition to AdaBIS".

AdaBIS Manager's Review Guidelines

It is very important that the project Manager perform a monthly review of the AdaBIS progress. A suggested outline guide is provided below.

AdaBIS Review Guidelines

1. Planning and Organization

   a. Project plan up-to-date and realistic.
   b. Task breakout established and realistic.
   c. Milestones attainable.
   d. Risks identified and mitigating actions current.
   e. Staff organization effective.
   f. Resources available, adequate, and effectively used.
   g. Client coordination effective.
   h. Team members adequate, knowledgeable, and trained.
   i. Manager achieving project visibility.
   j. Project measurement effective
   k. Project measurement include a future perspective.
   l. Manager's goals known to all team members.
   m. Standards and procedures understood by entire staff.

3. Design

   a. Design guidelines understood and followed.
   b. Physical design approach effective.
   c. Both top down and OOD methods in use or planned.
   d. Software Engineering principles in practice.
   e. Design implementation effective.
   f. Physical environment constraints considered.
   g. Models/constructs effectively used.
   h. Design review process effective
   i. Stability of functional and physical design.
   j. Effective use of CASE tools.
   k. Effective means to resolve design problems.
   l. Design include built-in controls, data integrity, and audit trail.

4. Development

   a. Use of prototype.
   b. Procedures documented, effective, and followed.
   c. Transition from design to development effective.
   d. Use of code generation and testing tools.
   e. Support environment effective and capable.
   f. Testing responsibility assigned/procedures in place.
   g. Target environment known and included in plans.
   h. Consultant services available or required.
   i. Interfacing technique established.

5. Testing

   a. Test planning complete and effective.
   b. Unit testing methods.
   c. System testing methods.
   d. Error recovery testing methods.
   f. Stress testing methods.
   g. Performance testing methods.
   h. Coordination for independent testing.

6. Documentation

   a. Single source/repository for system documentation.
   b. Applicable standards known and understood.
   c. Documentation schedule.
   d. Means to produce quality documentation.

7. Standards/Procedures/Reviews

   a. Standards selected, followed, and effective.
   b. Procedures effective and known to team members.
   c. All required standards and procedures available.
   d. Quality Assurance effort under manager's control.
   e. Quality Assurance effort effective.
   f. CM procedures in effect and under manager's control.
   g. Project progress reviews sufficient and effective.
   h. Productivity Improvement on-going.

A Comparison Between Functional Decomposition and Object-Oriented Design Methodologies — A Case Study

Ron Fulbright

Fluor Daniel

Abstract

This paper compares the original design effort of an inventory system, using a functional decomposition methodology and FORTRAN '77 as the implementation language, and the redesign effort of the same system using an object-oriented methodology and Ada as the implementation language. The redesign was done as an effort to assess the differences in the use of and potential benefits of an object-oriented development methodology. A phase by phase description of both development efforts is given and a comparison between the two development efforts is made noting advantages and disadvantages of using functional and object-oriented design techniques.

Introduction

Fluor Daniel is a private-sector engineering and construction company, and as a client-based company, we survive by providing services to our clients that are superior to those of our competitors. One service that we provide is software development particularly in the areas of process control systems and management and information systems. We are continually searching for ways to improve our software development abilities. If we can offer better software for less money and effort, we will stay ahead of the competition.

We currently use the "classical" software development model along with functional analysis and design techniques, commonly known as the "Yourdon technique" and referred to hereafter as functional decomposition. Recent years have seen the emergence of object-oriented analysis and design techniques as a promising way to develop software.

This paper examines the differences in functional decomposition and object-oriented techniques by focusing on a case study. The inventory system described in this paper was originally designed and implemented using functional decomposition and was implemented in FORTRAN '77 running on a micro PDP 11/73 under the RSX 11M+ operating system.

To analyze the differences between a functional decomposition and a object-oriented methodology, and to determine the benefits of using an object-oriented methodology, the system was redesigned using object-oriented techniques with Ada [1] as the target implementation language. The two development efforts are discussed and the resulting designs are compared.

General System Description

The inventory system was originally designed for use in the packaging area of a major paper products manufacturer. The system has three primary purposes:

1) Count the number of cases of each type of product produced
2) Direct cases to their proper destination in the warehouse
3) Provide operators with current inventory and routing information

Each case is encoded with a bar code that is read by bar code scanners mounted along the conveyor system at strategic points. There are two types of bar code scanners used in the system. The Type 1 scanners maintain a running count of the number of cases of each product. The counts are maintained in a table in the scanner's memory and are indexed by case code. The Type 2 bar code scanners maintain a list of case codes. When a case code on a passing case matches one of the case codes in the list, a hardwired signal is raised that engages a divert gate on the conveyor. This causes the case to be redirected and in this way, the case is directed to its proper location.

A host computer polls the Type 1 scanners periodically to obtain inventory counts. The inventory data are merged with static product description information, maintained on the host computer, and are stored in a file called the inventory file. This file is sorted on various keys and can be displayed or printed on request.

The host computer also allows the operator to view and change the static product information which is stored in a file called the product description file. The case destination can be changed by editing this file and when changed, the host computer changes the table of case codes maintained in the Type 2 scanners accordingly.

The system also generates printed reports on request and an end-of-shift report automatically. The page of data currently being displayed or the entire file may be printed on the request of the operator. The end of shift report provides a summary of the inventory counts of each case for that shift.

All communication with the scanners and operator terminals is via RS-232 using an ASCII character-

based protocol. The protocol used for scanner communication was custom designed for the application. There are two operator terminals and a printer in an office area near the host computer and a remote terminal on the warehouse floor.

The Functional Decomposition Design

The following is a description of the original design effort using functional decomposition techniques. An overview of the original methodology is given first followed by a phase by phase description of the development effort.

The Classical Software Development Model

In the original design we used the "classical" software development lifecycle model. The classical model is composed of five phases:

- o Requirements Analysis Phase
- o Design Phase
- o Coding Phase
- o Integration and Testing Phase
- o Maintenance Phase.

In the requirements analysis phase, we engage in dialogue with the client to determine the client's needs. It is the purpose of this phase to focus ideas, define the scope of the system, and express the functional requirements of the system in a concise manner. The result of this phase is a formal requirements document.

The design phase is actually composed of the preliminary design phase and the detailed design phase. In the preliminary design phase, the hardware and software components required to fulfill the requirements are identified. The features and requirements of the system are captured in top-level modules that are later broken down further. The detailed design phase consists of designing the software components and expressing this design in a manner that facilitates coding and testing. In this phase, the top-level modules are broken down into smaller modules and pseudocoded. The process of decomposition repeats until single function modules are produced. The result is a hierarchial structure of modules ranging from the top-level modules down to the lowest level, or primitive, modules.

In the coding phase, the designs expressed in the design phase are translated into executable code. Coding takes place in a top-down and a bottom-up manner, sometimes called the "sandwich" development approach. The top level modules are coded first with subordinate modules stubbed out. The primitive modules are coded and tested next. Coding then proceeds laterally through the mid-level modules.

The integration and testing phase is closely tied to the coding phase. Integration involves the functional joining of two or more modules. Testing involves verifying the functional correctness of the integrated modules per the original requirements. The coding phase and the integration and testing phase proceed in a highly parallel fashion. Usually, a group of modules are coded, integrated, and tested as a unit followed by the coding and testing of other groups of modules. Finally, the individual units are integrated.

In the maintenance phase, bugs are fixed and changes to the software are implemented in order to accommodate new or changed requirements. The maintenance phase lasts as long as the software is in use.

The Original Requirements Analysis Phase

Initial dialogue with the client revealed the major functions of the inventory system. The initial dialogue included a visit to the plant site, conversations with the plant operators who would use the system, and reviews with plant engineers. The result of this effort was a written requirements document that expressed the system in functional terms. The interaction of the operators with the system was described in some detail also. The content of this requirements document is similar to the general description earlier in this paper but is more detailed.

A list of functional requirements was made. This list initially contained every function that was identified during the requirements analysis phase. It was possible to group many individual functions from this initial list together under single, higher level functions. These higher level functions represented the major functions of the system and were identified as follows:

1.0 Communicate With User
- -display menus and data screens
- -perform actions requested by user

2.0 Edit Product Description File
- -find a record
- -add, delete, change a record

3.0 Communicate With Type 1 Scanners
- -read inventory counts
- -clear inventory counts

4.0 Communicate With Type 2 Scanners
- -read divert table
- -add,delete,change table entry

5.0 Perform 15-Minute Inventory Update
- -read latest inventory counts
- -update existing inventory file
- -add new inventory file records

6.0 Perform End of Shift Update
- -delete outdated inventory records
- -generate upload file
- -generate end of shift report

7.0 Communicate With Printer
- -print selected data screens
- -print selected files
- -print end of shift report

The major functions of the system were captured in a first level data flow diagram shown in Figure 1.

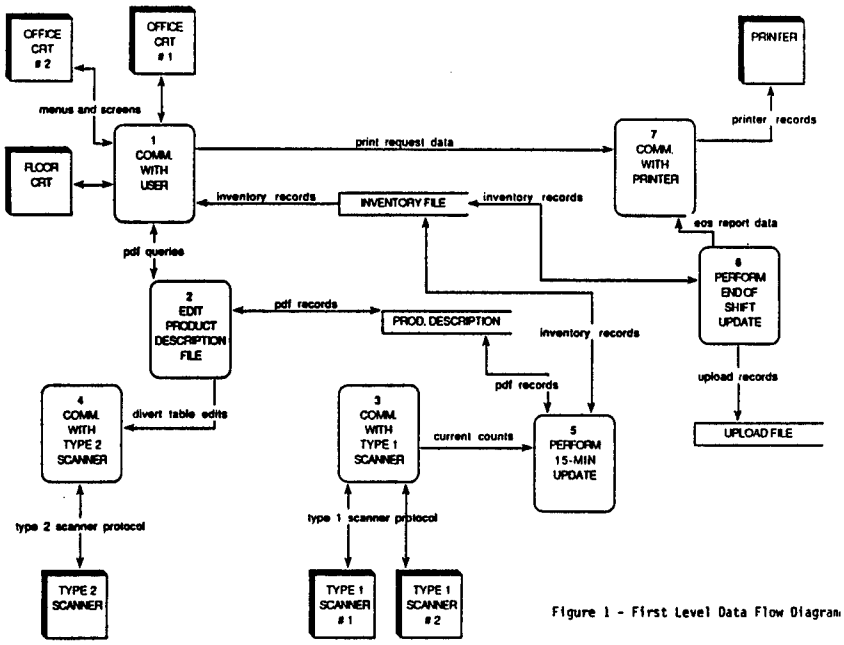


Figure 1 - First Level Data Flow Diagram

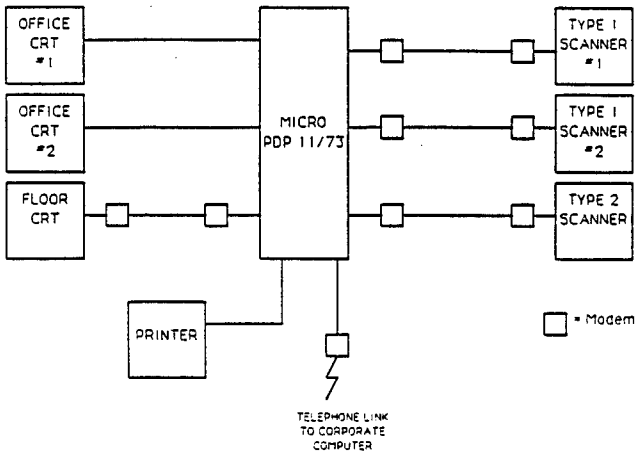


Figure 2 - System Diagram

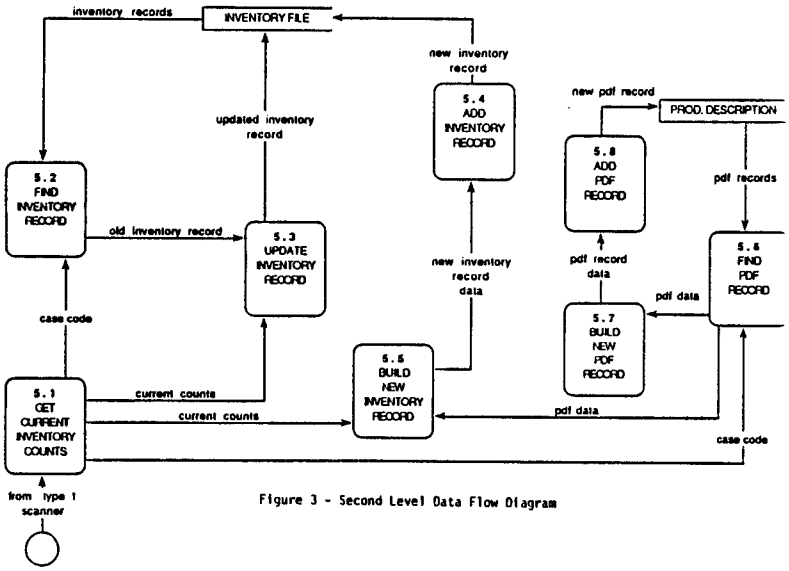


Figure 3 - Second Level Data Flow Diagram

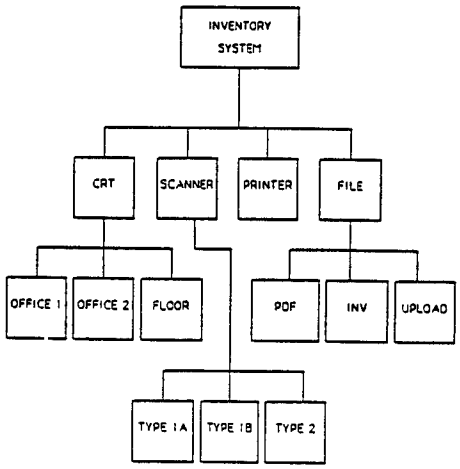


Figure 4 - Object Hierarchy Diagram

The Original Preliminary Design Phase

The first effort in the preliminary design phase was to decide the hardware necessary to implement the system. The use of bar code scanners was decided at this point as well as the need for a multi-user, multi-tasking host computer. The use of FORTRAN '77 was also decided at this time in the project. The configuration of the system was expressed in a system diagram that is shown in Figure 2.

The next effort in the preliminary design phase was to develop the overall structure of the software. The first task was to develop a set of second level data flow diagrams. This was done by breaking down the major functions, expressed in the top level data flow diagram, into smaller modules. These smaller modules, functions in their own right, were expressed in data flow diagrams as subordinates to the top level functions. It is important to note that the decomposition of the top level modules was performed on a functional basis. An example of a second level data flow diagram is shown in Figure 3.

The second task was to pseudocode the top level and secondary level modules. The general control structure of each module was captured in the pseudocode. Calls to other modules were identified in the pseudocoding process which defined the dependency between modules. An example of the pseudocode is shown below:

```
MODULE Perform 15-Minute Update
  get latest counts from scanners
  LOOP UNTIL all cases are processed
    find inventory file record
    IF case code found THEN
      update inventory file
    ELSE
      find product description
      CASE
      WHEN case code found =>
        build new inv. file record
        add inventory file record
      WHEN PDF data is incomplete =>
        build new inv. file record
        flag inv. rec. as incomplete
        add inventory file record
      WHEN case code not found =>
        build new PDF record
        build new inv. file record
        flag PDF record as new
        flag inv. rec. as new
        add PDF record
        add inventory file record
      END CASE
    END IF
  END LOOP
END Perform 15-Minute Update
```

The third task in the preliminary design phase was to identify the primitive modules. The primitive modules are those modules called one or more times from one or more of the higher level modules. The primitive modules form a set of utility subroutines which lie at the bottom of the module hierarchy.

The first, second and primitive level modules were represented in a hierarchial diagram.

The Original Detailed Design Phase

In the detailed design phase, the structure of the system took form. The top level functional modules were assigned to tasks, file formats were designed in detail, user interface screens were designed, and the protocols were developed for the bar code scanners. (Note that the mention of tasks here refers to tasks in the RSX 11M+ operating system environment and not to Ada tasks.) The actual code for each module was also sketched, following the pseudocode, in FORTRAN.

In general, a task was created to handle each one of the top-level modules. The result was a task to handle communications with each peripheral plus tasks to handle the 15-minute and end of shift updates. The tasking structure was essentially the same as the first level data flow diagram shown in Figure 1.

The detailed design of the system was captured in a series of diagrams. The tasking diagram was developed as well as diagrams showing the file formats and the communication protocol.

The Original Coding and Integration Phases

Coding proceeded in a sandwich fashion. The first components coded were the primitive modules. The pseudocode and the code sketches generated in the design phase were used as a guide in developing the actual code. The primitive modules were implemented as FORTRAN subroutines. Each primitive subroutine was tested separately to check its functionality against the functionality expressed in the design phase.

Once the set of primitive modules were coded, Coding of the higher level modules proceeded starting with the top level modules. These modules generally consisted of a loop which waited for an event (a keypress or a timed interval expiration) and responded to the event by calling one more subordinate modules. A module called by one of these top level modules was generally either one of the second level modules or one of the primitive modules.

The second level modules were initially generated as subroutine stubs. The subroutine was physically there and its parameters were in place but no actual code was provided. The stubs, together with the primitive subroutines, already coded, allowed the top level modules to compile and execute from the beginning of the coding phase so there was always a "working" program in front of the programmers. Initial versions of the program at this point did not actually perform any function other than calling subordinate modules but the exercise was important to insure the quality of the interfaces between modules. As the second level modules were coded one by one, the system grew in "stepwise refinement" fashion.

The Object-Oriented Design Effort

The following is a description of the redesign effort of the inventory system using object-oriented techniques. An overview of the methodology is given first followed by a phase by phase description.

The Object-Oriented Methodology

Object-oriented development revolves around the concept of an object. Objects are analogues of real-world devices, software entities, and data constructs. An object-oriented methodology seeks to model the target application as a collection of objects and a collection of manipulations on these objects, called methods.

Recognizing the benefit in being evolutionary and not revolutionary, it was not the intent of the redesign effort to overhaul the software development methodology used in the original development of the inventory system. In the redesign effort, we supplemented the original methodology by using object-oriented techniques at various points in the lifecycle. Functional decomposition was not totally discarded either. Instead, we combined functional decomposition and objective techniques.

The Objective Requirements Analysis Phase

For the redesign effort, the requirements analysis phase remained unchanged from that of the original design effort. In fact, the original requirements document was used in the redesign effort as a starting point for the object-oriented design phase.

For new projects in the future however, there would be some changes in the way the object-oriented requirements analysis phase would be conducted. Specifically, more attention would be given to identifying objects, both real and conceptual objects, in the requirements analysis phase.

The original functional analysis focused on the functional description of the system so that concern over entities like the files and the user interface screens was delayed until further in the development. Object-oriented requirements analysis would identify these as objects earlier in the development effort.

The functional analysis would not be totally discarded however. The object-oriented requirements analysis phase would consist of both types of analysis. There is benefit in viewing a system both in a functional context and an objective context. Identifying the functional requirements can indicate the need for objects that might otherwise escape detection. The converse is true also. Identifying objects require that functional analysis be conducted in order to determine the methods for the objects.

The Objective Preliminary Design Phase

The first task in the object-oriented preliminary design phase was to identify the objects in the system. There were three primary sources available from which objects could be identified: the original requirements document, the original data flow diagrams, and the system diagram.

Since the requirements analysis phase remained unchanged, the original requirements document was a valid resource to be used in the redesign effort. The original requirements document was used to identify objects by reading through the document noting nouns and verbs. The nouns were considered as candidates for objects and the verbs and adverbs were considered as candidates for methods.

Since the first level data flow diagram was developed as part of the requirements phase, it also was a valid resource for object identification in the redesign effort. Sources, sinks, and data stores were considered as candidates for objects and processes were considered as candidates for methods.

The system diagram was developed as part of the original preliminary design phase and was also a valid resource for object identification. The system diagram consists entirely of objects so each item on the system diagram was considered as candidates for objects.

First, a list of all candidate objects was developed drawing on the three sources mentioned above. Incidental objects and objects transparent to the software implementation such as cables, modems and multiplexors were discarded.

Analyzing this original list, it was realized that many objects were actually instantiations of higher level, more general, objects. In these cases, the objects were grouped together under the higher level object in an hierarchial fashion. This is similar to the grouping of functions in the original development effort.

Examples of this grouping are the CRT object and the FILE object. Originally there was an Office CRT #1, Office CRT #2, and a Floor CRT object defined. These three objects were grouped under the CRT object. The original objects Product Description File, Inventory File, and Upload File were combined under the object FILE.

The result was a list of high level objects and subordinates which were represented in an object hierarchy diagram shown in Figure 4.

The second task in the preliminary design phase was to identify the methods for each object. Again, the original requirements document, data flow diagram and the system diagram were used to identify methods. It was realized that not all methods required to implement all system functionality could be readily identified at one time. Rather, several iterations through the preliminary and detail design phases were required before a complete identification of methods was possible.

This is similar to the iterative nature of functional decomposition at this point in the original development effort. The iterative nature of the classical software development lifecycle is well known and it was no surprise that the object-

oriented approach exhibited the same iterative nature. The identification of methods is a form of functional analysis.

One difference between the redesign effort and the original effort was the emphasis placed on the data files. In the original effort, the files were known in the earlier phases but were not dealt with in detail until later in the development. In the redesign effort, the files were treated as top level objects.

The original implementation, designed by focusing on functional aspects, resulted in file manipulation subroutines being in many different modules. The redesign encapsulated all the file manipulations within a single object. This type of encapsulation lowers the complexity of the design and makes the final product more understandable and maintainable.

This was made apparent in the original system after a requirements change late in the lifecycle regarding the files caused the programmers to search throughout the entire implementation making modifications to, sometimes redundant, code. A bug was introduced in the system at this time because not all instances of file manipulations were found and changed. The bug was not discovered until late in the integration phase and the debugging and repair effort cost many manhours. Under the object-oriented design, all file manipulations are in a single place and can be easily changed without affecting the rest of the code.

The third task in the preliminary design phase was to specify the interfaces between methods. Again, a degree of iteration through the preliminary and detailed design phases was required. The interfaces depended in part on the implementation strategy used which was not determined until the detailed design phase.

All through the preliminary and detailed design phase, full Ada was used as the design language. As the object implementation and interfaces evolved, the infrastructure of the final code was created. The use of the Ada compiler during this effort was valuable as well. Interfaces and interdependencies between objects could be verified by compilation during the evolution of the object implementation.

The result of the design phase was a working program skeleton and was used as the starting point for the coding phase.

The Objective Detailed Design Phase

The first task in the detailed design phase was to determine the implementation strategy for each object. The use of Ada as the implementation language had an impact on the effort involved in this phase. Ada does not offer as much support for object-oriented development as do some other languages but Ada does provide several ways to implement objects.

The Ada package is one technique we used to implement objects. The object is the package itself and the procedures and functions are used to implement the methods for that object.

The Ada task is another technique we used to implement objects. Similar to a package, the methods for the object were implemented as task entries. The ability to define task types in Ada allow the instantiation of more than one of the objects.

Although we did not use them in the redesign effort, the Ada generic facility and Ada data structures can also be used to implement objects.

In the redesign effort, a mixture of implementation strategies were used which was one reason for the iteration through the preliminary and detailed design phases. On the first pass, every object was implemented as an Ada package and each method was implemented as a procedure in that package.

It was realized that multiple occurrences of some objects were better achieved if another implementation strategy was used.

A case in point is the object CRT. Originally, CRT was the package name and a collection of procedures implemented the methods for the object CRT:

```
package CRT_OBJECT is
   procedure INITIALIZE;
   procedure DISPLAY_MENU;
   procedure GET_USER_SELECTION;
   procedure DISPLAY_PAGE_OF_DATA;
   procedure CLEAR_SCREEN;
      . . .
end CRT_OBJECT;
```

With this implementation of CRT, the creations of the OFFICE_1, OFFICE_2 and FLOOR objects would have had to be via copying and renaming the package.

CRT could have been implemented using a generic package. With a generic package, the creation of the three subordinate objects would have been:

```
package OFFICE_1 is new CRT_OBJECT(...);
package OFFICE_2 is new CRT_OBJECT(...);
package FLOOR    is new CRT_OBJECT(...);
```

However, these would still be static entities. A better implementation method is one using tasks. Treating each CRT object as a separate task allows the execution of each to proceed in parallel. This more closely models the real world where there are actually three different terminals each operating at the same time.

One desired feature of the inventory system was the ability for additional terminals to be added in the future. In this light, it was decided to use a task type to implement the object CRT:

```
task type CRT_OBJECT is
   entry INITIALIZE;
   entry DISPLAY_MENU;
   entry GET_USER_SELECTION;
```

```
    entry DISPLAY_PAGE_OF_DATA;
    entry CLEAR_SCREEN;
      . . .
end CRT_OBJECT;
```

Multiple CRT objects were created by an array of task types:

```
CRT : array(1..N) of CRT_OBJECT;
```

An enumeration could have been used to enhance readability:

```
type CRTS is (OFFICE_1,OFFICE2,FLOOR);
CRT : array(CRTS) of CRT_OBJECT;

CRT(OFFICE_1).DISPLAY_MENU;
```

Multiple instances of the CRT object could be created dynamically and execute in parallel which satisfies all of the criteria for the object CRT. Note also how well the implementation follows the design. In the object-oriented design, the object OFFICE_1 is subordinate to the object CRT and DISPLAY_MENU is a method on the object OFFICE_1. The actual code follows this structure well.

The second task in the detailed design phase was to pseudocode the methods. Since the second level data flow diagrams were created in the original preliminary design phase, they were valid sources for information in this phase. The second level data flow diagrams were used to determine the underlying functionality for each method.

Again, full Ada was used as the pseudocode. Extensive use of comments were required in some cases but most of the overall control texture of a program unit could be written in compilable Ada.

The pseudocoding process had iterative impact as well. The process of pseudocoding identified new interface requirements in some cases and pointed out discrepancies in other cases. In such cases, the design was iterated back through the preliminary design and detailed design phases as mentioned above. The same iteration was required in the original development as well.

The Objective Coding and Integration Phase

The actual code for the redesigned inventory system was not developed in its entirety. However, some comments can be made on how the coding and integration effort would proceed given the object-oriented design.

The use of full Ada as the design language is a great benefit when it comes to the coding phase. Throughout the design phase, there is always a working version of the program available. At the end of the design phase, a collection of skeletal Ada program units exists. The structure and interfaces between the Ada units are already verified by successful compilation so all that remains is to flesh out the program units. Since each unit contains syntactically correct Ada pseudocode, the programmer's effort is reduced significantly.

Since the interfaces between the units have been verified by compilation, the units can be worked on independently by many different programmers with a minimum of integration problems. Each programmer could work on his or her unit in parallel with other programmers and be assured that the units would function correctly as long as the interfaces were not changed.

The separate compilation facility in Ada aids this type of development also. The specification parts of the Ada units have been compiled already as part of the design phase. Work may proceed on the bodies without recompiling the specification parts.

The use of object-oriented design techniques ensure that all methods for a particular object are contained within a given Ada program unit. This facilitates parallel development of the code as well.

For example, a programmer coding the object FILE can be assured that no other program unit in the system performs file manipulations. This aids in the integration of the program units too. The effect of changes or bugs are limited to a smaller section of code which can be quickly identified and repaired without affecting other parts of the code.

Any software development is an exercise in abstraction. Software seeks to model the real world via software constructs. The use of object-oriented techniques causes the design to look like the real world. That is, object-oriented design yields a closer abstraction than does other design techniques. The closer the abstraction is to reality, the more intellectually manageable the design is. If a piece of software is more intellectually manageable, it is more likely to be implemented correctly. Coding the objects and the methods in the redesign was a pleasure because one never had the sense of being lost in the design.

An abstraction that is close to reality also facilitates maintenance. Every system requires changes eventually. If a programmer is able to grasp the design and implementation of an application more clearly, he or she is more likely to implement changes to that system correctly and in a more cost effective manner.

Object-oriented encapsulation of methods and objects tend to limit the scope of changes. Changes made to one part of the system are less likely to have adverse, and sometimes unknown or hidden, effects in other parts of the system.

Conclusion

In the redesign effort of the inventory system, we successfully integrated object-oriented techniques into our existing software development lifecycle and showed that a clearer, more concise design followed.

We demonstrated that both functional decomposition and object-oriented analysis and design can be used to the mutual benefit of each other. We did this without significantly modifying the existing methodology which means that it is more likely to be accepted and used.

We showed that Ada is particularly well suited to this methodology and while, theoretically, any language could be used, Ada has certain features that support an object-oriented methodology well.

We see object-oriented techniques becoming more and more widely used and expect that the methodology will evolve along with the experience of the people using it so that, in the future, the methodology becomes even more object-oriented.

References

[1] Ada is a registered trademark of the United States Government (Ada Joint Program Office).

Bibliography

The Ada Programming Language Reference Manual. US Department of Defense, US Government Printing Office, February 1983. ANSI/MIL-STD-1815A-1983.

Alabiso, B., "Transformation of Data Flow Analysis Models to Object-Oriented Design", Proceedings, 1988 Object-Oriented Programming Systems, Languages and Applications (OOPSLA), San Diego, CA.

Bailey, S., "Designing With Objects", Computer Language, January 1989.

Booch, G., Software Engineering With Ada. Benjamin Cummings Publishing Company, Menlo Park, CA 1983.

Booch, G., Software Components With Ada. Benjamin Cummings Publishing Company, Menlo Park, CA 1987.

Bulman, D., "An Object-Based Development Model", Computer Language, August 1989.

Buzzard, G.D. and Mudge, T.N., "Object-Based Computing and the Ada Programming LANGUAGE", Computer, March 1985.

Celko, J., "Three Structured Approaches", Computer Language, March 1986.

Chandler, L., "Ada and Modern Software Design", Computer Language, March 1986.

DeMarco, T., Structured Analysis and System Specification. Yourdon Press 1978.

Harkleroad, "Analyzing Ada Concurrent Algorithms", Ada Letters, Vol III No. 2, March/April 1987.

Keuffel, W., "Structured Analysis: Under the Hood", Computer Language, January 1989.

Kiem, E., "The KEYSTONE System Design Methodology", Ada Letters, Vol IX No. 5, July/August 1989.

Lieberherr, K., Holland, I., and Riel, A., "Object-oriented Programming: An Objective Sense of Style", Proceedings, 1988 Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Conference, San Diego, CA.

Nierstrasz, O., "A Survey of Object-Oriented Concepts", Object-Oriented Concepts Databases, and Application. Kim, W. and Lochovsky, F. ed., ACM Press, New York, 1989.

Pascoe, G., "Elements of Object-Oriented Programming", Byte, August 1989.

Schiper, A. and Simon, R., "Traps Using the COUNT Attribute in the Readers - Writers Problem", Ada Letters, Vol IX No. 5, July/August 1989.

Tesler, L., "Programming Experiences", Byte, August 1986.

Urlocker, Z., "A Methodology for the Real World", Computer Language, January 1989.

Yourdon, E.,"The Year of the Object", Computer Language, August 1989.

Yourdon, E., Structured Design: Fundamentals of a Discipline of Computer Programming and Systems Design. Yourdon Press 1978.



Ron Fulbright is an Automation Engineer with Fluor Daniel in Greenville, SC and holds a B.S. in Computer Engineering from Clemson University. Mr. Fulbright is a member of the ACM and the IEEE.

An Ada Plasma Panel Controller: A Reuse Experiment

Captain Richard T. Mraz

United States Air Force Academy

Abstract

In this paper, we report the findings from our investigation of an Ada lifecycle management issue, software reuse. The results this project show that a good design is an essential requirement for reusable software. Our design is based on two guidelines. First, the designers must encapsulate those parts of the program that have a high potential for change. By segregating those parts of the design, we prevent changes from rippling through the program. Second, we recommend an isolation of operating system services. Since the operating system interface is not standard among Ada compilers, a virtual machine interface is suggested to reduce compiler dependent code. For our Plasma Panel Controller Project, where we followed these guidelines, the results show our Ada source code to be about 40% reusable and 75% portable.

Introduction

In 1985, the National Aeronautics and Space Administration's (NASA), Space Station Program Office issued the following statement,

> "The Space Station program has selected Ada for ALL software paid for by Space Station funds. The 'all software' literally means all types: application, operating system, user interface, data base management, and any other. Needless to say, waivers to the Ada mandate will not be readily granted or easy to obtain." [1]

While this directive influenced engineers throughout NASA, to the project managers at NASA's Johnson Space Center (JSC), in Houston, Texas, this policy required further investigation. The primary mission of JSC is to provide Mission Control operations for all U.S. manned spaceflight activities. Their mission is currently limited to Space Shuttle flights, but it will be extended in the future to include Space Station Mission Control.

Confronted with the software policy for Space Station and its future influence on their mission, managers at JSC moved to address large scale Ada software development. Some of the first topics under consideration included software design strategies, Ada problem domains, and lifecycle management issues. From these broad objectives, we started several Ada prototypes to look at specific problems. This paper reports findings from one project where we looked at Ada software reuse, a lifecycle management concern.

Paper Overview: In the remainder of the paper, we examine the reasons for this research. This is followed by a look at our problem domain, the plasma panel controller. Then, we describe the plasma panel design and our design guidelines in detail. We briefly look at our implementation and reuse results. Finally, we draw some general conclusion about our project.

Reasons for this Research

To provide a basis for this research, let's examine the requirements for the Space Station Control Center (SSCC). In the SSCC, there will be several specialized flight control positions. For example, some positions will analyze Space Station telemetry, others will monitor the health of the astronaut crew, and there will be flight controllers that watch the payloads and experiments on-board the orbiting platform. To assist them in their jobs, every flight controller will have a scientific workstation connected to local area networks. While there will be common software running on every workstation, we will also have programs to meet the specialized needs of each flight control position. Mindful of the preceding directive, **all** of this software is required to be written in Ada. We would like to make these programs as reusable and portable as possible. This is a reasonable goal because in the future we can expect the workstations to be replaced with new models or perhaps a workstation will be replaced with an entirely different computer. In addition to these hardware problems, we are also concerned about writing dependent Ada source code. These dependencies limit software reuse and portability. Based on these concerns, let's continue with an examination of our specific problem, the plasma panel controller.

Problem Domain

The goal of this project was to investigate software reuse with Ada. To meet this goal, we needed a worthwhile project that could not only give us clear results, but we also needed a problem that is likely to be confronted in the Space Station Program. After an analysis of several candidate projects, we selected the plasma panel controller. The reasons for this

choice are twofold. First, the plasma panel controller would give us insight into Ada software reuse situations. We envisioned software reuse because of the common software among flight control workstations. We also imagined a need to reuse the software on different computers, reuse the software with a plasma panel from another vendor, or perhaps compile the software with a different Ada compiler. Each of these situations poses a different challenge when writing a portable program. Second, a well known requirement states that Space Station software will have to control devices connected to computers. Such device control would be required both for ground based control centers and in the orbiting platform. Therefore, panel control is a suitable problem domain for our project. With this justification in mind, let's proceed with the design details of the Display Request Keyboard.

Plasma Panel Design

In the present Space Shuttle Mission Control Center, each flight control console has several keyboards and panels. Three of the more common panels are the Display Request Keyboard (DRK), the Multiple Command Module (MCM), and the Configuration Control Module (CCM).[2] The common features among these panels are communications to the Mission Operations Computer (MOC), key definitions, and keyboard layout. We selected the DRK for our experiment because we can find it on almost every console. The other two panels are specialized for flight control positions that have requirements to send commands to the Space Shuttle and to configure the mission control center.

Let's examine the DRK in detail. The DRK is a direct input device to the Mission Operations Computer (MOC). The flight controller uses this panel by pressing a sequence of keys where each key sends a 'display request' to the MOC. The display request typically identifies a program the flight controller wants the MOC to run. The results of these programs are transmitted to a display on the flight controller's console. One important feature of this Space Shuttle system is that the DRK panel contains a hardwired controller and it cannot be changed. Furthermore, the process of defining the 'display requests' associated with the keys is a time consuming task that involves re-wiring the console. Such limitations cannot be realistically or practically extended to the Space Station Mission Control Center.

As pointed out above, we needed a project where we could examine Ada controlling an external device. For our DRK, we used a plasma panel touch screen connected to a minicomputer's serial port via RS-232C. The panel used an Intel 80186 processor with 356 Kbytes of memory. The screen had a resolution of 60 pixels per inch with a screen size of 512 x 512 pixels. Besides having standard keyboard entry, the plasma panel was touch sensitive.[2] Now, with the stage set with this background information, let's proceed with the design details of the DRK panel.

Design Details

After a thorough analysis of the DRK panel, we started our design. For this project, we followed the Object-Oriented Design technique as defined by Booch.[3,4] Figure 1 shows the design architecture of the Plasma Panel Controller Project. While an interesting topic, it is beyond the scope of this paper to examine the details of our object design. Yet, simply stated, the design was overwhelmingly skewed to anticipate software reuse. We classified our software reuse concerns in three ways. First, we needed an adaptable software controller versus a fixed hardwired controller. Second, we needed a method to define unique flight control panels. Third, we needed to reduce both Ada compiler dependencies and machine dependencies. From these concerns we identified two design guidelines that proved to be very successful. First, we encapsulated those parts of the program that had a high probability of change, and second, we isolated the machine dependent operating system calls with a 'virtual machine' interface.

Design Rule #1 - Encapsulate Change: Let's start with our first design guideline, "identify those parts of the project with a high probability for change." The premise for this rule-of-thumb is straightforward. If we isolate the highly dynamic parts of the program, then we can prevent changes within these encapsulations from spreading throughout the majority of the code. For our specific project, the plasma panel controller, we identified three parts of the program that have a potential for change, the DRK Device, the DRK Definition, and the DRK panel controller. The DRK Device defines an interface to the external device, the DRK Definition provides a mini-database to catalog specific panel configurations, and the DRK panel controller defines the abstract state machine that emulates the current hardware panel.

Let's examine the reasons for these design decisions. First, the DRK Device creates an interface between the computer and the external device (see Figure 1, `Package DRK Device`). For this project, we encapsulated the 'definition' of our plasma panel touch screen in an Ada package. This package included operations to initialize the plasma panel, receive input from the panel, and send commands to the panel. While the rest of the program used this interface, the actual implementation of the device was hidden. In the package body, we simply translated the operations (defined in the package specification) into the panel's 'Escape-Command-Sequences.' By isolating the DRK Device in such a way, we could easily replace our plasma panel touch screen with another device without interfering with the remainder of the program. In addition, we could also reuse this package if we wanted to use the plasma panel for the MCM or the CCM keyboards.

The second area with a high potential for change was the DRK Definition. This package defines the labels on the keyboard, and more importantly, the display request associated with each key. The Mission Operations Computer uses the display request to identify the information that is transmitted to the flight controller's console. Furthermore, each flight control position has its own set of displays; hence, its own DRK panel definitions. Therefore, a robust design was needed to support a DRK configuration for any flight control position. Our design provided operations to recall a DRK panel from an external file (see Figure 1, `Package DRK Definition`). Through this interface, we could redefine the DRK data structures, change the implementation of those data structures, or modify the external file format without interfering with the remainder of the program. Furthermore, we can also reuse the DRK Definition packages if we needed to define the MCM or the CCM panels.

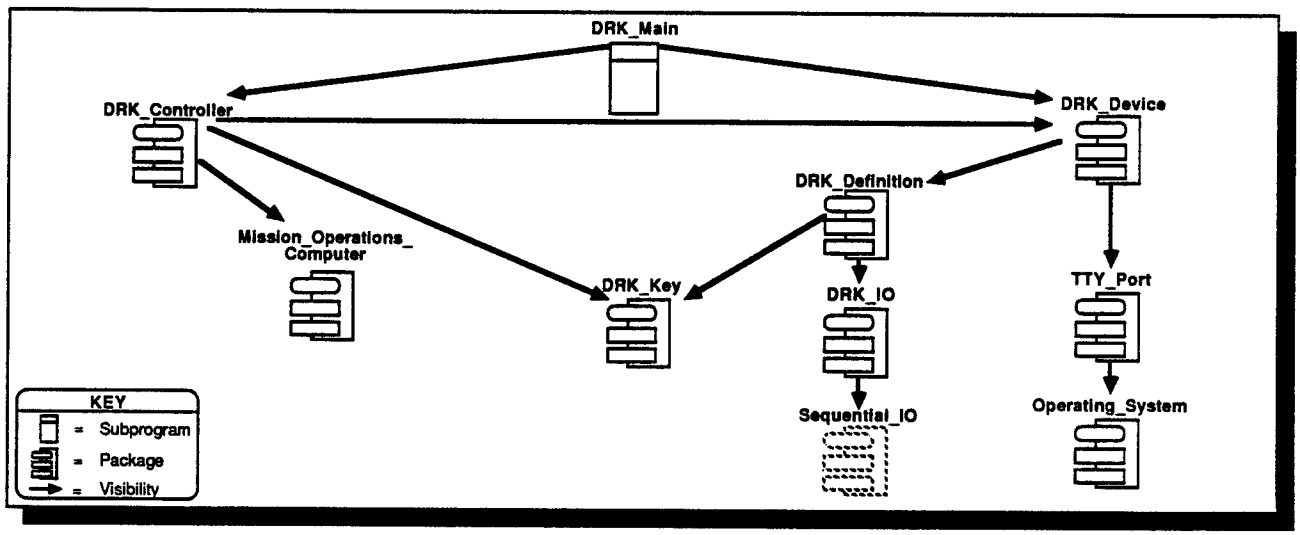


Figure 1: Plasma Panel Design Architecture

Finally, the third area with high potential for change was the DRK Controller. Our implementation of the controller emulated the functionality of the hardware panel currently in the Mission Control Center. The DRK Controller was written as an abstract state machine that accepts key presses from the user, keeps track of the state of the panel, and issues display requests to the Mission Operations Computer (see Figure 1, `Package DRK Controller`). Presently, the hardware panel in the Mission Control Center is not adaptable because it is hardwired. For our design, we encapsulated the DRK Controller into an Ada package. While the main program uses the interface to this package, the implementation of the state machine is hidden. Once again, without changing the interface, we could add functionality to the DRK panel, or we could change the DRK's function completely without repercussions throughout the entire program.

Design Rule #2 - Isolate Operating System Calls: Now, let's examine the second design guideline, isolate operating system services. Once again we need to look at the reasons for such a design heuristic. Simply stated, the Ada Language Reference Manual (LRM) does not define the interface to the operating system.[5] As one would expect, with no guidance from the LRM, compiler vendors define their own interface. If we used the operating system services provided by the compiler, we would be writing compiler specific code. In addition, such an interface may provide machine unique services that would result in machine dependent code. Therefore, our design rule reduces such non-portable techniques by encapsulating all operating system services into a 'virtual machine.' For our specific application, the plasma panel controller, we required access to the computer's communication lines (see Figure 1, `Package TTY_Port`). We needed to open communications with a serial I/O port, and we needed to read data from the port and write data to the port. For this design, we isolated these services in an Ada package. While the remainder of the program used our virtual machine interface, we explicitly defined each operation using the packages provided by the compiler. While we are still bound to compiler dependent code, we have effectively isolated the dependencies into one package. Once again, we can reuse this package when writing an application to emulate a Multiple Command Module or the Configuration Control Module.

Implementation: The resulting implementation of this design was straightforward. We used basic Ada data types, packaging mechanisms, control structures, and programming techniques. During this phase, we found almost any Ada reference book could answer our implementation questions. After we finished the code writing and testing phase, we examined the source code for reuse and portability. While a software engineering metric would be appropriate for this analysis, we could not define a metric robust enough to stand scrutiny. This is true for two reasons. First, reusable software tends to be highly domain sensitive, and, second, any quantitative measure we formulated could not be applied to all problem domains and to all languages. Therefore, we relied upon a qualitative measure.

Because our project was a prototype, we did not explicitly try to reuse the software by writing an MCM or a CCM emulator, nor did we actually port the source code to another machine. However, we examined the code to evaluate two reuse situations. For the first case, we looked at reusing our software with the other panels (the MCM and the CCM). We found the code to be about 40% reusable. For the second case, we looked at porting the software. The following scenarios were considered; (1) porting the code to another machine, (2) changing the plasma panel to another device, and (3) compiling the software with another compiler. Our code appears to be about 75% portable.[6]

Conclusions

At Johnson Space Center, NASA managers are concerned about large scale Ada software development. This paper addressed one lifecycle management concern, reusable software. As we approached this project, we had specific goals of examining the Ada Programming Language.

However, as the project developed, we found our attention drawn away from Ada and it moved toward the more general topic, software design. Therefore, we conclude that software reuse is synergistic. Our software must be designed with reuse in mind, and our languages must provide mechanisms that support reusable software.

For this project, we defined two design guidelines, and we used Ada, a language with good reuse features. For our first guideline, we recommend designers identify the parts of the program with a high potential for change. Each part should be encapsulated; hence, preventing a change from rippling through the rest of the design. For example, we isolated the DRK Device, the DRK Definition, and the DRK Controller. Now, we could easily swap in a different panel or modify the controller functionality without touching the majority of the source code.

Finally, if an Ada program uses operating system calls, we recommend a 'virtual machine' interface as a necessary part of the design. Since the Ada Language Reference Manual does not specify an operating system interface, each compiler vendor is free to define its own. Because compiler unique interfaces reduce software reuse and portability, the designers must isolate their effects through encapsulation. For this exercise, we placed all operating systems services into one Ada package. One could extend this idea of a 'virtual machine' interface by creating several packages each dealing with a particular part of the operating system. For example, we can imagine a File System package, I/O package, and Process Control package.

Notes

[1]Space Station Project - Plans and Status. Proceedings of the Joint Ada Conference. Fourth Washington Ada Symposium. 16 - 19 March 1987. Arlington, VA.

[2]Baker, Carolyn. A Mission Control Center (MCC) Console Module Emulation Prototype Using A Plasma Touch Panel. NASA Workstation Prototype Lab Newsletter. Volume 1 Number 3 May 1987.

[3]Booch, Grady. "Object-Oriented Development," IEEE Transactions on Software Engineering, SE-12: 211-221 (February 1986).

[4]Booch, Grady. Software Engineering with Ada. Second Edition. Menlo Park, California: Benjamin/ Cummings, 1986.

[5]ANSI / Mil-Std-1815A-1983. Ada Programming Language. 22 Jan 1983.

[6]Zola, Kenneth. "DRK Command Panel Prototype Report." Johnson Space Center - Workstation Prototype Lab Internal Report. 7 August 1987.

About the Author:

Captain Richard T. Mraz is an Instructor of Computer Science at the U.S. Air Force Academy in Colorado Springs, Colorado. He has taught introductory courses in undergraduate Computer Science. He received his B.S. in 1982 from the Air Force Academy, and his M.S. in Computer Engineering from the Air Force Institute of Technology. Captain Mraz is a member of the IEEE Computer Society.

Author's Mailing Address:

HQ USAFA/DFCS
Dept of Computer Science
Captain Richard T. Mraz
USAF Academy, CO 80840

Network Address: mraz@usafa.af.mil



A STRUCTURED STEPWISE REFINEMENT METHOD FOR VDM

Sue A. Conger*, Martin D. Fraser**, Ross A. Gagliano**, Kuldeep Kumar*,
Ephraim R. McLean*, G. Scott Owen**, and Vijay K. Vaishnavi*

* Department of Computer Information Systems
** Department of Mathematics and Computer Science
Georgia State University
Atlanta, GA 30303

Abstract

A proposal is made for improving the intuitive decomposition processes of formal Vienna Development Method (VDM) specifications which could be useful for Ada applications in MIS. The proposed approach is based on Structured Analysis and Design refinement heuristics, and is illustrated by decomposing the VDM specification of a business example.

1.0 INTRODUCTION

Abstraction and specification are keys to more effective programming, especially for languages such as Ada with modern features such as tasking. Because, however, the specification phase occurs early in the standard software life cycle, errors in this phase become deeply embedded in the program structure and, accordingly, are difficult to find and costly to correct.

Therefore, a design approach is needed that is methodical, unambiguous, and logically rigorous for developing and expressing specifications, especially for large systems. The corresponding code can then be created, tracked, and updated through the use of decomposable sub-specifications.

To ensure program correctness starting at the specification level increases the need for formalization. One formal method, called the Vienna Development Method (VDM), possesses a notation through which formal proof rules can be applied. Moreover, VDM has not only reached a level of maturity and acceptance, but also has been increasingly utilized in industrial settings [BJO87, BLO88, JON86].

Previously, a functional decomposition approach for the design phase was proposed [GAG88, OWE87]. This article presents a stepwise refinement process for VDM guided by Structured Analysis and Design Techniques (SADT), and illustrated in an example of a payroll system.

Supported in part under contract DAKF11-89-C-0014 from the U. S. Army Institute for Research in Management Information, Communications and Computer Sciences (AIRMICS).

1.1 The Vienna Development Method (VDM)

A key concept of VDM is successive stepwise refinement of software specifications. The specification is first written at its highest functional level. It can then be successively and hierarchically decomposed into a set of sub-specifications at the next lower level. Independently, each of the sub-specifications can be refined further into more detailed sub-specifications. The (n+1)th level sub-specification is considered to be the implementation of the nth level specification of the software system. The process of stepwise refinement can continue up to a point where physical implementation, either through software or hardware, becomes feasible.

By decomposing each of the specifications into a set of sub-specifications, the size and complexity of each of the specification problems is reduced [RAS85]. If such decomposition is done correctly, the sub-specification can effectively stand alone. The only requirement may be to satisfy each of the sub-specifications.

> "So modular development of software is possible - a large specification can be broken down into components and each component developed separately. When all of the code has been produced, it can be fitted together in just the right way to produce a program which satisfies the original large specification." ([AND88], p.6)

The concept of successive stepwise refinement is a part of a variety of software specification methodologies. However, most methodologies are limited to only providing a modeling frame which can be used to document the outcomes of the thought processes which result in successive refinement. As far as suggesting techniques for stepwise refinement, VDM is no exception. The process of successive refinement in VDM is mainly intuitive:

> "It is, however, important that the reader is not led to expect too much from this idea. Design requires intuition and cannot in general, be automated. What is offered is a framework into which the designer's commitments can be placed. If done with care, the verification then represents almost no extra burden." [JON86]

VDM methodology does not provide any detailed ("how to") guidelines for performing the successive stepwise refinements during the design process.

However, once the designer has thought through the processes involved and arrived at a next level refinement, VDM does provide both a language to express the next level design, and a formal method of proving the completeness and correctness of this design.

As a result of refinement, in theory at least, the (n+1)th level sub-specifications are the implementation of the nth level specification of the software system. However, errors of omission (incompleteness) and inconsistency between levels could mean that the next lower level specifications may not correctly implement the higher level specification. In VDM the formalisms and the inference mechanism of the VDM language can be used to prove if the next level refinement is a correct and complete implementation of its higher level specification.

1.2 SADT

Structured Analysis [DEM78] and Structured Design Methodologies [MYR78, YOU78] provide a set of heuristics for the functional decomposition of an information system. Though the actual decomposition itself ultimately depends on the structure of the problem space, and the domain experience and background of the designer/problem-solver, the heuristics provide a handy set of techniques which guide the designer through the process of stepwise refinement of the design. The structured analysis and design heuristics have been shown to be useful in the design of a variety of administrative and real-time time computer-based information systems.

The remaining article is structured as follows. The next section describes the Structured Analysis tools and heuristics for the functional partitioning of the system. Section 3.0 describes a business example and the structured analysis technique of hierarchical functional decomposition. Section 4.0 uses the structured analysis specification as a guide for developing formal VDM specifications and shows how the VDM proof mechanism can be used to verify the completeness and consistency of the lower level specifications. Section 5.0 shows how Ada tasking can be specified through VDM formalisms. Finally, section 6.0 presents our conclusions and suggestions for future research.

2.0 FUNCTIONAL STEPWISE REFINEMENT IN STRUCTURED ANALYSIS

The functional decomposition or refinement problem can be thought of as consisting of two distinct but related sub-problems:

a. Partitioning the overall system into sub-systems.

b. Arranging the partitioned sub-systems into a consciously designed structure.

Structured Analysis [DEM78] provides heuristics for approaching the partitioning sub-problem, while Structured Design [MYE78, YOU78] addresses the arranging or structuring aspect.

The primary tool used in structured analysis to describe and analyse a system is the Data Flow Diagram (DFD). The data flow diagram, at the highest level, describes the system as a data transformation process which receives input data-flows from its environment, and processes or transforms them into output data-flows which flow back to the environment. The highest level DFD is called a "Context Diagram."

A more detailed model of the overall system is built by partitioning the context diagram (i.e., the overall data transformation process) into sub-processes. Rather than using an "intuitive" or "brute-force" approach to partitioning, the partitioning is carried out by tracing either the input data flows forward, or output data flows backwards into the system. A sub-process is recognized at the point where a data-flow is operated upon or changes either its quantitative or qualitative attributes. The points of transformation of the data-flows thus provide the heuristics for partitioning the higher level transformation process. Additional heuristics (the rule of 7 plus or minus 2) are provided for managing the number of partitions at each level.

Each of the sub-processes may be further partitioned into their lower level sub-processes until bottom level processes called "functional primitives" are reached. Functional primitives are not partitioned further; instead they are described using "transform descriptions." Structured analysis methodology provides additional heuristics for recognizing the bottom level or the functional primitives. Finally a process of "Topological Re-Partitioning" may be applied to the hierarchically partitioned set to minimize the number and complexity of interfaces between the partitions [DEM78].

The hierarchically partitioned data flow diagrams have a precise set of syntactic and semantic rules (such as connectivity rules, data conservation rules, precedence rules) which need to be observed to ensure correctness, completeness, and consistency. Further precision is added by employing a set of syntactic rules which precisely and unambiguously define the data structures (data flows and data stores or files) providing the interface between the data transformation processes. The definitions of the data structures can also be hierarchically successively partitioned into lower level definitions until "self-defining terms" are reached. The definitions of data flows and data structures are stored in a "Data Dictionary" (DD).

A precise, unambiguous, and successively refined specification of the system can be developed using the syntactic and semantic rules for DFDs and the syntactic rules governing the DD. However, the process descriptions ("Structured English," decision-tables, or decision-trees) for functional primitives do not have the same level of precision as the DFD or DD.

The structure of the software components is designed using the "transform" or "transaction" analysis heuristics available in structured design

[MYE78, YOU78]. The design heuristics are applied to the data flow diagram to produce a structured arrangement of the modules called the "Structure Chart." These heuristics are designed to minimize coupling between and maximize cohesion within the data processing modules.

Thus a combination of structured analysis and structured design can be used to develop a hierarchically partitioned and structured (i.e. successively refined) specification of the software system. Though this specification is precise and unambiguous as far as the process structure and the data structures are concerned, the specifications of the functional primitives themselves are not very precise. Furthermore, the description is not formal in the sense that formal logic proofs can be applied to prove the correctness and completeness of the specification.

3.0 ILLUSTRATIVE SPECIFICATION

In this section we describe a payroll system which will be used to illustrate the use of structured analysis and design as a guide for stepwise refinement of VDM specifications. The payroll system receives a file of Time-Cards, which it processes against a DataBase of Valid Employees and Valid Account Numbers to produce a file of Paychecks. Those Time-Cards which do not pass the validation criteria are outputed onto an Error-list.

The context diagram for the Payroll System is shown in Figure 1. The first level partitioning for the system is shown in Figure 2, The Overview Diagram. In order to keep the illustration simple and manageable, a number of payroll functions, such as calculation of overtime, payroll deductions etc., have been omitted, and the system is not partitioned beyond the first level. Furthermore, the contents of the data flows and data stores (files) have been kept to the minimum needed for the specified system. Figure 3 shows the structure chart which has been derived as a result of applying transform analysis to the DFD in Figure 2. Finally Figure 4 specifies the Data Dictionary for the payroll system.

4.0 THE VDM TRANSFORMATION

The formal language of VDM used in the following specifications and proofs essentially follows [JON86] and [AND88]. The exceptions (for typographical reasons) are: x~ replaces a hook over x, and $\triangleq$ will mean "is defined as." Operations are used to specify the major modules of the design. Auxiliary functions are used to identify the logical properties of the pre and post conditions of specifications.

VDM moves from formal specification toward implementation by repeatedly decomposing complex specifications into sub-specifications until they can be easily coded. As discussed in Section 2.0, the technique we suggest to guide this stepwise specification refinement is based on the hierarchical partitioning techniques of data flow analysis.

This approach may considered similar to the use of the bottom level of an informal (lacking data flow definitions) flow chart to develop formal specifications by Andrews and Gibbons [AND88] with an important distinction. Our approach to specification decomposition is to track, at each level, the hierarchical partitioning of high level transforms with VDM specifications. The proof obligations incurred at each level by decomposition can also verify that the sub-specifications satisfy the previous level's specifications.

Discharging the proof obligations from level to level validates that the specifications corresponding to the transforms in the lowest level DFDs satisfy the specification corresponding to the single transform of the context diagram. Thus, validation of the specification of the entire software element of the system design is accomplished.

The technique for guiding operational decomposition with transform partitioning consists of:

a. representing the input and output data flows from and to externals in the DD in an abstract syntax,

b. giving a specification for each transform in the structure chart which has been derived from the DFD, and

c. combining (using the VDM combination constructs sequence, if-then-else, and while) the specifications according to the architecture provided by the structure chart. Proof that the combined specifications satisfy the specification at the higher level also verifies data flow connectivity.

This technique is illustrated with the payroll system example. Figure 5 gives the abstract syntax corresponding to the DD from Figure 4. The operation PAYROLL, specified in Figure 6.a, corresponds to the transform in the context diagram (Figure 1). Auxiliary functions used to express this specification are defined in Figure 6.b.

The payroll system is modeled at the highest level as a single transform which inputs and outputs data. The input data flows are:

a. a file of time cards modeled by a sequence (TcLst),

b. a data base of employee identifiers and pay rates modeled by a mapping (EmpDb) of (unique) identifiers to pay rates, and

c. a file of chargeable account identifiers modeled by a sequence (ActLst).

The output data flows are:

a. a file of invalid time cards modeled by a sequence (ErrLst), and

b. a file of processed paychecks modeled by a sequence (PcLst).

A conservation of time-cards constraint holds: A time card appears in TcLst if and only if it appears in exactly one of ErrLst and PcLst.

At the highest level there is one operation (or transform), PAYROLL, which maps Tc's to elements of either ErrLst or PcLst. A database (EmpDb) and a list (ActLst) of chargeable accounts provide the context in which the validity of a Tc is expressed. If a Tc is invalid, it is added to ErrLst; else a Pc is calculated and added to PcLst. The partitioning of the context diagram into the overview diagram (Figure 2) and structure chart (Figure 3) guides our choice of sub-specifications and combination constructs shown in Figures 7.a and 7.b.

4.1 Proof Obligations

The proof obligation is to show that the sequence "RDTC;VALACT;VALID;if-statement" satisfies the specification of the while-statement and that the while-statement satisfies the specification of PAYROLL given in Figure 6.a.

For example, the proof of "RDTC;VALACT;VALID;if-statement" is carried out by first deriving the post condition for the sequence RDTC;VALACT and composing it with the post condition of VALID to obtain the post condition for RDTC;VALACT;VALID:

let tc = tl(i) in aflg = act(acct(tc),al) and
iflg = idb(id(tc),db).

Next, the post condition of the if-statement is derived. Note that the post condition for the then clause can be conjoined with len el = len el~ because el is not changed in the then clause. Similarly, the post of the else clause can be conjoined with len pl = len pl~. By weakening each of the post conditions, we obtain the post condition of the if-statement:

(payrcd(tc,db(id(tc)),pl) or errcd(tc,el)) and
len pl + len el = len pl~ + len el~ + 1.

Finally, we combine S1 = RDTC;VALDT;VALID and S2 = if-statement in sequence. The post for S1;S2 is the composition of the two post conditions just obtained:

let tc = tl(i) in
((act(acct(tc),al) and idb(id(tc),db) and
payrcd(tc,db,pl)) or
(not(act(acct(tc),al) and idb(id(tc),db)) and
errcd(tc,el)) and
len pl + len el = len pl~ + len el~ + 1.

Because len pl~ + len el~ = i~, so len pl + len el = i~ + 1 = i, the required post condition of the loop body is obtained.

4.2 Error Detection

It is instructive to consider what happens if, for example, the designer missed the transform to validate the account number during the overview partitioning. Then VALACT would have been omitted, making the if-statement's guard "iflg" alone. The post condition for S1 would be post iflg = idb(id(tc),db) from which the desired post for the overall sequence cannot be deduced because introducing the conjunction with act(acct(tc),al) requires the strengthening, not weakening, of the post.

5.0 VDM AND Ada

Returning to the decomposition in Figure 7.b and Figure 2, note that the specifications VALACT and VALID call for operations that are not coupled and which work on different external files. Therefore, although these specifications are combined in sequence, the order of combination does not affect the derivation of the post condition of the sequence. That is, either pre condition can be conjoined with either post condition and the post's of VALACT;VALID and VALID;VALACT are the same because the conjunction act(acct(tc),al) and idb(id(tc),db) can be shown to commute.

These specifications can therefore be implemented in Ada as tasks. If the machine processing is actually distributed or parallel, the searches of the data sets represented by ActLst and EmpDb can be carried out concurrently and performance improved.

This is the case in a more general setting. Consider two specifications S1 and S2 such that (Figure 8):

a. {true}S1{R1} and {true}S2{R2}

b. there is s_i, s_j . $R1(s_0,s_i)$ and

$R1(s_j,s_2)$ and $R2(s_0,s_j)$ and $R2(s_i,s_2)$

Then S1;S2 has the post condition

there is s_i . $R1(s_0,s_i)$ and $R2(s_i,s_2)$

and S2;S1 has the post condition

there is s_j . $R2(s_0,s_j)$ and $R1(s_j,s_2)$.

Such specifications are candidates for implementation as tasks in Ada to clarify the implicit parallel processing design for which VDM does not provide an explicit formalism.

6.0 CONCLUSIONS

In this article we have demonstrated that structured analysis and design specifications can be effectively used to guide the stepwise refinement of VDM specifications. Furthermore, the proof obligations in VDM can be used to prove the correctness and completeness of these specifications. We have also shown how VDM specifications can be used to identify the possibility of parallelism which can be implemented in Ada as tasks.

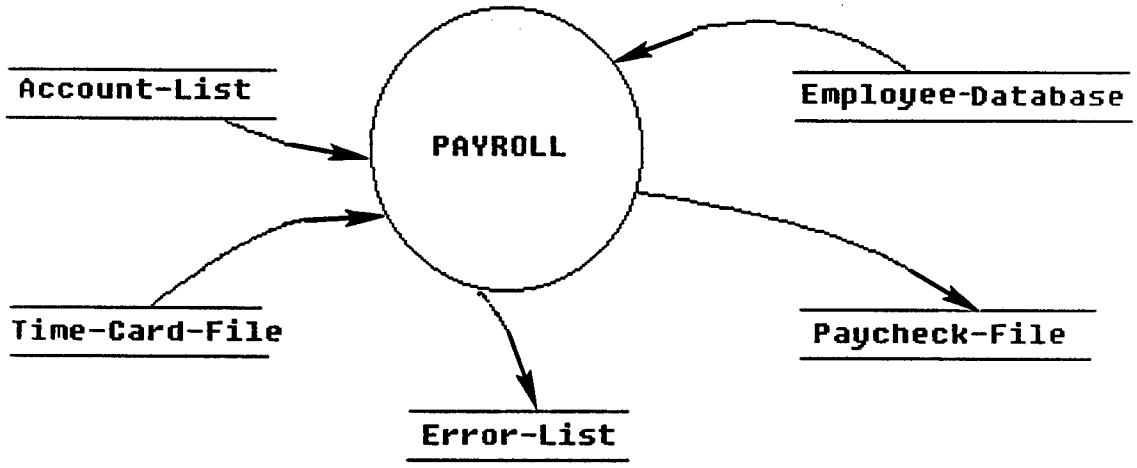


Figure 1. Context Diagram for the Payroll System

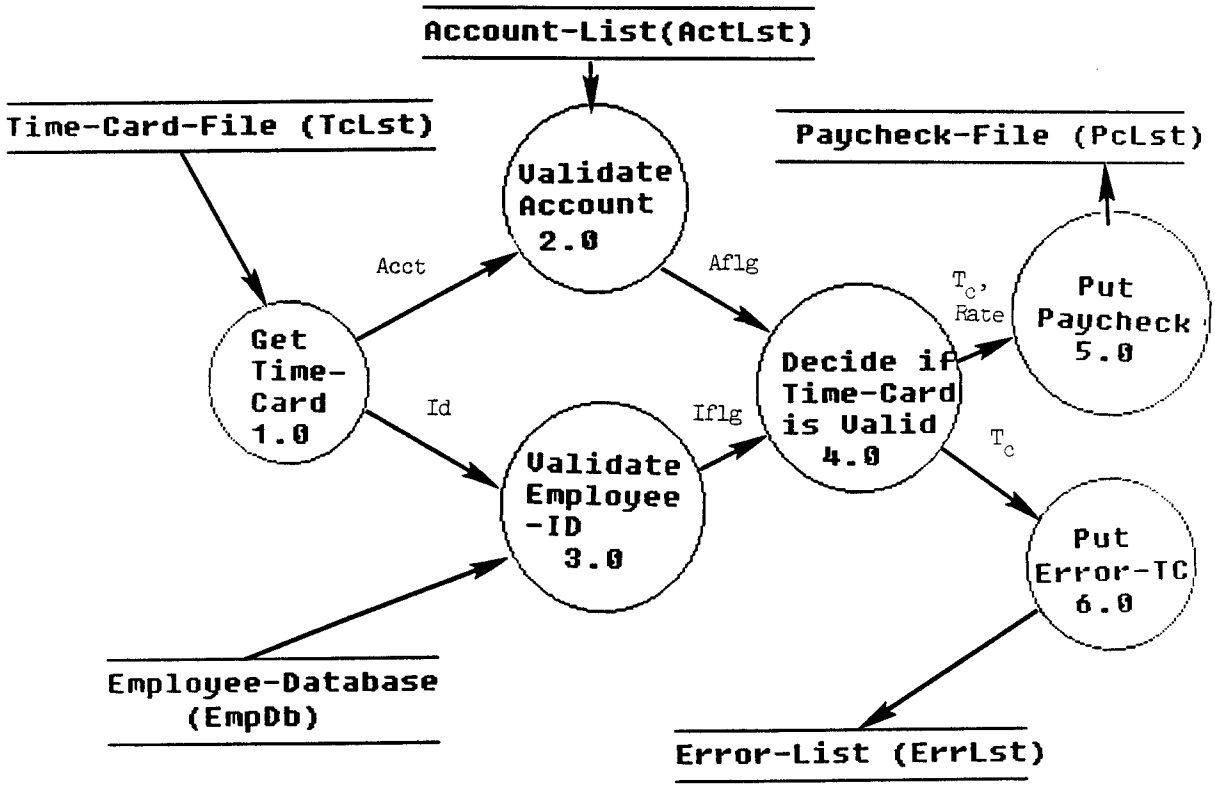


Figure 2. Overview Diagram of Payroll System (Diagram 0)

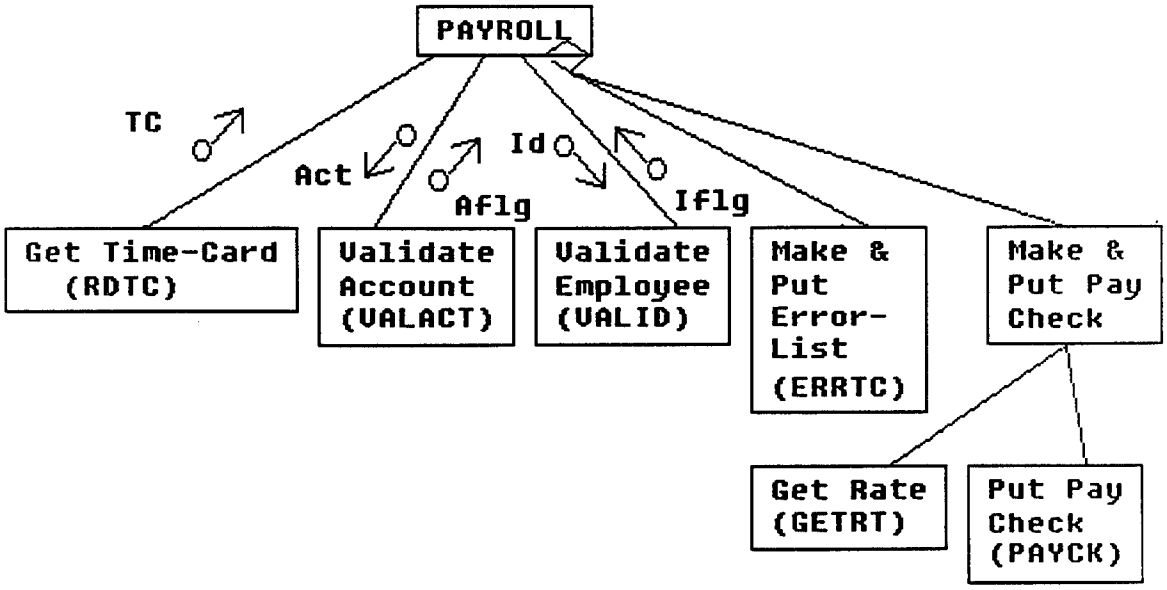


Figure 3. Structure Chart for Payroll System

Structured Analysis DD	VDM Names
Account	Acct
Account-DataBase = {Account}	ActLst
Account-Valid-Flag = [T/F]	Aflg
Employee-DataBase = {Employee-Id + Rate}	EmpDb
Error-List = {Error-Message}	ErrLst
Error-Message = Time-Card	Tc
Employee-Id	Id
Id-Valid-Flag = [T/F]	Iflg
Gross-Pay	GrPay
Hours	Hrs
Paycheck = Time-Card + Rate + Gross-Pay	Pc
Pay-Check-File = {Pay-Check}	PcLst
Time-Card = Id + Account + Hours	Tc
Time-Card-File = {Time-Cards}	TcLst

Figure 4. Payroll Data Dictionary

```
PAYROLL    =  map TcLst x EmpDb x ActLst to ErrLst x PcLst
TcLst      =  seq of Tc
              where inv-TcLst(s) ≜ card dom s = card rng s
Tc         =  compose Tc of  id:Id  acct:Acct  hrs:Hrs
Id         =  TOKEN
Acct       =  TOKEN
Hrs        =  REAL
EmpDb      =  map Id to Rate
Rate       =  REAL
ActLst     =  seq of Acct
ErrLst     =  seq of Tc
              where inv-ErrLst(s) ≜ card dom s = card rng s
PcLst      =  seq of Pc
              where inv-PcLst(s) ≜ card dom s = card rng s
Pc         =  compose Pc of  tc:Tc  rate:Rate  grpay:GrPay
GrPay      =  REAL
where inv-TcLst x ErrLst x PcLst(s1,s2,s3) = len s1 = len s2 + len s3
```

Figure 5. Abstract Syntax

```
PAYROLL
 ext rd tl:TcLst, rd db:EmpDb, rd al:ActLst, wr el:ErrLst, wr pl:PcLst
 pre db /= <> and al /= <> and pl = <> and el = <>
post len tl = len pl + len el and for all i belonging to inds tl .
     paid(tl(i),al,db,pl) or error(tl(i),al,db,el)
```

Figure 6.a. Specification, Context Diagram

```
act: Acct x ActLst --> BOOL
act(a,al) ≜ a belonging to elems al

idb: Id x EmpDb  --> BOOL
idb(id,db) ≜ id belonging to dom db

payrcd: Tc x Rate x PcLst --> BOOL
payrcd(tc,r,pl) ≜ mk-Pc(tc,r,r*hrs(tc)) belonging to elems pl

paid: Tc x ActLst x EmpDb x PcLst --> BOOL
paid(tc,al,db,pl) ≜ act(acct(tc),db) and idb(id(td),db) and
                    payrcd(tc,db(id(tc)),pl)

errcd: Tc x ErrLst --> BOOL
errcd(tc,el) ≜ tc belonging to elems el

error: Tc x ActLst x EmpDb x ErrLst --> BOOL
error(tc,al,db,el) ≜ not(act(acct(tc),db) and idb(id(tc),db)) and errcd(tc,el)
```

Figure 6.b. Auxiliary Functions

```
PAYROLL
 ext rd tl:TcLst, rd db:EmpDb, rd al:ActLst, wr el:ErrLst, wr pl:PcLst
 pre db /= <> and al /= <> and pl = <> and el = <>
var i:N
i := 0
;
pre i = 0
while i /= len tl do
  pre i = len pl + len el and i /= len tl
  inv i = len pl + len el
  rel i = i~ + 1 and let tc = tl(i) in
      ( act(acct(tc),al) and idb(id(tc),db) and payrcd(tc,db,pl) and
        len pl = len pl~ + 1 and len el = len el~) or
      ( not(act(acct(tc),al) and idb(id(tc),db)) and errcd(tc,el) and
        len el = len el~ + 1 and len pl = len pl~)
 post i = len pl + len el and i = i~ + 1 and let tc = tl(i) in
      ( ( act(acct(tc),al) and idb(id(tc),db) and
          payrcd(tc,db(id(tc)),pl) ) or
        ( not(act(acct(tc),al) and idb(tc,db)) and
          errcd(tc,el) ) ) and
      len pl + len el = len pl~ len el~ + 1
post len tl = len pl + len el and for all i belonging to inds tl .
     paid(tl(i),al,db,pl) or error(tl(i),al,db,el)
post len tl = len pl + len el and for all i belonging to inds tl .
     paid(tl(i),al,db,pl) or error(tl(i),al,db,el)
```

Figure 7.a. Specification, Overview Diagram

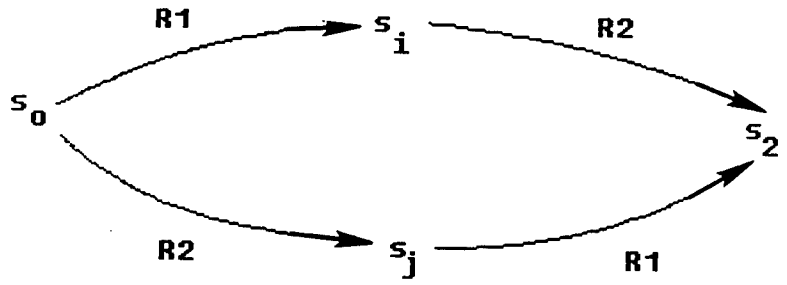


Figure 8. Condition for Parallelism

```
pre i = len pl + len el and i /= len tl
inv i = len pl + len el
var tc:Tc, aflg,iflg:BOOL, r:Rate
  i := i~ + 1
  ;
  RDTC(i:N1) tc:Tc
   ext rd tl:TcLst
   pre i belonging to inds tl
  post tc = tl(i)
  ;
  VALACT(acct(tc):Acct) aflg:BOOL
   ext rd al:ActLst
   pre true
  post aflg = act(acct(tc),al)
  ;
  VALID(id(tc):Id) iflg:BOOL
   ext rd db:Db
   pre true
  post iflg = idb(id(tc),db)
  ;
  pre true
  if aflg and iflg
  then GETRT(id(tc):Id) r:Rate
        ext rd db:Db
        pre idb(id(tc),db)
       post r = db(id(tc))
       ;
       PAYCK(tc:Tc,r:Rate)
        ext wr pl:PcLst
        pre true
       post payrcd(tc,r,pl) and len pl = len pl~ + 1
  else ERRTC(tc:Tc)
        ext wr el:ErrLst
        pre true
       post errcd(tc,el) and len el = len el~ + 1
 post len pl + len el = len pl~ + len el~ + 1 and
      ( payrcd(tc,db(id(tc)),pl) or errcd(tc,el) )
post i = len pl + len el and i = i~ + 1 and let tc = tl(i) in
     ( ( act(acct(tc),al) and idb(id(tc),db) and
         payrcd(tc,db(id(tc)),pl) ) or
       ( not(act(acct(tc),al) and idb(tc,db)) and
         errcd(tc,el) ) ) and
     len pl + len el = len pl~ len el~ + 1
```

Figure 7.b. Implementation of Loop Body

Bibliography

[AND88] Andrews, Derek, and Gibbins, Peter, An Introduction To Formal Methods Of Software Development, Units 1 to 4, The Open University Press, Walton Hall, Milton Keynes, UK, 1988.

[BJO87] Lecture Notes in Computer Science 252: VDM '87, VDM - A Formal Method at Work. Bjorner, D., Jones, C.B., Mac an Airchinnigh, M., and Neuhold, E. (eds.). Springer-Verlag, New York, 1987.

[BLO88] Lecture Notes in Computer Science 328: VDM '88, VDM - The Way Ahead. Bloomfield, R., Marshall, L., and Jones, R. (eds.). Springer-Verlag, New York, 1988.

[DEM78] DeMarco, T., Structured Analysis and System Specification, Yourdon Press, New Jersey, 1978.

[GAG88] Gagliano, R. A., M. D. Fraser, M. E. Schaefer and G. S. Owen. "Functionality in the Reusability of Software," Proceedings of the ACM 88 Computer Science Conference, 540-545, 1988.

[JON86] Jones, Cliff B., Systematic Software Development Using VDM, Prentice-Hall, New Jersey, 1986.

[MYE78] Myers, Glenford J., Composite/Structured Design, Von Nostrand Reinnhold Company, New York, 1978.

[OWE87] Owen, G. S., Gagliano, R. A. and P. A. Honkanen, "Functional Specifications of Reusable MIS Software in Ada," Proceedings of the Joint Ada Conference 5th National Conference on Ada Technology and Symposium, 19-26, 1987.

[RAS85] Rasmussen, Jens, "The Role of Hierarchical Knowlledge Representation in Decision Making and System Management," IEEE Transactions on Systems, Man , and Cybernetics, Vol. SMC-15, No.2, March/April 1985.

[YOU78] Yourdon, Ed. and Constantine, Larry L., Structured Design, Yourdon Press 1978.





Sue A. Conger is an Assistant Professor of Computer Information Systems at Georgia State University. She received her Ph. D. from New York University where she also served on the faulty. She has worked in management information positions for Mobil, Lambda Technology, Chase Manhattan, Educational Testing Service, and the U. S. Department of Agriculture. Her research interests include systems analysis, design methodologies, and CASE tools. She is a periodic contributor to the Auerbach Reports.

Martin D. Fraser is an Associate Professor in the Department of Mathematics and Computer Science at Georgia State University. He has worked as Statistical Analysis Manager, American Greetings Corp., Cleveland, an Information Systems Member at Western Electric, St. Louis, and a Software Analyst (Captain, USAF) at the Satellite Test Center, Sunnyvale. He has B.S. and M.S. degrees in Mathematics, and a Ph.D. in Mathematics, major in Statistics, from St. Louis University. His research interests are in computer science and statistics.





Ephraim R. McLean is Professor and George E. Smith Chair in Information Systems in the College of Business Administration at Georgia State University. Prior to joining the Georgia State faculty, he was the Chairman of the Computers and Information Systems area and the director of the Computers and Information Systems Research Program at the UCIA Graduate School of Management. He has an undergraduate degree in engineering from Cornell University and a Ph.D. from M.I.T.'s Sloan School of Management.

G. Scott Owen is Professor of Mathematics and Computer Science at Georgia State University. His research interests are in Computer Graphics, Visualization Techniques, Software Engineering, Ada Technology, Computer Science Education, and the application of computers to science and mathematics education. He received his B. S. from Harvey Mudd College in California and his Ph. D. from the University of Washington in Seattle.





Ross A. Gagliano is an Associate Professor of Computer Science at Georgia State University. He received his BS degree from the U. S. Military Academy, an MS in Physics from the U. S. Naval Postgraduate School, and an MS and Ph. D. in Information and Computer Science from the Georgia Institute of Technology. Prior to coming to GSU, he was a Senior Research Scientist at the Georgia Tech Research Institute. His major areas of interest include discrete element modeling, simulation, and software engineering.

Kuldeep Kumar is an Assistant Professor of Computer Information Systems at Georgia State University. He received the Ph. D. degree from McMaster University. His research interests include Function Point Analysis, CASE tools and MIS Life Cycle analysis.



Vijay K. Vaishnavi received his Ph. D. degree in electrical engineering from the Indian Institute of Technology, Kanpur, India. He is currently a Professor of Computer Information Systems at Georgia State University. He has authored numerous papers in algorithms, data structures, and other related areas. Dr. Vaishnavi is a senior member of the IEEE and a member of the Association for Computing Machinery.

Formal Design Methods for Dynamic Ada Architectures

Christopher M. Byrnes

The MITRE Corporation

ABSTRACT

The use of dynamic software architectures that change during the execution of a program offers some advantages over strictly static architectures. Dynamic architectures offer the possibility of being automatically extended to handle unforeseen changes in input and/or output, and they allow designers to quickly rearrange software modules for tradeoff and debugging purposes. But dynamic architectures must be used carefully to account for additional concurrency controls and resource contention; hence the need for design methods that help one to identify and minimize potential problems. This paper presents experiences and lessons learned from the use of formal design methods in creating dynamic architectures for an Ada implementation of the DIANA Query Language (DQL).

1 INTRODUCTION

Interest in the use of software architectures that change or mutate dynamically in response to system inputs and/or developer decisions comes from a variety of areas. Object-Oriented Design (OOD) methods have encouraged developers to think of the system as a network of communicating objects, where the interfaces that accept the messages between objects and the types of those messages are well-defined. The Ada programming language (DOD 83) supports these aspects of OOD; in addition Ada allows the dynamic creation and destruction of objects through task allocations and generic instantiations.

With the growing use of Ada in a variety of applications, there was a need to explore how the use of Ada's dynamic architecture features is reflected in a design method. The use of these features of Ada is a matter of some controversy (Firth 87). Software designers and evaluators need to know where their use is appropriate and how to best document the use of these Ada features. Fortunately the opportunity to address these issues during the design of a software technology product presented itself to us.

The Ada Skill Center at The MITRE Corporation identified a need to build a system to aid with the semantic analysis of Ada code and Program Design Language (PDL) delivered under government contract. An analysis system based on the Descriptive Intermediate Attributed Notation for Ada (DIANA) (Evans 83) was built; it is called the DIANA Query Language (DQL) (Byrnes 89). DQL allows queries to be formed based on the semantics of an arbitrary Ada program. These queries are formed from user menu selections and the query results are displayed on a user's workstation screen.

The design of DQL requires methods that handle DQL's use on a networked workstation. The numbers and types of workstation windows vary with the kinds (and sizes) of Ada software being analyzed and with the preferences of the analyst using DQL. Frequent changes in the configuration and relationships between these windows were expected. DQL's design team decided that this application was a candidate for dynamic architectures within an Ada implementation.

DQL currently exists as a prototype, running on a network of Sun 3® workstations. DQL's user interface and menu system were built on top of earlier work on the Electronics Systems Division (ESD) Acquisition Support Environment (EASE) (Byrnes 88). The DIANA intermediate forms that the queries operate against are provided by existing DIANA tree generators such as those provided by the Anna tools (Sankar 85) and the Intermetrics Ada Compilation System (ACS) (Intermetrics 87).

2 CURRENT ADA DESIGN METHODS

A variety of software design methods that are specifically targeted toward Ada software development have evolved. Examples of these include the methods of Booch (Booch 87), Buhr (Buhr 84), and Cherry (Cherry 86). Others have taken traditional software design methods such as Structured Analysis/Structured Design (SA/SD) (Yourdon 79) and have updated them to better support Ada (Wild 89). While the implementation details vary between methods, all these methods share some overall characteristics.

A designer using one of these methods works by defining a hierarchy of design entities and their interfaces. One of the products created when the designers apply these methods is an architecture that describes how these entities are related to one another. A static architecture (often represented graphically) shows how the data and control flow through the entities. The emphasis is placed on steady state processing, with the activities of start-up, shutdown, and exceptional conditions treated mainly as side issues.

® Sun and Sun 3 are trademarks of Sun Microsystems, Inc.

In addition to defining the entities within architectures in terms of the calls between them, these methods organize the entities by what "class" or type they belong to. The methods are oriented toward Ada implementations, so a "class" refers to Ada language entities such as task types and generic packages that can encapsulate both procedural (things that can be called) and data (things that can hold information or persistent state) abstractions. Some entities within an architecture are not part of a "class" (such as non-generic packages) or alternatively can be viewed as being the only example or instance of an undefined (or anonymous) "class."

For those entities such as generics and task types where multiple instances of the overall "class" are possible, these methods keep track of the instance objects and where they are created. The static architectures of these methods also define (through graphical notations) the generic units and task types as well as the objects (instances) of those generics and types. These architectures are static because while these methods define where objects are created, they provide less support for defining when (under what conditions) these objects are created.

The entities, objects, and Abstract Data Types (ADTs, combining procedural and information encapsulations) created during the initial portion of the design are mapped to Ada packages, subprograms, tasks, and type declarations during the detailed design portion of the method. Requirements traceability and other management information is often maintained as structured Ada comments in the Ada PDL and code being created. As the design progresses, the PDL in the bodies of these Ada design units is gradually refined until a full Ada implementation is generated.

3 REQUIREMENTS ON DQL'S DESIGN METHODS

When deciding what methods to use to develop DQL's design, the implementation team saw a need for methods that covered a variety of design issues. These included design issues often skimmed over by existing Ada design methods. This section outlines these design issues.

3.1 DYNAMIC ARCHITECTURES OF CONCURRENT ENTITIES

One requirement was that the design methods support dynamic architectures of concurrent entities. The DQL implementation involved windows operating on a workstation screen, where individual windows are created and destroyed during a user session. In keeping with the EASE system's use of interacting windows, DQL windows have operations (including creation and destruction) that result in changes to both themselves and other related windows. Therefore the design methods had to handle entities that are dynamically created and that can affect any other entities within the system. The design methods had to address when these windows (entities in the architecture) are created and when calls are made to each instance in addition to just defining the operations available to each window.

3.2 CLEAN MAPPING TO IMPLEMENTATION ENTITIES

Another requirement was that the design entities created by the methods must cleanly map to implementation entities. The intended target workstations for DQL use the Sun Visual/Integrated Environment for Workstations (SunVIEW)® (Sun 88) as the graphical workstation interface. Like other UNIX® environments, SunVIEW allows each major window to be its own independent UNIX process. A workstation windowing executive is responsible for distributing input/output events to the appropriate window and for managing communications between windows. This means that windows operate concurrently except for certain well-defined behaviors provided by the programmer.

DQL's design had to fit into the workstation environment and use its built-in capabilities as much as possible. Because DQL was not a workstation environment development exercise, its design was constrained by not requiring any major functions that the existing environment (SunVIEW) did not already provide. Therefore, DQL's design methods had to cleanly map onto SunVIEW's model of independent processes that cooperated through a (supplied) central environment coordinator.

In addition to being mapped onto the implementation entities provided by SunVIEW, the design methods used by DQL also had to produce design entities that are mapped onto Ada tasks. As described above, DQL's intended implementation involving interactive queries and windows implied dynamic entities at both a high level (the UNIX process/window level) and the lower levels of individual queries (and the subqueries that made up a large query). Independent threads of control are needed within some UNIX processes to manage these finer-grain issues. Therefore, DQL's design methods had to map onto Ada task types as well as they did onto UNIX processes.

3.3 DISTRIBUTION ACROSS A NETWORK

Another requirement on DQL's design methods was that they create entities that are distributed across a network of workstations. Early experience with DIANA trees had shown them to be very verbose; if each workstation and each separate process within a workstation had to build or read in these large tables, then the interactive response times expected of DQL would not be met. There was also concern that large DIANA trees could not be read in by one workstation due to the limited amount of local memory available. Creating DIANA trees from the original Ada source code can be computationally-intensive; only a powerful computer can do this within reasonable response times. Only a few server or mainframe computers on a network might have the necessary memory and processing power to handle large DIANA trees.

In addition to resource concerns, there was a desire to have DQL support a team of Ada analysts working on a network of workstations. Some of the resources this team uses are commonly shared, as with some network servers. Others are individual to an analyst such as the workstation and its displays. The DQL design methods had to create entities that might range from being exclusive to a user to being shared by many.

® SunVIEW is a trademark of Sun Microsystems.
® UNIX is a trademark of AT&T Bell Laboratories.

Because some performance and size issues are not fully understood until some implementation and modeling is done, DQL's design entities had to allow for redesign. Experience might show that DQL runs better if its components were distributed differently around the network. A DQL implementation might also be called on to analyze an unusual collection of Ada software (very large, multiple versions, classified software, etc.); a configuration of entities distributed over the network might have to be changed for the purposes of this analysis. Therefore, DQL's design methods had to create entities whose allocations onto a network of workstations were easily rebound as necessary.

3.4 ENCAPSULATION FOR MULTIPLE IMPLEMENTATIONS

Another requirement was that DQL's design methods establish encapsulated interfaces and behaviors to allow multiple implementations. While DQL's initial implementation was targeted toward Sun workstations running SunVIEW, eventually DQL would run on other vendor's workstations using industry standard windowing systems such as X Windows® (Scheifler 88). Any future porting effort depends on a design that is abstract enough to be mapped to these future targets. Therefore, DQL's design methods had to be able to describe a design entity's interfaces and behaviors for the benefit of these porting efforts.

3.5 CLIENT-SERVER RELATIONSHIPS

Another requirement was that the design entities had to support client-server relationships. Both the current (SunVIEW) and future (X Windows) workstation environments that support DQL use such as a client-server model of behavior. Any additional central repositories of DQL information or control (such as DIANA tree servers) also use this client-server model between the workstation's windows and the common program(s) shared by multiple users.

Within an object-oriented design method (such as those described earlier), client-server relationships establish which concurrent entities are passive (awaiting calls to provide a service), which are active (requesting services from other entities), and which are both. Client-server relationships also establish which entities must be synchronous or asynchronous with other entities. Because the existing workstation environment has entities based on clients and servers that define synchronous and asynchronous behavior, DQL's design methods must create entities that use the existing functionality and methods for tying objects together.

Designing (and reusing) such client-server relationships means establishing the communications protocols between the server and the multiple clients. Often it also means defining a "stream" model of how information flows back and forth. This was true for DQL because results from queries are modeled as intermittent long-lived streams of results generated as very large DIANA trees are searched for the occasional Ada construct of interest. Therefore DQL's design methods had to support the definition of streams of information flowing between servers and clients.

® The X Window System is a trademark of the Massachusetts Institute of Technology.

3.6 DANGERS OF INFORMAL APPROACHES

The requirements on the design methods described above could have been met by just adding some sort of free-form narrative to a standard Ada design notation. The existing notations and design rules as described in one of the standard texts (Booch 87) would have been used, with additional side comments added to cover one or more of these issues. This is a trend among some system designers, where a standard design method is used and any unique aspects of the application under development are handled as a side issue by secondary design notations.

This can be a dangerous trend, especially when the issues treated as "secondary" by a standard design method (and its tools) are in fact central to a particular application. Generally, software methodologists recognize that their approach cannot cover every possible application, so they anticipate (and usually encourage) the customization of the method to a particular application. The danger comes when the connections between the original method (and its associated model of computation) and the extensions created for an application are so weakened that the overall model of how the design fits together falls apart.

We wanted to avoid this problem by using an underlying collection of design methods that were strongly connected to one another. Ada was the base of the semantics of the methods and how they connected to each other. These connections between methods have to exist both at the visual level (so a designer knows where to turn next to follow an idea from one method's representation to another), and at the level of a computation model so the development tools, run-time behaviors, and designer's understanding of the system remained coordinated. DQL's designers felt that formal mathematical notations provided the best foundations for combining different design methods.

4 SOFTWARE DESIGN TECHNIQUES USED

In the previous section, a set of requirements on design methods to be used in DQL were defined. This section introduces the individual software design methods that were chosen for this design effort. This section also shows how each method meets one or more of the design issue requirements. These methods cover both how the design is organized into entities and how those entities are visually presented.

4.1 MATHEMATICAL FORMALISMS

One mathematical formalism used was first-order predicate calculus (Berg 82). Predicate calculus provides the standard "for all" ($\forall$), "there exists" ($\exists$), "implies" ($\rightarrow$), "such that" ($\Rightarrow$), "equivalent to" ($\Leftrightarrow$), "and" ($\wedge$), "or" ($\vee$), and "not" ($\neg$) operators that can be used to specify the relationships between entities in some domain. Some simple examples are shown below:

$$\forall\ A : \text{data_type} \Rightarrow f(A) = \text{true} \rightarrow g(A) = \text{false},$$
$$\exists\ B : \text{data_type} \Rightarrow B \Leftrightarrow f(\text{some_constant}).$$

The quantifiers (such as A and B above) of predicate calculus allow properties of data to be specified. Since DQL's de-

sign involved the creation and management of many types of data, predicate calculus was used to specify how data was related. Because this description of DQL's data is based on mathematics, there is less opportunity for vagueness and different interpretations than if informal natural language specifications were used.

Among the types of data that DQL's design created were Ada tasks (which are objects of an Ada task type). Predicate calculus allows designers to specify information about tasks, particularly those currently in existence (still running). Because Ada tasks can be of different types (and subtypes), quantifiers can be used to limit the tasks being described by a particular predicate calculus formula to just a subset of all the tasks that might exist at some point in time.

Another formalism (often used with predicate calculus) was axiomatic specifications (Goldblatt 82). Axiomatic specifications can be used to define a data type and/or an interface to a subsystem through the operations that can be performed. The classic example of an axiomatic specification is a stack. A brief excerpt from a stack's specification is shown below:

S : Stack, I : Item,
pop(*new* Stack) → **error**,
pop(push(S,I)) = I.

Note that some operations (such as popping an empty stack) are defined as illegal. Also note that the behavior of the data type can be defined by combining the primitive operations on the type (as in a "push" immediately followed by a "pop" returning the item "pushed"). Often the mathematical technique of induction is used, where an initial state of the data type (or package) is defined, and then later operations upon that state can be traced back ("inducted") to the original or some other observable well-defined state so an observer can deduce the behavior from analyzing previous calls. This allows axiomatic specifications to be used to define state machines.

DQL's design involved many subsystems whose behaviors and interfaces have to be encapsulated. The dynamic creation of entities such as workstation windows will also result in entities whose collective behavior (such as on a per-workstation or server basis) must be managed inductively while the individual behavior (what operations are legal) is specified axiomatically from the type (or class) of the entity. An example of induction is to define a basis case (or state) when an entity is created and then induct over the calls to that package's subprograms or task's entry points. DQL's design also had to address individual and collective behavior under steady state, exception (error), and termination conditions (both normal and abnormal/exception terminations).

Another formalism used was temporal logic (Hailpern 82), which can be used to define independent threads of control, the events they perform, and the partial or total order in which those events must occur. A simple example is shown below:

A, B, C : processes; D, E : events →
(A **performs** D **or** B **performs** D) **before** C **performs** E.

Here a partial order of who does what has been defined. DQL's design involved (often large) numbers of (semi) independent threads of control whose behaviors had to be described and coordinated with respect to each other. Since DQL's operations acted over periods of time (often of unknown length), DQL's design needed a method that had some notion of time and how activities existed in it. Temporal logic provided such a method that was formal enough to limit ambiguous interpretations and yet flexible enough to allow partial orderings so designers do not have to overspecify the exact order of every possible event for those events that did not affect one another.

4.2 GRAPHICAL NOTATIONS

As with other Ada implementation efforts, DQL's designers wanted to use a graphical notation to provide a higher-level view of the system's design than was seen with strictly textual notations. DQL's designers also wanted to use or extend graphical notations so that they covered all (or most) of the requirements on DQL's design methods (described earlier). This was particularly true of the dynamic architectures to be used in DQL. These graphical notations also had to maintain the same level of formality (of behavior and interface descriptions) that had been established by the mathematical formalisms described above.

DQL's graphical notation is based on standard Buhr diagrams (Buhr 84). Buhr diagrams allow software (or hardware) to be described by iconic units of design that can be arranged into hierarchies. These icons also define the interfaces that identify what services are provided by an icon (unit) to the rest of the design. Control and data flow arrows similar to those used by SA/SD show the interactions between these units.

The simple examples below show how Buhr diagrams can be used to define the architecture of a system. In figure 4-1 below, the parallelograms with straight lines (such as Parent) represent instances of Ada tasks. Those with dotted border lines (such as Node) represent Ada task types that may be allocated. Entry points to these tasks are represented by the text enclosed in a box (such as *input*) that acts as "ports" into the hidden state of the tasks. Buhr diagrams allow more complex rendezvous semantics to be specified graphically; for this example simple rendezvous is assumed. The large arrows indicate a rendezvous between tasks, with the data flow arrows (o→) showing what information is being passed. This example shows the three rendezvous numbered to indicate the order in which they occur.

Figure 4-1 and the rest of the figures in this section will show the creation of an architecture of Ada tasks to implement the following DQL query:

((**all subprogram_address** <source_range>) **union**
(**all entry_address** <source_range>);

where this compound query is broken down into two primitive queries (**entry_address** and **subprogram_address**) and a query (**union**) that will combine any results from the primitive queries into a single collection of results (in this case, of Ada addresses).

Note that while all the entry points are visible, only the Manager task calls the *subquery* entry point for the task Child_2. Only the Child_2 task calls the *input* entry point for the task Parent, so there is an implicit assumption (which will be

formalized) not stated in the diagram of which tasks will call which other tasks in this architecture.

Buhr diagrams can also be used to describe architectures that dynamically change over time. In figure 4-2 below, the architecture of the system is shown in its initial state. The overall task named Server has one entry point (*search_for*), a hidden Manager task that is currently running (having been activated when the Server was created), and a Node task type that may be instantiated. This architecture encapsulates the architecture described earlier in figure 4-1. Any caller to Server was unaware of these details, but an implementor must understand these interactions between calls to *search_for*, the Manager task, and the new Node tasks.

At some point this Server might be called on to do some processing, so a call to *search_for* caused the architecture of this system to be expanded, as shown in figure 4-3 below. The initial static architecture (as shown in figure 4-2) has been expanded through the creation of three tasks of the task type Node, indicated by the three large arrows. The architecture of the system now corresponds to the example shown in figure 4-1. The Manager task will also set up the relationships between those Nodes designated as children (running primitive DQL queries) and the Node acting as the Parent (running a DQL combining query). The details of these relationships are described later.

Buhr diagrams can be used to show steady-state processing of the units in the architecture. Figure 4-4 below shows the calls that might be made to the various entry points of the tasks created in figure 4-3. Even with a fairly simple architecture such as this one, trying to show all the rendezvous lines results in a cluttered diagram.

As a dynamic architecture terminates or deletes some of its units, a Buhr diagram can be used to show the removal of those units. Figure 4-5 below shows the architecture created in figure 4-3 at some point in the future when all of the processing done by the three dynamically created tasks is completed. The three tasks have terminated and have been removed (with any associated garbage collection, perhaps done automatically by the operating system), as indicated by the three arrows. This system has returned to its initial state as shown in figure 4-2.

In the examples below in figures 4-6 to 4-12, the complicated Buhr diagram shown above in figure 4-3 is presented as a Buhr diagram sequence. As shown in figure 4-1, Buhr's notation is able to define strict orderings of events. Within DQL, only some of these events need to be in strict orders; the use of partial orders minimizes premature serialization decisions.

A Buhr diagram sequence adds a third dimension of time. An individual Buhr diagram provides a snapshot of the overall system's state at a point in time. Only those packages, task activations, and generic instantiations that exist (have been elaborated) by then are shown. Only those control/data flows (represented with the usual arrows) active or in progress at that time are drawn. The goal is to provide in one Buhr diagram just those activities related to an individual transition in the system's state. Buhr diagram hierarchies can be used to isolate subsystem and unit body behaviors.

Each of the individual Buhr diagrams is assigned a number (as with the normal practice) or a name. A whole series of these diagrams is constructed from the individual diagrams; this forms the diagram sequence. By flipping through the individual diagrams in a sequence, an Ada designer can see how the system changes over time.

The following collection of diagrams illustrates such a sequence. These diagrams provide more details on how the tasks created in figure 4-3 are really constructed. Assuming an initial state as shown in figure 4-2, the first step in the creation sequence (shown in figure 4-6 below) shows the call to the *search_for* entry point of the Server task, which immediately results in a call to the *query* entry point of the Manager task. The second step in figure 4-7 below shows the creation of the first Node needed to compute this query, the Node named Parent. The third step in figure 4-8 shows the Manager passing the subquery to be processed to the *subquery* entry point of the Parent Node. The fourth and fifth steps (figures 4-9 and 4-10, respectively) show the creation and subquery assignment for the Node named Child_1. The sixth and seventh steps (figures 4-11 and 4-12, respectively) show the same thing for Child_2. With this completed, all the subqueries making up the overall query are running and possibly producing results (as shown earlier in figure 4-4).

Sometimes the individual steps in a Buhr diagram sequence represent opportunities for parallel behavior; in other cases a protected resource might force serialization of some of the steps. DQL's design included simple diagrams (acting like Petri Nets or Gantt Charts) that show how individual Buhr diagram sequences such as figures 4-6 through 4-12 are partially ordered. Each node on one of these simple Petri Nets will contain the number or name of an individual Buhr diagram.

The usual Petri Net semantics are used to show which of these Buhr diagrams (sequences) can be run in parallel and which must be serialized. The "enabling token" enters each node of the Net from its top (causing the behaviors in the associated Buhr diagram to occur) and then one or more enabling tokens are passed onto the next node. As with Buhr diagrams at the single "snap-shot" level of system state, the sequences and their Petri Nets can be arranged in hierarchies that help encapsulate behaviors.

In the earlier example of a Buhr diagram sequence (figures 4-6 through 4-12), the Server task begins in an initial (start) state (as shown by figure 4-2) and ends in a final state with the DQL query running (in its individual subqueries) as shown by figure 4-4. A Petri Net that shows the legal orderings of the seven subquery initialization steps is shown in figure 4-13 below. This Petri Net assumes that once the query to be done has been input, the three subqueries needed to implement it can be created in any order (or simultaneously, if the target computer is capable). If instead each Node task needed to be informed of its parent's identifier (such as an Ada task pointer) so it knows who to communicate with, then the Parent Node would have to be created before the two Children. Figure 4-14 below shows this change from the original Petri Net of figure 4-13.

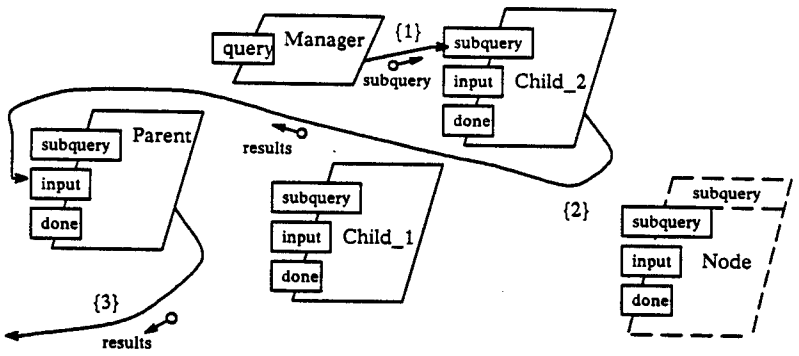


Figure 4-1. Simple Buhr Diagram Architecture

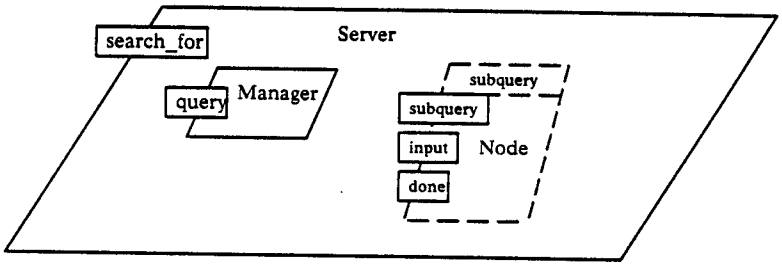


Figure 4-2. Initial Static Architecture

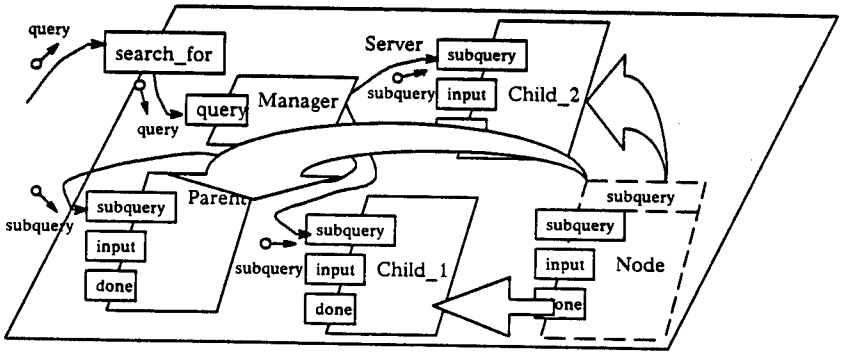


Figure 4-3. Dynamic Expansion of Architecture

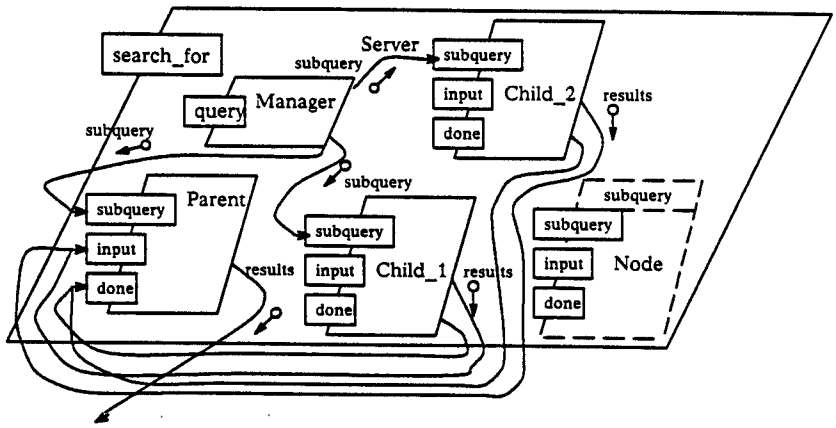


Figure 4–4. Steady State Processing

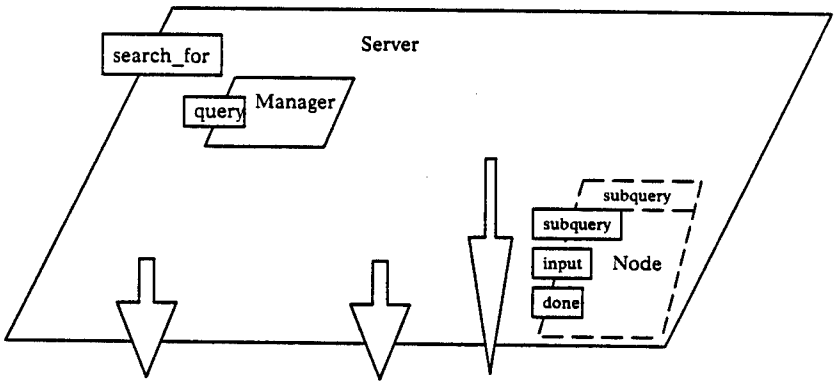


Figure 4–5. Dynamic Termination in Architecture

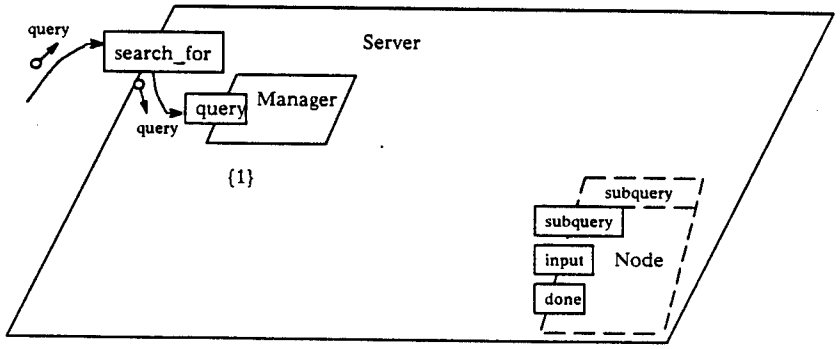


Figure 4–6. Manager Informed of Query to Search for (Step #1)

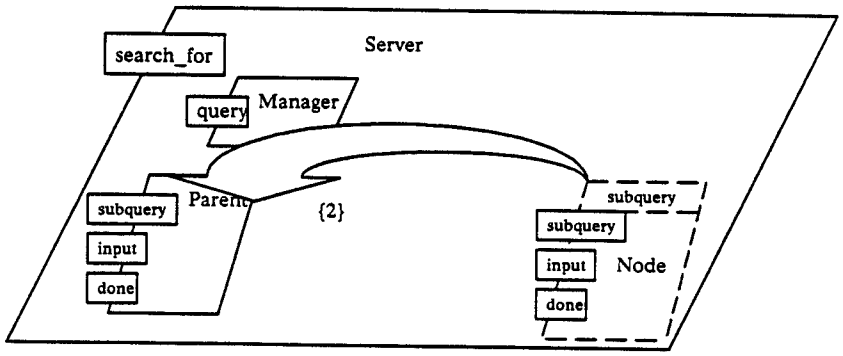


Figure 4–7. Creation of Parent Node (Step #2)

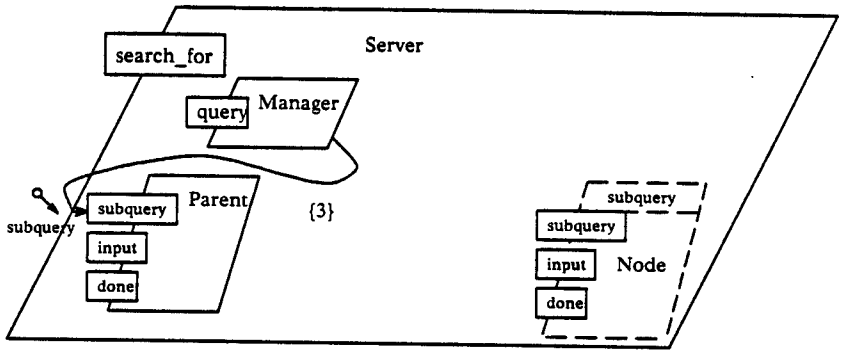


Figure 4–8. Parent Node Informed of Subquery to be Performed (Step #3)

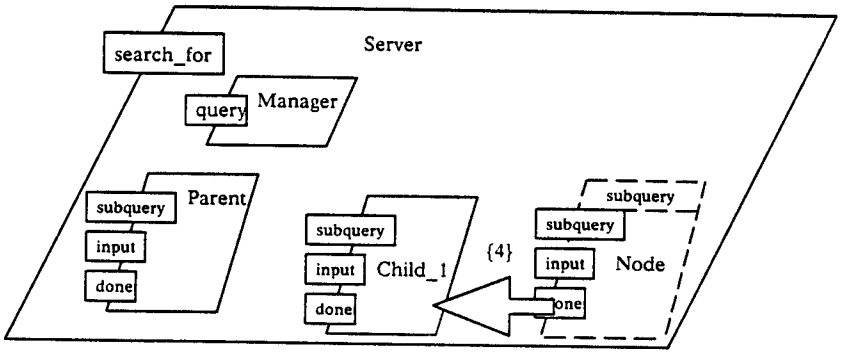


Figure 4–9. Creation of Child_1 Node (Step #4)

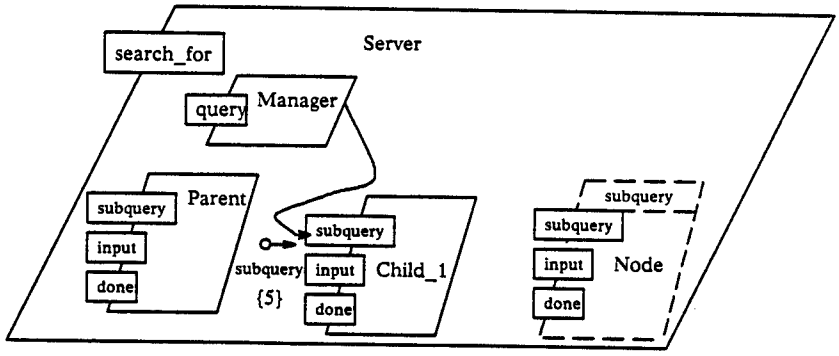


Figure 4-10. Child_1 Informed of Subquery to be Performed (Step #5)

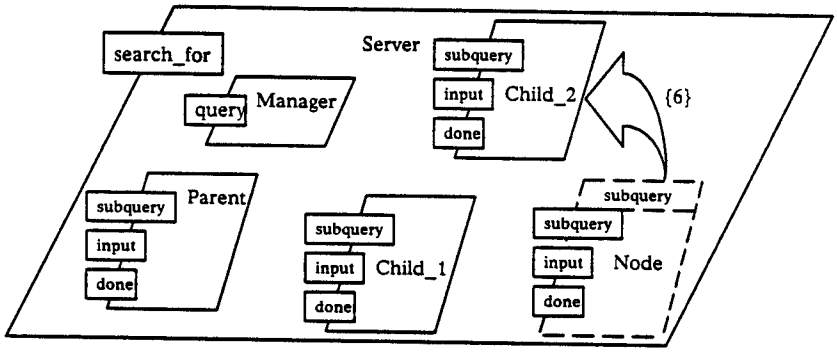


Figure 4-11. Creation of Child_2 Node (Step #6)

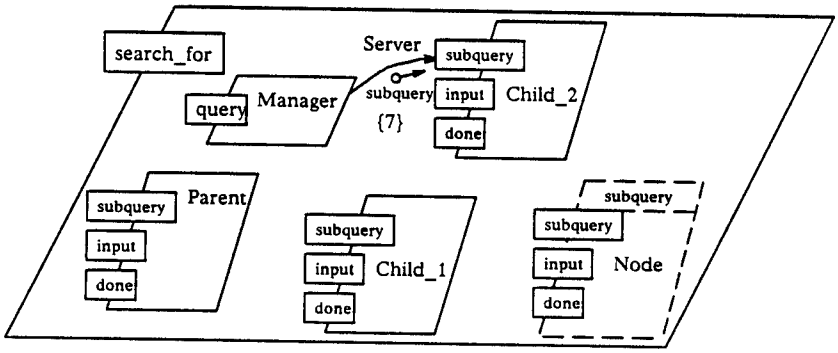


Figure 4-12. Child_2 Informed of Subquery to be Performed (Step #7)

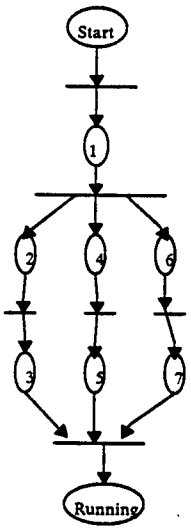


Figure 4-13. Petri Net for Buhr Diagram Sequence

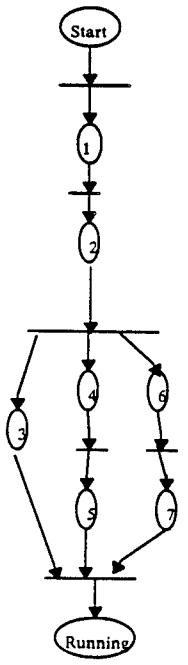


Figure 4-14. Alternative Petri Net for Buhr Diagram Sequence

4.3 TEXTUAL NOTATIONS

The main textual notation in DQL's design was Ada itself, used as a PDL. Standard Ada subprograms, packages, tasks, and generic units were used to define the units in the design. Ada's scope, visibility, and instantiation rules were used to define how these units were connected into an architecture. DQL's design began with the specification of these units, with the definition of the lower-level algorithmic details performed later as detailed design of the units' bodies.

In order to capture aspects of unit behaviors such as preconditions, postconditions, and state machine behavior (such as might be implemented as a package or task), a formal specification language was used. To remain consistent with the use of Ada, an annotation language was chosen that provides the formal notations described earlier (such as predicate calculus) within Ada. The Annotated Ada (Anna) (Luckham 87a) language provides such notations as well-structured Ada comments interspersed in normal Ada text.

The sample below shows a simple example of Anna usage. Note that all of the Anna constructs appear as Ada comments (after the "--" characters) and have a vertical bar ('|') as the Anna sentinel character. Also note that since Ada text cannot provide mathematical characters such as '∃,' '∀,' '⇒,' '⇔,' '∧,' and '→,' spelled-out English (ASCII) equivalents such as "exists," "for all," "=>," "<->," "and," and "->" are used instead.

```
    procedure delete_result(result : in out
                            result_ID);
--| <<Deleting_Existing_Result_Collection>>

-- preconditions on the input and current state

--| where result isin result_table'COLLECTION,
-- input result must be an allocated result_table

--|  result.current_readers.count > 0 =>
--|                          raise still_being_read,
-- there must be no readers of this table left

--|  raise still_being_read => result = in result,
-- this Ada exception will not cause result to be
-- changed here upon procedure termination

-- postconditions

--|  out (result not isin result_table'COLLECTION
-- this result's table data is reclaimed, and
--  so no longer valid

--| and (for all R : result_node_list =>
--|    (R isin in result_node'COLLECTION and
--|      in_here(in result,R) ->
--|     (R not isin result_node'COLLECTION))));
-- any results in this table are freed up
```

The Ada procedure specification for `delete_result` has had a series of Anna annotations defined after it (these annotations are collectively named by the Anna label `Deleting_Existing_Result_Collection` (the text enclosed by "<<...>>"). Several preconditions have been defined on the input parameters for any caller to `delete_result` such as

passing a valid result_ID and having no remaining users of result. If the check of the overall system state (the abstract count component of the result input parameter defined elsewhere for instances of result_table) does show someone else is reading this, then the Ada exception still_being_read will be raised and the input value result will be unchanged. Note that this formal specification can be more exact than either the normal Ada comments next to the annotations or this English narrative.

The postconditions for delete_result indicate that after a successful call to it, the result will no longer exist. Any individual result_nodes that happened to be in the result (as defined by the Anna virtual function in_here, defined elsewhere) at the time of the call (as indicated by the "in" modifier) are also reclaimed.

While Anna provides notations that allow formal definitions of most of the constructs in the Ada language, it does not include extensive support of Ada tasking. Here a separate Ada annotation notation known as the Task Sequencing Language (TSL) (Luckham 87b) was used. As with Anna, TSL uses structured Ada comments to specify the behavior of the Ada entity (in this case, Ada task types) the comment is placed against. These annotations are converted to run-time checks by TSL tools, but at the time DQL was designed those tools were not available. TSL was used instead as an unprocessed design notation.

TSL can be used to capture the behavior of the multiple threads of control (tasks) through temporal logic specifications. The TSL language defines activities and events that correspond to Ada tasking events, so designers can immediately refer to the behaviors of threads through their use of Ada tasks. TSL also allows designers to create their own user-defined events that are treated like the pre-defined Ada tasking events. This allows designers to tie in the behavior of major application-specific activities that might not involve Ada tasking or may be combined with tasking activities into the overall TSL description of how the threads of control behave.

The examples below are of some TSL specifications extracted from DQL. An Ada task type will be declared, with the structured annotations (indicated by the Ada "--" comment indication followed by the TSL '+' sentinel character) appearing after it. The Parent task created earlier in figure 4-7 is an example of the concentrator task types used in DQL to merge results from primitive subqueries together.

```
task type unary_concentrator is
  entry initialize(results  : result_ID;
                   parent   : parent_description;
                   operator : tree_node_type);
  entry result_available(result : result_node_list);
  entry result_finished;

-- User-defined events
--+ action input_read;
--+ action output_created;
--+ action output_completed;
--+ action parents_completion_notification;
```

These first four TSL statements specify some user-defined events that may be referred to in later TSL statements. Ada tasking events are implicitly declared; these user-defined events are explicit. The designer can indicate where these events occur by placing a TSL statement such as "--+ perform input_read;" in the body of this task (type) next to the Ada statement that performs this event. These events can be referred to in other TSL statements (such as temporal logic statements on correct event orderings) just like the standard Ada tasking events.

```
-- User-defined property of instances of this
--  task type
--+ property initialized_first(unary_concentrator)
--+                              : boolean := false;
```

These user-defined properties allow designers to quantify (over the tasks that are currently running) and test the state of tasks and instances of task types. The initialized_first property (of unary_concentrator, this and all other TSL examples in this section are for that task type) takes an instance of this task type as an argument and returns a boolean value of true if the (user-defined) activities associated with initialization have been done for this task. Those activities are defined in the body of this property. The default property value of false indicates that initially the task activations have not been initialized.

```
--+ <<Somebody_Calls_Initialize_Once>>
--+ when self activates then self accepts ?M at
--+                                    initialize
--+      where not initialized_first(self);
```

This TSL statement (which can be referred to by the label in brackets <<...>>, Somebody_Calls_Initialize_Once), indicates that when the currently executing task ("self") of this task type is allocated ("activates"), then that task will accept some unknown task (indicated by the placeholder "?M") at the initialize entry point. But this will only happen if the initialized_first property is false, indicating the task hasn't already been initialized since that should only be done once. Normally the Manager task (of figures 4-1 through 4-12) will be the one ("?M") who calls this task at initialize; but there is no need to "hard-wire" that assumption into this TSL specification, and so a placeholder is used instead.

```
--+ <<Nobody_Calls_Initialize_Afterward>>
--+ not self accepts ?M at initialize where
--+                 initialized_first(self);
```

This TSL statement is the opposite or negative of the previous one. After the earlier Somebody_Calls_Initialize_Once statement occurs, the property initialized_first will be true. Nobody_Calls_Initialize_Afterward disallows anyone ("?M") else from calling this task at initialize. These TSL statements begin to define both the temporal orderings of events and how the Ada "accept" statements in the task type's body (as well as other tasks in the system) must be constructed.

```
--+ <<No_Results_Accepted_Until_Initialized>>
--+ not self accepts any at result_available
--+      where not initialized_first(self);

--+ <<Nothing_Completed_Until_Initialized>>
```

```
--+ not self accepts any at result_finished
--+       where not initialized_first(self);
```

These two TSL statements disallow anyone from calling either the result_available entry or the result_finished entry until this task has been initialized. The earlier TSL statements indicate what has to be done to make that happen.

```
--+ <<New_Results_Are_Read_By_Unary_Concentrator>>
--+ when self accepts any at result_available
--+       then self performs input_read before
--+       self accepts any;
```

This TSL statement indicates what should happen when another task (a child task to this parent) calls this task instance with some results to process. The earlier TSL statements ensure that any initialization has been done properly. When a call to result_available is accepted, then this task will perform the user-defined action input_read, which was declared earlier. Later TSL statements and the body of the task will define what has to be done when input_read occurs; for example a parent task may have to be called. This TSL statement does state that all of these activities associated with input_read have to be done before this task will accept any other callers to it.

This task type has more TSL statements that further defined its behavior. For example, the activities associated with termination were defined. By defining actions such as parents_completion_notification and properties such as initialized_first, the designer can tie together events occurring in one task with those occurring in another, even if each task knows little about the others. This is how behaviors that cross task boundaries are defined.

5 CONNECTIVITY BETWEEN NOTATIONS

DQL's requirements on design methods led to the use of a variety of notations, but in order to produce a coherent design the methods and notations must relate to one another. These connections are needed at both the syntax and underlying semantic level. Too many current Ada design efforts have very weak connections between their notations. This section will define how the connections between the notations and the design methods were established for DQL's design.

5.1 BETWEEN ADA AND ANNA/TSL

As shown in the examples above, the textual annotations of Anna and TSL are connected to the Ada constructs they describe by how they are positioned in relationship to those constructs (as shown by the delete_result and unary_concentrator examples above). Currently Anna and TSL are separate notations, with TSL being used strictly for Ada task types. In addition to annotating standard Ada tasking events (such as rendezvous), DQL's designers created user-defined TSL events for major DQL activities that didn't involve Ada tasking but were related to important actions in DQL's behavior.

Therefore, the completed Ada source contained mostly executable Ada statements, interspersed with structured Ada comments providing even more detail on the software's behavior. Both the TSL annotations on standard Ada tasking events and the user-defined TSL event annotations describe behavior in terms of Ada's run-time model of Communicating Sequential Processes (CSP) (Hoare 85). The user-defined events are used to tie (non-tasking) design structures annotated with Anna into the (tasking) design structures annotated with TSL. At a syntactic level, there are connections between the Anna and TSL annotations (as in the user-defined actions for unary_concentrator).

Temporal logic provides a way to formally specify the behavior of "CSP tasks" through the definition of partial orders of correct process behaviors. Anna provides standard notations for mathematical formalisms such as predicate calculus and axiomatic specifications. TSL "properties" and user-defined events can be used to constrain the behaviors of dynamically-allocated task types, much as Anna can constrain the behavior of non-tasking Ada objects. So at a semantic level, the Ada, Anna, and TSL all describe the behavior of dynamically allocated (quantifiable) objects through their communications along the CSP model.

Having all these connections between the textual notations satisfies several of the requirements placed on DQL's design methods. Being able to quantify (over some or all of the objects) or constrain (over the partial orders of temporal actions) the behavior of instances of both tasking (through TSL) and non-tasking (through Anna) object instances supports the design of dynamic architectures of concurrent entities. The use of formal mathematical techniques as provided by Anna and TSL allows the behavior of an underlying unit to be specified abstractly, allowing implementors some freedom in choosing multiple Ada (or non-Ada) implementations.

5.2 BETWEEN BUHR DIAGRAMS AND ADA

The connectivity between standard Buhr diagrams and Ada source (or PDL) is straightforward and well-defined (Buhr 84). Note that because Buhr diagrams include graphical notations for Ada task call semantics (although the examples in this paper have been simplified), the behaviors of unit bodies as well as unit specifications are defined. Every icon on a Buhr diagram can be mapped to some Ada entity.

The axiomatic specifications and predicates defined in the Anna and TSL annotated Ada source provide a more precise description of unit and system behavior than were done graphically in Buhr diagrams. But, the Buhr diagrams have the advantage of being able to describe the interrelationships between specific instances of Ada objects. This is an advantage that a two-dimensional picture had over linear (one-dimensional) source code. The Anna and TSL formal notations tend to describe behaviors of classes of objects, while implementors and design reviewers want to see how instances of those classes are related as well. The individual Buhr diagrams in DQL's design provide examples of the state of the system at one point in time, showing how particular instances (corresponding to a particular application domain condition) of objects are related.

5.3 BETWEEN BUHR DIAGRAM SEQUENCES AND TSL

Just as simple Buhr diagrams show the calling, data flow, and hierarchical relationships among instances of objects (units) in the design, Buhr diagram sequences show how these relation-

ships change over time. The sequences try to provide a third dimension into the view of system behavior, just as simple Buhr diagrams try to provide a second dimension over linear text. Now sequences are the "classes" of how different "instances" of standard Buhr diagrams are related, just as a single Buhr diagram relates Ada task types and generic units to the entities that are created from them.

Each diagram in a sequence corresponds to a major event occurring during the execution of the system. These major events are defined as Ada tasking events (as defined by the Ada Language Reference Manual [ALRM] (DOD 83)) and by application-specific (user-defined) events defined as TSL "events." TSL's temporal logic notation can deal with both types of events.

These events are connected into Ada's tasking model of execution by placing TSL annotations on each Ada and user-defined event. These events are connected to each other textually (semantically) by defining TSL precondition and postcondition functions that define predicates on the partial orderings of events.

So as an event occurs (fires), its postconditions may match the preconditions of other Ada (or user-defined) events. This allows further events to occur. Ada's (CSP's) message passing mechanism through task rendezvous defines the run-time semantics of how these events behave as permanent and dynamic tasks communicate in a sequence.

The Petri Nets are used to provide a higher-level view into how the events in a sequence are ordered. While the semantic definitions of what correct partial orderings of events (individual Buhr diagrams) are defined in the TSL annotations, the Petri Net for each sequence provides this information at a higher level (on one page). The same event (TSL statement) might be applied to several different sequences. The Petri Net provides one example of how sequences of events are ordered.

5.4 AMONG BUHR DIAGRAM SEQUENCES AND THEIR THREADS

A decision the designers have to make is what are the major sequences (threads) that need to be specified in a Buhr diagram sequence and an associated Petri Net. Usually the application domain determines this choice, but without care the designers could devise thousands of different threads that are really variations on a few common themes. Alternatively, the designers could have created no sequences at all. This would require that any implementors or reviewers of the design derive sequences of interest on their own. Forcing readers of a design to create all the major sequences by themselves may be too much to ask when the readers are also trying to understand other aspects of the design. Designers should create some sequences as an overview and documentation of the system. Proper use of hierarchies in individual Buhr diagrams (abstracting behaviors into unit bodies), Buhr diagram sequences (where unit body behaviors can lead to "sequences of sequences"), and Petri Nets (defined hierarchically) can limit verbosity.

In DQL's design, a few major themes were used to define how many Buhr diagram sequences needed to be specified. One theme had to do with the creation of some major activity such as a command being issued at a workstation. The sequence then showed all the events that followed, such as forwarding the command to the proper handler and setting up communications paths. The completion of this sequence was the indication that initialization was complete.

Another theme addressed in DQL's design was shutdown (termination) of some activity started earlier. Many DQL commands cause work that has a finite lifespan to be performed. Any dynamically allocated resources have to be cleaned up. So the same care that was used to define how the architecture (Ada units) of the system was modified to handle a user's DQL request needs to show how the architecture is cleaned up as that request is satisfied.

Between initialization and termination there are many application-specific activities that might occur. Some are associated with steady-state processing; others are associated with exceptional conditions. The designers needed to be aware of what these application activities are so sequences are created that define the DQL system's response to them. While many steady-state events are just a specific ordering of Ada tasking events and their TSL annotations, exceptional conditions will often involve the use of Ada "exceptions." Ada exceptions, particularly user-defined exceptions, are often outside of the run-time behavior of Ada tasking. Anna is able to specify the conditions that led to an exception (such as the annotations for the `still_being_used` exception in the earlier `delete_result` procedure). Here the designer has to build connections between the Ada exception definition (and the associated Anna annotations) and TSL events, and also to place exceptions and their behavior in a Buhr diagram sequence.

Many of these major themes described so far have to do with responses to user inputs at a workstation. In DQL's design there were other themes related to how the server/client relationship between the disk and processing-intensive DIANA servers and the user interface-intensive workstations was managed. Sequences had to be defined that specified the communications protocols the servers and clients used.

These server/client communications protocol sequences were an example of hierarchies being used in Buhr diagram sequences. A goal of many of these protocols was to make sure all the relevant clients of a server were informed when the server's processing (usually a search of some DIANA tree) found something interesting. The communications protocol sequence is not interested in the details of how the server's tasks found these results, nor is it interested in what the client's (windowing) tasks do with the result once received. This higher-level protocol sequence concentrates on the mechanisms of how the relevant servers and clients communicate. This approach satisfied the requirement on DQL's design methods that a client/server relationship (as used in many multi-workstation windowing systems) be cleanly modeled.

In addition to being able to model internal client/server behaviors as communications protocols, the themes are used to address design issues of interest to the users of DQL. The threads of behavior are also mapped to what visibly happens on the workstation's screen. These threads define when windows are created, updated, and destroyed, particularly as changes in one window affect the behavior in other windows. Being able to describe both observable and hidden behaviors meant using de-

sign notations that applied to Ada and non-Ada implementation entities.

5.5 BETWEEN DESIGN AND IMPLEMENTATION

In the Buhr diagrams created for DQL's design, the standard assumption was that each concurrent entity was implemented as an Ada task (which is standard for Buhr's notation). However, in the actual DQL implementation not all entities are represented by Ada tasks or even implemented with Ada source code. At the time there was no Ada interface to the SunVIEW windowing system, so low-level C code was written to create and manage individual windows. Because SunVIEW's granularity level for concurrently operating windows was the UNIX process level, each DQL window was its own UNIX process.

DQL's implementation also involved multiple workstations. Communications between these workstations meant use of Sun's standard Inter-Process Communications (IPC) mechanism among UNIX processes since Ada tasks cannot be automatically allocated arbitrarily across workstation and process boundaries. Again there was no Ada interface to IPC, so more low-level C code needed to be written.

While system-level programming in C for dynamically-allocated UNIX processes and dynamically-initiated IPC calls can be done to almost any model of behavior, the use of guidelines allows the UNIX processes and IPC calls to behave very similarly to Ada tasks and rendezvous. There are major differences such as visible objects and blocking I/O properties, but by sticking with Ada design techniques that use the tasking model and rendezvous, the implementation of Ada models in UNIX and C code is easier. Using incompatible models of behavior (and design techniques) would have made this mixture of Ada and C code more difficult.

With the definition of the major events of system behavior (from the individual Buhr diagrams) and how those events are connected into sequences of activities (from the diagram sequences), the DQL implementors knew what to place in the C code (as well as the standard Ada) to meet the design. This satisfied the DQL design method requirement that design entities be mapped to Ada and non-Ada implementation entities. The use of models of behavior (particularly communications) that are mapped to client/server IPC calls as well as Ada met the requirement that DQL be distributed across a network.

Since an overall design was created that assumed Ada tasking as the main implementation entity and since Ada and non-Ada code was based on a common model of behavior (temporal logic specifications of CSP rendezvous), the decision on what entities needed to be implemented as UNIX processes versus Ada tasks and what communications needed to be done via IPC versus Ada rendezvous could be delayed for as long as possible. Besides, allowing reconfigurations as implementation issues were better understood; keeping a DQL implementation as a collection of Ada tasks within the same UNIX process had some debugging advantages. Ada symbolic debuggers were used to check the behavior of an implementation. Typically Ada's debuggers are much better at monitoring rendezvous behavior than cross-process UNIX debuggers; this proved an advantage when tracking down bugs.

6 REPRESENTING DYNAMIC STATES

As shown above, the individual Buhr diagrams and Buhr diagram sequences can be used to graphically represent the evolution and current state of the system's architecture. However, dynamic changes in the architecture involve more than just Ada task activation, termination, and rendezvous. These tasking events are usually a reflection of changes in the state of the system and its units. This section will describe how the graphical and textual notations used in DQL's design were used to represent major changes in system state as well as changes to the architecture.

6.1 SYNCHRONIZATION AS A DESIGN PROPERTY

The DQL design examples provided earlier show the architecture's evolution and how individual results are routed through the current architecture. In DQL's design an individual unit (instance of a task type) will often receive results as inputs from one or more child tasks, then perform some computation on these results, then forward the results to its parent tasks (as the parents' input). That parent could then forward further results to its parent. Note that the task type instances in this hierarchy are not necessarily of the same type; different requests for computations (queries) by a DQL user will result in different task types being used. Also note that while the example Buhr diagrams, shown in section 4, show this hierarchy of task instances arranged into a tree, the actual DQL design used caching of some intermediate results and reuse of currently running queries.

As a result, the architecture of the various instances of task types (units of the architecture) was very complex. Due to the interactive nature of the multiple workstation-based application, this architecture evolves over time. So when one task computes some results, there are a whole set of other tasks that need to be informed of this. This set will evolve at the same rate as the overall architecture.

DQL's design introduced the notion of units being in synchronization with one another. When one unit low in the architecture's hierarchy computes a result, then all the units above it in the hierarchy (in the set of units that depend on this unit's computation) get out of synchronization with respect to knowledge of the result. DQL's design had to show how this resynchronization process worked and how only those parent tasks in the set of dependent units were informed of this change (units not in the set should be unaffected).

Describing the detailed semantics of this resynchronization activity was fairly straightforward. The designers added the necessary Anna and TSL annotations to the Ada PDL. User-defined TSL "events" and functions were used to explicitly identify events related to determining the current set of dependent parent tasks and making sure all of them were notified. Buhr diagram sequences were created to show how the Ada tasking and user-defined events were connected into threads of behavior.

Just as the Buhr diagrams and diagram sequences showed graphically the relationships among instances of units, a graphical notation was needed to show this resynchronization process. Design reviewers had a better understanding of how this works if

they saw a picture. Since resynchronization is dependent on the current state of the architecture, this behavior needs to be projected on the same diagrams being used to show the architecture. A later section will show how the diagrams were modified to show both the events in a sequence (thread) and how the system state evolves.

6.2 SHADING OF DIAGRAM SEQUENCES TO SHOW ANIMATION

As shown above, there are a variety of changes that will occur to the state of the system and the units within as the architecture evolves during program execution. To provide a consistent view to readers of the design, these system state changes were projected onto the same Buhr diagram sequences used to show evolution in architectures. Deciding how best to project the information depends on the graphics resources available to the designers.

For DQL's design, the units (task activations) are shaded to show changes in system state. The graphics capability of the Interleaf Workstation Publishing System (WPS)® (Interleaf 86) document processing system used to format DQL's design had fairly good support for shading of polygons such as the Buhr notation's icons. Other modern document processing systems also allow shading as part of their graphics packages, so this graphics approach for showing system state could have broad application.

An alternative to monochrome shading would have been to use color or a combination of coloring and shading levels. This allows finer details of various state changes to be displayed on those output devices capable of color (none of which were available to DQL's designers). A concern when a wide range of graphical forms exist is that designers will get carried away with all those colors and shadings to produce designs that are virtually unreadable. The purpose of the graphics notations such as Buhr diagrams is to present the relationships among units in a concise and understandable form. Adding too much graphical clutter will eliminate this advantage.

The simple example below shows how shading is used, describing the resynchronization of units as one unit low in the hierarchy computes new results. This example shows a collection of Nodes similar to the ones shown earlier in figure 4-4. Here there are four dynamically created tasks; the first three are created as shown in figures 4-6 through 4-13. A fourth Node called Sub_Child was created that uses Child_2 as its parent (perhaps due to the structure of the query passed to the overall server task). Figure 6-1 below shows these four Nodes running, waiting for one of them to compute some results.

In figure 6-2 below, Sub_Child has found one or more results. This causes its ancestors (its parent Child_2 and grandparent Parent) to be out of date. The shaded tasks indicate that they need to be informed of these new results as they affect whatever (sub)queries they are working on. The discovery of new results by Sub_Child will be a user-defined (TSL) action since it may not involve any Ada tasking events yet. Note that the task Child_1 is not shaded since it is not a parent to Sub_Child and so will not be affected by any new results there.

® Interleaf and Workstation Publishing System are trademarks of Interleaf, Inc.

As shown in the TSL examples earlier, Nodes are responsible for seeing that their parents are updated when new results are computed. Figure 6-3 below shows the first step in that process, where Sub_Child calls its parent (Child_2) with the latest results it has found. As a result of this update call from its child Node, Child_2 is no longer shaded since it is now working with the latest information.

These update activities continue in figure 6-4 below as Child_2 calls its parent with any results it computes (based on the results received from Sub_Child in figure 6-3). This causes the shading to the Parent to be turned off as it is also now working with the latest results. Note that if Child_2 decided that the latest results from Sub_Child contained nothing that matched the subquery it was performing, this update of the parent Node (Parent) would be implicit (perhaps as a user-defined action) instead of explicit (as an Ada tasking rendezvous).

The Parent Node then might compute results to be passed on to something else. A Petri Net could be used to define the ordering of events in this sequence, although that is really not necessary in this example, since these events have to occur in order (in this example). This concludes the sequence of events as they relate to updates found in one node (Sub_Child). Any updates found in similar leaf nodes (such as Child_1) cause a similar sequence.

In addition to providing a pictorial representation of one aspect of the current state (after some activity occurs) of the system, the use of shading helps a reader of the design better visualize the dynamic activities or animations of the system. If a reader takes a stack of papers (each holding one Buhr diagram) representing a thread of activities and then quickly thumbs through them, the sequence appears to be animated. The effect is similar to a cartoonist's sketches of a brief scene.

As the shadings change at the same time as the architecture of units dynamically evolves, the reader (viewer) sees the connections between system state changes and system architecture changes. The extra work necessary by designers to create all these Buhr diagram sequences and shadings on appropriate units will be realized by future users of the design (especially implementors) as they get a clearer understanding of how the system operates. DQL's design made extensive use of modified formal design methods to support as much dynamism in response to user commands as possible. These graphic notations provide a way to capture and describe all these simultaneous changes.

6.3 INFORMATION FLOWS THROUGH ARCHITECTURE

The previous section showed how synchronizing the dependent tasks in a hierarchical architecture was portrayed by shadings of the units involved. When a reader "flips" through the pages of such a design, the combination of shadings and architectural changes makes the diagrams appear animated. In the case of synchronization of DQL design units, this animation will cause "ripples" or "waves" to flow through the architecture as the states (shadings) of the individual units change.

These "ripples" represent a higher or more abstract view of system behavior than threads (sequences of events). The ripples combine the effects of several threads, some acting in par-

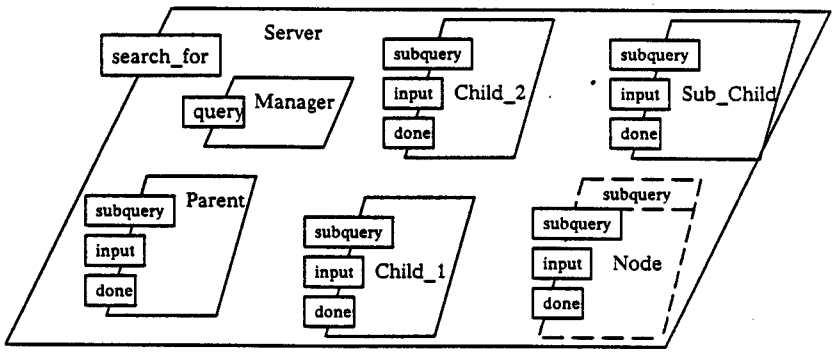


Figure 6-1. Steady State with Nodes Awaiting Results

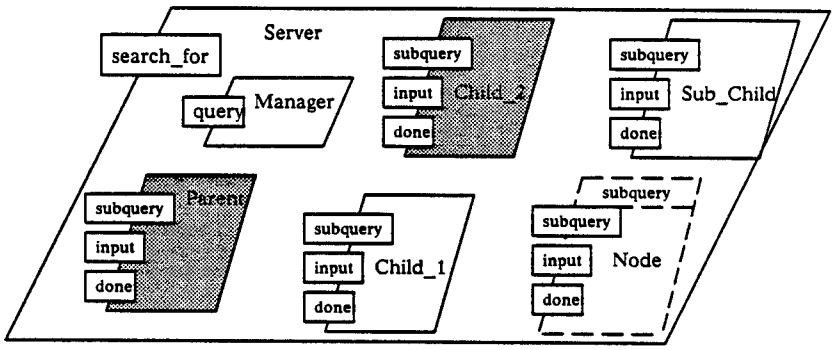


Figure 6-2. Sub_Child Computes Something; Parent Now Out of Date

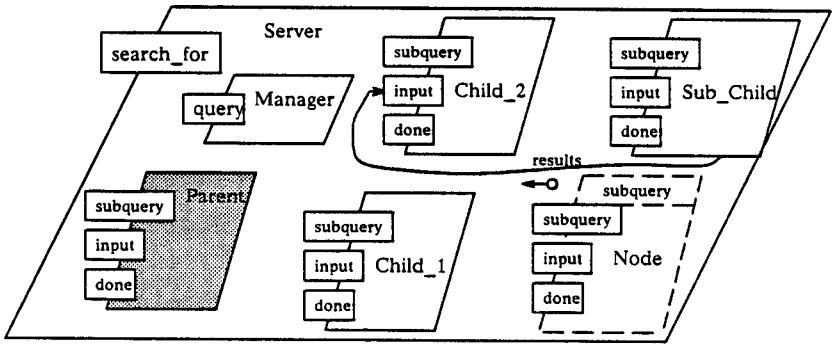


Figure 6-3. Sub_Child Informs Child_2 of Latest Results

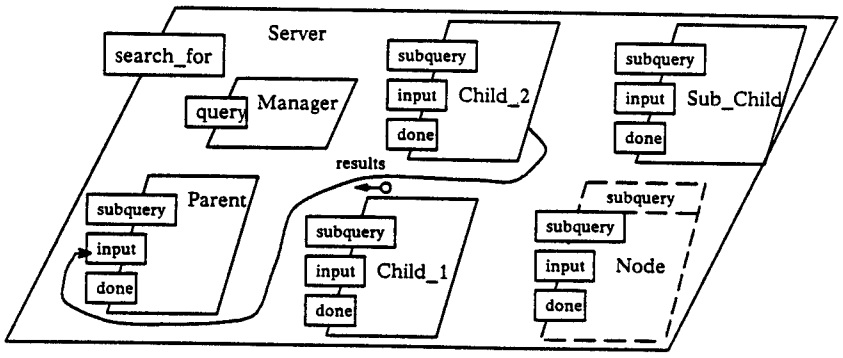


Figure 6-4. Child_2 Informs Parent of Latest Results

allel and some sequentially. A designer would like to specify appropriate behaviors using these very abstract descriptions as well as using lower level notations such as sequences.

For example, in DQL's design resynchronization will cause threads being executed to propagate the latest results through the architecture's hierarchy. For some DQL computations (primitive queries), the new results are immediately passed through a unit (task). This makes the ripples through the architecture appear smooth. In other DQL computations, a task will not forward the latest results until the stream(s) of input finish since the computation needs all results before any output is generated. Here the ripples will be compressed or collapsed into a few nodes until released by the closing of an input stream.

As a result of these possible behaviors, representing ripples as just collections of threads is not possible because sometimes the effect of a new result will not be seen in some higher level unit in the architecture until much later. A designer would still like to treat (successive) flows of information and the dynamic changes they cause to the architecture abstractly. In DQL's design, the interaction (or required non-interaction, such as ripples not overtaking each other) between these ripples was an example of complex information flows. We found that while representing ripples was syntactically possible (with shaded Buhr diagram sequences), representing the semantics of their behavior was more difficult.

Being able to abstractly describe higher level behavior becomes important when dealing with architectures involving multi-processor concurrent systems. The design cannot be bogged down in the details of locks and semaphores; instead the design must show how the processors work together with a minimum of conflict.

This is particularly important when dealing with server-client processing relationships. Computations or commands done for one window in one workstation may affect other windows (in the same workstation), other workstations, and one or more servers. A design will have to keep track of all these (simultaneous) flows through the architecture. As shown in the discussion on textual and graphical notations, any abstract description of behavior has to be tied to all the other notations used as well as the (formal) design methods used to produce them.

DQL's methods should work for tightly-coupled (shared memory) multi-processing systems as well as distributed (networked) systems. By using the same client/server models that both the Ada tasking system and (multi-processor) operating use, the boundary between those design entities that are Ada tasks and those that are operating system processes can be (relatively) transparent. As both additional workstations (users) are connected to the network and additional windows (requests for DQL query results) are created, the design methods can account for these dynamic changes regardless of where new tasks and/or processes are being created.

7 CONCLUSIONS

The implementation of DQL has been going well, with some partial releases of the tools to interested groups already underway. The developers feel that the definition of requirements on the design methods, the choice of design notations and methods that collectively meet those requirements, and the use of a common underlying model of computation as the basis of those notations has helped to produce a consistent design. The developers feel this is an activity that all major software development projects should go through before work begins.

The DQL implementation involved running programs on multiple (single processor) computers distributed over a local area network. But the issues raised in the design also apply to target computers that have multiple processors within the same box. Designers and programmers will try to take advantage of the additional processors in these advanced computers by spreading the workload among different processors, defining more concurrent activities, and using dynamically created entities to handle shifts in the workload. More applications will have to deal with these issues. Another issue is how any development team addresses these design concerns in any collection of design methods. The methods and notations used in DQL try to support these concurrency issues; they are much different from traditional design methods such as SA/SD. Development teams planning to use these earlier methods should be equally prepared to show how they handle concurrency and dynamic architecture issues.

ACKNOWLEDGEMENTS

The author wishes to thank David Emery for his hard work in reading the designs and implementing major portions of DQL. Thanks to Marlene Hazle for advice and support for both the DQL project and this paper on how we designed it. Thanks to Richard F. Hilliard and Steven Litvintchouk for their helpful comments. Also thanks to Professor David Luckham, Geoff Mendal, Doug Bryan, and everyone of the Stanford University Program Analysis and Verification Group for answering our many questions on their languages and tools. Finally, we extend the usual thanks to Linda Gaudet and her trusty word processor.

REFERENCES

(Berg 82) Berg, H. K., et al, *Formal Methods of Program Verification and Specification*, Prentice-Hall, 1982.

(Booch 87) Booch, Grady, *Software Engineering with Ada*, Benjamin Cummings, 1987.

(Buhr 84) Buhr, R. J. A., *System Design with Ada*, Prentice-Hall, 1984.

(Byrnes 88) Byrnes, Christopher, "ESD Acquisition Support Environment (EASE)," *Proceedings of the Sixth National Conference on Ada Technology*, U. S. Army Communications - Electronic Command, 1988.

(Byrnes 89) Byrnes, Christopher, "A DIANA Query Language for the Analysis of Ada Software," *Proceedings of the Seventh National Conference on Ada Technology*, U. S. Army Communications - Electronic Command, 1989.

(Cherry 86) Cherry, George W., *PAMELA Designer's Handbook*, Thought**Tools, Inc., 1986.

(DOD 83) Department of Defense, Ada Joint Program Office, *Ada Language Reference Manual*, ANSI/MIL-STD-1815A-1983, 1983.

(DOD 88) Department of Defense, Joint Logistics Commanders, *Defense Software Development Standard*, DOD-STD-2167A, 1988.

(Evans 83) Evans, Arthur, et al, *Descriptive Intermediate Attributed Notation for Ada Reference Manual*, TL-83-4, Tartan Labs, 1983.

(Firth 87) Firth, Robert, "A Pragmatic Approach to Ada Insertion," *Proceedings of the First International Workshop on Real-Time Ada Issues*, Ada Letters, Volume VII, Number 6, ACM Press, May, 1987.

(Goldblatt 82) Goldblatt, Robert, *Axiomatising the Logic of Computer Programming*, Springer-Verlag, 1982.

(Interleaf 86) Interleaf, Inc., *Workstation Publishing Software Reference Manual*, Interleaf, Inc., 1986.

(Intermetrics 87) Intermetrics, Inc., *ACS Compiler System DIANA Manual for IBM 370/UTS Ada Compiler*, IR-MA-786-0, Intermetrics, Inc., 1987.

(Hailpern 82) Hailpern, Brent, *Verifying Concurrent Processes using Temporal Logic*, Springer-Verlag, 1982.

(Hoare 85) Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

(Luckham 87a) Luckham, David, *Anna, A Language for Annotating Ada Programs: Reference Manual*, Springer-Verlag, 1987.

(Luckham 87b) Luckham, David, *Task Sequencing Language for Specifying Distributed Ada Systems*, Technical Report No. CSL-TR-87-334, Stanford University, 1987.

(Sankar 85) Sankar, Sriram, et al, "An Implementation of Anna," *Ada In Use: Proceedings of the Ada International Conference*, Cambridge University Press, May, 1985.

(Scheifler 88) Scheifler, Robert, et al, *X Window System: C Library and Protocol Reference*, Digital Press, 1988.

(Sun 88) Sun Microsystems, Inc., *SunView 1 Programmer's Guide, Release 4.0*, PN-800-1783-10, May, 1988.

(Wild 89) Wild, Fred, "The Ada Semantics of Buhr Diagrams: A Realtime Example," *ACM SIGAda Winter '89 Conference*, CADRE Technologies, Inc., 1989.

(Yourdon 79) Yourdon, Ed and Larry Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice-Hall, 1979.

BIOGRAPHY

Christopher Byrnes is a Member of the Technical Staff of the Software Center of The MITRE Corporation. He received his B.A. from Tufts University in 1976, his B.S. from the University of Lowell in 1980, and his M.S.E. from the Wang Institute of Graduate Studies in 1984. His interests include software development methods, analysis of design products, and the Ada programming language. His mailing address is: The MITRE Corporation, Burlington Road, M/S A156, Bedford, MA 01730. He can also be reached at cb@Mitre.org (InterNet) and at ...!decvax!linus!mbunix!cb (UUCP).



ADA EXCEPTIONS AND FAULT-TOLERANCE

Ronald J. Leach

Department of Systems & Computer Science
School of Engineering
Howard University
Washington, D.C., 20059

ABSTRACT

In this paper we consider some of the standard techniques for fault-tolerant computing and compare them to the Ada exception handling mechanism.

1. Introduction

One of the major features of the Ada language is exceptions ([BOOCH86]). An exception is defined as an unexpected event that causes suspension of normal program execution. Bringing an exception situation to attention is called raising an exception and responding to the exception is called handling the exception [LRM]. In this paper, we describe both predefined and user-defined exceptions in Ada and the manner in which they can be handled to produce fault-tolerant programs.

There are several predefined exceptions in Ada: they are CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, and TASKING_ERROR. In addition, many I/O operations lead to exceptional situations; these are included in the package IO_EXCEPTIONS. Thus we will also consider the type IO_EXCEPTION as being a predefined exception.

In this paper, we present some handlers for these exceptions and indicate strategies to be used when incorporating fault-tolerance in programs that generate exceptions. There has been a considerable amount of research in the development of fault-tolerant programming techniques such as N-version programming or rollback techniques. Exception handling in Ada will be considered in the context of these fault-tolerant techniques.

The first fault-tolerant strategy that we consider is the rollback technique of Randell [RANDELL75]. In this technique, the program is consider to be correct until it reaches an error state due to a *fault* (either hardware or software). After the fault occurs and is detected, program execution is halted temporarily so that a corrective action can be taken. This corrective action means resetting the program counter; changing the values of appropriate registers; clearing function calls, parameters, and return values from the system stack, etc. This is a very expensive operation in terms of execution time and the time needed for such a rollback is difficult to predict. Note that this is somewhat different from the Ada exception model in which control is passed to an exception handler and control of the program resumes from the point specified by the handler.

The second fault-tolerant programming technique that we consider is N-version programming in which several versions of the critical sections of programs are active ([CHEN75]). These versions are coded by different algorithms. The results of these algorithms are compared frequently and a determination is made by majority vote of what the correct state of the program actually is. This method has a very high but quite predictable overhead. See the references [COLEMAN&LEACH89] and [COLEMAN&LEACH90].

Ada uses a different philosophy for fault-tolerance in its handling of exceptions. After an exception is raised, the local block is searched for the existence of a handler for that exception. If none is present, then calling routines (functions, procedures, etc) are searched for handlers. This process continues until either an exception handler is found or control is assumed by the operating system. If the exception was raised in a task and no handler was present in the task, then the task becomes completed [LRM]. This is a different method since the program is not automatically rolled back to a point at which the program is presumed to be correct but instead is rolled forward to another point at which execution continues (assuming the existence of an exception handler.

In this paper, we consider the behavior of exceptions, exception handlers and fault-tolerance. Among the examples studied are ones involving system calls (DOS

and UNIX) as well as foreign code.

2. Exceptions in Ada

Much of the original motivation for designing Ada was the need for a better language to address unexpeced situations in real-time programming environments. Exception handling in Ada depends on what type of exception is encountered. The two types of exceptions are user-defined and predefined exceptions.

User-defined exceptions are exceptions that the user explicitly detects, raises, and handles. In other words, the user performs checks in their code for certain conditions. The user then raises the exception with the raise statement. The handling of the exception is achieved by appending one or more handlers to the end of the program block that the exception occurs in. The action of the exception handler once the exception has been detected and raised varies from one application to another. The action taken can range from simply terminating the program, to effectively handling the exception to allow continued program execution. Obviously, the actions taken are largely dependent on the application. For example, if a radar system installed in an aircraft detects and raises an exception, termination of the program would not be the appropriate method of handling this exception. In any critical situation such as this, program termination is to be avoided and an appropriate type of recovery action to regain the functionality of the system should be taken.

Predefined exceptions handle errors in the same way as user-defined exceptions. However, predefined exceptions are detected and raised in a different manner than user-defined exceptions. The predefined exceptions are raised by the exception handling facilities of Ada. The user is only required to specify in the exception handler the actions to be taken if an error occurs.

One important design considerations for the exception handling facility in Ada is that exceptions should add to execution time only if they are raised [ICHBIAH79]. Several techniques may be used to reach this goal and these techniques may differ from one implementation to another. However, they all use the idea of reactions rather than guards. The basic idea, as stated in [ICHBIAH 79], is that all processing costs should be concentrated on the treatment of the exception rather than its detection. Therefore the additional checks and processing time associated with detecting a possible exception before it occurs are eliminated.

Exceptions in Ada are events that prevent normal execution of a program. These events may be those defined by the user or the predefined exceptions in the Ada Language. For predefined exceptions, the idea of reactions rather than guards is preferred. However, for user-defined exceptions the user is required to detect the exception that he or she defines.

The Ada programming language has eight I/O exceptions which are raised in the following circumstances:

STATUS_ERROR: an attempt to use an internal file that has not been associated with an external file (i.e., has not been opened); or to open an internal file that is already open

CODE_ERROR: an attempt to perform an input operation on a file of mode out or an output operation on a file of mode in

NAME_ERROR: an attempt to associate an internal file with an external file if an invalid external file name is specified

USE_ERROR: an attempt to perform some input/output operation on an external file for which the implementation does not allow that operation

END_ERROR: an attempt to read past the end of an input file

DATA_ERROR: input data that is not of the expected form

DEVICE_ERROR: a problem with the hardware, software, or media providing input/output services

LAYOUT_ERROR: invalid Text_IO formatting operations

SOURCE : [COHEN 86]

The other predefined exceptions are:

CONSTRAINT_ERROR:ignoring subramge or array bounds

NUMERIC_ERROR: a numerical error such as overflow, underflow, imprecision in a computation, or division by 0.

STORAGE ERROR: when dynamic storage allocation is exceeded

TASKING_ERROR: an exception is raised in the calling task if the called task terminates before accepting the entry call or is already terminated at the time of the call.

PROGRAM_ERROR: when execution of a program reveals that the program is improperly formed such in the case of a function attempting to return without executing a return statement.

Examples of the raising of a PROGRAM_ERROR are shown in examples 1 and 2.

```
With BASIC_IO; use BASIC_IO;

function PERCENTAGE return float is
  n : integer;
begin
  get(n);
  if n in 0..100 then
      return float(n)/100.0;
  end if;
end PERCENTAGE;
```

Example 1

In example 1, the execution of the return statement is dependent upon the input value. Therefore, the detection of the error would require knowing the values of variables at run time. This presents problems in implementing an exception handler to detect this error.

Example 2 shows a PROGRAM_ERROR if there is a call to a subprogram before its body has been elaborated.

```
package FRACTION_PACKAGE is

  type FRACTION_TYPE is private
        .
        .
        .
  function"/"(LEFT,RIGHT:integer)
    return FRACTION_TYPE;
  ZERO : constant FRACTION_TYPE:= 0/1;
      --INCORRECT CALL ON "/"

private
        .
        .
        .

end FRACTION_PACKAGE;
```

Example 2

The call to "/" in the declaration of ZERO would cause a PROGRAM_ERROR because the body of "/" has not yet been elaborated.

3. What happens if an exception arises?

Should execution continue after an exception occurs? This section of the paper attempts to answer the proposed question. As stated in the introduction, the ability to handle erroneous situations is essential for reliability in real time systems. In many cases, these systems must be designed as highly fault-tolerant systems that should never halt. This requires an ability to handle situations which are likely to happen at any given time. The PROGRAM_ERROR exceptions generated in examples 1 and 2 are informative.

In example 1, the exception occurs because the program takes an execution path that was not expected. This could have been avoided by testing all possible execution paths *for this function* during the testing phase; the total number of such paths is likely to be small for most functions. Here the recovery block tehnique is appropriate at times. We simply call the function with an appropriate parameter if one exists. Note that no exception handler is likely to be available since the programmer was clearly careless in the original analysis and so is unlikely to look for an exception. If N-version programming is used, then, assuming that the majority voting is done by a separate *voter* task which is itself fault-tolerant, this problem is likely to be caught and damage will probably be restricted to a single task containing the offending function. In example 2, the exception is likely to be more amenable to an exception handler than to either of the other techniques.

For an example of another exception, consider the code in example 3.

```
package TRANSFER is
  use TEXT_IO;
  INF    : IN_FILE;
  OUTF   : OUT_FILE;
  C      : CHARACTER;
end;

package body TRANSFER is
  ...
  procedure READ_FILE is
  begin
    open(INF,"SOURCE");
    open(OUTF, "DESTINATION");
    loop
      Get(INF,C);
      Put(OUTF,C);
    end loop;
   EXCEPTION
      when END_OF_FILE =>
        Put(OUTF,EOF);
        Close(INF);
        Close(OUTF):
  end;
  ...
end TRANSFER;
```

Example 3

This illustrates a case where exception handling is used to treat an event which is certain to happen, reaching the end of a file. (This example could be formulated with an explicit check for each iteration). Assuming the file to be very large, the body of the procedure READ_FILE may be represented as an infinite loop, and the final actions of the procedure performed by the exception handler END_OF_FILE. The procedure READ_FILE transfers the characters from the file SOURCE into the file DESTINATION. At each iteration, Get is called and eventually an END_OF_FILE exception will occur. As a result, the corresponding handler will be activated and its execution will complete the execution of READ_FILE, thus returning control back to the body of TRANSFER. Thus the exception handling technique is more appropriate .han

either N-version or rollback techniques in this example.

Example 3 shows that exception handlers can be viewed as substitutes ready to take charge of the operations in case of errors. Furthermore, program termination should only occur in the event that the exception is not properly handled; otherwise the program should continue normal execution. In other situations, exceptional events can be considered as terminating conditions so that when an exception occurs in a given program unit, control will be passed to an exception handler but will never return to the point where the exception occurred.

The possibility of I/O errors occurring must be considered in any robust software design. In traditional languages, the handling of abnormal cases such as these must be programmed explicitly similarly to the normal expected case with the result that the error handling code can grossly distort the structure of the main program flow. In Ada, the actions to be taken when an error occurs are kept separate from the main program flow so that no such distortion takes place. The simplest and probably the most common approach to error handling is for each function to return some status information indicating the success or failure of the operation for which it is responsible. It should be emphasized that the ability to recover from errors should be considered as a facility for handling predictable errors (such as bad input data formats) in a structured way and as a method for providing a layer of fault-tolerance against those errors which are largely unpredictable.

4. Fault-tolerance and exceptions

In the previous sections, we considered some examples of simple exceptions and described some approaches for treating them. Two of the three simple exceptions reacted well to Ada exception handlers; we suggested N-version programming to improve the fault-tolerance of the other one. We briefly discuss a complex topic - how to provide fault-tolerance for programs that use multiple languages (foreign code in Ada).

The difficulty is caused by the fact that different languages allow different actions to be taken in exceptional situations. If a fault occurs, which language has the responsibility for treating the fault? The "best" answer is probably to use the language with the best fault recovery features.

Example 4 shows a mixed language program with an obvious fault - an attempt to compute the logarithm of a negative number.

```
with TEXT_IO; use TEXT_IO;

procedure TEST is

package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
package L_FLOAT is new FLOAT_IO (FLOAT);
use L_FLOAT;

function SIN (X:FLOAT) return FLOAT;
function LOG (X:FLOAT) return FLOAT;
pragma INTERFACE (C, SIN);
pragma INTERFACE (C, LOG);

a, b, c, t: FLOAT := -0.3;

begin

put(a*SIN(t)+b*SIN(2.0*t)+c*SIN(3.0*t));
new_line;
put(a*LOG(t)+b*LOG(2.0*t)+c*LOG(3.0*t));
new_line;
end TEST;
```

Example 4

The output obtained from running this example on a SUN 3 running Verdix Ada 3.4 is given below.

```
 4.93046876905157E-01 ** MAIN PROGRAM ABANDONED -- EXCEPTION "numeric_error" RAISED
```

The program is quite interesting. This particular version of the compiler did not comment about the fact that the C functions sin() and log() expect an argument of type double; this is in keeping with the "who cares" attitude of C. An exception was raised and so the program halted. How can the program continue execution and remain in a reasonably well-understood state? Putting in the simple exception handler

```
exception
   when NUMERIC_ERROR =>
        put_line("oops!");
```

gives the expected output

```
 4.93046876905157E-01
oops!
```

but we are uneasy about the state of the system stack. At this point, the rollback technique is probably appropriate. We should clear the stack ourselves in the foreign code, writing the exceptional features into it. Since the SUN uses the UNIX operating system, we could use the UNIX C functions setjmp() and longjmp() to clear the stack, placing the program back into a state of (presumed) correctness.

The situation is much more severe if we use foreign code on a system that allows a programmer more access to the underlying hardware. PC's running DOS have obvious problems in this area. We suggest that foreign code be avoided on such systems except for straightforward access to DOS commands.

5. Conclusions

Some exceptions arise from a violation of the language rules, such as an attempt to divide a number by zero or to access a non-existent component of an array. It is interesting to note that in C, Pascal, and Fortran, errors such as these are handled by simply issuing a warning message and then aborting the program. In Ada, the programmer can specify what actions should be taken and then continue program execution. Others include those which are detected through explicitly programmed actions. For example, if at some point X should be positive the programmer may write:

```
if (X < 0)
    ERROR... .
```

In Ada, the only difference between these two classes of errors is the way that they are detected and signaled. From that point on, they are handled, in the same way, regardless of which class the error belongs to.

Note also that a program may produce incorrect answers that are not exceptions. In a case such as this, N-version programming may be the only solution to fault-tolerance.

The rollback technique is appropriate for programs with foreign code if there is a direct interface to operating system calls.

A major design goal for any program should be to make it robust. Methods available in a given programming language should be utilized to make a program robust; in Ada these methods are elegant. However, they do not provide a complete treatment of fault-tolerance and thus other methods such as N-version programming and rollback techniques are also needed at times.

REFERENCES

BOOCH86: Booch, Grady, "Software Engineering with Ada, Second Edition", The Benjamin/Cummings Publishing Company, Inc., Melno Park, CA., 1986

CHEN78: Chen, L., and Avizienis,A., "A Fault-tolerance Approach to Reliability of Software Operations", Proceedings of the 8th International Conference on Fault Tolerant Computing, June 1978.

COHEN86: Cohen, Norman, "Ada as a Second Language ",McGraw-Hill, Inc., New York, 1986.

COLEMAN & LEACH89: "Performance Issues in C Language Fault-Tolerant Software", Computer Languages vol 14, No. 1 (1989), 1-9.

COLEMAN & LEACH 90: "N-Version Programming Using the Ada Tasking Model", to appear.

ICHBIAH79: Ichbiah, J., "Rationale for the Design of the Ada Programming Language", 1979.

LRM "Ada Language Reference Manual"

RANDELL75: B. Randell, "System Structure for Software Fault Tolerance", IEE Trans. Softw. Engr. vol SE1, No. 2, June 1975.

Acknowledgement

Research support form the Maryland Procurement Office, the Naval Surface Warfare Center, and the Army Research Office during the time that this research was performed is gratefully appreciated.

BIOGRAPHY

Ronald J. Leach is a Professor in the Department of Systems and Computer Science at Howard University. He has B.S, M.A. and Ph.D degrees in Mathematics from the University fo Maryland at College Park and an M.S. degree in Computer Science from Johns Hopkins. His research interests include computer graphics, analysis of algorithms, user interfaces, and software engineering (especially software metrics and Ada programming)

A BETTER APPROACH TO SOFTWARE ENGINEERING

Herbert P. Woodward

TRW Systems Integration Group, Fairfax, Virginia

This paper will describe how major advances have been made in the development of software through the use of Ada engineering. It will describe the Ada software approach, process model, use of Ada as a design language, and Ada environment and tools. It will discuss the above in the context of the experience gained on the major Ada activities going on at TRW (the nation's largest Ada contractor) and conclude with current and projected results using Ada engineering.

INTRODUCTION

Ada is as important for what it promotes as for what it does. Good software engineering is fundamental to obtaining high quality, reliable, reusable, portable, maintainable software. But few general purpose languages embody and promote good software engineering practices and none to the extent that Ada does. This critical characteristic of Ada when properly utilized and coupled with an excellent approach, process, and environment can result in significant improvements in software quality, maintainability, reusability, reliability, portability, productivity, and requirement satisfaction.

APPROACH

An engineered approach to software development is embodied in the spiral model as developed by Dr. Barry W. Boehm, when he was the Chief Scientist for the Defense Systems Group of TRW. Figure 1 portrays this model. It is a risk-driven approach that emphasizes prototyping and incremental deliveries to allow a more realistic approach to requirement satisfaction than in more traditional waterfall models. It provides structure to the concept of "design a little, build a little," and allows evaluation and feedback on each cycle of software development. Each spiral model cycle follows the same sequence of steps--identify objectives, constraints and alternatives, evaluate alternatives and resolve risks relative to objectives and constraints and develop the appropriate product, then review and plan the next cycle. This provides a flexible, iterative answer to the problem of software development. It has the following particularly positive features:

- "It fosters the development of specifications that are not necessarily uniform, exhaustive, or formal, in that they defer detailed elaboration of low-risk software elements and avoid unnecessary breakage in their design until the high-risk elements of the design are stabilized.
- It incorporates prototyping as a risk-reduction option at any stage of development. In fact, prototyping and reuse risk analyses [can be] used in the process of going from detailed design into code.
- It accommodates reworks or go-backs to earlier stages as more attractive alternatives are identified or as new risk issues need resolution. . . .
- It focuses early attention on options involving the reuse of existing software. The steps involving the identification and evaluation of alternatives encourage these options.
- It accommodates preparation for life cycle evolution, growth, and changes of the software product. The major sources of product change are included in the product's objectives, and information hiding approaches are attractive architectural design alternatives in that they reduce the risk of not being able to accommodate the product-charge [sic] objectives.
- It provides a mechanism for incorporating software quality objectives into software product development. This mechanism derives from the emphasis on identifying all types of objectives and constraints during each round of the spiral.
- It focuses on eliminating errors and unattractive alternatives early. The risk-analysis, validation, and commitment steps cover these considerations.
- For each of the sources of project activity and resource expenditure, it answers the key question, 'How much is enough?' Stated another way, 'How much of requirements analysis, planning, configuration

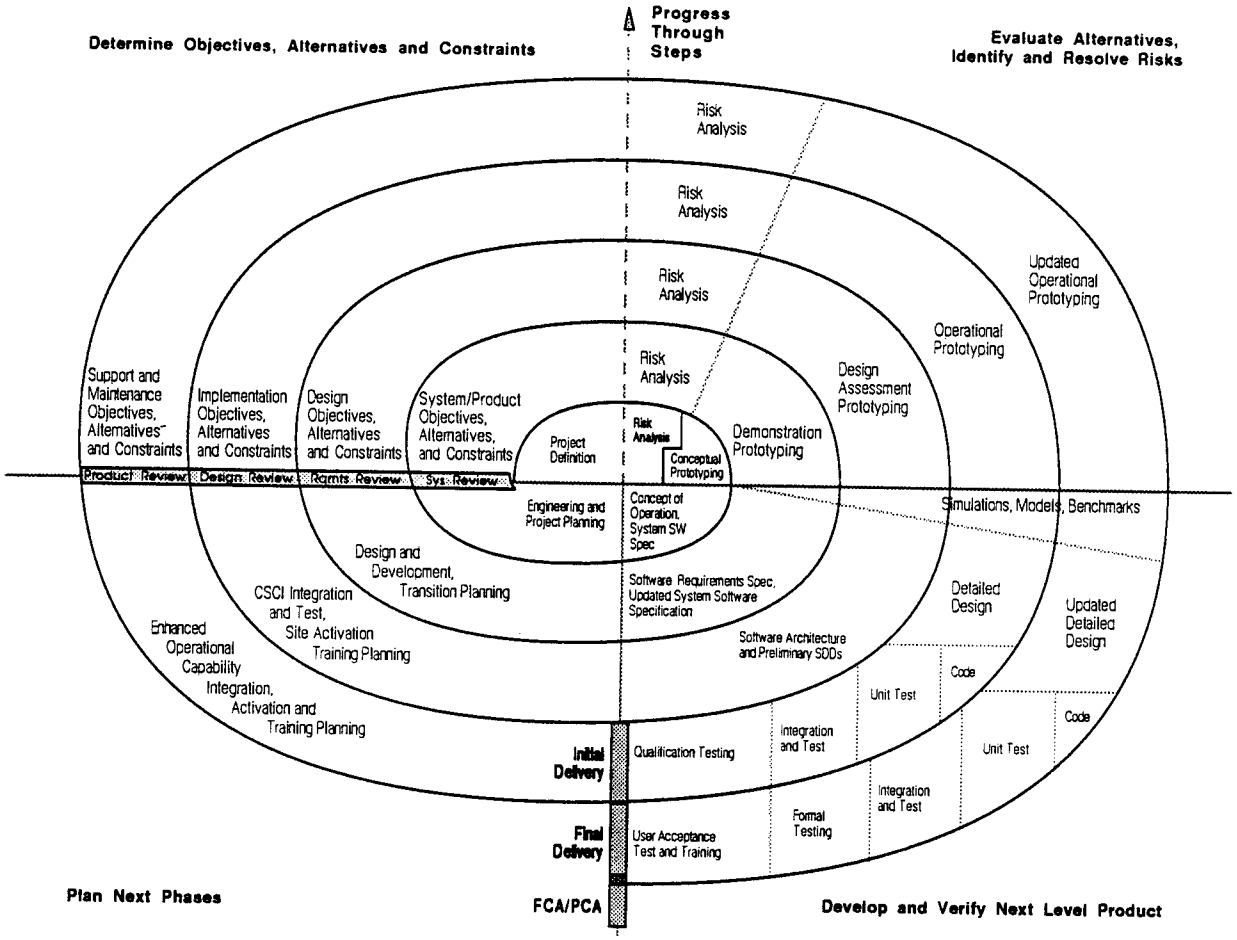


Figure 1. Spiral Model of the Software Development Process

management, quality assurance, testing, formal verification, etc. should a project do?' Using the risk-driven approach, one can see that the answer is not the same for all projects and that the appropriate level of effort is determined by the level of risk incurred by not doing enough.

- It does not involve separate approaches for software development and software enhancement (or maintenance). This aspect helps avoid the 'second-class citizen' status frequently associated with software maintenance. It also helps avoid many of the problems that currently ensue when high-risk enhancement efforts are approached in the same way as routine maintenance efforts. . . .

- Overall, risk-driven documents, particularly specifications and plans, are important features of the spiral model. Great amounts of detail are not necessary unless the absence of such detail jeopardizes the project. "[1]

- "It provides a viable framework for integrated hardware-software system development. The focus on risk-management and on eliminating unattractive alternatives early and inexpensively is equally applicable to hardware and software. "[2]

The model reflects the underlying concept that each cycle involves a progression through a specific set of steps for each portion of the product and for each of its levels of elaboration, from an overall concept of operations document to the coding at each individual program.

A typical cycle starts with the elaboration of the objectives of this portion of the product (performance, functionality, etc.), identifying alternate means of implementing this portion of the product (build, buy, reuse, etc.) and the

determination of the constraints imposed on the application of the alternatives (cost, schedule, etc.).

The next step is to evaluate the alternatives relative to the objectives and constraints. Frequently, this process will identify areas of uncertainty that are significant sources of project risk. If so, the next step will be the formulation of a cost effective strategy for resolving this risk. In a typical project situation this will entail additional prototyping, benchmarking, and/or modeling. Once the risks for this particular portion of the spiral model have been evaluated and appropriate actions defined, the next step will be to implement the actions. In the typical project application this will result in a further definition of the product through some level of prototyping until it is feasible to define the product to the extent that detailed design activities can take place and a waterfall model approach (detailed design, code, unit test, etc.) can be applied. After the applicable product has been developed and verified then planning for the next phases of the spiral model cycle will take place. This step will finish with a review of these plans (in a typical project this will be a formal review such as a requirements review, design review, etc.) and a formal commitment for the next cycle of the spiral model.

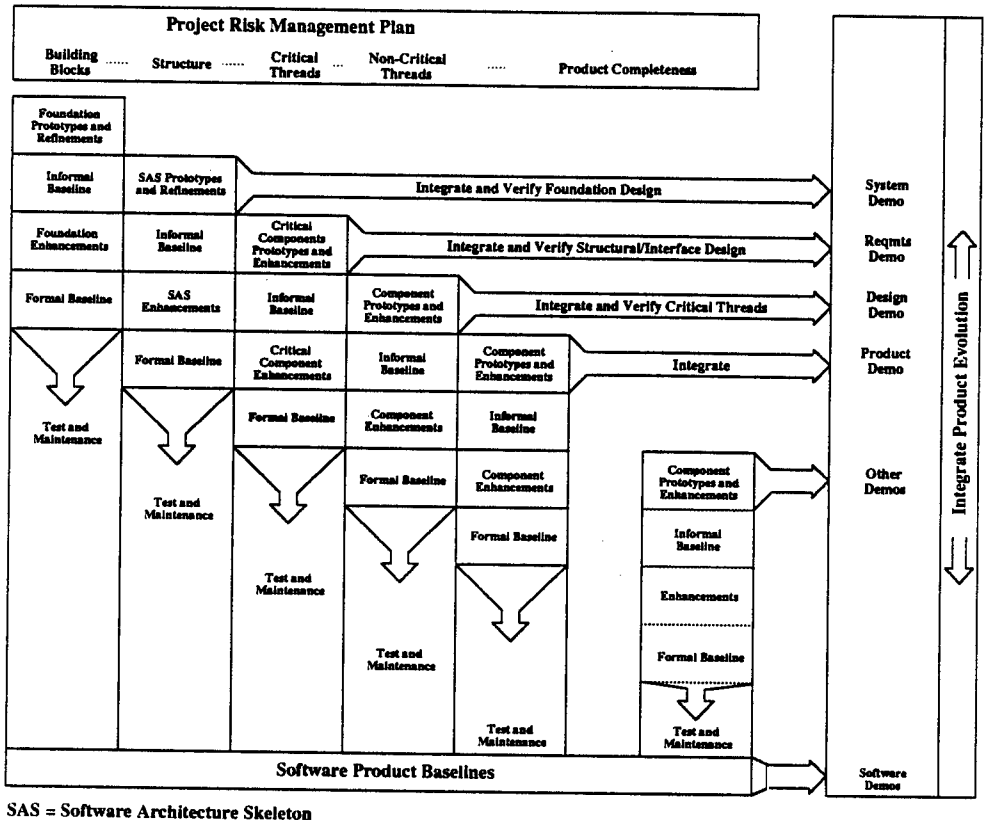
There are a couple of additional key aspects of the spiral model to note. First, it supports a series of evolutionary prototypes (demonstration, assessment, operational, etc.) that we have found are the best way to ensure that the requirements are correct and properly defined. Second, it supports incremental development ((Initial Operational Capability (IOC) delivery, Final Operational Capability (FOC) delivery)) which we have found is the best way to ensure that the requirements are being properly implemented and meet the true needs of the user. This ability of the spiral model to respond to real-world perturbations and influences is important in meeting today's complex software systems needs.

PROCESS

The proper use of Ada engineering is best exemplified in the Ada Process Model developed by Walker E. Royce, TRW's Chief Engineer for a major Ada development Program. Because Ada not only supports, but promotes good software engineering, it can be very effectively used as a life cycle language. The development of an integrated, efficient life cycle Ada process--(an Ada Process Model) is critical to the success of any Ada software development. The model that has been developed is a uniform application of incremental development coupled with a demonstration-based approach to design review that provides continuous insight into the development process. It enables the "design a little, build a little" approach to become reality and provides continuous "touch and feel" throughout the life cycle development. Figure 2 shows this process. It has major advantages over early process models because it not only has the advantage of building from the top down, but because the use of Ada supports partial implementation, the structure is real, can be easily evolved, and supports demonstrations of each level of development. Further, each build and subsequent demonstration validates the process as well as the structure. Finally, each level can constitute a formal baseline and be controlled.

It is important to recognize that large complex systems must evolve to a successful result, and not attempt "to do it all the first time" or rigidly attempt to define the total needs of the program at the beginning. Some design breakage due to unknowns, misconceptions, etc. will happen in the real world and must be accounted for, and the best way found to handle this is through an innovative approach to incremental development, as portrayed in the spiral model approach to software development. The Ada process model is in total concert with this spiral model approach with each build being a selected subset of the total software capability which implements a specific cycle of the spiral, and mediates the risks identified for that cycle. It is important to note that the process at each level provides the prerequisite components for smooth development of the levels that follow it, and provides for early prototyping of the development process itself. In addition, the early builds serve as "guinea pigs" for exercising the process that will be used in the subsequent builds.

The Ada process model takes full advantage of Ada as a design and implementation language and supports partial implementation (abstraction). Each component starts out as partial implementations of Ada program units maintained in compilable Ada format. Ada is used as an ADL (Ada Design Language) and is compilable Ada with placeholders for pending design details. This provides an outstanding means for supporting incremental development. It also provides easy and complete traceability of the components because the component structure is real (as opposed to paper) and evolves as the components become fully implemented rather than being displaced, as is often the case in paper designs, because little insight into the components integrated operations, performance and structure is provided at the early stages of design and development. Further, demonstrations of actual integrated components at early stages of the design and development are possible, allowing the design risks to



SAS = Software Architecture Skeleton

Figure 2. Ada Process Model

be mitigated early where changes are reasonably easy to implement, rather than at formal integration test time where changes are difficult, time consuming, and expensive. Focusing on integration during the design phase rather than the test phase is more cost effective, and leads to a far better chance of meeting total program schedules and costs. The increments of each level or build of the Ada process model are first structured at a high level of abstraction and refined, the various increments are then integrated and verified, and each level culminates in a demonstration of capabilities providing real "touch and feel." Another major advantage to the Ada process model is that it is multi-dimensional. Each build represents a layered architecture approach as well. The first build addresses the foundation components which are typically

1. the components needed to construct a software architecture skeleton
2. "the instrumentation needed to monitor a demonstration (i.e. to make the software execution visible)
3. the components needed to drive demonstrations (e.g., test scenario construction, and execution)
4. the components needed to do demonstration/test data reduction
5. development tools
6. reusable utilities
7. commercial off-the-shelf or existing versions of the above.

Note that the items above represent components which are likely to be depended on by large numbers of development personnel. Their early availability, usage, feedback and stabilization represents a key asset in avoiding downstream breakage and rework. "[3]

A critical element of this build is the development of a software architecture skeleton (SAS). This is fundamental to the success of the Ada Process Model. ". . . . Although different application domains may define the SAS differently, it should encompass the declarative view of the solution which identifies all top level executable components (Ada Main Programs and Tasks), all control interfaces between these components, and all type definitions for data interfaces between these components. Although a SAS should compile, it will not necessarily execute without software which provides data stimuli and responses. The purpose of the SAS is to provide the structure/interface baseline environment for integrating evolving components into demonstrations. The SAS represents the 'forum' for interface evolution between components. It is important to construct a candidate SAS early, evolve it into a stable baseline, and continue to enhance, augment, and maintain the SAS as the remaining design evolves. . . ."[4]

The SAS is expected to change and evolve but the ability of the process model to control this without resulting in design instability is another big plus.

Subsequent builds, as shown in Figure 2 focus on the critical components to verify the critical threads and then the remaining components are developed in a logical fashion that represents reasonable development packages. As the figure also shows, the completion of each of the build demonstrations results in the establishment of a formal baseline of the component build which can be formally controlled by configuration management. The importance of demonstrations as the primary product of the Ada process model needs to be emphasized.
". . . .Traditional software developments under the current military standards focus on documentation as intermediate products. To some extent, paper is certainly useful and necessary. However, by itself it is inadequate for large systems. Fundamental the Ada Process Model is forcing design review [sic] to be more tangible via visibly demonstrated capabilities. These demonstrations serve two key objectives:

1. The generation of the demonstration provides tangible feedback on integratability, flexibility, performance, interface semantics and identification of design and requirements unknowns. It satisfies the software designer/developer by providing first hand knowledge of the impact of individual design decisions and their usage/interpretation by others. The generation of the demonstration is the real design review. . . .
2. The finished demonstration provides the monitors of the development activity (users, managers, customers, and other indirectly involved engineering performers) tangible insight into functionality, performance, and development progress. One sees an executing Ada implementation of important and relevant capability subsets, but not necessarily on target hardware. "[5]

As a final note, it is important to recognize that the separation of specifications and bodies, packages, sophisticated data typing, and Ada's expressiveness and readability are powerful features in providing a well-engineered development approach.

ENVIRONMENT

The actual production of the Ada code and its associated documentation is best done in what can be termed an "Ada Factory." This is an integrated set of hardware and software that enables the software design, development, testing and documentation to be done in an essentially automated fashion with built-in standards, compliance verification, metrics collection, and progress monitoring. This environment provides high productivity and quality, traceability, and management visibility. The environment is capable of automatically generating 2167a documentation, and facilitates updating design information, code, and documentation while maintaining consistency and control. Figure 3 shows this Ada Factory with its hardware complement and the types of tools that support each phase of the Process Model cycle.

This "Ada Factory," often called an Ada Programming Support Environment (APSE) was devised as a true life cycle support environment, starting with the premise that you could input a set of requirements into this environment, perform requirements analysis and prototyping on these requirements and settle on a requirements baseline. Then, from this baseline, design the system graphically, develop and update the code, test the results, and produce the documentation, all done in an automated and integrated fashion. All of these activities could be conducted while maintaining traceability, configuration control, quality and productivity metrics, and having available the necessary editors, analyzers, debuggers, checkers, formatters, optimizers and design aids typically available in any good environment. The decision was made to automate not only the forward progress paths through the spiral model (such as PDL generation from design graphics), but feedback loops as well. The "Ada Factory" therefore includes "reverse engineering" capabilities, enabling the regeneration of design diagrams from updated Ada code. The automated reverse engineering greatly reduces the amount of rework

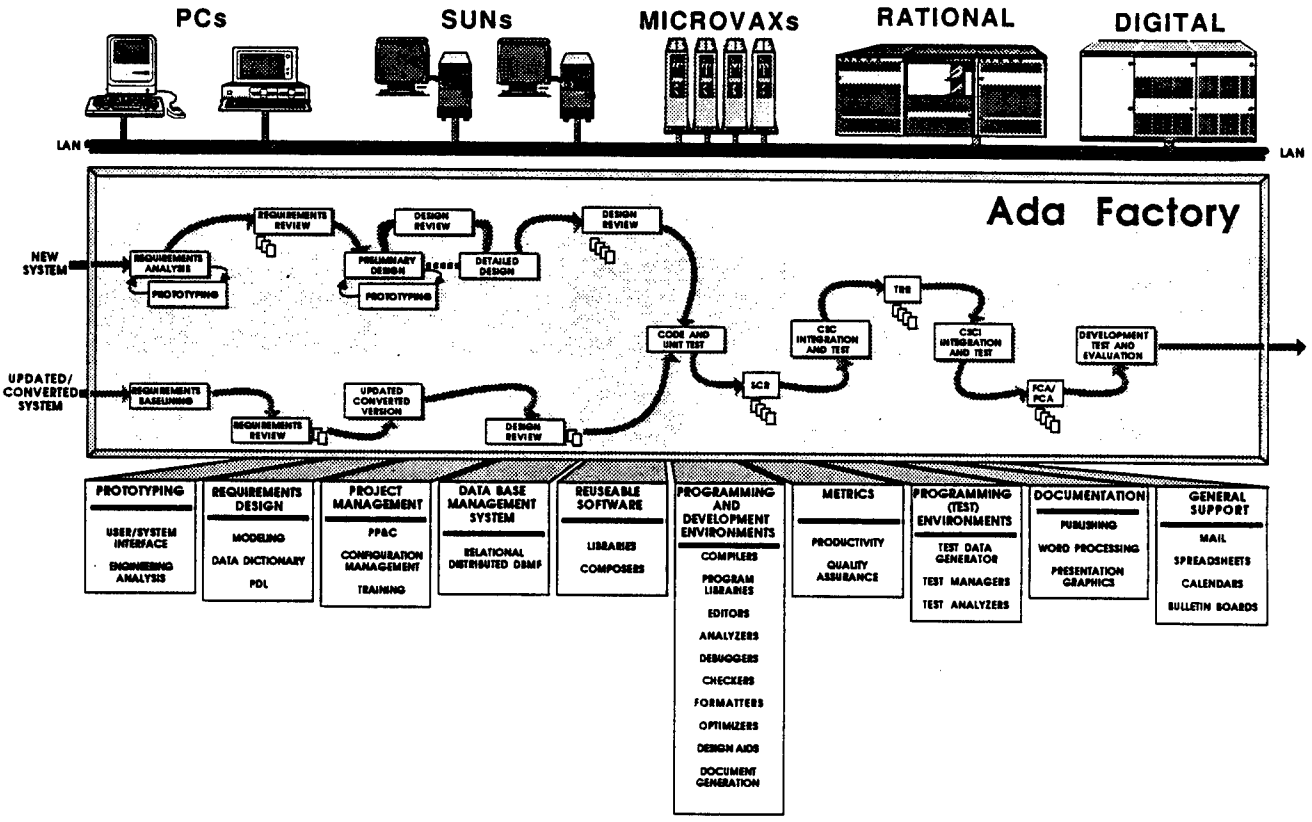


Figure 3. Ada Factory Approach

required to maintain consistency between various elements of the development process, such as graphics, code, test plans and cases, and textual documentation.

The particular capabilities that were necessary and the life cycle process chosen to be incorporated in the "Ada Factory" are shown in Figure 3. To best meet user needs the factory was built around a Rational processor with its Design Facility software as the Ada engine, a Digital Equipment 8xxx computer as the file server, and SUN 3 workstations for the graphics, document production, and Human to Machine Interface (HMI) front end, as these represented what were found to be the most sophisticated and robust Ada support components available when work on the factory began. A number of COTS tools were then successfully integrated, and additional capability was built where off-the-shelf software was not available to provide a complete and fully integrated Factory.

The "factory" supports the entire life cycle software development process starting with the requirements definition activities. This is accomplished through the structured input of requirements and expansion information on each requirement to the Rational Design Facility tool. Although the requirements information is mainly textual, tabular, and graphical, it is structured within skeletal Ada constructs which can be optionally migrated into the Ada design. The Rational machine maintains configuration control over updates to requirements information, including components such as graphics which exist on other platforms. The Rational can then automatically generate the 2167a Software Requirements Specification document. This document is produced within the environment in Interleaf (a publishing system) mark-up format, including the merged graphics images from other platforms. Requirements for each segment may be further ascertained through the use of prototyping and simulation. The environment includes a sophisticated user interface prototyping tool providing full windowing and graphics capabilities. Prototypes developed using this tool are automatically generated into Ada code implementing identical functionality.

Once requirements for each spiral segment have been identified and baselined, the next step is to evaluate design alternatives for satisfying the requirements. The "factory" supports the selection of alternatives in two fashions. First, architectural paradigms can be reviewed through a graphics browser which stores information on various Ada design "models." The Ada code implementing each model can also be reviewed through the browser, and selected in entirety or in part for reuse within the system under design. An Ada reuse library containing a selection of low-level data abstractions, general purpose tools and application-specific reusable components is then used to support the definition of design alternatives.

Design alternatives are evaluated by engineers through the prototyping of representative threads through the design. Each alternative can first be conceptually modeled in graphic format, and then refined as ADL. The environment supports the automatic generation of ADL on the Rational machine from graphics images developed on the SUN workstations. Designers can then make full use of the Rational environment to further specify data types, interfaces, and architecture. This design activity is supported by the Rational design facility which enforces project-defined coding standards and collects supplemental design information to be included in the Software Design Document (SDD). Throughout the design process, graphics can be regenerated on the SUN workstations from ADL on the Rational using a reverse engineering capability. This ability provides a high-level representation of the system for manager and customer review with the added confidence factor of a direct tie between the graphics and the current implementation.

Once a particular design alternative has been selected, the completion of the implementation in Ada code is supported in three ways. First, the code that defines the HMI interface of the system is automatically generated from the user prototype developed during the requirements analysis activities. Second, the software architecture model library provides code "templates" which can be modified to provide the desired functionality. Third, the environment includes a number of tools which encapsulate functionality for database interactions, message handling, interprocess communication, and formatted object (such as military message) parsing and generation. The availability of these tools significantly reduces the amount of Ada code which must be produced by developers. In addition to automatic code auditors which verify compliance to standards, developers can browse the standards library while they are in the process of developing a module. A developer simply highlights a particular construct for which he or she is uncertain of the constraints, and requests to view the standards which pertain to that construct. This in-process help greatly limits the amount of rework necessitated by a post-completion code audit.

The test phase of each spiral model cycle is likewise supported through an integrated toolset. Both requirements-based (white box) and code-based (black box) testing techniques are accommodated in tools which automatically generate test scenarios, test drivers and stubs, and test data from the requirements information and code on the Rational environment. Although these tools are Rational dependent, the drivers, stubs and data which they produce are entirely portable to the target environment.

The remaining environment components serve project management objectives. The environment automatically collects both software quality and completion metrics on the Rational and provides management reports on the SUN workstations. This information can then be used to automatically update project schedules and staffing levels. Standard project management activities such as WBS definition, staff allocation, cost projections, critical path analysis and labor curve levelling are preformed using COTS project management software on SUN workstations.

All products are integrated through a publishing system. The outputs of each tool are either automatically produced or converted into a standard format, allowing the ready document text, briefing slides, graphics, metrics reports, test results, Gantt and PERT charts, etc. There is also sign assimilation of items produced outside of the environment, either through electronic transfer and conversion, or through image scanning.

The "Ada Factory" provides automated and integrated support for all lifecycle phases, and is fully compliant with the spiral development and Ada Process models. It has proven to be an outstanding complement to the Ada Process Model in terms of applying principles and practices. Although the full factory capability is not yet complete, the initial capability has produced significant increases in productivity while providing much better tracebility and quality control than the standard compiler based with non-integrated support tools (tool bag) environment that is more commonly employed.

CONCLUSION

The bottom line is that using Ada engineering to drive the approach, process and environment for software development has resulted in a high productivity, low cost

approach to software development that is significantly better than what can be achieved in non-Ada general purpose language approaches, particularly in terms of life cycle cost savings, maintainability, portability, and reusability. Although Ada is certainly not the answer to all large development problems, we have shown that it is a better solution to most software systems development problems and when looked at the terms of a classic FORTRAN or COBOL baseline we are seeing the following results:

- Higher productivity due to the Ada factory (2 to 4 times SLOC/Man-Month over traditional productivity).
- Less required changes due to prototyping and incremental deliveries (25-50% savings over life cycle).
- Better quality, portability, maintainability, reusability, and reliability due to Ada software engineering (25-50% savings over life cycle possible).

NOTES

1. Barry W. Boehm, "A Sprial Model of Software Development and Enhancement," Computer, May 1988, p. 68.
2. Ibid., p. 69.
3. Walker Royce, "TRW's Ada Process Model for Incremental Development of Large Software Systems" [paper], p. 6.
4. Ibid., p. 7.
5. Ibid., p. 8.

REFERENCES

[Boehm 1981] Boehm, B. W., Software Engineering Economics, Prentice-Hall, 1981.

[Boehm 1985] Boehm, B. W., "The Spiral Model of Software Development and Enhancement", Proceedings of the International Workshop on the Software Process and Software Environments, Coto de Caza, CA, March 1985.

[Royce 1989] Royce, W. E., TRW's Ada Process Model for Incremental Development of Large Software Systems.



HERBERT P. WOODWARD

ONE FEDERAL SYSTEMS PARK DRIVE

FP2/1213

FAIRFAX, VA 22033

BIOGRAPHY

Mr. Woodward is presently the TRW Deputy Project Manager for the Army Worldwide Military Command and Control System (WWMCCS) Information System (AWIS) project, the largest Army Ada development activity contracted to date. He has a BS in engineering from MIT and an MS in Management from USC. He has been intimately involved in the software crisis for almost 30 years and is an acknowledged authority on software engineering and management in general and on Ada in particular.

INCREASING SYSTEM RELIABILITY THROUGH MULTI-LEVEL FAULT TOLERANCE

Darren J. Stautz

SofTech Incorporated

Abstract

In highly reliable computer systems a failure must be detected and recovered from while minimizing the system outage and preventing data loss. This paper examines the multi-level fault tolerance approach employed on the USAF SPACECOM Granite Sentry program. Three levels of fault tolerance are used to increase the system reliability. The first level is the use of redundant hardware and software. At the second level software processes control the redundancy, monitor the hardware and software, and detect failures. Finally, the application processes temporarily store intermediate results to disk files while processing to ensure no loss of data due to failure.

Introduction

The USAF SPACECOM Granite Sentry Program is responsible for upgrading or replacing portions of the North American Aerospace Defense Command (NORAD) Computer System (NCS) and Modular Display System (MDS). Granite Sentry will integrate the missile (NCS) and Modular Display System (MDS). Granite Sentry will integrate the missile warning, space surveillance, atmospheric defense, and intelligence information functions of the NORAD Command Center (NCC).

The Ada application processes developed to receive, process, and display the mission critical information are required to be extremely reliable with no loss of data. Specifically, Granite Sentry shall not lose more than one in 10^5 messages. Two methods are often employed to increase the overall system reliability. First, fault avoidance techniques are used to ensure that the system design and development is performed in such a manner to achieve the desired level of system reliability. Secondly, fault tolerance techniques are used to increase the system reliability in the presence of faults. Several definitions of fault tolerance exist. See [AND81], [AVI79] , and [SIE82]. However, all of these definitions require the use of protective redundancy to increase the system reliability by accepting the fact that the system will suffer faults during operation.

This paper is concerned with the implementation of the fault tolerance by Granite Sentry to meet the high reliability requirements. Fault avoidance techniques were also used. See [GOY89] for the Granite Sentry design philosophy.

Key to the multi-level fault tolerance approach are three software processes. The Status Monitor maintains the Granite Sentry hardware and software status. The Hardware Monitor determines the health of the Granite Sentry hardware and reports that health to the Status Monitor. Lastly, but most

important, is the Process Monitor which manages the redundancy and provides application process failure detection and recovery.

The multi-level approach to increase the system reliability is discussed in the following sections. First, the Granite Sentry Architecture is presented, describing the redundant hardware and software. Secondly, the process monitor design is highlighted. Finally, the role of the application processes in recovery is examined.

Granite Sentry Architecture

Granite Sentry is a message driven system. Messages are received by gateway processes from two external sources and directed to the appropriate missions for processing. Both air and space related messages are received from the Communications System Segment (CSS) while Missile related messages are received via the NORAD Computer System (NCS). The three missions (air, space, and missile warning) process the messages and forward them to the workstations where graphical and tabular displays are developed and maintained. In addition, users can input messages at the workstations for processing. The workstations provide a menu driven user interface to Granite Sentry.

Granite Sentry is being developed in a phased approach. The first phase was to develop the atmospheric defense (air) mission. Each phase adds a fully functional mission to the preceding phases. The second phase, currently in progress, is developing the missile warning portion. The next phase will implement the space mission of Granite Sentry. Figure 1 illustrates the Granite Sentry Architecture as of the Phase 2 development [GOY89]. For Phase 3, additional processors will be added for the space mission. The complete Granite Sentry system will be built in six phases over a seven year period.

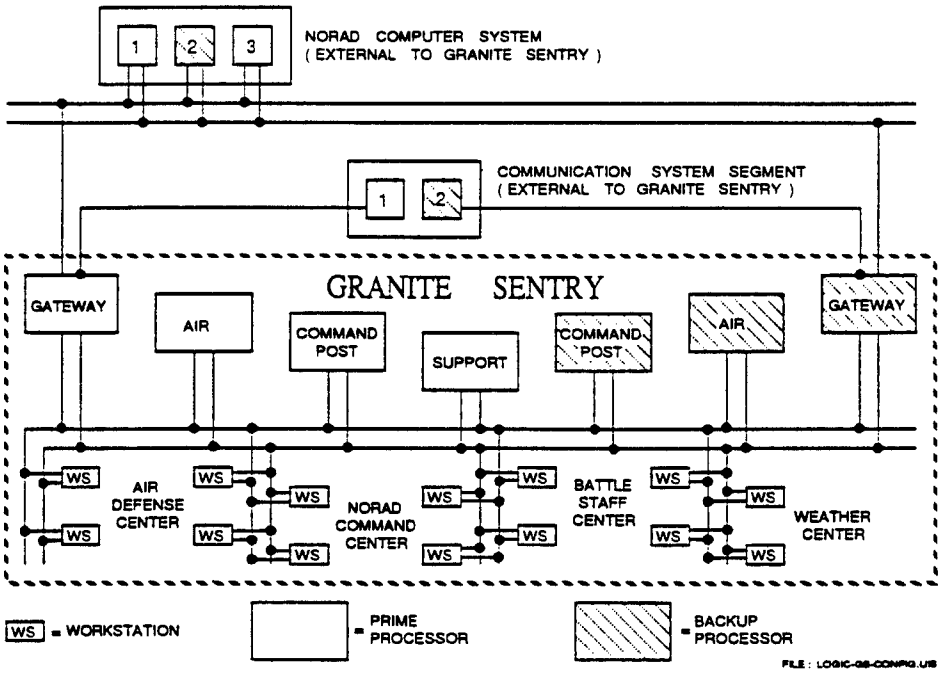


Figure 1. Granite Sentry Architecture

Redundant gateway processors provide the external interfaces between Granite Sentry and the NCS and CSS. The determination of the prime NCS gateway process (processor one or processor two) is assigned by the NORAD Computer System (NCS) with the backup NCS gateway process executing on the other processor. Similarly, the prime and backup CSS gateway process is determined by the Communication System Segment (CSS).

Likewise, five redundant mission processors host the air mission processes, missile warning processes, and support software. In the case of the mission processes, the prime and backup status is determined by the process monitors and is described in the following section. Several workstations are distributed throughout the Cheyenne Mountain Complex (CMC) to provide the user interface.

The gateway and mission processors are VAX 8550's with the five mission processors forming a VAXCluster. The workstations are VAXStation 3520's. The software is developed using the Ada programming language to execute under the VAX/VMS operating system.

On each processor, a software process exists to periodically confirm that the hardware is functional. The hardware monitor determines the hardware status and must send a health message periodically to a master hardware monitor. If a health message is not received within two time periods, then the hardware is assumed to have failed.

When either the hardware monitors or the process monitors detect failure, the failure is reported to the status collecting process, the status monitor. The status monitor maintains the complete Granite Sentry status information in a local database and this information is sent to the workstations to be displayed graphically. Additionally, alarms are generated and sent to each of the workstations to notify the system operators in case of either hardware or software failures. The process monitor is examined in more detail in the following section.

Process Monitor

The name process monitor is really a misnomer, in that it is responsible for monitoring and controlling a processing string that may range from one to several Ada application processes. A processing string is a related set of processes that must operate on the same processor. A nonrecoverable failure in one of the processes results in the termination of the other processes in the processing string. The Granite Sentry processing strings are delineated by the Computer Software Configuration Item (CSCI) boundaries.

The processing strings currently monitored by process monitors include the CSS gateway, NCS gateway, air mission, missile warning mission, status monitor, and the workstation software. Additionally, process monitors exist on each processor and workstation for system processes to synchronize the clocks and to monitor the hardware. Phase 3 of Granite Sentry will add the space mission and another gateway processing string. The number of processes in a processing string vary from two in the air mission process monitor to more than ten in the workstation process monitor.

The process monitor was originally designed to monitor and control only one application process. This required mailbox communication among the other process monitors of the other application processes in the processing string in case of failover. However, since the Ada programming language provides concurrency through the use of

tasks, the process monitor was redesigned to monitor and control the entire processing string. The Ada tasking leads to a cleaner, more maintainable design and reduces the failover time significantly.

The chief data structure of the process monitor is the application monitor task type. An object of this task type is created for each of the application processes in the given processing string. The application monitor task uses VAX/VMS system services to create a termination mailbox for the application process that it monitors. Another system service is called to create the application process and associate it to the termination mailbox. The task then is suspended until the application process terminates and the VMS operating system records the reason for termination in the termination mailbox. The task reads the mailbox and uses the reason for termination to determine the appropriate recovery action.

Two possible recovery actions exist; failsoft and failover. Failsoft is defined as restarting the failed application process and normal execution is resumed. No change in the prime or backup status occurs. Failover, however, results in the termination of the other application processes in the processing string and allowing a backup processing string on another processor to assume the prime role.

Figure 2 graphically depicts the process monitor design for the version that controls mission software redundancy. In this instance, the process monitor is controlling a processing string consisting of three application processes. Three objects of the application monitor task type are created. Each of the tasks is waiting on the termination mailbox associated with the application process it is monitoring. The process monitor main procedure requests the prime lock that is controlled by the VMS lock

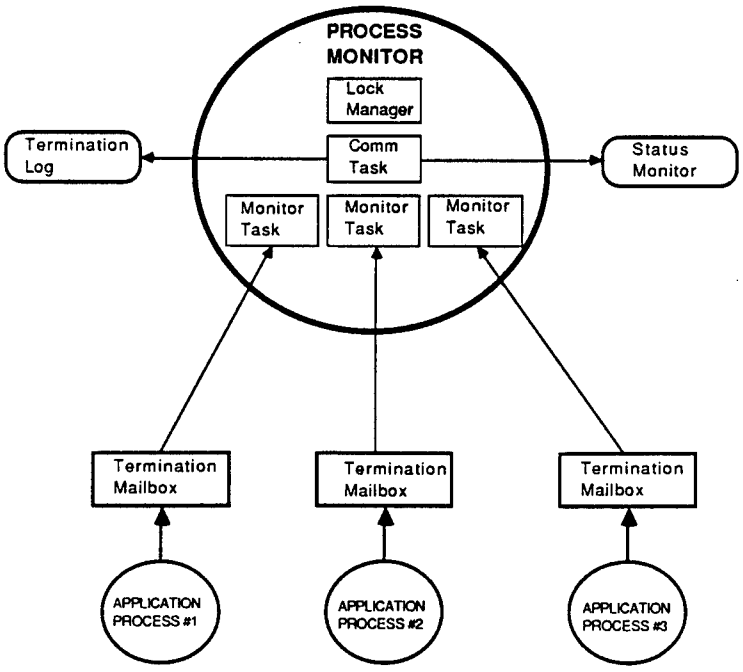


Figure 2. Process Monitor Design

management services for the whole cluster. If the process monitor receives the lock, then an event flag is set to inform the application processes that they are prime. If the lock is not granted, then the process monitor waits in a queue until the prime processing string terminates and the lock is finally granted. The gateway and workstation process monitors do not utilize the lock management services due to the external interface controlling the prime/backup status of the gateways and no failover takes place in the workstations.

Another task synchronizes the process monitor's communications. The process monitor reports significant events such as process startup, termination, restart, shutdown, and failover to the status monitor. Additionally, a log of every process termination, including the name of the process, the time of failure, the reason for failure, and the recovery action implemented, is recorded for maintenance programmer review.

The application monitor task type processing is highlighted in the following pseudo-code. Both the interface to the VMS system services and the Communication routines are encapsulated in Ada generic packages. The Recovery Services package is the instantiation of the VMS System Service generic. The Communications package is the instantiation of the Communication routines generic.

At the system startup time, the process monitors are started. The process monitor in turn starts the application processes it is monitoring. The process monitors vie for the lock to determine the prime/backup status of their mission processes. Figure 3 demonstrates the redundant versions of a mission process monitor being started on two processors.

The process monitor on processor one received the lock and notified its set of processes that they are prime via the set event flag. The process monitor on processor two is queued for the lock and its set of processes are informed that they are backup via a cleared event flag. Only when the process monitor receives the lock is the event flag set to inform the application processes of their prime status.

Failure detection is depicted in figure 4. The second process terminated abnormally and the reason for termination was recorded in the termination mailbox by the VMS operating system. The application monitor task then determines the recovery action based on the reason for termination.

The failure recovery is implemented by both the process monitor and the application processes. The process monitor determines whether the recovery action is failsoft or failover. The application process must be able to continue processing without loss of data. Examples of both failsoft and failover recovery actions are presented in the following paragraphs.

In the case of failsoft, as shown in figure 5, the process monitor simply restarts the failed application process. The application process must be able to resume processing where it left off. First, it must review the contents of its safe data store in memory to determine the data it was processing at the time of failure. The data is retrieved and then processed. The communications mechanism will queue any messages delivered to the process until it can resume normal processing.

Figure 6 illustrates failover. The other processes in the processing string are terminated by the process monitor and the lock is released allowing the backup set of

```
begin -- APPLICATION_MONITOR

    - -  Determine the process to monitor
    accept STARTUP ( PROCESS : in PROCESS_NAME_TYPE )

    - -  Create the Termination Mailbox
    RECOVERY_SERVICES.CREATE_TERMINATION_MAILBOX ( MAILBOX )

    - -  Create the Application Process and associate it to the
    - -  termination mailbox.  The environment includes the
    - -  executable image, the priority, and process quotas.
    RECOVERY_SERVICES.CREATE_APPLICATION ( PROCESS
                                           MAILBOX,
                                           ENVIRONMENT )

    - -  Verify that the Process was created and is executing
    RECOVERY_SERVICES.CHECK_FOR_PROCESS ( PROCESS )

    - -  Report the process startup to the status monitor
    COMMUNICATIONS. REPORT_EVENT (  PROCESS,
                                    STARTUP )

    MAIN:
    loop

    - -  Suspend until the process terminates
    RECOVERY_SERVICES.WAIT_FOR_COMPLETION ( PROCESS,
                                            TERMINATION_CAUSE )

    - -  Assess the Failure and determine the Recovery Action
    RECOVERY_ACTION := DETERMINE_RECOVERY_ACTION ( TERMINATION_CAUSE )

    - -  Log the Process Termination
    COMMUNICATIONS.LOG_TERMINATION ( PROCESS,
                                     TERMINATION_CAUSE,
                                     TIME,
                                     RECOVERY_ACTION )

    case RECOVERY_ACTION is
        when RESTART =>
            - -  Restart the failed process
            RECOVERY_SERVICES.CREATE_APPLICATION ( PROCESS,
                                                   MAILBOX,
                                                   ENVIRONMENT )

            RECOVERY_SERVICES.CHECK_FOR_PROCESS ( PROCESS )

            - -  Report the Restart
            COMMUNICATIONS.REPORT_EVENT ( PROCESS,
                                          RESTART )
        when SHUTDOWN =>
            - -  Report the Shutdown
            COMMUNICATIONS.REPORT_EVENT ( PROCESS,
                                          SHUTDOWN )

            - -  Terminate the Task
            exit MAIN loop
        when FAILOVER =>
            - -  Report the Failover
            COMMUNICATIONS.REPORT_EVENT ( PROCESS,
                                          FAILOVER )

            - -  Shutdown the other processes in the processing string
            SHUTDOWN
            exit MAIN loop
      end case -- RECOVERY_ACTION
   end loop MAIN
end APPLICATION_MONITOR
```

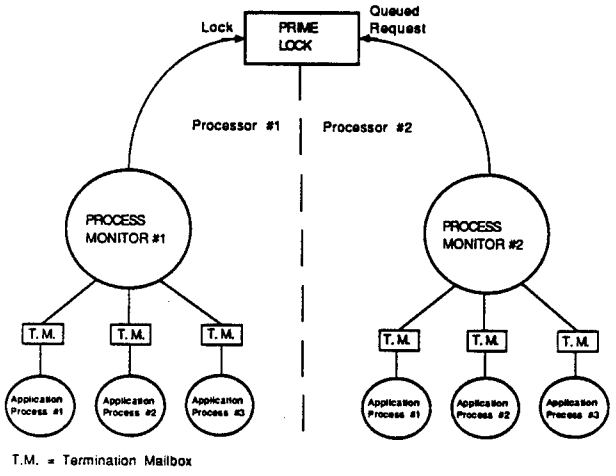


Figure 3. System Startup

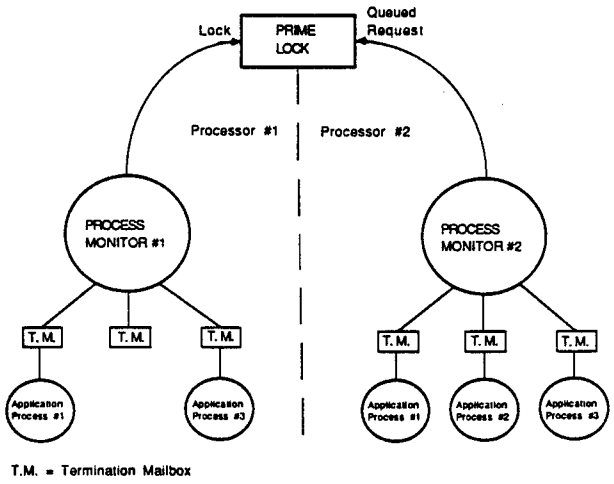


Figure 4. Failure Detection

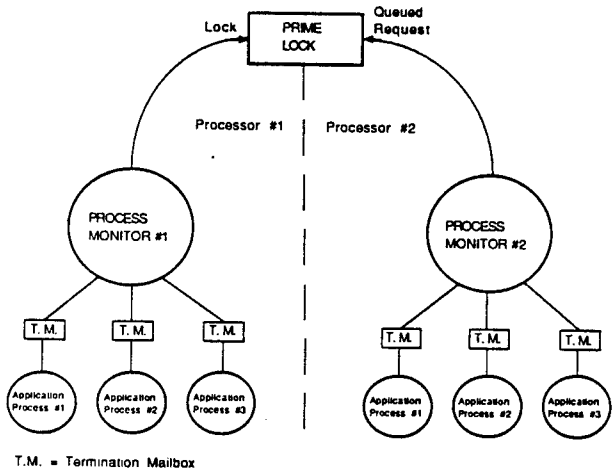


Figure 5. Failsoft Recovery

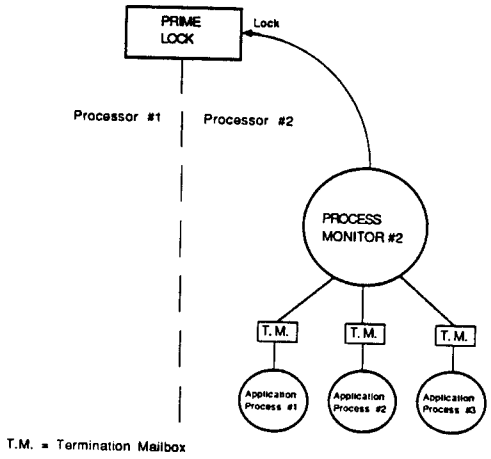


Figure 6. Failover Recovery

processes to assume the prime status. Human interaction is required to restart the process monitor on processor one to bring it up as a backup. The new prime application processes must determine the data the failed application processes were processing at the time of failure and recover in a similar manner as the application process recovery in failsoft conditions.

The application processes are coded to handle only the exceptions that can be recovered internally. If a fatal exception is raised, the exception is propagated until it results in the termination of the process. The operating system provides a traceback delineating where the exception occurred and the nested calling structure of the procedures leading to the location of the exception. This traceback information is preserved in an output file that is associated with the application process when it is started by the process monitor. The traceback information is valuable to the maintenance programmers to decipher why the failure occurred.

Conclusions

The three levels of the Granite Sentry fault tolerance were described herein. The first level being redundant hardware and software. The second level is the software processes that monitor the hardware and software. Key to this level is the process monitor which performs redundancy management, failure detection, and implements the recovery at the high level. Finally, the application processes complete the recovery by maintaining a safe data store to resume processing at the point of failure.

The process monitor was designed to be highly reliable and maintainable. The majority of the processing was designed to use the operating system services available. These system services have been thoroughly tested promoting reliability. Maintainability is incorporated through the use of Ada constructs such as generics and the application monitor task type. A generic Ada package encapsulates the interface to the operating system services thus isolating the system dependencies to a single package. Similarly, the communication routines are encapsulated in another generic package.

The process monitor is not a cure-all. It is possible for a common software design flaw in an application process to cause failure in both the prime and backup versions. Whenever failover occurs, however, an alarm is generated warning the system operators that the prime set of processes has failed and that the backup is now assuming control. These alarms will serve to minimize the system outage due to common mode failures.

The process monitor has proven to be a very effective software fault tolerance technique on the Granite Sentry program. During integration, the process monitor served as a valuable debug tool in detecting failures without interrupting system processing. Additionally, by preserving the traceback information, the reason for termination and the location of the point of failure are recorded to determine what "bugs" exist in the software.

In summary, the process monitor provides the following advantages:

a) Consistent and accurate failure detection,

b) Intelligent failure recovery based on the reason for process termination,

c) Reduced application complexity due to the transparent nature of the process monitor in providing fault detection and recovery,

d) High reliability through the use of proven operating system services,

e) Ease of maintenance through the use of advanced Ada constructs such as generics and task types, and

f) System maintenance assistance through the preservation of the traceback information and the termination log to record the reasons for the abnormal termination for review by maintenance programmers.

References

AND81 T. Anderson, P.A. Lee, Fault Tolerance - Principles and Practice, Prentice Hall, 1981.

AVI79 A. Avizienis, "Towards a Discipline of Reliable Computing", IFIP Working Conference on Reliable Computing and Fault Tolerance in the 1980's, Proceedings of the Europe IFIP 79, London, September 1979, pp 701-705.

GOY89 M. Goyden, "The Software Lifecycle with Ada: A Command and Control Application", Tri Ada, Proceedings of the Tri Ada Conference, Pittsburgh, October, 1989, pp 40-56.

SIE82 D. Siewiorek, R. Swarz, The Theory and Practice of Reliable System Design, Digital Press, Bedford, MA 1982.

Acknowledgements

I want to thank my colleagues on the Granite Sentry program and at SofTech for encouraging me to write this paper and their support in reviewing it. I especially want to thank George Macpherson, Mike Goyden, and Bill St. John for their many reviews and comments. Rich Wallace provided the original concept of using the VAX/VMS system services to monitor and control the Granite Sentry application processes. Lastly, but definitely not least, I want to thank Marti Devine for the technical production of the paper.

About the Author

Darren J. Stautz has just completed the requirements for his Master's Degree in Computer Science at Chapman College. He will graduate in May. He received his B.S. in Computer Science at Seattle Pacific University in 1984. After graduating, he joined the Boeing Aerospace Company to work on the Integrated Fault Tolerant Avionics System (IFTAS) Research and Development project. In 1987, he joined SofTech Incorporated to develop C3I systems in the Ada programming language. Readers may correspond with the author at SofTech, Inc., 1670 Newport Road, Suite 300, Colorado Springs, Colorado 80916.

DESIGNING PARALLEL DATA TYPES FOR ADA

Paul B. Anderson
Planning Research Corporation
1500 Planning Research Drive
McLean, Virginia 22102

E. K. Park
Computer Science Department
U.S. Naval Academy
Annapolis, MD 21402

Abstract

The design of a set of Ada packages defining parallel data types is described. The parallel data types and operations defined on them are intended to provide natural Ada constructs for exploitation of the data parallel Connection Machine. Implementation concepts are described and Ada restrictions which affected the design are identified. Particular attention is given to the design impact stemming from provisions for use of Ada tasking.

1. INTRODUCTION

1.1 Background

Since the 1970s, the Department of Defense (DoD) has been supporting development and use of the Ada programming language. Designed to meet the needs of both large and small DoD software systems, Ada provides many features which promote software development productivity, ease of maintenance, efficiency, and software reusability.

Use of Ada has been mandated by the DoD for several years for new embedded and mission critical systems. Only now, however, is it becoming widely accepted, as positive experiences with Ada systems spur wider use. Ada's long term life cycle benefits may ultimately cause its spread into the scientific domain, displacing Fortran as the language of choice.

The DARPA Strategic Computing Initiative, which has sponsored the development of the highly parallel Connection Machine (CM), must maintain the ultimate goal of placing this unique machine in operational DoD systems. An efficient and useful interface between Ada and the Connection Machine would demonstrate how such powerful and unique hardware could be accommodated within an Ada environment. As will be discussed below, this can be done without sacrificing the benefits of the Ada language.

1.2 Objectives

The primary objective of this work is to develop a software binding or interface between the Ada language and the CM using the underlying CM machine language, PARIS. This binding is based on a series of parallel data types which correspond with data fields allocated on the Connection Machine. These parallel data types are similar to arrays in that a single identifier refers to a set of values of a single type. The main difference is that the data values are stored in the Connection Machine, one element per processor, and that operations take place elementwise and in parallel. Essentials elements of the Connection Machine architecture are discussed in Section 2.

The binding is intended as a mechanism for application software, called client software (or just client) in the remainder of the paper, to harness the Connection Machine processor array using the standard Ada package mechanism. The client view of this interface may be sufficiently general to apply to other Single Instruction/Multiple Data (SIMD) computers and to vector processors. The interface is not expected to be appropriate for Multiple Instruction/Multiple Data (MIMD) computers which are already supported by Ada's intrinsic tasking features [1, 2].

Programming the Connection Machine presently involves the use of specialized algorithmic techniques and specialized languages. Languages in use for the Connection Machine include:

- PARIS: the underlying machine language;
- *LISP: a series of parallel extensions built on Common LISP;
- C*: an extended C based on the concepts of the C++ language; and
- Connection Machine Fortran: based on recommendations for the proposed Fortran-8x standard plus extensions.

None of these languages is entirely satisfactory. Use of LISP in scientific and engineering programming is not common and, although *LISP allows access to all CM capabilities, significant user resistance and a shallow learning curve have been experienced. C* represents a unique language, implemented only on the Connection Machine. The linguistic ability of C* to extract full Connection Machine performance without resorting to PARIS calls has not been established. Connection Machine Fortran may ultimately provide the most machine independent programming environment since it is based on a proposed standard. However, Fortran is primarily used for scientific applications and is not particularly suitable for other uses such as embedded DoD applications.

The interface will be implemented within the Ada language as several packages. No language syntax extensions are anticipated, implying that programs written using the interface packages will be considered to be pure Ada programs under DoD Ada policy. The interface should be general enough to apply to other fine-grained parallel machines and, hence, could be adapted to those machines without change to the external client software view. This should enhance transportability of applications across at least some family of parallel machines. A side benefit of using the package concept as the implementation method is that a completely simulated interface can be provided, thus simplifying application development.

2. CONNECTION MACHINE ESSENTIALS

In the past several years, there have been a number of new, powerful machines introduced in the supercomputer field. Most of these machines have been somewhat smaller and cheaper than traditional supercomputers such as the Cray series. Sometimes called mini-supercomputers, the architectures of these machines vary dramatically, employing pipelining techniques, parallel processors and even horizontal microcoding. One of the most promising of these new computers is the Connection Machine.

The Connection Machine is a massively parallel computer [5]. It achieves processing rates above one million instructions per second (1,000 MIPS) and above 1 billion floating point operations per second (GFLOPS) by using a very large number of very simple processors. Cray X/MP processors achieve performance in the hundreds of million floating point operations per second (MFLOPS).

The Connection Machine is a specialized machine for computationally intensive applications. Special programming techniques are used for the machine, thus programmers must learn to think of parallel solutions, unlearning old habits from sequential machines and sequential languages.

Figure 1 shows the layout of the basic components of the Connection Machine. The Connection Machine is attached to a front end computer bus much as are array processors such as those built by Floating Point Systems and others. The front end machine has its own processors, memory and peripherals and controls overall execution of the combined system. The microcontroller joins front and back end machines and also implements higher level variable bit length operations using one bit wide Connection Machine processors. Each Connection Machine processor has local memory for storage of data only. Programs are stored in the front end machine memory. The box surrounding the processors denotes the hypercube communication system described in Section 2.2. The hypercube is used for interprocessor memory references.

2.1 SIMD Architecture

The Connection Machine consists of 4,096 (4K) to 65,536 (64K) one bit wide custom CPUs, all synchronously executing a single instruction stream on different data items. This type of parallelism is known as Single Instruction, Multiple Data or SIMD.

The Connection Machine is designed to operate on arrays of data spread across the array of processors. Usually a single element of a parallel array of data is stored on one processor. Operations on the parallel data are carried out synchronously by all processors, operating on their elements of parallel data. For cases where the computation should take place in a subset of processors (for example to avoid dividing by zero), each processor has a context flag. If the context flag is turned on, the processor executes the instructions broadcast to all processors. Otherwise the processor waits. A parallel *if-then-else* construct can be built using the context flag mechanism. The boolean condition is evaluated in each processor on local data elements. Those processors which satisfy the condition continue to execute. Those not satisfying the condition are idled by resetting their context flags. Once execution of the *then* part is complete, context flags are inverted and the *else* part is executed. This process is called processor conditionalization and the set of context flags is called the processor context.

Other types of parallelism also are represented by contemporary machines such as the Multiple Instruction, Multiple Data (MIMD) Encore Multimax machine. In MIMD architectures,

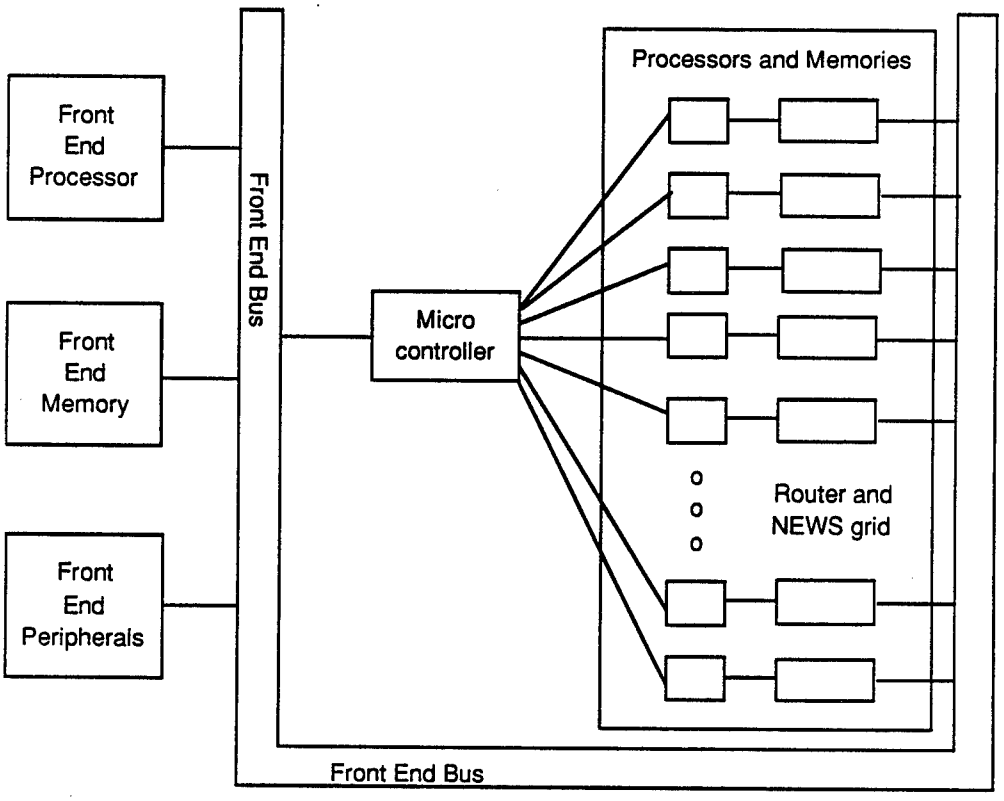


Figure 1. Connection Machine Layout

each processor executes independently. Processors must occasionally synchronize with other processors and exchange data as in an Ada rendezvous. Another common parallel architecture, exemplified by the Cray 1, uses pipelining. Pipelined machines rely on the assembly line principle to speed up certain regular operations such as moving, adding or multiplying vectors and matrices.

2.2 Hypercube Network

The processors of the Connection Machine each have local memory and can perform operations only on local data. To perform operations on data from another processor, a high capacity switching network is provided. Conceptually forming a 16 dimensional hypercube, this network is the origin of the Connection Machine name.

The hypercube organizes the processors into a 16-dimensional array with (Fortran) dimensions (2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2). Each single processor is connected to the 16 other processors whose subscripts match the given processor's subscripts except in one position. For example, in a four-dimensional hypercube, the 16 processors are arranged in a (2,2,2,2) array. Processor (1,1,1,1), for example, is connected to processors (1,1,1,2), (1,1,2,1), (1,2,1,1), and (2,1,1,1).

Getting from one Connection Machine processor to any other requires traversing multiple legs of the hypercube, moving from a processor to one of the 16 it is connected to at each step. Eventually, the message to send or receive data gets through, after traversing up to 16 legs.

A hypercube network provides many parallel paths for data messages, but is not a complete cross-bar switch. This means that there will be some contention for connections from messages being routed through the network. Messages that need the same connection must take turns using it, resulting in delays. The pattern of message senders and receivers determines the contention that will occur, and hence the total delivery time of all messages. A random pattern of messages takes on the order of 800 microseconds to transmit.

The advantage of the hypercube is that many transmissions can be in progress simultaneously. By contrast, a standard network such as Ethernet is designed for the situation where, on average, few transmissions are in progress at any point in time. On the SIMD Connection Machine, all processors do the same operation at the same time, resulting in either completely idle periods on the communication system, or totally busy conditions.

A complete cross-bar switch, providing a connection between all pairs of processors, would clearly be the optimal network from a performance point of view. However, a cross-bar switch would require about N^2 connections whereas a hypercube uses about $N \log_2 N$ connections. For 65,536 processors, the number of connections required by both systems is very large, but the cross-bar needs about 4,000 times more connections than the hypercube.

While the hypercube provides general processor to processor communications, more restricted communication patterns based on the regular grid structure of a VP set (see Section 2.3) are available and are much faster. Regular grid communication operations are known as NEWS operations where NEWS is an acronym for North-East-West-South, the grid directions available on the early Connection Machine model CM-1. Current machines support generalized grid communications over more than 2 dimensions at rates an order of magnitude faster than general communication using the router.

2.3 Virtual Processors

The large number of processors in the machine is best exploited by allocating one processor per data element. The definition of a data element depends on the problem and may vary during program execution.

For many problems, even the 65,536 processors of a full machine are not sufficient. For example, a 1024 by 1024 pixel image requires one million processors if one processor operates on one pixel. When the number of processors needed is greater than the number of physical processors in the machine, a mechanism called virtual processors is used.

Virtual processors are multiplexed by the hardware over the physical processors. Multiplexing divides the speed of the machine and its memory capacity by the number of virtual processors being simulated by each real one. For example, if there are four times as many virtual processors as real ones, the machine executes each operation at about one fourth the usual speed and has one fourth the memory per processor. The most attractive aspect of this arrangement is that the hardware takes care of all details of the multiplexing.

3. DESIGN CONCEPTS

The high level, basic requirements of the Connection Machine Ada interface are as follows:

1. Provide a set of parallel data types and operations on them.
2. Be as consistent with the standard Ada types and operations as possible, including use of compile-time type checking rather than run-time checking.
3. Implement the interface within Ada limitations.
4. Provide access to all major Connection Machine facilities including parallel arithmetic/relational/logical operations, scans, conditionalization, VP sets, router communications, and news communications.
5. Provide the same functionality as in the *LISP language.
6. Provide a basic demonstration capability with the potential to grow into a complete, useful, and efficient product.

Expansion of requirements into actual operators, procedures and functions has been guided by the Ada Language Reference Manual (LRM) [2], and by the developing Fortran 8x standard [7]. The Fortran 8x standard was found to be more helpful in designing a parallel Ada capability than the specific *LISP language functions because Ada and Fortran are much more closely related than Ada and LISP. The Fortran 8x language proposal includes extensive provisions for array operations and these have been adopted, to the extent possible, in the Connection Machine interface.

3.1 Parallel Data Types and Operations

The interface software provides a number of capabilities including:

1. Parallel variable (pvar) data types, in five different type groups:
   - Address pvars (linear, grid and relative grid);
   - Integer pvars corresponding to *LISP SIGNED-PVARs;
   - Boolean pvars;
   - Natural pvars corresponding to FIELD- or UNSIGNED-PVARs; and
   - Two floating pvar types corresponding to SINGLE- and DOUBLE-FLOAT-PVARs.
2. Operations on pvar data types including the standard Ada operators appropriately overloaded, along with most Fortran 8x array intrinsic functions and elemental operations;

3. Processor selection and subselection operations for conditionalization.

Major Connection Machine hardware (PARIS) capabilities are represented in an approach similar to that used in the TMC *LISP extensions [4]. There are, however, significant differences between *LISP and the Ada interface which are principally the result of being consistent with the underlying languages. Some of these differences represent significant Ada shortfalls. See Section 4.4.

The data types are all of fixed length, rather than client specifiable as in *LISP. Strong typing, a central feature of Ada, is preserved in the interface. This results in compiled code which contains no run-time type checking. The interface encapsulates the implementation of parallel variables and operations on them, thus preserving Ada's abstract data type and object-oriented programming features. The abstract data types and operations provide a base from which new client-defined types can be derived and constructed. The interface relies heavily on operator and procedure overloading which makes programming uniform, simple, and easily understood. Operator, function, and procedure overloading will permit compile-time selection of the appropriate procedure based on supplied operands. Standard Ada keyword argument association, default values, and overloading make the extensions appear similar to the *LISP functions in their use of optional and keyword arguments.

Pvar types are implemented as a set of derived record types. The five client types are:

- Boolean_pvar (length 1 bit);
- Integer_pvar (32 bits);
- Natural_pvar (32 bits but limited to 31 bits in front end variables);
- Single_float_pvar (32 bits); and
- Double_float_pvar (64 bits).

Because pvars are actually record types in Ada, it is not possible to establish bounds for derived pvar types.

The pvar record consists of a length field, and extent set number, a storage segment number, and a CM field ID. Extent and storage segment are discriminants for the record. As an example, an Integer pvar object describes a Connection Machine parallel variable. This variable consists of a set of 32 bit values, stored one value per virtual processor. The extent set parameter of the record determines how many dimensions the array of values has and the length of each dimension.

Pvars are allocated on the CM heap when their Ada declarations are elaborated. Deallocation must be performed by the client due to a lack of finalization processing in Ada. This is aided by the concept of a storage segment which allows storage reclamation without explicitly deallocating each variable.

3.2 Extents

Extent sets correspond to Virtual Processors sets (VP sets) in *LISP and PARIS. Extent sets are client-defined and establish the number of dimensions and the size of each dimension in a virtual processor set. A pvar object's extent set association which determines the object's size and shape is given by the extent discriminant and is set at allocation time. As in *LISP, Ada pvars have the exact size and shape as the extent set in which they are allocated.

Procedures are provided for selecting only a particular region or subregion within an extent set. The region can be specified using masks or using processor index ranges. Selected subregions provide another masking capability which is implicitly applied to operations. If explicit mask parameters (see Section 4.5) are supplied, selection masks may be considered to be logically ANDed with a selected subregion mask. Selected subregions remain in effect within an extent set even though the extent set may not be the active one. A mask of currently active processors in an extent set can also be obtained. Operations on pvars in different extent sets can be intermixed freely but, as in the underlying PARIS, cross-extent operations are limited to router operations which are implemented with the ASSIGN procedures and REF functions described in Section 3.3.

Figure 2 shows the basic structure of an interface function implementing the sum of two integer parallel variables. This example is not generic hence is only indicative of operations to be performed. The result is allocated in the same extent (VP set) and storage segment as the left operand. A run time check is performed to ensure that both operands have the same extent. Once this is verified, a critical section is entered to prevent other Ada tasks from using the CM hardware (these critical sections are discussed further in Section 4.2). The PARIS call is performed which actually does the addition. The critical section then ends and the result is returned.

```
function "+"(left, right: Integer_pvar)
         return Integer_pvar is
  result: Integer_pvar
            (extent -> left.extent,
             segment -> left.segment);
```

```
begin
  if left.extent /= right.extent then
    raise CROSS_EXTENT_ERROR;
  end if;

  connection_machine.lock("+");
  activate_extent(left.extent);

  CM_add_3_1L(result.field,left.field,
              right.field,left.length);

  connection_machine.unlock;
  return result;
end "+";
```

Figure 2. Example Operator Code

3.3 Assignment and Communication

The ASSIGN procedures exist primarily because Ada does not allow overloading of the usual assignment operator, ":=". One advantage of the procedural form over an operator form is that it allows conditional or guarded forms (those involving the WHERE and EXCEPT_WHERE masks) and nonlocal assignments to be naturally included in a unified view. Ada overloading makes the syntax appear to have many optional parts.

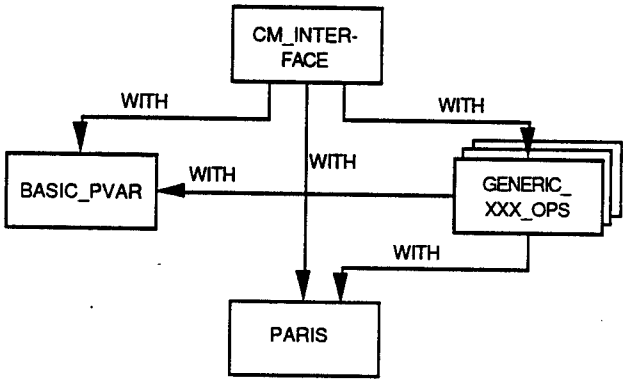
The overloaded ASSIGN functions provide assignment operations including the equivalent of *LISP's *PSET function. The most basic ASSIGN procedure performs a simple assignment. Variations are provided to provide assignment of a constant to a pvar. Other variations provide for storing computed values (scalar or parallel) into a pvar in other processors or for storing a scalar value into a single processor. Many ASSIGN variations include two optional mask parameters which provide conditional assignment. The WHERE and EXCEPT_WHERE parameters accept Boolean_pvars. If the WHERE parameter is specified, only those processors with the value true in the mask pvar will be active. If an EXCEPT_WHERE mask is specified, only those processors with the value false in the mask will be made active. When both are specified, both conditions are applied simultaneously. Only those processors with true in the WHERE mask and false in the EXCEPT_WHERE mask will be active.

A number of overloaded REF functions correspond to the *LISP PREF!! function. In both cases, REF stands for "reference." These functions involve interprocessor communication to obtain data values needed by processors which are stored in different CM processors in the array. The PARIS "get" and "send" operations are ultimately employed which use the general CM router hardware. REF operations are also provided to obtain a single value from a specific processor using one of three addressing methods. The CSHIFT and EOSHIFT functions also perform interprocessor communication but use nearest neighbor grid communication which is much faster than router operations.

The REF communication operations are similar to those provided in *LISP. CSHIFT and EOSHIFT are as defined in Fortran 8x.

3.4 Implementation Concepts

The CM interface consists of several packages as shown below. It was a goal that client software would see one package as the sole interface to the capability. The CM_INTERFACE package is the only one intended to be directly accessed by the client software.



Parallel data types, operators, procedures and functions defined in CM_INTERFACE are expected to be made directly visible via the "use" statement. While the use statement should be considered carefully, it seems appropriate here. Without it, operators would have to be qualified. The use statement would allow the following code to be written:

```
X,Y,Z: INTEGER_PVAR(SEGMENT,EXTENT);
...
ASSIGN(Z, X + Y);
```

The other packages are:

- BASIC_PVAR, which provides the basic pvar types, constants, and global variable definitions;
- GENERIC_ASSIGNMENT_OPS, containing generic ASSIGN procedures to be instantiated for each pvar type;

- GENERIC_REFERENCE_OPS, which defines the generic REF procedures to be instantiated for each pvar type;
- GENERIC_BOOLEAN_OPS, which provides generic procedures and functions for the boolean pvar type;
- GENERIC_INTEGER_OPS, which provides generic procedures and functions to be instantiated for each integer pvar type;
- GENERIC_FLOAT_OPS, which provides generic procedures and functions to be instantiated for each floating point pvar type;
- GENERIC_SHIFT_OPS, containing generic CSHIFT and EOSHIFT functions;
- GENERIC_MISCELLANEOUS_OPS, which provides other generic procedures; and
- PARIS, which is the needed definitional "glue" between Ada and the C-PARIS library.

To prevent client assignment and possible corruption of internal fields it was originally intended that the pvar types provided in CM_INTERFACE be limited private types. However, restrictions on use of private types in generic instantiations and use with discriminants led to public type definitions. Although not the best Ada practice, it is consistent with *LISP. Also, limited private types would have been private to only one package and hence required accessor functions to be used by secondary packages. Since generic routines were forced into secondary packages, most implementation code would have been required to use the accessor functions.

The packages other than CM_INTERFACE, while they can be used by client software, are intended as private implementation detail. The interface organization is the result of restrictions documented in Ada LRM Section 7.4.1 (paragraphs 3 and 4) and apparent constraints on the use of a generic procedure in the same scope as its definition. Rules in LRM Section 7.4.1 also preclude use of private types in instantiation. Segmentation of the interface into many packages clearly makes the implementation easier to maintain but does allow more user access to the implementation even though such access would be explicit.

The structure of the generic routines in the interface is governed by implementation issues. The PARIS library is very regular in its syntax which permits several client operations to be implemented by a single generic routine. This is true for relational and logical operations and for some arithmetic operations such as addition, subtraction and multiplication. PARIS even provides support for mixed parallel and scalar operations for the noncommutative operations of subtraction and division. There are differences in some PARIS operations and the generic routines reflect these differences. For example, PARIS division operations are unique in that they use the zero divide flag. PARIS exponentiation operations have unique operand forms. In the case of division, the Ada procedure controls the additional complexity by using an additional generic parameter. Exponentiation functions use special generic routines.

Generally, there are separate generic interface routines for integer and for floating point operations. Floating point operations in PARIS make use of a separate set of result flags and have two length operands, one for the significand and another for the exponent. Integer operations have only a single length. Depending upon the operation group, integer generic routines may also divide into signed and unsigned versions.

The package specifications developed to implement the interface can be compiled on any validated Ada compiler which supports the pragma INTERFACE for the C programming language. As the PARIS capabilities are essential to providing access to the Connection Machine, support for C language library routines is essential.

4. ADA DESIGN DRIVERS

This section contains a series of discussions documenting the features or limitations of Ada which affected the overall design of the binding packages.

4.1 Extent Sets

The concept of an extent set is intended to map directly onto a CM VP set but the name "extent set" was chosen to be independent of VP sets. This is to provide a generalized capability with possible applicability under other future circumstances. As in *LISP, pvars have the exact size and shape as the extent set in which they are allocated. This approach is not consistent with that used in Fortran 8x where results of some functions such as MATMUL, SPREAD, SUM, and PRODUCT have different shapes than their operands.

The extent set of a pvar is explicitly set as a record discriminant when the variable is allocated. This differs from *LISP where dynamic association of pvars and extent sets occurs. Explicit association under client control seems more appropriate in Ada and is essential to support Ada tasking (see Section 4.2). Because the extent set of a parallel variable is dynamically determined it must be checked at run time.

The interface does not support the concept of a current extent set. The primary impact of this is that some operations, such as

explicit scalar to parallel conversions, require an extent set number to be specified so that the result value can be allocated in the proper extent set. In most other cases the extent set of the result is determined from the extent set of the operands.

4.2 Exceptions and Tasking

Neither current CM software nor the VAX/VMS (the current development host operating system) implementation of Ada tasking permits the use of multiple simultaneous Connection Machine interfaces. The Connection Machine PARIS interface has many implied global parameters which, for example, determine which hardware port number is being used and which is the current extent set. This makes it difficult for independent Ada tasks to perform operations on different Connection Machines or on different portions of the same Connection Machine.

To cope with the situation it is necessary to introduce critical sections into most routines to guarantee that only one task has access to the Connection Machine and its global variables. Since the identity of the currently executing task cannot be determined within the interface/binding packages, the implementation must not assume that any previously established condition such as machine context is still active. This reduces the possible efficiency of the implementation because the needed machine context must be completely established for each operation. Also, handling of temporary variables is likely to be more inefficient than a compiler code generator might be able to achieve. On the other hand, strict validation procedures will probably work against highly vectorizing or parallelizing Ada compilers and handling parallel operations within an Ada tasking framework would still be difficult.

Exception handling and tasking complicate handling of CM storage deallocation because they impede or destroy the last-in-first-out sequencing that is a fundamental part of stack allocation and so force allocation on the Connection Machine heap. Of the two areas, tasking has the most severe impact because of the unpredictable ordering of deallocation. By ensuring that both exception handlers and main line code release the correct amount of Connection Machine space, exceptions can be handled efficiently within the present scheme. To aid in efficient release of storage, each allocated variable is associated by the client with a storage segment number. Segments are allocated and deallocated by client software using support routines. All variables associated with a segment can then be deallocated in a single call.

4.3 Generic Restrictions

The design of the basic pvar type is influenced by Ada language capabilities and restrictions. Because a pvar must record at least a field ID and length, a record structure is essential. Since operators may only have mode "in" parameters, the record must be unchanging. Since there are no generic record type parameters, direct use of records would preclude many generic procedures and functions. This would be a serious drawback as generic procedures and functions significantly compress the amount of coding needed. As a result of these factors, generic pvar parameters are specified as private types. In order to use the record components, interface implementation routines must copy the parameters into basic pvar record temporaries using UNCHECKED_CONVERSION. While it is not the best practice, it would be confined to the implementation packages and use in client software would not be required.

4.4 Lack of Nested Structure Extensibility

A basic difference between Ada and Common LISP as implementation languages lies in the area of blocks and other nested structures. It is simple in Common LISP to extend the structure of the language using the macro features. This is largely because all standard structures of LISP uses the same syntax as is used for both function and macro calls. In contrast, Ada block structure and nested structures such as if-then-else are predefined in the language via special syntax and reserved words. Ada procedures and operators have invocation syntaxes which are distinct from control structures such as if-then-else and case.

The *LISP *LET form, used to obtain local parallel variables, is a typical area of difference between Ada and Common LISP. It is simple in Common LISP to implement *LET using and expanding on existing Common LISP features. Because LISP structures are nested parenthetically, it is straightforward to allocate new variables and to deallocate them once the body of the form has completed its execution. In contrast, while it is possible to arrange for automatic allocation of parallel variables, Ada does not provide sufficient capability to allow for their automatic deallocation. This is important because the programmer will be forced to explicitly specify when deallocation is to take place. This presents an opportunity for errors of two kinds: prematurely deallocating space still needed and failure to deallocate space which is no longer accessible. A guaranteed deallocation procedure such as is present in the language C++ could compensate for the lack of extensibility in block structure.

The interface package uses Fortran-8x style conditionalization to extend the *LISP approach to processor conditionalization.

In *LISP, conditional operations are enclosed in a selection form such as *WHEN, *IF, or IF!!. The corresponding Fortran 8x feature is the WHERE statement. The Fortran WHERE is not nearly as satisfactory as the *LISP conditionalizations because WHERE cannot be nested and the body can contain only array assignment statements. Seemingly as compensation, Fortran-8x array extensions frequently include a mask for selecting elements on which to operate. The Ada interface extends the mask concept by providing two optional masks, one to select elements and the other to exclude elements. Both masks can be used together and a clear and useful meaning is possible. Conditionalized operators are possible but are not implemented because a clumsy function call syntax would be required rather than infix operator syntax.

Each extent has an associated active area which can be set and reset by client software with supplied procedures. This default active area is intended to reduce the need for WHERE and EXCEPT_WHERE masks.

The relational functions EQ, NE, LT, LE, GT, and GE were defined primarily for use in masks (WHERE and EXCEPT_WHERE parameters.) The standard Ada relational operator symbols "=", "/=", "<", "<=", ">", and ">=" could not be consistently used as Ada restricts equality to deliver a result type of Boolean (a scalar value). This seems reasonable in many contexts but not in this one. The Ada requirement appears to stem from the definition of "/=" in terms of "=".

5. CONCLUSIONS

The preliminary design of a Connection Machine interface to be built in the Ada programming language has been completed. The design provides data parallel operations equivalent to operations found in the Connection Machine *LISP programming language and preserves many of the inherent advantages of the Ada language. Design limitations have also been discussed, especially with respect to the influence of restrictions in the present Ada language definition.

Package specifications for the four packages constituting the interface have been written and compiled with the VAX/VMS Ada compiler. Current efforts are underway to implement the bodies of these packages.

ACKNOWLEDGEMENTS

This work is supported under contracts by the U.S. Naval Research Laboratory. The authors would like to thank Henry Dardy and Elizabeth Wald, both of the Connection Machine Facility, for their support of this project. Machine resources of the Connection Machine Facility were used in this work.

REFERENCES

[1] Barnes, J. G. P., *Programming in Ada*, Addison-Wesley, London, Second Edition (1984).

[2] United States Department of Defense, *Reference Manual for the Ada Programming Language*, U. S. Government Printing Office, Washington D.C., (1983).

[3] Thinking Machines Corporation, *PARIS Reference Manual*, Thinking Machines Corporation, Cambridge Mass, (1988).

[4] Thinking Machines Corporation, *The Essential *LISP Manual*, Thinking Machines Corporation, Cambridge Mass, (1986).

[5] Hillis, W. D., *The Connection Machine*, MIT Press, Cambridge, Mass., (1985).

[6] Metcalf, M. & Reid, J., *Fortran 8x Explained*, Oxford University Press, London (1987).

[7] American National Standards Institute, *American National Standard for Information Systems Programming Language Fortran S8 (X3.9-198x) Version 104* (June 1987).



Paul Anderson received his MS degree in Computer Science from the University of Wisconsin. His interests include parallel computation, optimization, and software engineering. Mr. Anderson is a member of the Senior Technical Staff of Planning Research Corporation.



E. K. Park received his PhD in Computer Science from Northwestern University. His research interests include software engineering and Ada, distributed computing, Ada for parallel processing, and fault tolerance. Dr. Park is currently on the faculty of the Computer Science Department at the United States Naval Academy.

The Portable Common Execution Environment (PCEE): An Approach to System Software for Large, Distributed Systems

Charlie Randall
GHG Corporation

David Auty
Softech

Alan Burns
University of Bradford

Charles W. McKay
University of Houston - Clear Lake

Pat Rogers
Software Arts and Sciences

Abstract

The Portable Common Execution Environment (PCEE) project addresses the support of large, complex, distributed computing applications with mission and safety critical components that operate non-stop for an extended life-time. It focuses on the system software, the interface to applications, and the system architecture necessary to reliably build and maintain such systems. The services provided in the PCEE are built from distinct software modules arranged in layers and operating on top of a run-time system. Futhermore, for the types of systems the PCEE addresses, the traditional division of issues into host and target environments is not sufficient. PCEE recognizes a third environment for integration.

1. Introduction

The Portable Common Execution Environment (PCEE) project is a research effort addressing the life cycle support of large, complex, non-stop, distributed computing applications with Mission And Safety Critical (MASC) components. Such applications typically have extended life-time (e.g., 30 years) requirements, such as the NASA Space Station Freedom Program. PCEE focuses on the system software, the interface to applications, and the system architecture necessary to reliably build and maintain such systems. The requirements extend from the target system environment to the integration environment, and ultimately to the host environment. The integration environment serves as the single logical point of software test, integration, deployment, and configuration control whereas system development occurs in the host environment. Life cycle issues include an integrated approach to the technologies (environments, tools, and methodologies) and theoretical foundations (models, principles, and concepts) that span these three environments. The project goals include the advancement of the edge-of-knowledge, the state-of-the-art, the state-of-the-practice, and narrowing the gaps among them.

The scope of the effort is necessarily broad, as it addresses the problems faced in a project such as the Space Station Freedom Program. There are, however, substantial research foundations to support development across the breadth of the project. Furthermore, with a foundation and framework which addresses the broadest scope, areas can be developed with the expectation that they can and will scale down appropriately to improve software engineering in less demanding applications.

At its core, PCEE consists of a set of policies for the management of distributed computing services and resources in a non-stop, secure and safe environment. Its primary aim is to provide a portable interface to a fault tolerant, distributed, realtime set of integrated system software. In doing so it inevitably prescribes certain properties for the underlying software and hardware. It therefore can perhaps best be seen as an interface specification plus the minimal architecture needed to build/implement the interface. It provides a common interface to both hide and support the use of differing instruction set architectures, data bases, data communications systems, bare machine implementations and operating systems without regard to their underlying implementations. Derived from this core are the requirements for support elements including rules for evolution and modification and the integration and host development environments.

Since application software will be written primarily in Ada, the interface that PCEE provides will take the form of Ada package specifications. These library packages extend the view that an application has of the underlying computing environment. Different applications will need different views and therefore a number of distinct packages will be made available. Note that some PCEE interface packages will only be available to authorized host environment tools; for example the basic runtime library may only

be used by appropriate compilers. Other libraries will be used directly by the application code; an example here would be an extended set of runtime features to control, say, scheduling.

It follows from the above point and from the need to partition the software components of the underlying system software (for fault tolerant as well as good engineering reasons) that the services provided in a PCEE are built from distinct software modules arranged in layers and operating on top of a runtime system (RTS). A bare machine approach is taken, i.e., the RTS is supported by a kernel that interacts directly with the hardware. Later sections in this paper will look at each PCEE service starting with a minimal view of the hardware configuration.

The PCEE uses an object oriented paradigm. An object may be an arbitrarily complex piece of data (known as a data object), a subprogram, a package, an individual thread of control, or even a complete program. All objects are seen as instances of some abstraction and therefore all operations that can be applied to any object. type are visible and are declared with the object's abstraction. In essence the PCEE, together with the applications it supports, forms an object management system with distribution supported at the object level. This can accommodate both pre- and post-partitioning approaches, and coarse or fine grain granularity.

The PCEE is *portable* because applications programmers and users can rely upon Ada interfaces designed to facilitate portability. The interfaces do not depend on any specific system, machine, or vendor, but can be used with any system, machine, or vendor product encapsulated under PCEE interfaces. This does not preclude any specific vendor product, machine, or system from being used, it only hides these products from the user and prevents any direct dependency on those products. By placing an interface over these products, their replacement or inclusion is not apparent to the user. This establishes a more flexible or robust system. Software developed for PCEE systems are then also portable and easily moved from one such system to another. The interface also permits a new generation of integrated system software to evolve as replacements for what are now separately developed subsystems which often contain conflicting or repetitious components.

The PCEE is *common* in the sense that within a distributed computing system, the PCEE defines the management of sharable services and resources commonly and throughout the system. While there will be heterogeneous hardware facilities and differences in underlying implementations, PCEE provides the common framework which ensures the desired goals to maintainability, safety, and security. What is not intended by "common" is a life cycle dependency upon any vendor-specific products for the data base management system, data communications system, operating system, or instruction set architecture.

It supports the *execution environment* by establishing policies, providing non-stop operation of sharable services and resources, and supporting a consistent conceptual model for the target environment in which the programs will execute. The integration environment is included in this in so much as it is incorporated into the actual execution of programs, i.e, the monitoring and controlling of programs at runtime. The development host environment is defined only to provide the information base upon which the maintenance and evolution of the system are dependent.

The PCEE effort has two perspectives to consider. In the first perspective, it is viewed as PCEE the project. This view's scope covers the three environments: host, target, and integration. It involves exploring issues for creating, maintaining, testing, and executing programs for large, complex, non-stop distributed systems as they proceed through the three environments. In the second perspective, the PCEE is viewed as the working execution environment. For this view, the PCEE is regarded as an executable environment and everything that entails. Only those safety and portability issues that are relevant to executing programs across different systems are of importance.

This paper addresses both perspectives, often without specifically naming the view. The context in which the term PCEE is mentioned should make the intended perspective obvious.

1.1 Relationship to Other Systems

The PCEE is not an operating system, data communications system, or data base management system, but instead presents an integrated set of virtual interfaces to services and resources provided by the underlying system. Virtual interfaces identify only what services and resources are provided, consumed, or affected. For each of the services and resources, the interface identifies how well, and under what circumstances it must be supported. The PCEE's appearance to applications software and users is basically the same whether it sits on an operating system or the runtime kernel associated with some bare machine system. All implementation details are hidden from the user. The PCEE hides the underlying system software implementations so that a new generation of integrated system software may begin to emerge as a replacement for the

fragmented and poorly integrated components of today.

The Ada package specification supports the goal of a virtual interface directly. The functions, procedures, types, and variables declared in the package specification identify which resources and services are available and how to use them. The implementation details are hidden in the package body and can be changed or ported to another system without affecting the users of the provided services and resources.

Not all of the potential interactions, uses, or implementations involved in a long-lived system are knowable during its initial design. New hardware will probably be added to the system. Better software might replace or enhance the initial software. In any case, these systems will need the ability to take advantage of new technology. To accomplish this, the PCEE interfaces must be extensible to insure the best possible means of meeting the system requirements are available.

Several works have influenced the development of the PCEE approach. These include the Clear Lake Models for Life Cycle Support Environments and for Runtime Support Environments [16, 20, 21], the Ada RunTime Environment Working Group's (ARTEWG) work as documented in their Catalog of Interface Features and Options (CIFO) [1], the MARS project (MAintainable Realtime Systems) [14], the Alpha Kernel (part of the Archons project at Carnegie-Mellon University) [19] and the DIADEM project [2]. For a brief review of these see the PCEE Concept Document [3] or the respective documentation.

PCEE avoids any dependencies upon any proprietary or particular

(1) operating system,

(2) data base management system,

(3) data communications system, or

(4) instruction set architecture.

Many current life cycle environments are unable to meet some or all of the requirements of the distributed systems of concern to the PCEE. By not directly depending on present systems, the ability to add future capabilities is not impaired. With an appropriate virtual interface, the PCEE can be freed of any such dependencies.

1.2 Safety, Security, and Reliability

Safety and security are important aspects of any distributed system (or, for that matter, any system upon which life and property depend). Incorporating and sustaining safety and security in systems is not a simple matter. The system must guard itself against any action, intentional or accidental, that attempts to compromise its integrity. This requires considering safety and security requirements at each point of the system's life cycle. Classes of faults and fault combinations should be identified and prioritized according to their probability of occurrence during execution and the consequences of not properly dealing with them. A safe system is able to monitor its situation and detect faults that enter the system state vectors as soon as possible, firewall their propagation, analyze their effects, and recover safely. For systems upon which life and property depend, piggy-backing on inappropriate models and obsolete technology is too risky. The use of Ada and a well designed runtime support environment provide exceptional assistance towards this goal.

Distributed systems which support a diverse group of users are especially vulnerable to problems which result from improper access to information, processes, and other resources and services. At the minimum, protection is necessary for inadvertent access due to program or operation error. At the other extreme, deliberate disruption must be prevented. The PCEE project seeks to provide security to at least the multi-level security class B3 according to the Trusted Computer System [9].

Large, complex, non-stop, distributed systems with long life expectancies increase the difficulty for instilling reliability. Furthermore, large systems are usually developed by large teams, often spread over a wide area, adding to the management and integration complexity. Distributed systems involve a greater testing, verification, and integration challenge than that associated with uniprocessor systems. Evolving systems with long life expectancy encounter a greater number of requirement and environment changes that in turn increase the overall complexity of a system. Each of these further reduces the possible reliability of these systems.

The use of high level languages, especially Ada, has lead to improvements in overall software reliability. The use of Ada eliminates many types of errors and enhances expressiveness. For example, strong typing and exception handling help eliminate many of the errors that greatly decrease software reliability. Other enhancements such as generics, tasking, and abstract data types improve the expressiveness of the software by reducing the amount of detail that must be specified.

To meet the demands for reliability, several possible solutions have been suggested for these problems. These include fault avoidance, fault elimination, fault tolerance, fault prediction, reuse, formal specification, and formal verification. Only some of these are currently applicable to the PCEE work.

1.3 Characteristics Of Potential Systems

An example program which satisfies the characteristics of systems described for use with the PCEE is the NASA Space Station Freedom Program. The work presented in this paper has used the space station as a focus for the analysis, and as a yardstick for assessing the PCEE designs and concepts. The analysis is not, however, limited to this one application.

It has been estimated that 10 million lines of application code (in Ada) will be needed for the space station. If one also considers systems code and host environment concerns then a figure closer to 100 million is reached. If we focus on the on-board execution environment, the space station has the following pertinent characteristics. It will:

(1) be large and complex,

(2) support non-stop execution,

(3) have a long operational life,

(4) experience evolutionary software changes (i.e., dynamic reconfiguration),

(5) consist of a mixture of safety and mission critical modules (i.e., high reliability requirements including fault tolerance),

(6) have components with hard realtime deadlines,

(7) make use of multiprogramming,

(8) have distributed processing,

(9) possibly contain heterogeneous processors, and

(10) support applications written in Ada.

It follows that the Ada programs must execute in an environment that combines features of redundancy, deadline scheduling, fault tolerance and survivability, network operating systems, network communications, and data bases. The PCEE is an attempt to specify an architecture that is applicable to the space station and similar large projects. Moreover, the PCEE is not just aimed at supporting Ada but also is designed to be implemented in Ada.

2. Proposed Model Of Distribution For PCEE

In proposing a model of distribution for the PCEE the following points were taken into account:

(1) The virtual node concept in Ada is weak; although not precluding its use eventually, it is too early to fix upon a unit of distribution; also its visibility in the design phase may be problematic.

(2) Post-partitioning with a fine granularity gives maximum flexibility for the testbed.

(3) To be able to provide fault tolerance at the application layer some meta-information must be available to this layer.

The model envisages a network of clusters. A cluster is defined, informally, to be one or more processors that share a single runtime support system.

With this view of distribution we can isolate three layers:

(1) network layer - the physical network,

(2) system layers - the runtime support system, other system software systems, and the predefined messages they exchange, and

(3) application layer - application programs.

A post-partitioning approach is taken. To provide the necessary meta-information the application layer will have the following available:

(1) A package that defines a type for CLUSTER_ID.

(2) A prescribed attribute which can be applied to any program object (i.e., task, procedure or variable). It will return the cluster upon which that object currently resides. This attribute will be called 'CLUSTER.

(3) A function that will return the current cluster of the current thread of control:

```
function My_Cluster return Cluster_ID;
```

3. Fault Tolerance

With the three layered model a number of fault modes are possible. These correspond to where the fault is recognized and where the control of error recovery is placed. In this discussion we are not concerned with errors that are diagnosed and dealt with at the network layer. Our primary focus is the

with at the network layer. Our primary focus is the situation where the system layer is required to transparently (to the application) support fault tolerance and survivability. When this is not possible, the next level of support is to inform the application layer of the existence of an error; with the application controlling recovery.

It is first necessary to note those faults that are possible in the kind of system of which the space station is typical. Two generic faults in any distributed system are processor failure and network error (including lost messages, duplicated messages, corrupted messages, and network partitioning). Other generic faults may include loss of: memory components, bus components, system software components, and other sharable system services and resources.

We assume, for this work, that there is sufficient redundancy at the network layer, and known techniques for the systems layer, to enable any network error to be repaired below the application layer. Therefore, to demonstrate fault tolerance and survivability features of PCEE, consider the consequences of processor failure.

There are also system layer techniques available that will isolate a failed processor and attempt to keep faults associated with the failure from entering the system state vector. Such fault avoidance techniques are but one of many approaches which attempt to keep applications consistent. These techniques, though worthy of further study, are rejected here for the following reasons:

(1) The runtime support system will have to be considerably more complex (in size and function) as the techniques needed to keep a consistent state for entire programs across a distributed system are nontrivial and sometimes impossible.

(2) A downgraded system may present performance difficulties to the applications. Rather than downgrade all, load shedding may be preferable. Only the system software can determine which load components, among all of its supported applications, should "survive" and which should be shed.

(3) Because of long life operation it will be desirable to repair or replace the processor (if it cannot be rebooted). In the interim, the system response to the faults associated with this failure may need improvement. This may require human interaction with the system software but through the integration environment.

It is also assumed that there is not sufficient processor redundancy to mask any processor failure.

There are two cases of immediate interest. In the first case, more than one application with MASC components is utilizing the cluster that has just suffered a processor failure. Since a processor is a sharable resource and the MASC components of different applications should have no requirement to know of one another's existence, the coordination and control of recovery cannot and should not be application directed. The second case -- where only one application with MASC components is utilizing the cluster -- presents the case where the semantics of the application provide the best vehicle to direct certain forms of fault tolerance and survivability control and where the MASC components of the single application do not share the cluster with other MASC components of other applications, PCEE will support an externally directed recovery (i.e., from the application).

4. Ada and Dynamic Change Management

Many large realtime systems are expected to have a long, non-stop, operational life. It cannot, however, be assumed that the code that initially started executing will remain appropriate for the complete lifetime of the system. Indeed it will more likely be the case that the system will be subject to evolutionary change. This can reflect redeployment of the original code, bug fixes, or the addition of new functionality. These needs exist for both the application and the PCEE system software.

With distributed systems it is possible to change one part of the system while the rest still executes and remains unchanged. Dynamic alterations are therefore more feasible than in uniprocessor environments.

As a term, "dynamic change management" [5] is not universally accepted. Sloman et al [23] use the phrases "topology flexibility" and "time domain flexibility": the former to identify redeployment (of existing code); the latter to signify the incorporation of new code into a running program.

Some languages defined for distributed computing specifically address dynamic reconfiguration. Conic, for example [23], uses task modules as the unit of distribution and deployment. Tasks can be stopped, new modules loaded and linked, and then started. Module interfaces must not change but the modifications that are allowed are controlled and can take place while the system is executing (i.e., other modules are unaffected).

Ada does not address issues of distribution. Moreover its reference manual [8] embodies a view of binding that is static. To change any component of an Ada program requires the program to be relinked with execution starting at the appropriate place (i.e., elaboration of library units; elaboration of declarative part of main subprogram, etc.).

Once a new component is added it is clear that a new program is created; but in the discussion above it is also clear that the new program did not start its execution at the beginning of the program as discussed in the Ada reference manual. One must conclude that the kind of dynamic flexibility needed in projects such as the space station was not explicitly accommodated in Ada and may be of questionable legality.

5. Approaches to Reliability

As previously stated, reliability is a necessary but not sufficient component for a safe distributed system. With respect to reliability there are several technical questions that must be answered. Specifically, how to build reliability into distributed systems that:

(1) will suffer component failure;

(2) cannot guarantee an ideal communications subsystem;

(3) cannot guarantee the absence of all software errors;

(4) cannot guarantee the absence of errors involving human interactions;

(5) may suffer acts of providence which are not avoidable but which requires survivability strategies to be enacted;

(6) may suffer combinations of the preceding which will severely tax the capabilities of supporting fault tolerance and survivability;

(7) evolves incrementally over a long development period and must be sustained over an indefinite period;

(8) has some components which operate non-stop, and are unattended but allow changes in functionality;

(9) has a requirements mix ranging from hard-constraint realtime applications to data-driven applications.

Large distributed systems simultaneously provide higher and lower reliability; higher in the probability that some part of the system is operational at any time and lower in the probability that some part of the system is not operational at any time. Therefore, system design must not depend upon all components being operational at once. In particular, system software must support the survival of:

(1) processor failures (crashes); and

(2) communications failures.

The basic point is that system software represents the last line of defense in supporting MASC components in non-stop, distributed applications such as the PCEE is intended to address.

An aid to writing such system software is the transaction mechanisms which are based upon sets of software instructions which execute with the effects of atomic actions. Much of our use of nested transactions is built upon earlier work by Moss [18].

One approach to enhance reliability is through redundancy. By having multiple copies of the system, or parts thereof, the system can be made more reliable. Depending on the circumstances, there are two possible approaches to redundancy, passive and active. In passive redundancy, one or more copies of some or all of the system are kept in storage at remote sites. Should the primary copy then suffer a failure, one of the other copies can then be used to carry out the necessary activities. This approach is enhanced by maintaining sets of checkpoints for the executing copy with sufficient state information to permit the backup copy to proceed from a known point of safe recovery. In active redundancy, the copies at other sites are actively running simultaneously with the primary copy. This arrangement facilitates two possible forms of reliability, which can, in turn, be combined. In the first of these approaches, when the primary copy fails, a second copy is available to take over at the point where the primary copy failed. In other words, at the point of failure, system operation is switched from the primary copy to a secondary copy. The second approach also has all of the copies running simultaneously. But in this case, the results of the secondary copy(ies) are used to verify operations carried out by the primary copy.

Another possible approach to reliability is fail/stop processing, i.e., when the system running a process fails, that process stops completely. That assumes that the process no longer has any effect on the rest of the system as a whole. While this assumption does make things easier to handle, it simply does not work for large, complex, distributed systems. In the real world, such processes do not fail "nicely". They often continue to, or at least attempt to, carry out

various operations, sometimes creating highly undesirable situations or conditions.

PCEE assumes just the opposite -- that processes do not fail/stop. Instead, PCEE takes into account that a process might have failed, at least partially, and that the information coming from a remote process might not be correct. Extra steps are taken according to the context of circumstances to insure that any crucial data or information is correct.

6. Computing Environments

For the types of systems the PCEE addresses, i.e., large, complex, non-stop, distributed systems, the traditional division of issues into the host and target environments is not sufficient. PCEE recognizes three types of software environments, each with different functional and operations requirements. These are:

(1) Host: providing for the planning, development, and maintenance of all software products.

(2) Target: providing for the loading and operation of the application system.

(3) Integration: providing for external monitoring and control of the target system, including on-line software and configuration changes.

The goals of PCEE place demands on the architecture of the host, target, and integration environments, particularly at their points of interaction.

6.1 Host Environment

Within the host environment (HE), solutions to process automation and computational problems are proposed, analyzed, developed, and sustained. Each site in the distributed host network utilizes a common framework and set of tools, rules, and components to develop and maintain configuration control over its parts of the overall system. The complexity and magnitude of large software systems demands a sophisticated process and information architecture, this being the principal area of technology development to support PCEE.

The software for the target environment is created in the host environment. Deliverables for the target environment are provided to the integration environment which is then responsible for installation, integration, and operation.

To the maximum extent possible, the host environment will use off-the-shelf tools, rules, procedures, components, and frameworks to support the life cycle of MASC components in PCEE. However, certain restrictions are believed to be essential upon these selections and at least eight needed tools/tool sets are unlikely to emerge as commercial products during the development of the PCEE proof-of-concept prototype.

The restrictions call for a life cycle project object base with fine grained semantic modeling support as the foundation for the framework portion of the host environment. It is hoped that such foundations will be available within the first two years of the project so that appropriate on-line semantic models and object management systems to be deployed in the target and integration environments can be developed and sustained in the host environment. However, there are at least four reasons why the project cannot wait beyond the second year for the availability of a commercial or prototype framework with fine grained semantic modeling support for an object management system (OMS) and library management system (LMS). (The LMS manages both the collections of individual objects and the views in which they appear.)

First is the ease of maintenance, modifications, and extensibility. To the extent that the framework and tools are available in Ada form, project risk management, productivity, and extensibility/modifications will be better supported by the host environments.

Second is the desire to leverage standards for the representation of on-line semantic models such as IRDS (Information Resource Dictionary Standard, ANSI 1988). Such a standard could facilitate the availability of commercially available tools and the transfer of research results.

Third, and most importantly, PCEE's support for MASC components in non-stop, distributed systems is entirely dependent upon the semantic richness to be designed into a new generation of integrated system software. As we learn more about these requirements and their implications in the target environment, we must immediately begin to assess the impact upon the requirements and implications for the integration environment. In turn, the combined impact upon the host environment must be understood. Thus the researchers must be able to make and control changes to the semantic components of objects that exist in different forms across all three environments.

Finally, the research team also wants to leverage and contribute to the growing body of knowledge which extends on-line constraint management for the enforcement of semantic integrity in object management systems. Typically called "knowledge

based systems", the association of predicates, triggers, and demons with constraints on the attributes of objects and their relationships has obvious potential for improving integrity enforcement at runtime in all three environments. For all these reasons, the researchers hope to be able to acquire and use an OMS and LMS appropriate to support the research goals of PCEE.

Eight host environment tools/tool sets are envisioned for the PCEE [3]:

(1) Distributed System Modeling Tool (DSMT)

(2) Partitioning and Allocation Tool (PAT)

(3) Distributed Simulation Tool (DST)

(4) Program Building Tool (PBT)

(5) Semantic Debugging Tool (SDT)

(6) Methodology Support Assistants (MSA)

(7) Quality and Safety Management Assistants (QSMA)

(8) Expert System Aids (ESA)

6.2 Target environment

The target environment (TE) is the support environment for execution of solutions developed in the host environment. Operational software is deployed, operated, and eventually retired here. The target environment encompasses the broadest scope of operational support requirements as it must support the full range of embedded hard realtime processing to multi-user workstation applications. All applications share, however, the need for remote operational and configuration control. The PCEE assumes the demanding requirements of providing a single site image of control over the full spectrum of deployed software.

As will be explained in a later section on software components, the logical architecture of the PCEE software consists of the following major layers and subsystems:

(1) Mission and Safety Critical (MASC) Kernel: This software surrounds the bare machine -- either a von Neumann or an object oriented architecture -- to provide a policy-independent set of mechanisms which support all higher layers and subsystems of PCEE. Currently, twelve sets of components and issues not typically found in today's kernels are included in the MASC kernel requirements. These enhancements are specifically intended to facilitate non-stop distributed support of the collective requirements of MASC components for fault tolerant, survivable, extensible, and realtime operations.

(2) RunTime Library (RTL): These Ada library packages run on top of the MASC kernel to meet the requirements of the Ada standard for runtime support of Ada applications software.

(3) Extended RunTime Library (XRTL): These Ada library packages may run on the MASC kernel or, in some instances, a combination of the MASC kernel and portions of the RTL. These are legal extensions to the minimum requirements of the Ada standard for runtime support of Ada applications software. For any given cluster of a distributed systems the visible (to higher levels of software) interfaces of the XRTL and RTL provide the interface to the Distributed Operating System (DOS). Additional Ada packages are structured into the: Distributed Configuration Control System (DCCS), Distributed Communication System (DCS), and Distributed Information System (DIS). Collectively, these four subsystems -- DOS, DCCS, DCS, and DIS -- provide the PCEE to the programs and components of the Distributed Application System and its users.

6.3 Integration Environment

The integration environment (IE) is added to the above commonly recognized environments in recognition of the unique requirements to monitor and control the configuration and deployment of operational software. Once deployed, operational software forms the deployment baseline. Within the deployment baseline the integration environment normally has no direct function other than to monitor system and subsystems performance. Under emergency conditions, commands from the IE may be used to assist or modify reconfiguration of the target environment resources.

A main function of the integration environment will be to extend the baseline by introducing new application programs which may contain distributed components. This requires four distinct steps:

(1) Verify that the new application software is in an appropriate state to be transferred to the target. This stage is, essentially, an interaction between the host and integration environments. However, the IE may need behavioral measurements from the target as well to accomplish this verification.

(2) Pass to the effected target clusters the appropriate load-module objects. This step includes the loading of redundant copies as required for operational performance and/or fault tolerance.

(3) Load the load modules within the target clusters. Following this step the load map will be updated for all target system objects including shadow copies if they exist.

(4) Assuming the load was successful, start the application via a start message to the appropriate main module.

For terminating programs, the IE will follow a similar procedure to "unload" modules.

Note that some applications will run non-stop (having once started) while others may be invoked periodically (either automatically or under human control). For this latter class there may be some flexibility as to where a new execution (or part of it) is placed. However, all allocations must have been verified by the integration environment.

The other main action of the integration environment is to change or reduce the baseline under various, especially abnormal, conditions. Activities of this nature include:

(1) symbolic debugging,

(2) dynamic reconfiguration of application software,

(3) reconfiguration of the established runtime data or object base,

(4) aborting transactions,

(5) aborting applications,

(6) changing priorities of application tasks,

(7) reducing PCEE services.

Human interaction within target applications occurs in two quite distinct ways:

(1) via an application, or

(2) via the integration environment.

A user interface within an application is the responsibility of the application itself.

Interactions with the integration environment will need to be sanctioned at various levels of privilege. The following (partial) list shows some of the requests a human might make of the integration environment. The ordering here implies increasing privilege:

(1) Obtain monitoring information.

(2) Load and execute a previously verified and deployed version of some application.

(3) Deploy (having first verified) a new application.

(4) Dynamically move persistent objects or reconfigure application or systems code.

(5) Reduce baseline activity because of abnormal conditions.

7. Hardware Architecture

The processing elements of the hardware are assumed to be grouped into clusters; the clusters themselves are linked via local area or even wide area networks (including gateway connections). At the basic level of communication between clusters a reliable service is not assumed.

Clusters do not share memory with each other and have the following properties. Clusters:

(1) can fail completely,

(2) can experience faults that will lead to downgraded operation,

(3) can be added at any time,

(4) can be removed at any time,

(5) can have their software change (dynamically),

(6) have access to memory subsystems including some stable storage,

(7) may control access to system resources,

(8) have the capability to be restarted after failure,

(9) contain processors of the same type, i.e., they run the same instruction code and have identical data representation (different clusters need not use the same processor type).

The stable storage, which is used to construct a fault tolerant distributed information system, will exhibit high reliability but will nevertheless have a small probability of failure, i.e., it cannot be considered to be truly permanent memory. Within the long operational life of the space station this probability

may become a liability and other techniques (such as redundant copies) will need to be provided by the appropriate system component.

Communications support is necessary to support remote procedure calls (RPCs), asynchronous signals, and the Ada rendezvous. Sequential forms of remote communications are best provided through RPCs, which are effected when a called subprogram resides on a remote processor. RPCs must adhere to the rules of Ada with additionally defined failure semantics and behave as if the communicating partners reside on a single processor. Asynchronous signals do not wait for a response, thereby providing communication services without delaying the signalling process. They are necessary when a condition needs to be communicated without the signalling processor relinquishing or slowing execution of its thread of control. For safe and fault tolerant communication of application information requiring synchronization of two threads of control, the Ada rendezvous is essential. Actual implementation is based on the ISO/OSI seven layer model and is extended to include the Ada rendezvous.

Distribution of logical entities within the PCEE should be addressed at the most general level. Distribution across WANs of integrated LANs requires solutions to the problems of unreliable communications and unique identification across networked processors. The PCEE must incorporate transparency of location, replication, parallelism, and fault tolerance. The extent to which the PCEE achieves these goals and the overhead incurred will affect its overall performance.

8. Kernel Approach

Individual applications provide those services and resources which are not shared by other applications. The kernel, on the other hand, provides the mechanisms to support the libraries of services and resources that are shared by applications. Independent applications then depend on the underlying system to provide services and resources that are sharable, while they are responsible for those services and resources they require which are not intended to be sharable.

As processors are not required to be of the same type, a kernel layer is placed on top of the hardware. It hides features that are machine-dependent and provides a rich set of mechanisms to support MASC components in PCEE. This minimal kernel supports the basic runtime libraries system and will typically include functions such as disk I/O services, memory management, process management, runtime error handling, basic interrupt handling, and the primitives for constructing atomic actions.

Process management will consist of a policy free dispatcher. Schedulable objects (typically Ada tasks) that are eligible to execute will be held on an appropriate data structure. The implications being that they are queued in priority order according to context sensitive schedulers that reflect current policies for the integrated management of all sharable services and resources. Measures must be taken to ensure that the kernel and its data structures are fault tolerant.

The full runtime environment is implemented via kernel mechanisms and library modules. It will support functions such as:

(1) dynamic memory management including virtual memory services;

(2) object storage management;

(3) an execution environment for Ada tasking (e.g., task load, create, etc);

(4) scheduling services for realtime control processing.

9. Software Architecture

For all software components it is possible to take three views of their functionality, structure, and availability:

(1) definition of the objects that are exported from the component (together with the operations that can be applied to these objects). For many components the objects may need to be subdivided into classes, with the understanding that not all classes will be available to all "clients" and that under abnormal operation a component may have its functionality restricted.

(2) definition of the subcomponents from which the main component is built. This definition will include the relationships between the objects of the component itself and those of any hidden subcomponent.

(3) definition of the external clients (other PCEE components or application software) that can use (or are using) the objects (or classes) available.

There is a need to formally (or at least precisely) specify these views for each component of the PCEE. One approach would be to make use of semantics models of the EA/RA form.

To the application layer only view (1) is significant. This gives the interface to the component. By contrast the integration environment will need to have access to all three views.

Each view has a static and a dynamic perspective. The static perspective encompasses the complete interface, all subcomponent relations, and all access rights to the interface (for security). The dynamic perspective gives the current usage of the interface and subcomponents as the applications execute.

The integration environment will obtain the static information from the host environment. The dynamic behavior requires the monitoring of the target environment.

The following software components have been recognized as necessary in the PCEE.

9.1 Distributed Applications System (DAS)

The Distributed Applications System (DAS) is the top layer of the software components at which applications usually operate. This layer serves to separate the specific application concerns from those of the underlying components. It addresses the perspective of the applications programmer and user. Standards are defined at this level to ease development and use of applications by the user. Other considerations include the interface to the rest of the software components of the PCEE and which services and resources are available at this level.

There are several different classes to describe the types of potential applications. For each particular class, context sensitive services and resources are available for applications identified by that class. These services and resources are available through the other software components. For a new application to use any component not defined as being available to its class, the activity must be sanctioned by the integration environment.

Some of the different classes to be investigated initially are grouped as follows:

(1) distributed/non-distributed,

(2) dependable/non-dependable, and

(3) realtime/non-realtime.

In all cases, applications have access to the basic runtime library (RTL). If they need distributed capabilities, the services of the DCS and DIS are available through the XRTL. Adding realtime concerns also generates the need for the extended runtime library (XRTL). Finally, applications requiring reliability may make use of the full capabilities available through the entire PCEE.

9.2 Distributed Information System (DIS)

One of the consequences of multiprogramming is that there will be many objects that are shared between applications. The Distributed Information System (DIS) component on each cluster gives access to a distributed object base system which specializes in information services and resources for defined contexts and provides the operators by which these objects may be manipulated.

For a new application, its use of existing DIS objects will first need to be sanctioned by the integration environment. Applications may also be allowed to add to the DIS object base.

To protect persistent objects from failures in particular applications (and from system failures) all operations on DIS objects are implemented as transactions. An application may group these transactions together thereby exploiting nested transactions.

All objects reside within the stable storage of some cluster. Each cluster's DIS knows what objects it has access to. It can also find out which cluster holds any specified object. That is, each cluster knows which libraries have locally stored components and the identification of these components. The cluster also knows how to find the directories for each library in the event a remote object is needed for a local client. The movement of objects is however under DCCS control. If the DIS gets a request (from an application) to access a non-local object then the appropriate (external) DIS will be sent a message via the DCS. This message will take the form of a remote procedure call and is thus itself a transaction. In general objects will remain stationary (stay at the same cluster). Thus, by maintaining audit trails for an application, then after the first access request for each remote object, the DIS will have knowledge of where all objects an application uses are located. Any change to the location of an object must be registered with the integration environment which knows which applications use it and therefore which DISs need to be informed.

An application executing a remote procedure call or remote rendezvous will interact with both the DIS and DCS. The destination of the cluster will be given by the DIS; the actual call being undertaken by the DCS.

To increase reliability an object may be duplicated on more than one cluster. The details of duplication

are however transparent to the application (and to the transaction manager in the RTS). The DIS will give access to one site and will update shadow copies whenever any commit is done to update the main one. Often, read access will be provided to the most convenient available copy. Write access will begin with the primary copy and continue with the shadow copies.

In general an application will manipulate objects that are either:

(1) managed by the DIS, or

(2) held completely within the application space.

An application may use any fault-tolerant technique it wishes to control its own objects (including programming its own shadow copies). It may however make use of the DIS even when the object is not shared. This is especially true if key information associated with the application must persist among multiple iterations of the program or its components. If this is the case, it can take advantage of:

(1) transaction updates, and/or

(2) duplication.

To summarize, the DIS interface must:

(1) Define the objects encapsulating sharable and/or persistent information that are accessible to applications (under access control for security).

(2) Define the operations for manipulating these objects; these will be implemented as transactions.

(3) Allow transactions to be nested by defining top-level transaction "start" and "commit" operations.

(4) Allow new objects (and operators) to be added to the DIS (under the control of the DCCS and integration environments).

(5) Allow objects to be deleted (under appropriate control).

9.3 Distributed Communication System (DCS)

Communications resources and services which are shared among multiple application programs or users in a distributed, on-line environment should be accessible to the application at the Distributed Communication System (DCS) through a virtual interface set. The DCS may also view the DIS as a user and vice versa. For example, a user request from the DAS to the local DIS for a resource of data might result from a request that is transparent to the user to the local DCS to obtain the data resource from a remote site.

All low-level communications appropriate to network traffic are hidden within this component. It provides higher level message abstractions whenever the referent of an executing instruction is determined to be located at one or more remote clusters. In particular the DCS must provide:

(1) asynchronous sends (including broadcasts) and associated buffers (mailboxes),

(2) remote procedure calls, and

(3) remote rendezvous.

The semantics of asynchronous send is send without blocking and receive without acknowledgement. However use of this primitive is limited as the receipt of any such message cannot be assumed by the sender within a predefined time period.

To implement the latter two it will be necessary for the DCS to create surrogates in the RTS of the designated cluster.

A remote procedure call has many of the desirable attributes of an atomic action; it is therefore defined in PCEE to have exactly-once semantics. This usually means that the call also specifies the number of times it has been sent in addition to the call message. Thus a receiving site that creates an agent to locally represent the caller may unsuccessfully send such a reply status. The caller, not receiving the acknowledgement, repeats the call but with a new value for the number of times sent. The called site realizes this is the next transmission of a previously acted upon call and refuses to create an orphan agent. Instead, it continues to acknowledge the original message. Within a realtime, fault tolerant context this definition has to be weakened somewhat. By use of stable storage and a commit protocol it is possible to ensure that a called procedure's execution which was never acknowledged by the caller (as indicated by incremental repetitions of call message) will leave the distributed (persistent) information system unchanged (see discussion on DIS). However, an external event may have been triggered by a partial execution of the procedure (prior to cluster failure for example). The required semantics for remote procedure call therefore cover the following three cases:

(1) procedure executes exactly once with any changes to persistent objects being committed;

(2) procedure's execution did not begin; an exception being raised (say, PROCEDURE_START_ERROR);

(3) procedure's execution did not complete. No persistent objects have been committed; a distinct exception being raised (say, PROCEDURE_TERMINATION_ERROR).

As a remote rendezvous will effect the thread of control of another (remote) object, the only semantic information returned to the caller will be either that the rendezvous was successful or:

(1) it failed because of language defined semantics (i.e., TASKING_ERROR or other application defined exception); or

(2) it failed because of a failed cluster (i.e., CLUSTER_ERROR).

Note that there is an important difference between the raising of TASKING_ERROR and CLUSTER_ERROR. If a rendezvous request generated TASKING_ERROR then all subsequent calls must produce the same effect. This is not the case with CLUSTER_ERROR; a second rendezvous attempt may indeed succeed.

The necessary mechanisms to implement a remote rendezvous are discussed by Burns, Lister and Wellings [4]. Experimental work with remote rendezvous has been undertaken by Honeywell [6] and as part of the DIADEM projects [2].

The interface that DCS provides has two important classes. The actual objects for making use of remote procedure calls and remote rendezvous are made available to the application software via tools in the host environment. However the libraries that define the necessary exceptions may be made directly available to the application.

Within the PCEE, other logical messages will need to be constructed and communicated (e.g., remote elaboration, two-phase commit, etc.). These will be made available at a layer above the DCS and will use DCS facilities for resolution.

Within the DCS consideration must be taken of the processor types upon which the two partners in the communication are executing. If the processors are of the same type (i.e., they have the same data representation) then no internal data transformations are necessary or desirable. However it is possible that heterogeneous processors are involved; in this case a transformation protocol is needed. The protocol must be extensible as new processor types may be added to a running system.

9.4 Distributed Configuration Control System (DCCS)

The purpose of the Distributed Configuration Control System (DCCS) is to provide configuration control of execution environment services and resources based upon semantic models of the stable interface sets. The DCCS virtual interface set has visibility of the DAS, DIS, DCS, and DOS. This visibility provides a unique opportunity to exploit known semantics about the various components that provide services and resources of the three distributed services to monitor, manipulate, and control distributed processing:

(1) programs can be distributed dynamically,

(2) processes can be advanced or blocked,

(3) parameterized performance monitoring can be enabled or disabled, and

(4) interactive debugging and reconfiguration can be supported among remote sites.

In addition, the DCCS offers a unique opportunity to provide constraint management for the other four systems. For key states and state transformations of MASC components in any of the four, constraints may be specified along with predicates and assertions for responses.

This a much higher semantic level of configuration-control service than is typically found in underlying operating systems.

The primary function of the DCCS is to be the window on the target environment for the integration environment. Under normal operation the DCCS will monitor the behavior of the cluster and report this (using the DCS) to where the functionality of the integration environment resides. Other normal facilities that it will control or monitor are:

(1) The introduction of a new object to a cluster.

(2) The movement of an object from one cluster to another (or the simple removal of an object).

(3) The loading of a subprogram (or task) object (load module) for subsequent usage.

(4) The start-up of a program.

(5) The unloading of a program object (load module).

(6) The introduction of new RTS objects.

(7) The dynamic replacement of a program object by an equivalent (extended or modified) object.

Under abnormal conditions the integration environment may need to reduce activity in order to deal with a potentially hazardous situation. In this eventuality the DCCS may be called upon to:

(1) abort a program object (or atomic action),

(2) freeze a program object (i.e., give it no further processor time until the hazard has been dealt with), or

(3) reduce the availability of software components by limiting access to their interfaces.

9.5 Distributed Operating System (DOS)

The design of the Distributed Operating System (DOS) for the PCEE is influenced by the following:

(1) an explicit set of policies for managing the integrated operation of all categories of system services and resources that are to be sharable among independent application program components and users;

(2) an explicit set of management modules (software, hardware, and operational interfaces) to implement and enforce policies; and

(3) a precise model of the operating system which provides rules and guidelines for modifications including extensions, regressions, and reconfigurations.

The list of functions supported by the full runtime environment is given in the section discussing the kernel. The major interfaces by which the software system interacts with the kernel are the DOS. They can be identified as the:

(1) Standard runtime library (RTL) available to the Ada compiler and to other components of the PCEE.

(2) Extended runtime library (XRTL) available to the applications programmer for realtime computations (i.e., ARTEWG CIFO). Provisions for deadline scheduling are included.

(3) Atomic action (transaction) system interface.

All RTL services (e.g., high level, context-sensitive scheduler; transaction manager; etc) are provided as schedulable objects for the kernel; they will typically have higher priorities than application tasks and may additionally have kernel level privileges. This approach allows the kernel to be policy free, with the policy dependent modules running as "tasks" above the mechanisms of the MASC kernel.

One consequence of this architecture is that the functionality of the RTS can be changed during execution. To this end the highest priority "task" at any cluster will be one that has the capability to effect other RTS or application tasks. This configuration task can only be called by the DCCS or, in certain contexts, by the integration environment via a command interface.

10. Conclusions

The progress of the PCEE is checked by developing and refining appropriate conceptual models. Once these models are accepted, they are mapped to a fully instrumented, highly reconfigurable testbed which supports

(1) rapid prototyping and simulation;

(2) proof-of-concept demonstrations;

(3) empirical evaluation of design alternatives; and

(4) traceable causes and effects across all three environments.

The hardware of the PCEE testbed focuses primarily on the target environment. To demonstrate specific aspects of the PCEE, a scale-downed approach to the conceptual model is used to address a wide area network of local area networks ("WAN of LANs") in each environment where the LANs may contain clusters of parallel processing resources. The typical configuration envisioned consists of 3 LANs with 3 computing clusters per LAN, and 3 processors and 2 stable storage devices per cluster. We have selected three cluster types based on:

(1) 80386 microprocessor chip with MultiBus II

(2) 68030 microprocessor chip with VME Bus

(3) 960 microprocessor chip with BiiN proprietary bus.

Currently, we have only the first of these in place.

In summary, the motivation behind the design of the PCEE is the life cycle support of non-stop, distributed, mission and safety critical (MASC)

systems. To this end the following features are significant:

(1) An extended runtime library for realtime computation.

(2) Redundancy at the kernel level so that non-fatal processor failures can be trapped and corrected.

(3) An interface to stable storage at the kernel level, so that transaction commit protocols can be implemented.

(4) An information system that supports nested transactions and, separately, object replication.

(5) A communication system that hides communication failures (but will notify users of processor failure).

(6) An integration environment that controls changes to the baseline.

(7) An integration environment that can abort (or reduce) functionality so that critical systems can be executed more effectively.

Bibliography

1. ARTEWG (ACM SIGAda Ada RunTime Environment Working Group), A Catalog of Interface Features and Options for the Ada Runtime Environment, ARTEWG Interfaces Subgroup 3, Release 2, December 1987.

2. Atkinson, C., T. Moreton and A. Natali; Ada for Distributed Systems; Ada Companion Series, Cambridge University Press, 1988.

3. Auty, D., A. Burns, C. McKay, C. Randall, and P. Rogers, PCEE Concept Document, RICIS SE.16, University of Houston - Clear Lake, Nov. 1989.

4. Burns, A., A. Lister, and A. Wellings, A Review of Ada Tasking, Springer-Verlag, Heidelberg, 1987.

5. Burns, A., A. Wellings, "Dynamic Change Management and Ada", Journal of Software Maintenance (to be published) 1989.

6. Cornhill, D. "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets", Proceedings of the IEEE Computer Science Conference on Ada Applications and Environments, pp. 153-162, 1984.

7. Cornhill, D. "A Survivable Distributed Computing System for Embedded Applications Written in Ada", Ada Letters, Vol. 3, No. 3, 1983.

8. DOD, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, 1983.

9. DOD, Trusted Computer System Evaluation Criteria, DOD 5200.28-STD, 1985.

10. Enslow, P. "What is a 'Distributed' Data Processing System?", Computer, Vol. 11, No. 7, January, 1978.

11. Hutcheon A. D. and Wellings A. J., "Ada for Distributed Systems", Computer Standards and Interfaces, Vol. 6, No. 1, pp. 71-82, 1987.

12. Kamrad, M., R. Jha, G. Eisenhauer, and D. Cornhill, "Distributed Ada", Ada Letters, Vol. 7, No. 6, pp. 113-115, 1987.

13. Knight, J. and J. Urquhart, "On the Implementation and Use of Ada on Fault-Tolerant Distributed Systems", IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, May 1987.

14. Kopetz, H., A. Damm, C. Koza, M. Mulazzani, W. Schwabi, C. Senft and R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach," IEEE Micro, pp. 25-39, February 1989.

15. Liskov, B., and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3 (July), pp. 381-404, 1983.

16. McKay, C., D. Auty, and R. Charette, "A Study to Identify Tools Needed to Extend the Minimal Toolset of the Ada Programming Support Environment (MAPSE) to Support the Life Cycle of Large, Complex, Non-Stop, Distributed Systems Such as the Space Station Program", NAS9-17010, 1986.

17. McKay, C., D. Auty, and K. Rogers, "A Study of System Interface Sets (SIS) For the Host, Target, and Integration Environments of the Space Station Program (SSP)", SERC (UHCL) Report SE.10, NCC9-16, 1987.

18. Moss, J. E. B., "Nested Transactions: An Approach To Reliable Distributed Computing", MIT/LCS/TR 260, April 1981.

19. Northcutt, J., Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel, Academic Press, Boston, 1987.

20. Randall, C., and P. Rogers, "The Clear Lake Model for Distributed Systems in Ada", SERC (UHCL) SE.9, NCC9-16, 1987.

21. Randall, C., P. Rogers, and C. McKay, "Distributed Ada: Extending the Runtime Environment for the Space Station Program", Sixth National Conference on Ada Technology, pp. 134 - 142, March, 1988.

22. Rogers, P. and C. McKay. "Distributed Program Entities in Ada", First International Conference on Ada Programming Language Applications for the NASA Space Station, pp. B.3.4.1 - B.3.4.13, 1986.

23.Sloman M., Magee J. and Kramer J., "Building Flexible Distributed Computing Systems in Conic", Distributed Computing Systems Programme, ed Duce D.A., Peter Peregrinus Ltd., IEE, pp. 86-106, 1984.

24. Wellings, A. J. and Hutcheon A. D., "The Virtual Node Approach to Designing Distributed Ada Programs", Ada User, 1988.

Authors

Charlie Randall is a senior systems engineer with GHG Corporation. His research interests include distributed systems, concurrent programming, realtime systems, and Ada. He is past vice-chair of the Clear Lake Area SIGAda group. He received his B.S. in Physics from Louisiana State University in 1980, a M.S. in Physical Science from the University of Houston-Clear Lake in 1983, and is currently working on a M.S. in Computer Science at the University of Houston-Clear Lake.

David Auty is the Houston Operations Technical Coordinator for Softech. His interests include system analysis and design of realtime, distributed systems in Ada. He received his B.S. in Computer Science from Brown University in 1979.

Alan Burns is a Senior Lecturer in Computer Science at the University of Bradford. Beginning January 1990, he will be a Reader in the Computer Science Department at the University of York, UK. He gained D.Phil in 1978 following a First in Mathematics in 1974. Research interests include Real-Time Systems and their Programming Languages; author/co-author of 6 book and over 70 technical papers.

Dr. Charles McKay is Director of the Software Engineering Research Center and the High Technologies Lab at the University of Houston-Clear Lake where he is also a full professor of Computer Science. He has over 20 years of consulting, research, and development experience in industry, government, and academia. He is the author of three textbooks and numerous papers and reports. He is currently the chairman of the Interfaces Subgroup of ARTEWG.

Patrick (Pat) Rogers is a consultant through his firm Software Arts and Sciences and a Principle Member of the Ada Runtime Environment Working Group (ARTEWG). He is the past Chair of the Clear Lake SIGAda group. He holds a B.S. and M.S. in Computer Science from the University of Houston at Clear Lake. His current research interests include distributed systems, software engineering, realtime systems and Ada.

Ada COMPOSITE BENCHMARK FOR INTELLIGENCE / ELECTRONIC WARFARE SYSTEMS

Arvind Goel
Unixpros Inc.
16 Birch Lane
Colts Neck, NJ 07722

and

Mary E. Bender
U.S. Army CECOM
Center for Software Engineering
Ft. Monmouth, NJ 07703

ABSTRACT

This paper describes the process of developing an Ada composite benchmark to model a class of real-time applications. A composite benchmark for a system looks at the interaction between features of the system rather than just the performance of individual features. The class chosen to model was the Communication/Electronic Intelligence (COMINT/ELINT) systems of Intelligence/Electronic Warfare (IEW) used by the U.S. Army. Although the benchmark was developed for COMINT/ELINT the process can be easily extended to other classes of applications. The benchmark development process involves: determining the characteristics of a system, programming representative parts and algorithms of the system in Ada, and then running the benchmark on various host/target configurations.

1.0 INTRODUCTION

A goal of Ada was to provide a language that supports modern software engineering principles in the design and development of both real-time and non real-time systems. Many of the current Ada implementations do not allow the development of reliable embedded systems software without sacrificing productivity and/or quality. With the growing list of validated Ada compilers, software developers must be able to select an Ada compiler and its runtime environment so that timing and storage requirements of real-time embedded systems are met and efficient, quality systems are produced.

To determine the suitability of Ada compiler systems for embedded applications, a benchmarking effort was undertaken by the Center for Software Engineering (CSE), U.S. Army CECOM, Ft. Monmouth, NJ. The first step involved the identification of Ada features of interest for real-time embedded systems. By analyzing the real-time characteristics of the COMINT/ELINT class of IEW systems, the description of a composite benchmark for these systems was produced. [1]

The next step involved the development and execution of real-time benchmarks to measure the identified features. Benchmarks were developed that measure the performance of Ada individual features, determine Ada runtime system implementation dependencies, and program the real-time algorithms found in real-time systems. [2] The final step involves the development of the composite benchmark based on the initial description. [1] This paper describes the effort involved in developing this composite benchmark for the COMINT/ELINT class of IEW systems.

The organization of the paper is as follows. Section 2.0 describes the procedure for evaluating Ada compilers using runtime performance benchmarks. Section 3.0 discusses the concept of composite benchmarks. Their purpose and relationship to an overall Ada benchmarking process is addressed and the process of developing them is described. Section 4.0 describes the capabilities of the COMINT/ELINT systems and how they have been implemented in the sample composite benchmark. Section 5.0 describes the testbed hardware and software. It also gives more implementation details and some preliminary results obtained during the execution of the benchmark. Finally Section 6.0 describes some of the lessons learned.

2.0 Ada COMPILER EVALUATION USING BENCHMARKS

To design and implement systems in Ada, it

is essential that the performance and implementation characteristics of an Ada compiler system meet the requirements of the system being designed. Technical criteria for evaluating Ada compilers include: compiler operation, compiler/linker features, development tools, documentation, library options, program/code management, optional (third party) software components, compile time performance, Ada Runtime System (RTS) implementation dependencies, and runtime performance of Ada code. For programming IEW systems, the most important criteria for selecting Ada compilers is the runtime performance of Ada code as well as the implementation dependencies of the Ada RTS. To measure the performance of various Ada language features and implementation dependencies of the RTS, benchmarks are run on the Ada compilation system in question and the results analyzed.

The use made of performance measurements depends on one's purpose in conducting benchmarks. The purpose may be to compare implementations for general maturity or to determine which is most suitable for a particular application. In the first case, evaluation of benchmarks will be implicitly based on assumptions about which performance characteristics are indicative of maturity. In the second case, the performance characteristics deemed most important will depend on the requirements of the application. An implementation with a low score for a particular performance characteristic may still be most appropriate for a given application.

Ada benchmarking can be approached in four ways:

a. design benchmarks to measure execution speed of individual features of the language;

b. design benchmarks that determine implementation dependent attributes (this and the previous approach are referred to as microscopic benchmarks);

c. design benchmarks that measure the performance of commonly used real-time Ada paradigms; and

d. design composite benchmarks which include representative code from real-time applications.

The ultimate goal of any benchmarking effort would be to have sets of benchmarks, where each set evaluates an Ada compiler for a particular class of real-time applications. Each set should include microscopic as well as composite benchmarks. Performance characteristics considered important for a real-time embedded application depend on the requirements of the application. For a class of real-time embedded applications, it is possible to define the distinguishing characteristics of such a system. Specific microscopic benchmarks that measure those Ada features can then be extracted from the general set to form a set that evaluates a compiler system for that particular class of real-time systems. [1]

The first step in selecting an Ada compiler is to run the composite benchmark for that class of applications on a set of Ada compilers. Based on the results of the composite benchmark, the remaining Ada compilers are then evaluated using microscopic benchmarks. Real-time paradigms can then be run on Ada compiler implementations to determine the efficiency of various real-time paradigms that are important for an application.

2.1 PERFORMANCE OF INDIVIDUAL FEATURES

This approach measures the execution speed of individual features of the language and RTS by isolating the feature to be measured to the greatest extent possible. Such benchmarks are useful in understanding the efficiency of a specific feature of an Ada implementation. For example, a benchmark that measures the time for a simple rendezvous can be run on two Ada compiler systems. The advantage of such an approach is performance evaluation without bias towards any application. The problems with such an approach include determining and isolating the features of the language and RTS that are important for real-time system applications. Also, this approach requires a significant number of tests and the numbers produced have to be statistically evaluated to determine general performance.

2.2 RUNTIME SYSTEM IMPLEMENTATION DEPENDENCIES

These benchmarks are concerned primarily with determining the implementation characteristics of an Ada RTS. The scheduling algorithm, storage allocation/deallocation algorithm, and priority of rendezvous between two tasks without explicit priorities are some of the many implementation dependent characteristics that need to be known to determine if a compiler system is suitable for a particular real-time application. A major effort in such benchmarks involves interpreting the results obtained by running the benchmarks and drawing the correct conclusions. A detailed description has been provided to help interpret the results. [2]

2.3 REAL-TIME PARADIGMS

This approach involves programming algorithms typically found in embedded systems. For example, a scenario in real-time systems might be a producer that monitors a sensor and produces output asynchronously and sends it to a consumer. The producer task cannot wait for a rendezvous with the consumer (who might be doing something else) as the producer task might miss a sensor reading. To program this paradigm in Ada requires three tasks: a producer task, a buffer task that receives input from the producer task, and a consumer task that receives input sent to it by the buffer task.

Macro constructs are defined as a set of Ada statements that perform a well defined process e.g. semaphores, mailbox, etc. For real-time systems, real-time paradigms can be identified and programmed in Ada using macro constructs. These benchmarks can be run on Ada compiler implementations and statistics gathered on their performance.

2.4 COMPOSITE BENCHMARKS

A composite benchmark is defined as a program, within the context of the application domain, that looks at the interaction between Ada features rather than the performance of individual features themselves. It involves the use of typical code segments from a given application collected into a program whose overall performance is measured. An example of employing this approach is the Ada Avionics Test Program Package developed by SofTech Inc. for the Air Force. The AATPP is a collection of Ada programs which are representative of real-time avionics application software. These Ada programs provide general information concerning an Ada compiler system's real-time processing performance. The purpose of running a composite benchmark is to aid in the selection of a suitable compiler and runtime for a particular application.

The advantage of the composite benchmark approach is, for a given application domain, running this benchmark on different compiler implementations enables a straightforward selection. Also there are no complex combinations of feature-by-feature evaluations to consider, and no surprises stemming from an unenlightened evaluation. The difficulty with composite assessments is their narrow scope of usefulness and their bias towards the domain of applications from which the benchmarking code was selected.

This paper focuses on designing composite benchmarks for real-time systems, specifically the COMINT/ELINT class of IEW systems used by the U.S. Army. The next section describes the steps followed for developing a composite benchmark for these systems. This particular method of developing a composite benchmark can also be extended to any class of real-time systems.

3.0 HOW TO DEVELOP A COMPOSITE BENCHMARK

Developing a composite benchmark description for a particular class of systems is not a simple task, but once developed the benchmark could be used to aid in the selection of Ada compilers for any system in the given class. The following three steps are followed when developing composite benchmarks:

a. identify the common capabilities of the particular class of systems by studying the requirements and functions of the systems within the class. Particular emphasis should be placed on the real-time characteristics of the class of systems being analyzed;

b. define and analyze each capability. The description should include all functions common to the systems in the class. If a particular function is common to several of the systems within the class, it should be included because the function may be performed in a new system being developed. Define the interactions and interfaces among the capabilities; and

c. develop Ada package specifications and code.

3.1 IDENTIFICATION OF COMMON CAPABILITIES

The first and foremost step is to identify the common capabilities of the class of real-time systems for which the benchmark is being designed. In order to do this, it is important to define the characteristics of real-time systems.

A real-time system constantly monitors, analyzes and responds to external real world events in a time critical fashion. Real-time computer applications are characterized by interaction with real-world (physical) events, over whose timing they have little or no control. It is therefore a requirement of the software that it control its own execution timing so as to synchronize with the real world. If the

timing of the software does not meet certain constraints, the entire system of which the software is a part may fail. Besides having timing constraints, real-time systems typically have a number of important secondary characteristics derived from their functions of monitoring and controlling physical processes. These characteristics include concurrency, high reliability and fault tolerance, resource constraints (processor time, memory, power, and weight) and low-level interactions between the software and application-specific hardware devices.

The high-level functions performed by real-time systems are as follows:

a. monitor - connection of a physical event to a computer system so that data pertaining to the physical event can be collected;

b. analyze - portion of software that determines the next course of action based on the data collected;

c. respond - portion of software that executes the course of action determined from the analysis.

The capabilities of the COMINT/ELINT class of IEW systems are described in Section 4.0. The relationship of these capabilities to the real-time functions are:

a. intercept capability performs the monitor function;

b. direction finding, emitter location, and analysis capabilities perform the analyze function; and

c. reporting performs the respond function.

3.2 DEFINE AND ANALYZE EACH CAPABILITY

The next step involves the definition and thorough analysis of each capability. The major goal in this step is to understand the high-level as well as low-level functionality of each capability and to produce a description of the functional and performance requirements of each capability. This step consists of two phases: analysis and design. The analysis phase deals with the "what" of system development. At this stage the high level requirements for each capability are identified. When the initial analysis phase is completed, what the system must do is known in sufficient detail to start identifying the program modules and start specifying the data that will accomplish the tasks and subtasks. In the design phase, each system capability is further broken down into more specific details. They are analyzed with respect to lower-level characteristics of real-time systems. The objective is to break down the high level description of the major capabilities into details like periodic tasks, aperiodic tasks, throughput, timing and memory constraints, and input data required. This step also involves determining the interactions and interfaces among the different subsystems.

3.3 DEVELOP Ada PACKAGE SPECIFICATIONS

The last step involves:

a. developing the specification of each program module from which coding can be started. This includes the identification of the Ada constructs that will be utilized in the program; and

b. determining the output that will be produced by the benchmark as well as the performance measurement of typical Ada language features such as tasking, exception handling, and rendezvous during different stages of the program.

At this point, the development of detailed Ada code can begin.

4.0 DESCRIPTION OF THE COMPOSITE BENCHMARK

This section describes the five capabilities of COMINT/ELINT class of IEW systems - intercept, direction finding, emitter location, analysis, and reporting. It also describes how the capabilities have been implemented in Ada as a composite benchmark.

The interaction of the five functions of this class of systems is described as follows. The intercept capability monitors frequencies to detect signals. Upon detection of activity on a frequency, the intercept capability calls upon the direction finding capability to determine the direction of the signal detected. When the direction of the signal has been determined, the direction finding capability calls upon the emitter location capability to compute the location of the detected signal. When the location has been estimated, the signal analysis capability determines the type of the signal that has been detected by the intercept capability and passes the signal type to the intercept capability. This information is then relayed in the form of a report to field commanders to enable them to take appropriate action.

The basic design philosophy applied was the use of Ada tasking and rendezvous to control the execution of the software. This enabled the Ada code

to execute for the most part asynchronously, with synchronization applied only when necessary. The software demonstrates a typical Ada-style usage of the Ada language features of multitasking, rendezvous, generics, exception handling, separate compilation, and packaging. The software has been designed to capture real-time processing in terms of computational speed, task rendezvous speed, generic code execution speed, and exception processing speed. Additional compiler performance data such as program load module size are determined for different compiler systems.

4.1 INTERCEPT IMPLEMENTATION

This capability performs automatic acquisition of unknown signals. It searches frequency bands to find and catalog unknown signals. It involves two different search capabilities: general search and directed search. The intercept capability consists of two library level Ada tasks that perform the general search and directed search capabilities. These two tasks execute periodically and their function is to search frequency bands for activity. There are two intercept manager tasks that are designed to control the execution of the two intercept capability tasks. The intercept manager tasks performs rendezvous with the two intercept capability tasks that signal the presence of activity at a particular frequency. The rendezvous frequency can be increased which increases the number of detected signals. Each intercept capability task is designed to perform data input, processing, and output within an infinite loop control structure as shown in Figure 1.

```
task body //task name// is
        // local declarations//
begin
        loop
                //perform data input//
                //perform data processing//
                //perform data output//
        end loop;
end;
```

Figure 1

4.1.1 GENERAL SEARCH IMPLEMENTATION

General Search (GS) is a broad based sampling of frequency activity. It monitors a number of frequency bands for emitter activity and reports the occurrence of detected signals. This involves automatic environment mapping within selected frequency bands with associated geographic areas of interest and selected signal types. GS has been implemented as a periodic Ada task (GS_TASK) which searches frequency bands for activity. The task steps through the defined frequencies at a rate of at least 50 frequencies per second (this rate can be increased). Some frequencies are excluded from being monitored. The benchmark specifies the frequency bands, exclusion of frequencies, and signal class/types to be detected.

The GS task is supplied with the following data: start frequency, stop frequency, frequency step size, receiver bandwidth size, and signal class/type. The data also includes exclusion frequencies that are specified to inhibit the reporting of signal activity at specified frequencies.

The task maintains an activity table that contains entries for the most recent GS and the manual direction finding that the system performed. It also contains the time for first and last intercept and a location estimate for each entry for which direction findings have been taken (Direction Finding is described later in this paper).

4.1.2 DIRECTED SEARCH IMPLEMENTATION

Directed Search (DS) is the automatic intercept of specific known signals. It involves automatic environment sampling at discrete frequencies, and it revisits known emitters. The DS operation monitors a list of individual frequencies specified in a DS plan and reports newly active signals. The DS capability is also implemented as a periodic Ada task (DS_TASK) which searches only a few frequencies for known signals. An activity table is maintained for each specified frequency. The table includes an activity counter for each signal detected and the time of the last intercept.

Each specified frequency has an associated priority assigned to it that is used to guarantee minimum revisit intervals. The software determines the best DS frequency to be examined while taking into account the relative priorities of the frequencies. The software steps through the frequencies in the DS plan at a minimum rate of 50 entries per second.

Monitor DS (highest priority) entries have a revisit rate at least 5 times the rate for priority DS entries. Normal DS (lowest priority) entries have a revisit time interval as determined by the number of entries in the DS plan. The data needed by this task includes the priority associated with the frequencies being monitored and the number of automatic Direction Finding requests to be made for each frequency. The DS plan contains at least 125 entries, including 20 priority DS entries and two monitor DS entries.

4.2 DIRECTION FINDING IMPLEMENTATION

Direction Finding (DF) capability is implemented as an Ada task (DF_TASK). When the intercept capability detects signals, DF requests are made. DS_TASK or GS_TASK does a rendezvous with DF_TASK that requests it to perform the direction finding for the signal detected. The DF task then makes various measurements that provide an indication of the direction from which a frequency signal originated. The measurement process consists of several sequentially executed tasks that conclude with the generation of a Line of Position (LOP). The DF task accepts input data from the DF related equipment and the magnetic field converter via an analog-to digital converter.

An LOP consists of two pieces of data: a Line of Bearing (LOB) and the location of the receiving measurement equipment. An LOB is a line drawn from the measurement platform location at the angle (relative to north) that a signal arrived. When a LOB becomes referenced to a position (platform location at the time of the bearing), it becomes an LOP.

The DF task performs direction finding according to the following priority: monitor DS, priority DS, normal DS, and finally GS. A local queue of pending DF requests is maintained. The DF task also computes and formats an LOP message for a given DF request within 2.25 seconds of receipt of the request.

4.3 EMITTER LOCATION IMPLEMENTATION

The Emitter Location (EL) capability computes the location of an emitter signal. DF_TASK does a rendezvous with the EL task (EL_TASK) after computing LOPs. The EL task uses the LOPs to compute the best estimate of the emitter location. Upon receipt of LOPs from a DF request the EL task attempts to associate the LOP set with existing locations in a file. If an association is found, the LOP set is assigned to the corresponding location; otherwise, the task attempts to generate an emitter location estimate. If a reasonable emitter location estimate cannot be determined, the LOP set remains unassigned. The benchmark is capable of computing a location from five LOPs within 300 milliseconds.

4.4 ANALYSIS IMPLEMENTATION

This capability determines the type of the signal that has been detected by the intercept task. The EL task calls an entry in the analysis task which then determines the type of the signal detected. Signal analysis data results (detected frequency and signal type or modulation) are then displayed. The signal classification task processes signal classification requests at a sustained rate of up to 20 per second without losing data.

4.5 REPORTING IMPLEMENTATION

The reporting capability generates a full report of the final data that has been gathered for a intercepted signal. The data specification parameters allow the operator to view DF data associated with a single frequency or a frequency range, a specific time span within which the data was collected, or a specific geographical area. These data specification parameters are included in any combination.

5.0 STATE OF THE BENCHMARK

At the time of writing this paper, code is being developed for the benchmark. Coding for the intercept, direction finding, emitter location, and analysis capabilities is nearly complete and some preliminary results have been obtained. The code is being developed on a Verdix Ada compiler system that is hosted on a SUN 3/60 and targeted to the Motorola 68020 bare machine.

5.1 TESTBED HARDWARE AND SOFTWARE

The setup used for benchmarking can be summarized as follows:

Host - Sun 3/60, running Sun Unix 4.2 Release 3.5

Compiler - Verdix Ada Development System targeted to Motorola MC68020 targets, release 5.41

Target: - GPC68020 (based on Motorola MC68020 microprocessor) multibus-compatible computer board having 12.5 Mhz MC68020 microprocessor, a MC68881 floating point co-processor, and 2 megabyte of RAM.

The GPC68020 has two serial lines with one RS232 line and one line which may be configured as either RS232 or RS422. The second serial port was connected to the SUN 3/60 serial port and used for downloading object code to the GPC68020 computer. The code is compiled on the SUN 3/60 workstation and then downloaded to the bare target and executed via tools available from the compiler vendor. The Verdix compiler cross-compilation system contains tools for compiling, linking, downloading, and executing target programs. There is also a cross-debugger that enabled debugging of programs running on the target.

This benchmark will also be run on an HP-Ada compiler that is hosted and targeted on a HP 9000/350 machine (running HV-UX V 6.5).

5.2 BENCHMARK EXECUTION

All the tasks are implemented as library level Ada tasks, so that all tasks have been created and are in the execution state during startup of the composite benchmark. During the execution of the benchmark, the two Search tasks (GS_TASK and DS_TASK) print the rate (per second) at which the frequencies are being monitored. This rate can be increased or decreased based on user supplied input data. The tasks also calculate the actual rate at which they are monitoring the frequencies. The actual rate depends upon a number of factors such as the number of signals detected at each frequency, time spent for rendezvous with the DF task, and the overall runtime speed of the Ada code for a particular Ada compiler system/hardware configuration. The number of rendezvous performed between a Search task and DF task is equal to the number of signals detected during a complete scan of the frequencies being monitored. The DS task also calculates the revisit time interval for the highest priority and lowest priority frequencies. This also depends on the number of signals that have been detected at the particular frequencies being monitored.

The DF task also operates in an infinite loop control structure. It accepts entry calls from GS_TASK and DS_TASK and then performs some mathematical computations to compute 5 LOPs for the detected signal. Elapsed times are measured from the point DF_TASK finishes a rendezvous with either GS_TASK or DS_TASK to the time an LOP has been computed. Since the DF task performs direction finding according to priority, entry call families have been used to implement prioritized queues. If DF_TASK takes a long time to perform the computation of a LOP, then the task will not be able to accept rendezvous calls from either DS_TASK or GS_TASK fast enough, so that the whole system processing may not meet its timing requirements. The timing requirements are that the DF task has to compute an LOP message for a given DF request within 2.25 seconds after it performs rendezvous with either of the two Search tasks.

After the DF task has finished calculating 5 LOPs for a detected signal, this data is passed to the EL task via rendezvous. EL_TASK operates in an infinite loop control structure and has an accept inside the loop statement. After obtaining data from the DF task, the EL task looks up existing locations to generate an EL estimate. The timing requirements that are measured is the elapsed time measured from the point EL_TASK does a rendezvous with DF_TASK to the point the EL has been determined (timing requirements are 300 milliseconds).

After an EL has been calculated, EL_TASK calls an entry in the Analysis task (AN_TASK) which determines the type of signal detected by the intercept task. The basic structure of the Analysis task is also an infinite loop that has an accept statement. Up to twenty rendezvous calls per second should be handled by the Analysis task without jeopardizing the timing requirements of the whole system. This again depends on the rendezvous time between DF_TASK and AN_TASK as well as the number of requests being made for signal classification by EL_TASK to AN_TASK.

The reporting capability is implemented as a function call within the Analysis task. A full report is printed out on the line printer for a signal whose type has been determined by the Analysis task. Elapsed times are measured from the time a signal is detected (by either of the Search tasks) to the time when finally a report is printed for that detected signal by the Analysis task. A minimum of three rendezvous are necessary to generate this report from the time the signal was detected.

Preliminary results on the Sun/Verdix/68020 configuration indicate that the system meets its timing requirements when the rate of frequency scan is 50/sec and the number of signals detected is 10 or less. When the frequency scan rate is increased to

around 200/sec, and the number of signals detected is at least 50 per scan, the timing constraints of the system were not met. The benchmark will run for a considerable period of time during which signals are detected by the GS and DS capabilities at a rate that is controllable by the benchmark user. As the rate of signal detection is increased, the pressure on the other tasks increases to meet their timing requirements. The maximum number of signals that can be detected per second without violating the timing constraints as listed in the benchmark description will be calculated. This will be dependent on factors such as rendezvous times, memory allocation/deallocation patterns during the execution of the benchmark, task creation/termination timings, task abortion timings, and numeric computation capabilities of the Ada compiler being used. Again, these are preliminary results and more detailed results will be available when the effort is completed.

Comparison of the results obtained by execution of the benchmark on the HP-Ada compiler hosted on top of Unix to the Verdix results executing on a bare Motorola 68020 should provide an idea of how the HP-Ada compiler running on Unix meets the real-time requirements of the composite benchmark. It should also indicate the ease of transporting and using the composite benchmark for testing a number of compilers.

6.0 CONCLUSIONS

This paper describes the process of development of a composite benchmark for evaluating Ada compilers for COMINT/ELINT class of real-time systems. Some of the lessons learned during this process are:

a. a major effort required in the benchmark development process was the thorough and correct analysis of the system requirements and translating them into corresponding Ada tasks or functions;

b. if there are too many tasks in the system, it is quite possible that a particular Ada compiler system may not meet the performance requirements of the COMINT/ELINT class of IEW systems. Also, overall program execution time may be increased due to the extra Ada tasks; and

c. since the design depends heavily on Ada tasking features, preliminary results indicate that rendezvous timing as well as context switching time for a particular Ada compiler system are extremely important if a system has to satisfy its timing constraints even during high loads.

The basic philosophy used to design this benchmark was;

a. to identify the system capabilities that perform the high-level monitor, analyze, and respond functions for real-time systems;

b. break these system capabilities into lower-level real-time requirements; and

c. determine tasks and procedures as well as the interfaces between the tasks.

The basic philosophy used in the composite benchmark process can thus also be used in designing a composite benchmark for other real-time systems by identifying its capabilities that perform the real-time functions and then breaking them down into the lower-level real-time details.

REFERENCES

[1] CECOM Center for Software Engineering Technical Report C02 092 LA0003, "Establish and Evaluate Ada Runtime Features of Interest for Real-time Systems", Final Report delivered by IITRI, Lanham, MD, October, 1988.

[2] CECOM Center For Software Engineering Technical Report C02 092LY 0001 00, "Real-time Performance Benchmarks For Ada", Technical Report, March, 1989.

[3] Ada Compiler Evaluation Capability (ACEC) Technical Operating Report (TOR) User's Guide, Report D500-11790-2, Boeing Military Aerospace Co., P.O. Box 7730, Wichita, Kansas, 1988.

[4] R.M. Clapp et al., "Towards Real-time Performance Benchmarks for Ada", Communications of the ACM, Vol. 29, No. 8, August 1986.

[5] "Proceedings of the International Workshop on Real-time Ada Issues", UK, 13-15 May, 1987, pages 10-11.

ABOUT THE AUTHORS

Arvind Goel received his B. Tech. degree in Electrical Engineering from IIT, Kanpur, India in 1980 and MS degree in Computer Sciences from the University of Delaware in 1982. He is the founder of Unixpros Inc. where he is working on developing composite benchmarks to model a class of real-time systems. He is also working on CASE tools and their application to real-time systems.

Mrs. Mary E. Bender is a computer scientist with the Center for Software Engineering, U.S. Army Communications-Electronics Command, Ft. Monmouth, NJ. She is the project leader for their technology program in Ada real-time applications and runtime environments. She is a principal member of ARTEWG (the SIGAda Ada RunTime Environment Working Group). She received a B. A. degree in Computer Science from Rutgers University, New Brunswick, NJ.

A Proof Method for Ada/TL Specifications

William Hankley and **James Peters**

Department of Computing & Information Sciences
Kansas State University, Manhattan KS 66506

Abstract -- This paper introduces a proof system for the tasking specification language called Ada/TL. The purpose of Ada/TL is for writing constructive specifications that can be used for the design of Ada tasking systems. Ada/TL extends Ada specification of tasks with local temporal assertions about the behavior of each task and with system assertions that specify required constraints about the interaction between tasks. Proof of an Ada/TL specification consists of showing that the system specification is consistent with all local task specifications. Proof of the tasking specification reduces verification of the corresponding Ada code to showing that each task satisfies its own specification.

The proof system is illustrated in this paper using a small tasking system example with simple rendezvous operations. The proof system consists of proof rules for the constructs and temporal operators used in Ada/TL, progress axioms about the behavior of the specified Ada programs, representation of program behavior as a computation graph, and finally, an inductive technique for proving invariant properties over states of the computation graph.

Index Terms -- Ada, concurrent systems, specification, verification, software engineering.

1. Introduction

This paper introduces a proof system for the tasking specification language called Ada/TL, which we introduced in a previous paper [14]. We begin with a preview of the subject and its relationship to other work.

Whereas Ada specification of tasks consists of declarations of tasks and related data types, Ada/TL extends Ada specifications by adding local temporal assertions about the behavior of each task and global system assertions that specify required constraints about the interaction between tasks. Ada/TL merges the style of the concrete syntax of Ada specifications with more abstract notations of temporal logic. General features of Ada/TL are illustrated in Section 2.

Proof of an Ada/TL specification consists of showing that the system specification is consistent with all local task specifications. Unlike verification of programs, there is no proof of individual task properties, which are taken as basis for design of task bodies. Since the system property generally shows relations between tasks, proof of the system property involves joint description of the threads of behavior of the related tasks. Once the system property has been verified, it does not need to be verified again for the program implementation. Thus, proof of the tasking specification reduces verification of the corresponding Ada code to showing that each task satisfies its own specification.

The proof system for Ada/TL consists of inference rules for the constructs and temporal operators used in Ada/TL, progress axioms about the behavior of the specified Ada programs, and an inductive technique for proving properties as invariant over possible states of program execution. A proof deals with the possible execution behavior of the specified program even though the program does not exist but is only specified. Execution behavior is represented as a computation graph. These components of the proof system are developed in Sections 3 and 4. The proof system is illustrated in this paper using a small tasking system example with simple rendezvous operations. The example system and verification are covered in Section 5. The proof system as developed herein includes only inference rules for constructs of the example system. Features for exception handling and timed rendezvous are not included.

2. Background Work

The Ada/TL language and proof system follow from three veins of related work:

a) verification of and proof systems for concurrent programs,
b) use of temporal logic for specification and verification of programs,
c) development of modular specifications of programs.

The following summaries illustrate some works in each of these three areas, but the references are not meant to be inclusive.

Owicki and Gries [24] applied Hoare's method to verification of concurrent programs. They used a form of Algol as the algorithm language. Apt [1] presented a more formal proof system, using CSP as the algorithm language. Later Apt provided a further formal foundation for the proof system [2]. Barringer and Mearns [6] developed a formal proof system using a subset of Ada as the base language. All of these works introduce auxiliary variables to express program assertions. The auxiliary variables may be seen as variables necessary for expressing external specifications. None of these works use temporal specifications.

Specifying the behavior of programs with temporal logic was formulated by Burstall [9], Pnueli [25,26,27], and Manna [20,21,22]. Such early work consisted of annotation of program code with temporal assertions and a proof method similar to hand tracing of assertions through the program code. Assertions were based on linear-time temporal logic. Pnueli and DeRoever [28] illustrated a proof system for Ada rendezvous based upon a semantics using temporal logic. Manna and Pnueli [23] provided a general guide to construction of proof systems. Ben-Ari, Manna, & Pnueli [7] broadened the focus to use of the more general branching-time temporal logic. Clarke [10] illustrated a simple verification method for branching-time assertions for finite state systems.

The concept of modular specification of programs was evidenced in Ada and other languages that distinguish declaration and implementation parts. Yet early papers on specification and/or verification dealt mainly with global properties. Lamport [16] carried the modular style to formal specifications. Jackson [15] illustrated using the Vienna Development Method for Ada package specifications.

3. Ada/TL Specifications

Ada/TL provides a notation for specifying the constraints on the local behavior of individual tasks and the global behavior of interaction between tasks in concurrent and distributed systems. It merges ideas from three styles of specifications:

1) It is an extension of Ada task specifications defined in the Ada Language Reference Manual (ALRM) [11].
2) It uses the model-based style of VDM (Vienna Development Method) [8] to specify data items and the effect of operations on such items.
3) It uses a variation of the linear-time temporal operators defined in Ben-Ari et al. [7] to specify the sequential behavior of individual tasks. It uses branching-time operators to specify properties of task interaction that are not constrained by individual task specifications.

This section summarizes information from [13, 14].

Ada/TL notation is illustrated using a simple tasking system in Figure 1. The specification is a generic package since ItemType is identified only as pending and ProducerType is given only as a task type, and the number of producers is not identified. Type "pending" denotes types which are used in the specification but are otherwise unspecified. The package specification consists of three task specifications (described below) followed by a **SysProperty** which specifies task interactions not constrained by the task specifications.

Task specifications include entry declarations as in the ALRM. Entry specifications are annotated with non-temporal **in** and **out** assertions, as in VDM specifications. Task specifications also define static variables like Q, and operations (procedures or functions) like Def, which are referenced as part of the task specification. We refer to these

```
generic
  type ItemType is pending;     -- specification parameters
  MaxSize: natural pending;

package Producers_Consumer is

  task type ProducerType is                -- task specification
    procedure Def(Item: out ItemType); -- define Item
    property □ ◇ seq(Def(Item), Queue.Put(Item) );
  end ProducerType;

  task Consumer is                         -- task specification
    procedure Cons(Item: ItemType);        -- consume Item
    property □ ◇ seq(Queue.Get(Item), Cons(Item));
  end Consumer;

  task Queue is                            -- task specification
    Q : seq of ItemType;                   -- state variable
    init Q'length = 0;                     -- initial assertion
    entry Put( Item: ItemType);
      in Q'out = Q'in + Item;
    entry Get( Item: OUT ItemType);
      out Q'in = Item + Q'out;
    property Q'length ≤ MaxSize;           -- global assertion
  end Queue;

  SysProperty ∀□ (all P: ProducerType:
      P:Def(Item) imp ◇ Consumer:Cons(Item) );
end Producers_Consumer;
```

Fig. 1. Sample specification.

items as externally observable, in that they are used in description of the task specification but they are not accessible by other tasks, nor are they necessarily part of the task implementation. The observable items may become items in the task body or they may correspond to the auxiliary variables mentioned in Section 2. The last component of each task specification is the **property** assertion which specifies the linear-time temporal behavior of the subject task, including interaction with other tasks and access to either global items or its own observable items.

In Figure 1, an unspecified number of Producer tasks each repeatedly execute the sequence of generating an Item using the Def operation and then putting the Item into the bounded Queue task. The temporal quantifiers □ (always) and ◇ (eventually) together are read as "infinitely often". The "seq(...)" operator specifies a temporal sequence of predicates. The predicate "Def(Item)" is interpreted as the control predicate "at(Def(Item))" which has the side effect of instantiating Item. The predicate "Queue.Put(Item)" is interpreted as the control predicate "at(Queue.Put(Item))" which means that the Producer task calls the entry Put in task Queue with the previously instantiated value of Item.

The entry assertions of task Queue specify that it implements the abstract type queue using the data item Q. The single Consumer task repeatedly removes an Item from task Queue and performs the operation Cons.

The **SysProperty** asserts that over all paths of execution, every data Item instantiated by some Producer task P calling its operation Def will eventually be used by the Consumer task operation Cons. The branching-time quantifier $\forall\Box$ indicates that the "always" quantifier holds over all paths of the system execution. The system property seems obvious from the task specifications, but it is not explicitly stated in the task specifications because producer and consumer tasks never directly interact.

In summary, non-temporal predicates express constraints about parameters and observable variables. Control predicates express events of execution of entry procedures and observable operations. They are interpreted as using the "at" predicate [16], although the "at" is not written. They constrain instances of "in" parameters and they instantiate values of "out" parameters. Temporal predicates are composed using temporal quantifiers applied to either non-temporal or control predicates. Temporal predicates express sequence, repetition, or eventuality of predicate conditions.

Task names used in predicates allow three kind of qualification:

T1. op	.....	specified task calls entry op of task T1; this follows Ada notation
T1: op	.....	task T1 performs operation op; this notation is used in the system property
T1'attribute	...	an attribute of T1 is evaluated.

Ada/TL introduced the task attribute STATE which is a record of all information related to states of execution of tasks. STATE components include the local clock for the task, state of execution [3], execution stack for operations performed by the task, and all entry queues. These notations will be explained when used within specification examples.

The temporal meaning of task specifications is based upon a Kripke structure model [31]. The execution behavior of a task T can be represented as a sequence of states $h = (s_{T0}, s_{T1}, s_{T2}, \ldots)$. The notation $K_{i,h}(P)$ denotes that P is true in the i th state of h. Temporal quantifiers are defined in terms of the Kripke structure:

always $K_{i,h}(\Box P) == (\text{all } j:\ j \geq i:\ K_{j,h}(P))$

eventually $K_{i,h}(\Diamond P) == (\text{exists } j:\ j \geq i:\ K_{j,h}(P))$

before $K_{i,h}(P \text{ before } Q) ==$
$(\text{exists } j,k:\ i \leq j < k:\ K_{j,h}(P) \text{ and } K_{k,h}(Q))$

sequence $K_{i,h}(\text{seq}(P)) == K_{i,h}(\Diamond P)$
$K_{i,h}(\text{seq}(P_1, P_2, \ldots, P_n)) ==$
$K_{i,h}(P_1 \text{ before seq}(P_2, \ldots, P_n))$

infinitely often $\Box\Diamond P == (\Diamond P) \text{ before } (\Box\Diamond P)$

and similarly for other operators.

Branching time quantifiers are defined in a similar fashion, but over composite states of a system. A system state S_i is a tuple $\langle G_i, s_{\alpha i}, s_{\beta i}, s_{\gamma i}, \ldots\rangle$ where G_i is a record of all global data items, and $s_{\alpha i}, s_{\beta i}, s_{\gamma i}, \ldots$ are states of the component tasks of the system. The system path $H = (S_0, S_1, S_2, \ldots)$ defines a computation tree of possible behaviors because of possible non-deterministic interleaving of individual task behaviors. The branching-time quantifiers are written as $\forall\Box$ and $\forall\Diamond$ to emphasize that they must hold for all system paths composed from individual task paths.

4. Proof System

The proof system is comprised of inference rules for the constructs used in Ada/TL and the actual proof method. This section presents a minimal set of inference rules and the general proof structure. First, additional notation is presented.

The essence of the proof technique is to show evaluation of the system property over all system paths that can be composed from task paths determined by the task properties. To analyze task paths we need further notation for describing task states. Such notation is part of what is known as an observer languages for the specification language. We use two observer operators which are called location markers: $\blacktriangledown$, read as "before" and defined below, and $\Diamond$, read as "before or at". The $\Diamond$ operator is exactly the eventually operator. When $\Diamond$ is used in specifications, it is read as "eventually"; when it used to represent task states, it is read as "before or at". The $\blacktriangledown$ operator is defined as follows:

AT(P) == true iff P contains a true conjunct at(op)

$K_{i,h}(\blacktriangledown P) ==$ not $K_{i,h}(AT(P))$ and (exists j: j>i: $K_{i,h}(P)$)

$K_{i,h}(P \blacktriangledown) ==$ (exists j: j<i: $K_{i,h}(P)$) and not $K_{i,h}(AT(P))$

That is, $\blacktriangledown$ marks a task state that is before (or after) a state that satisfies predicate condition P. If P includes a control predicate then the control predicate is not yet (or no longer) true. The following properties can be shown:

$\blacktriangledown\Diamond P = \blacktriangledown P$

$\blacktriangledown\ \Box\Diamond P == \text{seq}(\blacktriangledown P, \Box\Diamond P)$

$\blacktriangledown\ \text{seq}(p_1, \ldots, p_n) = \text{seq}(\blacktriangledown p_1, \ldots, p_n)$

$\text{seq}(p_1, \ldots, p_i \blacktriangledown, \ldots, p_n) = \text{seq}(p_1, \ldots, \blacktriangledown p_{i+1}, \ldots, p_n)$

$\text{seq}(p_1, \ldots, p_n \blacktriangledown) = \text{seq}(p_1, \ldots, p_n)\blacktriangledown$

Using the location markers, we represent that one state progresses to the next state as: $s_{T1} \Rightarrow s_{T2}$. The following progress axiom must hold:

Axiom P.1: Local Progress:
Let p_i be a satisfiable predicate without temporal quantifiers and not involving rendezvous:

$$\blacktriangledown p_i \Rightarrow \Diamond p_i$$
and
$$p_i \Rightarrow p_i \blacktriangledown$$

As a consequence, we also have:

$$seq(p_1, \ldots, \blacktriangledown p_i, \ldots, p_n) \Rightarrow seq(p_1, \ldots, p_i \blacktriangledown, \ldots, p_n)$$

Following are inference rules for a small part of Ada/TL. These are given in a simplified form (ignoring preconditions for parameters). The emphasis is on readability of the rules. The semantics of Ada/TL tasks are meant to be exactly the semantics of Ada. The inference rules give no new information about Ada. The rules merely enumerate in a formal way certain behavior of Ada so that the behavior rules can be cited within steps of proofs of system properties.

Axiom E.1: System Elaboration:
Let M be the main system specification.

all H : $K_{0,H}$ (M:elab)

Meaning: The initial state of any possible path of execution is the elaboration of the system body.

Rule E.2: Component Elaboration:
Let M be any specification module, and
let C = {C_i} be the set of components of M.

M:elab
(all C_i in C : $\blacktriangledown$ C_i:elab) and $\blacktriangledown$ M:init

Meaning: Elaboration of M leads to elaboration of its components, and satisfying the initial predicate of M.

Rule E.3: Component Initialization:

M:init
$\blacktriangledown$ M:prop

Meaning: Initialization of M leads to satisfying its specification property.

Rule R.1: Simple rendezvous:
Let C be a client task, S be a server task, E be an entry of S.

a) C:S.E
in(C:S.E) and
S'State.Q(E)'new = S'State.Q(E) + callrec(C)

b) S:E
in(S:E)

c) in(C: S.E), in(S:E), S'State.CallerId = C
C:S.E $\blacktriangledown$, S:E $\blacktriangledown$,
S'State.Q(E)'new = tail(S'State.Q(E))

Meaning: (a) When a client task calls an entry of a server, the call record is enqueued. (b) The simple accept statement makes local progress. (c) When a server accepts an entry for caller C, and caller C is in the call to the entry, then both tasks progress to states after the call and accept states. We have ignored restating the constraints of the in and out assertions of the entry E since these are stated as part of the specification of the task entry and it is expected that specification of the client task will satisfy the constraints of the entry. Constraints on the length of rendezvous queues is also ignored.

Finally, we give the structure for verification of system properties with the $\forall\Box$ quantifier. Generally the collection of all possible system paths defines an infinite computation tree [10]. To reduce the computation tree to a finite structure, we identify characteristic states. A characteristic state S is a state that occurs repeatedly in the system behavior. It is characterized by some predicate. For example, if all task properties have the form $\Box \Diamond \phi$, then a characteristic state may be that each task τ is in the state $\blacktriangledown \phi_\tau$. The proof structure consists of the following parts:

1) Show that the initial state
leads to one or more characteristic states.
2) Show that each characteristic state
leads to another characteristic state.
3) Show that the system property holds for each state
of the finite model determined in steps 1 and 2.

In showing steps of progress of tasks, operations local to tasks can be interleaved in unknown order. Task behaviors are synchronized at points of rendezvous. We show this by grouping proof steps in the form of a concurrency map [32]. For example, with two tasks U and V, proof steps are organized in the style shown on the next page. The proof lines are interpreted as representatitive of equivalent ordering of the lines. Lines 1 through 5 can be interleaved in any order which preserves the U1,U2 ordering and the V1,V2,V3 ordering. Lines 6 and 7 can be occur in reverse order, but they must occur after lines 3 and 5.

```
1     U1
2               V1
3     U2
4               V2
5               V3
      ==========
6     U3
7               V4
      ==========
```

5. Example Proof

Specification of an example system called Sync_Pkg is given in Figure 2. The generic specification allows an arbitrary number of User tasks which each perform some operation on a shared resource. User tasks repeatedly acquire permission from the Server task, use the resource, and then release the permission back to the Server.

```
generic
 type ResourceType is pending;
 type ResultType is pending;
 type OpType is pending;

package Sync_Pkg is

 resource: ResourceType;

 task Server is
  entry Get(TaskToGetCapability:          IN TaskName);
  entry Done(TaskToReleaseCapability:     IN TaskName);
  property
   □ ◇ seq( Get(TaskToGetCapability),
            Done(TaskToReleaseCapability)
           and
(TaskToGetCapability=TaskToReleaseCapability)          ) ;
 end Server;

 task type UserType is
  procedure Def(OpChoice: OUT OpType);
     --select operation to be performed on resource
  procedure Perform( OpChoice: IN OpType;
                     Result   : OUT ResultType);
     --perform operation OpChoice on resource and
     -- return Result
    out   map(OpChoice, resource, Result);
  procedure Cons(Value: IN ResultType);
     --consume Value
  property
     □ ◇ seq(Def(OpChoice),
             Server.Get(STATE.TaskName),
              Perform(OpChoice, Result),
             Server.Done(STATE.TaskName),
              Cons(Result)  );
end UserType;

  SysProperty
    ∀□ (all U1, U2: UserType and U1 /= U2:
          not(in U1: Perform  and in  U2: Perform));

end Sync_Pkg;
```

Fig.2 Sync_Pkg Specification

The system property for Sync_Pkg is a safety property. It requires that no more than one User task accesses the shared resource at the same time. This mutual exclusion property is achieved by the tasking system protocol, but that is not proved by the task specification.

For the proof example, assume an instantiation Sync_2 of Sync_Pkg with two User tasks. We use several short notations. S2 represents the system Sync_2, U1 and U2 represent the User tasks, and S represents the Server task. ø1, ø2, and øS represent the task properties for U1, U2, and S. States are numbered as:

n	n th state of S2 task
1:n, 2:n, 3:n	n th state of U1, U2, or S tasks

The queues S'State.Q(get) and S'State.Q(done) are represented simply as get and done.

Part 1:

An evident characteristic state is when U1 and U2 are both before or at their request to S and S is before the get entry. The corresponding predicate is:

(◇ U1:S.get) and (◇ U2:S.get) and (▼ S.get) and |get|≤2 and |done|=0

state S2	U1	U2	S	condition	justification
0				S2:elab	E.1, sys elab
1			3:0	S:elab	E.2, comp elab
			3:1	S:init	E.2, comp elab
				= |get|=0 and |done|=0	
			3:2	▼ S:get	E.3, init
2	1:0			U1:elab	E.2, comp elab
	1:1			▼ ø1	E.2, comp elab
	1:2			◇ U1:S.get and |get|≤1	P.1, local prog
3		2:0		U2:elab	E.2, comp elab
		2:1		▼ ø2	E.2,comp elab
		2:2		◇ U2:S.get and |get|≤2	P.1,local prog
4				S2:elab ▼	P.1,local prog

Discussion: The characteristic state is <4, 1:2, 2:2, 3:2> . Progress is blocked pending the S:get operation. U1 and U2 are before or at their Ui:S.get operation.

Part 2:

state S2 U1 U2 S	condition	justification
	3.2 ▼ S:get and |get|≤2	assumed
	3.3 S:get	R.1b, rendezvous
1:3	◇ U1:S.get	assumed
1:4	U1:S.get	R.1a, rendezvous
	3:4 S:get and get.caller=U1	from 1:4, assume U1 is 1st
	3:5 S:get ▼ and |get|≤1	R.1c, rendezvous
	3:6 ◇ S:done	P.1, local prog
1:5	U1:S.get ▼	R.1c, from 3:4
1:6	U1:perform	P.1, local prog
1:7	U1:S.done and |done|=1	R.1a, rendezvous
	3:7 S:done	R.1b, rendezvous
	3.8 S:done and done.caller=U1 and get.caller=done.caller	R.1c, rendezvous from 3:4
	3.9 S:done ▼ and |done|=0	R.1c, rendezvous
	3.10 ▼ øS	P.1, local prog
	3.11 ▼ S.get	P.1, local prog
1.8	U1:S.done ▼	R.1c, from 3:8
1.9	U1:S.cons	P.1, local prog
1.10	▼ ø1	P.1, local prog
1.11	◇ U1:S.get and |get|≤2	R.1a, rendezvous

Discussion: Without loss of generality, we assume U1 makes progress through the rendezvous and U2 (and other User tasks) are blocked. Alternately, U2 may progress and U1 would be blocked. In either case, starting in the characteristic state, the system progresses through a sequence of states back to the original characteristic state. Thus the general computation tree is reduced to a finite state behavior. The proof depends on induction in the same manner that proofs of sequential loops use depend upon induction.

Part 3:

1. For all states in the Part 1 proof, no Ui:perform operation is yet started. Hence, the system property is trivially true.

2. For all states in the Part 2 proof, only one user task, arbitrarily selected as U1, can do its perform operation. U2 is blocked in the U2:S.get rendezvous. Again, the system property holds.

4. Conclusion

The purpose of the proof system is to provide the basis for developing proofs of consistency for Ada/TL tasking system specifications. As a result of proving the consistency of a system property relative to local task properties, it is then only necessary to prove that task bodies correctly implement corresponding task specifications. This proof system introduces the "before" and "before or at" markers as part of an observer's language for discussing predicates about tasking system behavior. This observer language is important because it provides a means of making temporal proofs more tractable. We are continuing work to extend the proof system for general forms of rendezvous.

References

[1] K.R. Apt, N. Francez, W.P. DeRoever, "A Proof System Transactions for Communicating Sequential Processes", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 3 (July 1980), 359-385.

[2] K.R. Apt, "Formal justification of a proof system for communicating sequential processes," *Journal of the ACM*, Vol. 30, No. 1 (1983), 197-216.

[3] A. Burns, A. Lister, A. Wellings, *A Review of Ada Tasking*, LNCS 262, Springer-Verlag, 1987.

[4] H. Barringer, "Axioms and Proof Rules for Ada Tasks", *IEE Proceedings*, Part E: *Computers and Digital Techniques*, vol. 129, No. 2 (Mar 1982), 39-48.

[5] H. Barringer, R. Kuiper, A. Pnueli, "Now You May Compose Temporal Logic Specifications", *Proceedings of the 16th ACM Symposium on Theory of Computing*, Washington, D.C. (1984), pp. 51-63.

[6] H. Barringer, "A Proof System for Ada Tasks", *The Computer Journal*, vol. 29, No. 5 (Oct. 1986), 404-415.

[7] M. Ben-Ari, Z. Manna, A. Pnueli, "The Temporal Logic of Branching Time", *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*. Jan. 26-28, 1981, 164-175.

[8] D. Bjorner, C. Jones, *Formal Specification and Software Development*. NJ: Prentice-Hall, 1982.

[9] M. Burstall, "Program proving as hand simulation with a little induction", *Proceedings of IFIP Congress*, 1974, Stockholm. Amsterdam: North-Holland, pp. 308-312.

[10] E. Clarke, M. Browne, E. Emerson, A. Sistla, "Using Temporal Logic for Automating Verification of Finite State Systems" in *Logics and Models of Computer Science*. NY: Springer-Verlag, 1985, 3-26.

[11] US Dept Defense, *Reference Manual for Ada Programming Language*, ANSI/MIL STD 1815A-1983. NY: Springer-Verlag, 1983.

[12] W. Hankley & J. Peters, "A Proof Method for Ada/TL", TR-CS-89-11, Kansas State Univ, 1989.

[13] W. Hankley & J. Peters, "Temporal Specification with Ada/TL," TR-CS-89-7, Kansas State Univ. ,1989.

[14] W. Hankley & J. Peters, "Temporal Specification of Ada Tasking," *Proceedings of the 23rd Hawaii International Conference on System Sciences* (January, 1990).

[15] M. Jackson, "Developing Ada programs using the Vienna Development Method", *Software--Practice and Experience*, vol. 15, no. 3, March 1985, pp. 305-318.

[16] L. Lamport, "Specifying Concurrent Program Modules", *ACM Trans. on Programming Languages and Systems*, vol. 5, no. 2, April 1983, 190-222.

[17] L. Lamport, "Control Variables are Better than Dummy Variables for Reasoning about Program Control", Tech Report, Digital Equip. Corp., May 5, 1986.

[18] L. Lamport, "A Simple Approach to Specifying Concurrent Systems", *CACM*, vol. 31, no. 1 (Jan. 1989), 32-47.

[19] B. Lampson, "Atomic Transactions," *Distributed Systems: Architecture and Implementation.* NY: Springer-Verlag, 1981, 246-265.

[20] Z. Manna, A. Pnueli, *The Modal Logic of Programs, Automata, Languages and Programming*, LNCS 79. Berlin: Springer-Verlag, 1979, 385-409.

[21] Z. Manna, A. Pnueli, "Temporal Verification of Concurrent Programs: The Temporal Framework of Concurrent Programs", *The Correctness Problem in Computer Science* R. Boyer, J Moore. (eds), Orlando, FL: Academic Press, 1981, 215-272.

[22] Z. Manna, A. Pnueli, *Verification of Concurrent Programs: Temporal Logic Principles*, LNCS 131. NY: Springer-Verlag, 1981.

[23] Z. Manna, A. Pnueli, "How to Cook a temporal proof system for your pet language", *Proc. ACM Symposium on Programming Languages*, Austin, TX, (Jan. 1983), 141-154.

[24] S. Owicki & D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica* 6 (1976), 319-340.

[25] A. Pnueli, "The Temporal Logic of Programs", *18th Annual Symposium on the Foundations of Computer Science*, IEEE, Nov. 1977, pp. 46-57.

[26] A. Pnueli, "The Temporal Semantics of Concurrent Programs" in *Lecture Notes in Computer Science 70*, Springer-Verlag, NY, 1979, pp. 1-20.

[27] A. Pnueli, "The temporal Logic of concurrent programs", *Theoretical Computer Science*, vol. 13 (1981), pp. 45-60.

[28] A. Pnueli, W. DeRoever, W.P. "Rendezvous with Ada--A Proof Theoretical View", *Proceedings of the AdaTEC Conference on Ada*, Arlington, VA, Oct. 6-8, 1982, 129-137.

[29] A. Pnueli, "In transition from global to modular temporal reasoning about programs", in *Logics and Models of Concurrent Programs* , K. Apt. (ed) Series F: Computer and System Sciences, vol. 13. Springer- Verlag, 1985.

[30] D. Reed, "Implementing Atomic Actions on Decentralized Data", Vol. 1, No. 1, *ACM Transactions on Computer Systems* (February 1983), pp. 3-23.

[31] S. Kripke, "Semantic Analysis of Modal Logic", *Z. Math. Logil Grundlagen*, vol. 9 (1963), pp 67-96.

[32] J. Stone, "A Graphical Representation of Concurrent Processes", *Sigplan Notices*, vol 24, no 1 (Jan 1989), pp 226-235.

Automatic production of Mil/DoD/NASA-Std Documentation

Lee Jensen and Thomas S. Radi, Ph.D.

Software Systems Design 3627 Padua Avenue
Claremont, California 91711
Telephone (714) 625-6147 FAX (714) 626-9667

ABSTRACT

This paper describes how software design documentation for Military, NASA and DoD standards, such as DoD-Std-2167A, can be produced in an accurate and cost-effective manner. The documentation is produced by analyzing design code and extracting the required information.

The software designs produced during both Preliminary and Detailed design phases are expressed in Ada (with Ada/PDL extensions). The designs are a combination of Ada source code and Ada PDL constructs in the form of comments and are therefore fully compilable by any Ada compiler. As a by-product of the design process, these Ada designs contain much of the information required by Military or DoD standards.

Information required by Mil/DOD/NASA standards is obtained from by a thorough analysis of the design. The design is parsed by ADADL, an intelligent Ada PDL processor which is the equivalent of the front end of an Ada compiler. ADADL determines the relevant information directly from the design as expressed in source code.

The tool which assembles the extracted information into the proper document format is called DocGen. The DocGen output is an ASCII file which is compatible with Interleaf, LaTex, Context/Doc, nroff/troff and/or VMS/RUNOFF, as well as other text processors and publishing systems.

Introduction

Producing Mil-Std documentation is a tedious and time consuming process. Each required document has a rigid format which must be thoroughly understood before accurate documentation can be produced. Typically engineers developing the software system must spend large amounts of time understanding and writing the complex documents.

This paper describes a documentation tool, called DocGen, which generates documents from information found in design databases, pseudo code and source code (see Figure 1). The tool quickly and accurately provides documentation which adheres to standards (DOD 2167A, DOD 2167, NASA, NSA, MIL-STD 1679...). DocGen also provides the flexibility to create custom or tailored documents with minimal effort.

Many document generators support only the early design phases by providing a skeletal outline of the document, leaving most of the work for the developers. DocGen is capable of generating complete documentation for most phases of the software development life cycle. For example, in the design phase, DocGen has access to the detailed information from the source code which is not available to other document generators. Therefore DocGen can produce a more complete document. The philosophy of utilizing information from the appropriate CASE tools for each life cycle phase allows DocGen to produce complete document for that phase.

The ability to analyze and collect documentation information from design databases is provided through other CASE tools in the AISLE family available from Software Systems Design (see Figure 2). DocGen is the documentation generator for the AISLE toolset.

An automatic document generation process

In describing the DocGen tool, DOD-STD-2167A DI-CMAN-80012A, the Software Design Document, will be used to illustrate the document generation process. As stated in this standard, "The Software Design Document (SDD) describes the complete design of a Computer Software Configuration Item (CSCI) as composed of Computer Software Components (CSCs) and Computer Software Units (CSUs)."

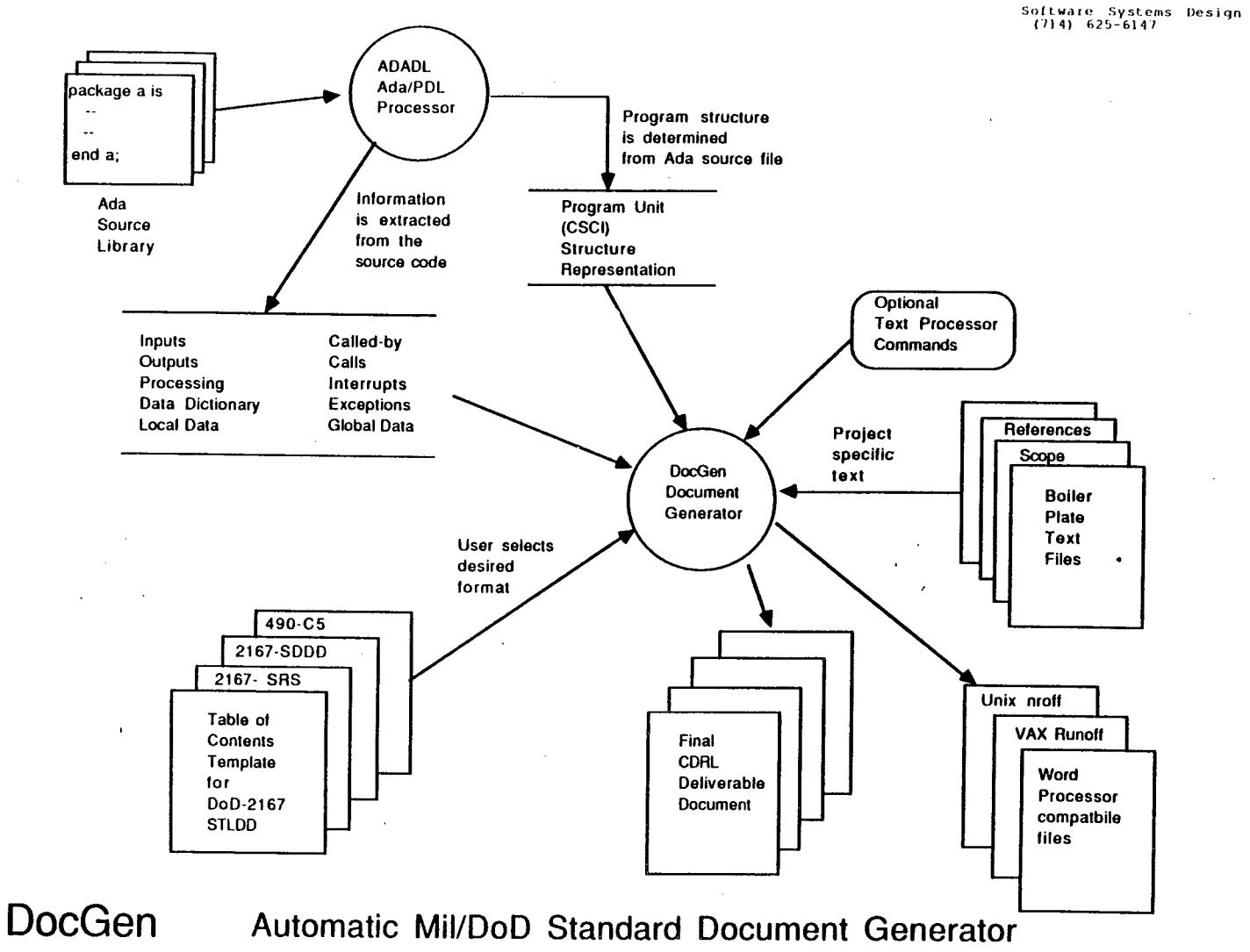

Software Systems Design
(714) 625-6147
package a is
--
--
end a;
Ada
Source
Library
ADADL
Ada/PDL
Processor
Program structure
is determined
from Ada source file
Information
is extracted
from the
source code
Program Unit
(CSCI)
Structure
Representation
Inputs
Outputs
Processing
Data Dictionary
Local Data
Called-by
Calls
Interrupts
Exceptions
Global Data
Optional
Text Processor
Commands
DocGen
Document
Generator
Project
specific
text
References
Scope
Boiler
Plate
Text
Files
User selects
desired
format
490-C5
2167-SDDD
2167- SRS
Table of
Contents
Template
for
DoD-2167
STLDD
Final
CDRL
Deliverable
Document
Unix nroff
VAX Runoff
Word
Processor
compatbile
files
DocGen
Automatic Mil/DoD Standard Document Generator

Figure 1.

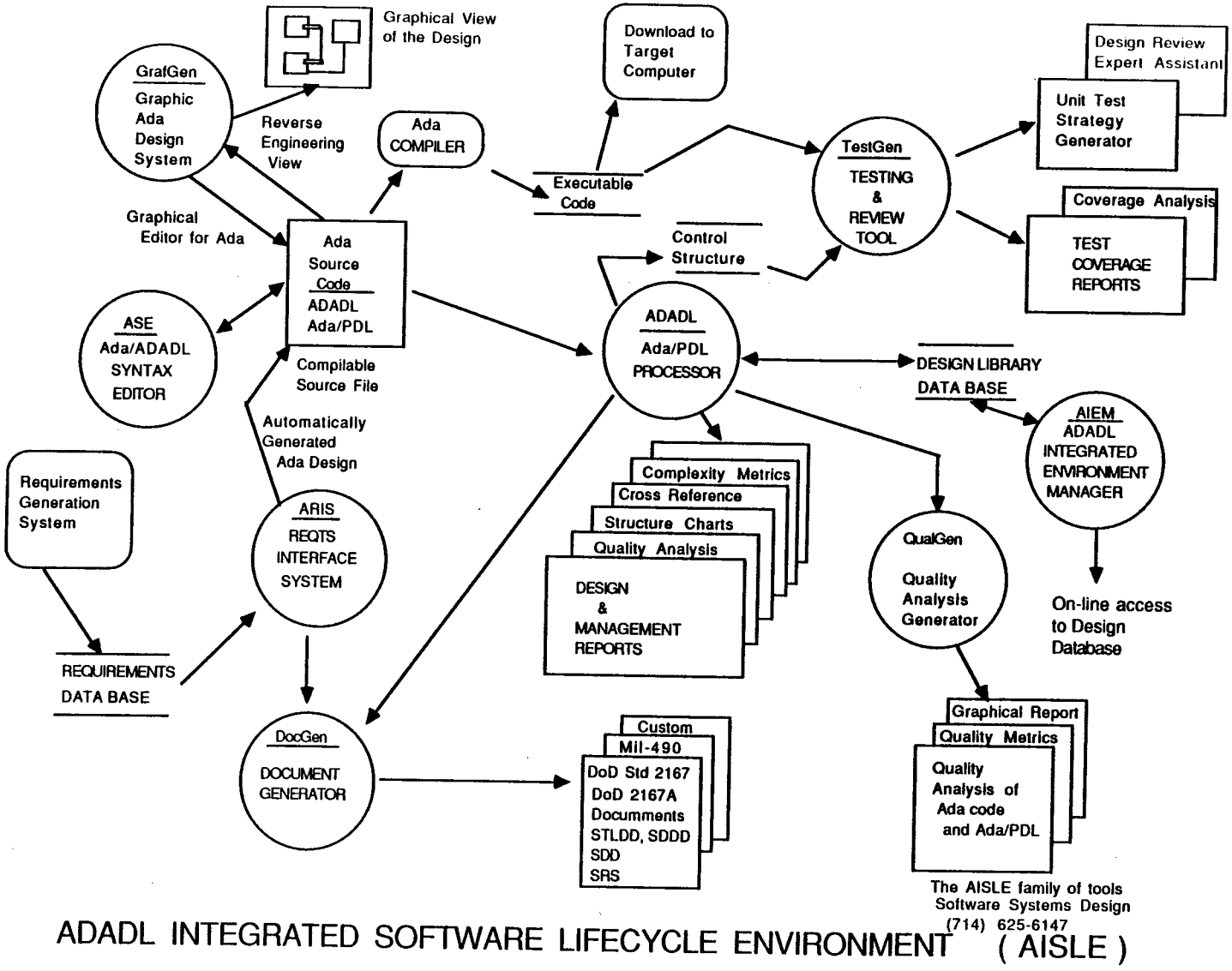

Graphical View of the Design
GrafGen
Graphic Ada Design System
Reverse Engineering View
Ada COMPILER
Download to Target Computer
Design Review Expert Assistant
Unit Test Strategy Generator
TestGen
TESTING & REVIEW TOOL
Coverage Analysis
TEST COVERAGE REPORTS
Executable Code
Graphical Editor for Ada
Ada Source Code
ADADL
Ada/PDL
Control Structure
ASE
Ada/ADADL SYNTAX EDITOR
Compilable Source File
ADADL
Ada/PDL PROCESSOR
DESIGN LIBRARY DATA BASE
AIEM
ADADL INTEGRATED ENVIRONMENT MANAGER
Automatically Generated Ada Design
Requirements Generation System
ARIS
REQTS INTERFACE SYSTEM
Complexity Metrics
Cross Reference
Structure Charts
Quality Analysis
DESIGN & MANAGEMENT REPORTS
QualGen
Quality Analysis Generator
On-line access to Design Database
REQUIREMENTS DATA BASE
DocGen
DOCUMENT GENERATOR
Custom
Mil-490
DoD Std 2167
DoD 2167A
Documments
STLDD, SDDD
SDD
SRS
Graphical Report
Quality Metrics
Quality Analysis of Ada code and Ada/PDL
The AISLE family of tools
Software Systems Design
(714) 625-6147
ADADL INTEGRATED SOFTWARE LIFECYCLE ENVIRONMENT (AISLE)

Figure 2.

Many document generators such as those associated with Structured Analysis/Structured Design tools are only capable of generating SDD document skeletons based on the CSC's in the data flow diagrams or structure charts. They do not have detailed information about the preliminary and detailed designs, such as whether a data item is local data, global data, etc.

In the typical 2167A software development life cycle, the preliminary design will be represented as a structure chart, Booch diagram, or Ada program units.

There are three parts to preliminary design: definition of program structures; definition of data structures; and description of top level algorithms.

Graphical methods such as Booch diagrams and structure charts provide support for the definition of program structures but lack support for data structures and algorithms. The use of an Ada Program Design Language (PDL), such as ADADL, during preliminary and detailed design provides the support for all three parts of preliminary design.

This preliminary design is further refined into detailed design and finally source code. At each step, the design may be submitted to ADADL to generate the information needed by DocGen to produce the SDD. According to the DoD-STD-2167A format, the SDD is required to contain information about both the preliminary and detailed design.

To generate an SDD, DocGen uses information extracted by ADADL from the preliminary or detailed design. The ADADL processor finds much of the required information in the pseudo code or source code automatically through its compiler front end. Data dictionary definitions (ADADL "--%" construct), entered for each entity are used by DocGen where natural language descriptions are required. Processing is described using the ADADL pseudo code construct ("--|"), for those who use pseudo-code, or from the executable code for those who like to use Ada itself as the design language.

Documentation which cannot be obtained from parsing the source code may be entered as supplementary text. The construct "--#extract {keyword}" followed by the appropriate documentation is entered and then extracted by ADADL for use by DocGen. The "keyword" can be user defined, for example "--#extract mem_timing" in the example ADADL input file shown in Figure 3.

DocGen accesses the documentation based on the "keyword". Automatically extracted information is referenced by reserved "keywords" which are described in Appendix A.

```
  FUNCTION Size_Board RETURN integer IS
--%The board size is entered by the user and is checked
--%to verify that the entered value is within 4 to 12.
--%The size of the board is returned as an integer.

--#extract HLD
--#Finds the size of the chess board by asking the user.
--#end

--#extract limitations
--#There are no special limitations on the use of
--#this program unit.
--#end

--#extract requirements
--#No explicit requirement is satisfied by this routine.
--#end

--#extract relationships
--#This routine supplies board size to calling routines.
--#end

--#extract mem_timing
--#There are no memory/timing constraints on this
--# routine.
--#end

Board_Size: integer; --%The size of the chess board.
  PACKAGE Int_Io IS NEW integer_io(integer);

   BEGIN
--| Print "program to solve the N queens problem" on the screen;
--| LOOP on unacceptable input
--|  Print "enter the board size in range 4..12" on the screen;
--|  Read the user's response from the keyboard;
--|  IF the entered Board_Size is within acceptable limits THEN
--|    RETURN the Board_Size
--|  END IF
--| END LOOP

    put_line ("program to solve the n queens problem");
    LOOP
       put ("enter the board size in range 4..12 ");
       int_io.get(Board_Size);
       put_line (" ");
       IF Board_Size >= 4 and Board_Size <= 12 THEN
          RETURN Board_Size;
       END IF;
    END LOOP;
    END Size_Board;
END Set_Board_Size;
```

Figure 3. Portion of an ADADL input file.

Using the information as derived from the source code, DocGen assembles the document based on a Table of contents template file. Appendix B contains a portion of a Table of contents file for the SDD. This file is essentially the table of contents for the final document. It contains commands instructing DocGen how to create the document. The file outlines the format of the document and specifies the location of paragraphs, paragraph headings and the documentation information they will contain. The documentation information is placed by using the extract keywords described above. Text or graphics files may also be included to further augment the document.

The table of contents template instructs DocGen to loop through the CSCI architecture or program unit hierarchy in order to generate the same documentation for different units. The loops are controlled by variable indices in the paragraph numbers of the table of contents template. See paragraph 4.1 of the table of contents file in Appendix B.

The CSCI architecture can be the default hierarchy as determined by the declaration structure of the source code, or the hierarchy can be explicitly specified using embedded ADADL commands in the source or design code.

DocGen is intended as a document generator which works in conjunction with text processors or desk top publishing systems. There are many excellent publishing systems available. DocGen output is compatible as input to any of these tools to produce polished final documents automatically. DocGen allows the embedding of the publishing tools' commands into the DocGen output. Table of contents templates exist to produce files that are compatible with nroff, VAX/RUNOFF, Interleaf, LaTex and other publishing systems. The DocGen output may then be input to the publishing system to produce the final document. The portion of the automatically produced SDD that relates to the example shown in Figure 3 is shown below. This document was produced by using the nroff utility and the nroff table of contents as shown in Appendix B.

4.1.1. Function Set_Board_Size.Size_Board (CSU)

Purpose:
This function prompts the user to specify the board size.

The board size is entered by the user and is checked to verify that the entered value is within 4 to 12. The size of the board is returned as an integer.

Design specification/constraints:
No explicit requirement is satisfied by this routine.

Design:

Input data elements:
No parameter inputs for Function Size_Board.

Output data elements:
Return Type: Integer

Local data elements:
Board_Size
Defined As: Integer
The size of the chess board.

Interrupts and signals:
No interrupts handled by Function Size_Board.

Algorithm/Logic Flow:
Print "program to solve the N queens problem" on the screen;
loop on unacceptable input
| Print "enter the board size in range 4..12" on the screen;
| Read the user's response from the keyboard;
| if the entered Board_Size is within acceptable limits then
<—return the Board_Size
| end if
end loop

Error handling:
No exceptions found for Function Size_Board.

Use of other elements:
Procedure specification Text_Io.Put_Line
Procedure specification Text_Io.Put
Procedure specification Set_Board_Size.Size_Board.Int_Io.Get

Data structures:
No data types found for Function Size_Board.

Limitations:
There are no special limitations on the use of this program unit.

CONCLUSION

A tool has been developed which is capable of automatically generating documentation for most life cycle phases. Currently this tool, DocGen, can generate documents for the preliminary and detailed design phases of software development. DocGen's philosophy of incorporating information from other CASE tools used throughout the life cycle facilitates more complete automatically generated documents.

In the near future, other CASE tools will become more tightly integrated with DocGen in order to produce complete documents for additional life cycle phases. The ARIS tool currently processes requirements analysis tool databases (ie. Teamwork, Software through Pictures, Excelerator) to produce an Ada preliminary design. DocGen will be integrated with ARIS in order to produce the Interface Design Specification, Interface Design Document and Software Requirements Specification for DOD-STD 2167A. DocGen will also be integrated with TestGen, a design and source code testing tool, in order to produce the Software Test Plan, Software Test Description, and Software Test Report for DOD-STD 2167A. Tighter integration with RETT, a requirements traceability tool, will give DocGen the ability to provide complete requirements traceability throughout the documents.

Appendix A - Information automatically extracted by the ADADL processor

"keyword"	Description
In	All program unit "in" and "in out" parameters are shown. The parameters are identified by the name of the parameter, the type definition (including any initialization), and the data dictionary definition of the object.
Out	All program unit "out" and "in out" parameters are shown. The parameters are identified by the name of the parameter, the type definition (including any initialization), and the data dictionary definition of the object.
Called_by	A list is created for each program unit which identifies all program units which are invoked by that particular program unit.
Invocation	A list is created for each program unit which identifies all program units which invoke that particular program unit.
Local	All data which is declared in a program unit is identified. The objects are identified by the name of the object, the type definition (including any initialization), and the data dictionary definition of the object.
Global	All data which is used in a program unit but is not local data is identified as global data. The objects are identified by the name of the object, the type definition (including any initialization), and the data dictionary definition of the object.
Exceptions	All exceptions are identified. Each exception name and data dictionary definition is listed.
Interrupts	All interrupts are identified. Each physical address corresponding to an interrupt is listed along with the data dictionary definition for the interrupt.
Pseudo_code	ADADL pseudo-code (--| construct) is copied from the source code and pretty-printed to show the control structure and the nesting levels.
Purpose	The data dictionary definition (--% construct) for each program unit.
Code	The executable Ada code (between the "begin" .. "end" for the program unit) is copied from the source code. This may be used as the processing section of the documentation by those developers who use "pure" executable Ada rather than pseudo-code.
Other_inputs	All objects that are declared as (non-parameter) "other inputs" in the design using the ADADL "--#other inputs" command will be listed along with their data dictionary definitions and their declarations. The objects listed as "other inputs" will be combined with the parameter inputs in the output file.
Other_outputs	All objects that are declared as (non-parameter) "other outputs" in the design using the ADADL "--#other outputs" command will be listed along with their data dictionary definitions and their declarations. The objects listed as "other outputs" will be combined with the parameter outputs in the output file.
Declaration_tree	The declaration tree determined by ADADL from the design files program unit declaration hierarchy. "declaration_tree" must appear as a major paragraph in the DESIGN.TOC file in order for the information to be included in the document.
Invocation_tree	The invocation tree determined by ADADL from the design files program unit invocation hierarchy. "invocation_tree" must appear as a major paragraph in the DESIGN.TOC file in order for the information to be included in the document.

Appendix B - A portion of SDD Table of Contents template file

```
T1.".NH 1\n" Scope
.XS \" SN string provides section numbers in toc.
"\*(SN Scope"
.XE
.LP
c include scope.inc
T1.1 ".NH 2\n" Identification
.XS
"\*(SN Identification"
.XE
.LP
c include ident.inc
T1.2 ".NH 2\n" System Overview
.XS
"\*(SN System Overview"
.XE
.LP
c include sysoverview.inc
T1.3 ".NH 2\n" Document Overview
.XS
"\*(SN Document Overview"
.XE
.LP
c include docoverview.inc
T2. ".NH 1\n" Referenced Documents
.XS
"\*(SN Referenced Documents"
.XE
.LP
c include refer.inc
T3. ".NH 1\n" Preliminary Design
.XS
"\*(SN Preliminary Design"
.XE
T3.1 ".NH 2\n" CSCI overview
.XS
"\*(SN CSCI Overview"
.XE
.LP
c include cscioverview.inc
T3.1.1 ".NH 3\n" CSCI Architecture
.XS
"\*(SN CSCI Architecture"
.XE
.NH 4
"Declaration Tree"
.LP
"The following is a graphical description of the declaration tree."
c table ".DS B",".DE"
.\" block center the declaration tree
M3.1.1.01 declaration_tree
c table ".nf",".fi"
.\" reset the table bracket commands
c process_pufile 3_levels
.\" Use .LP to command nroff to separate paragraphs
 3.1.1.X ".NH 4\n" (CSC)
 3.1.1.X.01 ".LP\n" purpose Purpose: \n.br
 3.1.1.X.02 ".LP\n" relationships Relationship to other CSC's: \n.br
 3.1.1.X.Y ".NH 5\n" (Sub-level CSC)
 3.1.1.X.Y.01 ".LP\n" purpose Purpose: \n.br
T3.1.2 ".NH 3\n" System states and modes
.XS
"\*(SN System States and Modes"
.XE
.LP
c include states_modes.inc
.NH 4
"Invocation Tree"
.LP
"The following is a graphical description of the invocation tree."
c table ".DS B",".DE"
.\" block center the invocation tree
M3.1.2.1 invocation_tree
c table ".nf",".fi"
.\" reset the table bracket commands
T3.1.3 ".NH 3\n" Memory and Processing Time Allocation
.XS
"\*(SN Memory and Processing Time Allocation"
.XE
 3.1.3.X ".NH 4\n" (CSC)
 3.1.3.X.01 ".LP" mem_timing
T3.2 ".NH 2\n" CSCI Design Description
.XS
"\*(SN CSCI Design Description"
.XE
c no_csu
 3.2.X ".NH 3\n" (CSC)
.XS
"\*(SN #unit_type# #unit_name#"
.XE
 3.2.X.01 ".LP\n" purpose Purpose: \n.br
 3.2.X.02 ".LP\n" requirements Requirements allocated: \n.br
 3.2.X.03 ".LP\n" pseudo_code Execution control and data flow:
 3.2.X.031 ".LP\n" invocation Invocations:
 3.2.X.032 ".LP\n" called_by Called by:
 3.2.X.033 ".LP\n" in Inputs:
 3.2.X.034 ".LP\n" out Outputs:
 3.2.X.Y ".NH 4\n" (Sub-level CSC)
 3.2.X.Y.01 ".LP\n" purpose Purpose: \n.br
 3.2.X.Y.02 ".LP\n" requirements Requirements allocated: \n.br
 3.2.X.Y.03 ".LP\n" pseudo_code Execution control and data flow:
 3.2.X.Y.031 ".LP\n" invocation Invocations:
 3.2.X.Y.032 ".LP\n" called_by Called by:
 3.2.X.Y.033 ".LP\n" in Inputs:
 3.2.X.Y.034 ".LP\n" out Outputs:
 3.2.X.Y.Z ".NH 5\n" (Sub-sub-level CSC)
 3.2.X.Y.Z.01 ".LP\n" purpose Purpose: \n.br
 3.2.X.Y.Z.02 ".LP\n" requirements Requirements allocated: \n.br
 3.2.X.Y.Z.03 ".LP\n" pseudo_code Execution control and data flow:
 3.2.X.Y.Z.031 ".LP\n" invocation Invocations:
 3.2.X.Y.Z.032 ".LP\n" called_by Called by:
 3.2.X.Y.Z.033 ".LP\n" in Inputs:
 3.2.X.Y.Z.034 ".LP\n" out Outputs:
.\"
.\" This section of the TOC file produces the portion of the Software
.\" Design Document (SDD) applicable during the detailed design phase.
.\"
```

Appendix B - continued

```
T4. ".NH 1\n" Detailed Design
.XS
"\*(SN Detailed Design"
.XE
cyes_csu
cprocess_pufile 2P_levels
  4.X ".NH 2\n" (CSC)
.XS
"\*(SN #unit_type# #unit_name#"
.XE
  4.X.01 ".LP\n" pseudo_code Relationship to other CSU's:
  4.X.Y ".NH 3\n" (CSU)
.\" to prevent exceeding the maximum line width, the table of contents
.\" entry for this header is produced by the next command.
T4.X.Y.001 \n.XS\n\*(SN #unit_type# #unit_name# \n.XE
.\" this sequence produces a table of contents entry
  4.X.Y.01 ".LP\n" purpose Purpose: \n.br
  4.X.Y.1 ".LP\n" requirements Design specification/constraints: \n.br
T4.X.Y.2 ".LP\n" Design:
  4.X.Y.2.01 ".LP\n" in Input data elements:
  4.X.Y.2.02 ".LP\n" out Output data elements:
  4.X.Y.2.03 ".LP\n" local Local data elements:
  4.X.Y.2.04 ".LP\n" interrupts Interrupts and signals:
  4.X.Y.2.05 ".LP\n" pseudo_code Algorithm/Logic Flow:
  4.X.Y.2.06 ".LP\n" exceptions Error handling:
  4.X.Y.2.07 ".LP\n" invocation Use of other elements:
  4.X.Y.2.08 ".LP\n" data_types Data structures:
  4.X.Y.2.09 ".LP\n" limitations Limitations:
M5. ".NH 1\n" global CSCI Data \n.XS\n\*(SN CSCI Data\n.XE\n.LP
T6. ".NH 1\n" CSCI Data Files
.XS
"\*(SN Data Files"
.XE
T6.1 ".NH 2\n" Data File to CSC/CSU Cross Reference
.XS
"\*(SN Data File to CSC/CSU Cross Reference"
.XE
.LP
cinclude dataxref.inc
—
—
—
```

THE INTELLIGENT TEST BED: A TOOL FOR SOFTWARE DEVELOPMENT AND SOFTWARE ENGINEERING EDUCATION

Sue A. Conger*, Martin D. Fraser**, Ross A. Gagliano**,
Kuldeep Kumar*, Ephraim R. McLean*, G. Scott Owen**

*Department of Computer Information Systems
College of Business Administration
Georgia State University
Atlanta, GA 30303

**Department of Mathematics and Computer Science
College of Arts and Sciences
Georgia State University
Atlanta, GA 30303

Abstract

Software engineering can become easier to teach through the use of automated tools. Software engineering courses typically include management, methodology, quality, and maintenance topics. Within methodologies, multiple methodologies with similar but different concepts can become confusing to students. In addition, Computer Aided Software Engineering tools that automate methodologies are becoming more common and are relevant to software engineering. The Intelligent Test Bed, an automated classification of methodologies and Computer Aided Software Engineering tools, can facilitate both the teaching process and the learning process. It can be used to broaden the scope of student assignments and to provide instructional support. This paper discusses the design of the Intelligent Test Bed and its relevance to the teaching of software engineering.

Introduction

An "Intelligent Test Bed" (ITB) is a hypertext tool that categorizes and associates application development life cycle phases with software design methodologies and Computer Aided Software Engineering (CASE) tools. The ITB was developed as part of The Assessment and Development of Software Engineering Tools project sponsored by the U. S. Army Institutes for Research in Management Information, Communications and Computer Sciences (AIRMICS)*. The purpose of the ITB is to support management assessment of methods and tools in building ADA MIS applications.

Three audiences can gain useful insights into methodologies and CASE tools by use of the ITB. First, researchers and developers can determine the availability, feasibility and similarity of tools that track systems development and maintenance. Second, managers and analysts have a basis to compare available products and a means to comprehend advancements in the several fields. Third, educators can use the ITB to promote understanding of the relationships between different methodologies, CASE tools and development phases.

* Supported in part under contract DAKF11-89-C-0014.

In this paper, we focus on the rationale of design decisions for the ITB, then discuss the importance of the ITB in software engineering education. The next section presents the rationale and design decisions that preceded ITB development. Then, the actual ITB design and implementation is presented. Third, actual and potential uses of the ITB in software engineering education are discussed. Finally, several issues of continuing importance in software engineering and education are raised.

Design Rationale

The team first identified phases of an application development life cycle as one major dimension with methodology classes as the other major dimension of the Intelligent Test Bed. Considerable discussion about the granularity and detailed contents of the two main dimensions ensued. In this section, first the GSU life cycle development is discussed. Then, the design of the methodologies dimension is presented.

Life Cycle Definition

Issues relating to application development life cycle included:

1. Whose life cycle should be used?
2. Whose definitions of phases should be used?
3. Is it more helpful to be fine grained or coarse grained in relating methodologies and CASE tools to phases?

Which Life Cycle: Four main life cycles were considered before defining a "Georgia State Development Life Cycle" (GSDLC) which abstracts from the others. The life cycles considered include DoD Standard #2167a, Boehm's[1] Waterfall life cycle, IEEE[8] life cycle standard definitions, and Davis and Olson's[5] life cycle. The DoD standard is the primary life cycle used throughout the defense establishment. Boehm and IEEE are both standards used in teaching software engineering. Davis and Olson are recognized as authorities in MIS software architecture design.

Since the project focusses on an MIS audience, a wider rather than narrower scope of life cycle activities is appropriate. The scope of intended activities range from project inception through to project retirement. None of the life cycles are comprehensive although the Davis and Olson, and IEEE are both more inclusive of front-end activities than the others, while quality assurance and configuration management are only mentioned in Boehm. Thus, a comprehensive application development life cycle was defined by the team. The phases of the Georgia State Development Life Cycle include: Initiation, Feasibility, Analysis, Conceptual Design, Design, Coding and Unit Testing, Testing, Implementation, Operations and Maintenance, and Retirement (See Attachment 1). Two universal activities performed in every phase of the life cycle include Verification and validation, and Configuration management. Table 1 presents a summary of the GSDM phases with a mapping to each of the other four life cycles evaluated. This mapping is also implemented in the ITB to provide an on-line directory to those more familiar with one or another of these life cycles.

Phase Granularity: The issue of granularity of phases, expecially for testing, was also debated. The team decided that a coarse grain approach provides the most useful information to users who are otherwise required to paginate through many levels of detail simply to compare two methods. The implementation of the ITB, then, provides a fine grained level of detail on each method which is accessed through a coarse grained approach.

Phase Definition: Having a multi-functional project team caused each definition of each phase to be evaluated in terms of understanding and built-in meanings ascribed by different reference disciplines. For instance, the phase called "Conceptual Design" (See Attachment 1, 4.0) describes the period of time during which the proposed logical system is defined in detail, including designs for screens, reports, function definitions, data bases, data entry, etc. Depending on academic persuasion and work experience, this phase might also be called preliminary design, detailed logical design, external design, or software requirements specification (as distinct from software requirements analysis). Because of the potential for ambiguity, each phase definition was evaluated to ensure a rational placement within the life cycle, and a mapping to the other potential synonyms for the term. Thus, Attachment 1, which provides the phases and definitions of each phase, also contains a list of synonyms for each phase. Phase definitions reflect the activities in each phase as identified and extended by the GSU team.

Methodology Definition

Discussions similar to those described above determined the methodologies and Computer Aided Software Engineering (CASE) tools to be included in the ITB. A methodology is a comprehensive set of procedures, tools, and techniques designed to work together to transform an information system from its current state to a new, desired state. A CASE tool is an automated tool, facility or environment that is used in the production, enhancement or maintenance of software. CASE tools are rapidly automating many of the representation forms whose development is guided by the methodologies available. The issues to be decided with respect to methodologies are which methodologies to include, which CASE tools to include, and the granularity of their descriptions.

Which Methodologies:

There was immediate agreement about the major methodologies to include in the ITB: DeMarco[6], Yourdon and Constantine[11], Ward[10], Jackson[9], Finkelstein and Martin[7], and Booch[2]. The issue then was one of deciding to include or omit lesser known methodologies of authors such as Mumford, Warnier/Orr[10], Checkland[3], Langefors and others. One factor determined the decision: the extent to which research on the method would be required to instantiate the information. The time required to design one four level entry for one phase of a life cycle for one methodology is approximately three hours if the person developing the entry already has working knowledge of the method. In addition to this time, research on methods in which the designer had less knowledge was approximately two hours per four level entry. The effort was, in short, labor intensive. To further research methodologies that were heretofore unknown would require at least one week per method. Finally, a trade-off between CASE tools and methods was required for completion of the work in the allotted time. Thus, from a practical perspective, the works included needed to be limited to those already known.

Which CASE Tools: CASE tools range from documentation support (e.g. Excelerator) to application development facilitation with complete code generation (e.g. Information Engineering Facility)[4]. There are over two hundred CASE products on the market ranging from $25 to $250,000 and over. The team selected representative, high-end tools that are advertised to support most of the project life cycle, that are specifically coupled to a methodology, and/or are used in Army work. At least one tool is selected for each methodology. This selection ensures coverage for each class of methodology and is acknowledged as representative rather than as an exhaustive reference.

Granularity of Methodologies and Case Tools: The granularity discussion centered on whether or not to attempt to classify methodologies and CASE tools or to simply provide one row for each with no classification. We again opted for coarse granularity. The major argument for coarseness is that managers always want the summary first to be followed by details as needs require. With this in mind, coarseness allows classification of major activities with details at lower levels of information, and with completeness as time permits. Second, the alternative was to list of potential activities for a given phase then within cells identify the methodologies which support that activity. This was determined to be a neverending, not particularly useful method of classification

Three major classes of methodologies: Process, Data, and Object are provided in the ITB. Process methodologies represent the approach to system life cycle development that take a structured, top-down approach to evaluating processes in the problem domain and the data flows with which they are connected. The documentation produced by the process approach for analysis phase activities, for instance, (cf. Yourdon and Constantine, 1978; DeMarco, 1978) includes but is not limited to data flow diagrams, data dictionary, structured English, decision tables/trees.

The data methodology is that approach to system life cycle development that first evaluates data and their relationships to determine the underlying data architecture. Then, outputs are mapped onto inputs to determine the processing requirements[10]. The documentation produced by the information engineering data approach for analysis for instance (Finkelstein and Martin, 1979), includes entity relationship diagrams, entity hierarchy charts, process hierarchy diagrams, and process dependency diagrams.

The object oriented methodology is that approach to system life cycle development that takes a horizontal view of application objects, their allowable actions, and the underlying message passing requirements to determine the system architecture. The data/action components are encapsulated in abstract data types to promote modularity, information hiding, functional cohesion, and minimal coupling. The documentation produced by the object approach (cf. Booch, 1983; Berard, 1986) for the analysis phase includes, but is not limited to a succinct paragraph fully describing the system and its relevant components, an object list, an action list, object/action attribute lists, and message lists.

In summary, the purpose of this methodology classification is, first to provide a meta-organization for managers who may want to select from among only those methods that concentrate on one particularly complex aspect of their application, e.g. data. Next, the classification, by its definition omits some methodologies that might be added later in other categories but are omitted because of DoD current interests (e.g. Soft Systems Methodology[3].]). Thus, the use of these three method classes bounds the ITB and concentrates attention on those method classes that are included. Finally, by relating methodologies and CASE tools to one or more of these three classes, we can immediately separate those that support a method (e.g., Information Engineering Workbench, and data methods) from those that do not (e.g. Excelerator).

ITB Design

The Intelligent Test Bed provides an automated, visual method of presenting different application development methodologies and their organizing schemas. The ITB was developed under a hypertext software package. Hypertext is the name used to describe software that supports unlimited linkages between information that is located in different two-dimensional tables. At this time, the ITB is a series of such linked two dimensional tables. Although the initial ITB development concentrates on the construction of a tools data base, the intelligence in the ultimate ITB entails a system which will accept desired tool attributes, based on the life cycle, and match them to features of either available or potential tools.

Figure 1. shows the initial ITB window as a series of rows and columns, each with its own specific meaning. Each numbered column across the screen represents a phase of the GSU development life cycle as discussed in the preceding section and defined in Attachment 1. The title "GSU Phases" is a button that opens to another window that automates Table 1., showing each phase of the GSU life cycle and the associated reference phases in the four reference standards. Under each phase number heading are "R" buttons that can be used to show the name of the GSU phase if desired.

Each row of the table corresponds to one of the three classes of methodologies evaluated as part of this work: process, data, and object. Selection of a row opens a window that lists the methodologies included in that classification scheme.

Each cell of the top-level table, i.e., each particular row/column combination, accesses a list of the methodology authors whose work is used to instantiate the ITB. For instance, the conceptual design/object intersection accesses a list including Booch and Berard.

Selection of one entry from the list opens another window displaying requirements of that methodology for that particular phase of the life cycle. In the example in Figure 2. there are five requirements for object oriented conceptual design: a paragraph, object list, action list, object attribute list, and action attribute list.

For each entry in the requirements list, we can further access another window that identifies the definition, an example, and a list of supporting CASE tools. Definitions are developed at two levels. First, a definition of the design documentation element is provided. At a lower level of detail, a second set of definitions of the components of the particular documentation element are given. For instance, a data flow diagram is defined as a graphical representation of the processes, data flows, data stores, and external entities in an application system. At the lower level, each of the components (e.g. process, etc.) are also defined. The examples provided in the ITB for each requirement include examples of all possible component parts of the documentation element. All examples use an order entry system to relate the entries to an MIS application.

Navigation throughout the various levels of the ITB are provided for both expert and novice access. For novices, all screen interactions can be accomplished through the use of a mouse or other point-and-pick device. In the expert mode, command entries can be made through the use of function keys or through multiple key combinations. Macros of frequently used key sequences can also be developed.

Three options for table/window management are available. A table can be moved "behind" another one simply by clicking on the window of the desired table, which is then brought to the "front" of the screen. Tables that are no longer needed can be closed in two ways. The novice mode uses a selected option from a pull-down menu. The expert mode uses multiple key command entry.

ITB Use for Education

This section discusses the use of the ITB for software engineering eduction. Software engineering here refers to the application of well-defined policies and procedures to the development of application software. Software engineering is a superset of systems analysis and design and assumes a knowledge of representation forms, pros and cons of a variety of methodologies, in addition to alternatives for configuration management, programming language selection, and assurance of application quality. Teaching software engineering, then, requires coverage of many, unrelated topics to a sufficient level of detail that the student gains more than a passing appreciation for the subject matter.

The ITB facilitates the teaching of software engineering in several ways. First, the introduction of method classes gives a structure to discussion of multiple methodologies. Second, and most important, student learning is facilitated through concrete examples and comparisons across the methodologies. Third, with the ITB as a reference, the number of related topics can be expanded. Finally, an entire schema of ideas linking life cycle phases, methods, and CASE tools can be demonstrated. Each of these facilitating uses of the ITB is discussed in this section.

Sructuring discussion: In teaching methodologies at least one representation method from each class is discussed at GSU. Demarco, Yourdon and Constantine, and Ward provide the basis for process oriented discussions; Jackson's JSD, and Finkelstein and Martin's Information Engineering methodologies provide the basis for data oriented discussion; and Booch provides the basis for object oriented discussion. If there were no overall classification scheme for discussing these methods, the linkages from, for instance, DeMarco's analysis to Yourdon, et al's design would be less obvious and easily overlooked. In addition, focussing the discussion on methodology classes, a more natural presentation of methods in the context of hardware and software technology of the time is possible. Finally, providing students with a hierarchy of information helps them form their own mental model of methods and how they fit together that they might otherwise not form.

Facilitation of learning: Two types of knowledge are inherent in all methodologies: declarative and procedural. Declarative knowledge is that knowledge describing the steps or actions to be performed. Procedural knowledge is that knowledge describing how to perform each step. With respect to software development methodologies, procedural knowledge includes, for instance, how to make decisions about what entities to include in a system, how to select between a file and a message, or what level of abstraction to introduce a "local" file in a design.

Abstract discussion of these topics, which is in any text on the subject, does not give students any real feel for the procedural "how" of the methodology. Most class time is spent on actually developing cases that provide enough variation and difficulty to cover the most ambiguous and frequently occurring types of decisions made during a design.

The use of the ITB helps facilitate the abstract procedural learning process by giving students who desire more repetition and contact a multi-dimensional resource for comparing methodologies at their own speed and in their own way.

Also, the ITB facilitates learning by supporting a different type of student assignment. Usually, in the software engineering classes, student assignments are case problems of applications that they then analyze and design. With the ITB, student assignments can also include research on representation forms different from those covered in class, comparison of different representations of the same information across methodologies, and/or assignments to develop a representation using the ITB as the basis for the fundamental declarative and procedural knowledge. With these assignments, students experience a richer learning experience, and the researchers obtain valuable feedback on the problems and/or advantages of using a tool such as the ITB. So far, feedback is mostly positive and students are enthusiastic about having a resource that has unlimited patience and no time limits on use. The major complaint is that the ITB is not populated with enough information to support the broad range of student interests. That students make this complaint is heartening because it expresses the desire to learn more than we are currently teaching.

Expansion of Topics: As Computer Aided Software Engineering (CASE) becomes more pervasive in industry and the government, software engineering courses, which are already pressed for time in covering the requisite topics, must also expand to include CASE. Further development of the ITB to include more details on CASE tools, such as integration across phases and level of intelligence included for evaluating completeness and correctness of design, is planned. When this activity is complete, ITB use in class can be expanded to highlight the limitations of different CASE tools and to provide a basis for class evaluation of CASE products.

In the future, the ITB is one vehicle that allows coverage of more topics in less time because of its presence. When the ITB is populated with a critical mass of information on methodologies and CASE tools, classroom discussion then becomes the vehicle for identifying high level concepts, organizing information hierarchies, and procedural information which is not codified in texts or the ITB. The ITB then becomes an integral part of the curriculum to support learning of the declarative, comparative, and contrastive information for both methodologies and CASE tools. Eventually, the ITB can be worked into the syllabus to interleave classroom contacts with ITB contacts. Assignments can include both classroom and ITB related issues to be examined, cases to be developed, or comparisons to be made.

Linking Schema: One side benefit of the ITB in classroom use is that a linking schema of ideas between life cycle phases, methodologies, and CASE tools can be discussed. The discussion centers on issues about what it means for a CASE tool to support a methodology, the extent to which artificial intelligence can be applied to methodology representations in CASE tools, and the problems with technology driven fields such as management information systems. Students are encouraged to link what they have learned both from their work experience and from their other academic experiences to the dilemma that we still do not build very good systems. Some of the questions debated include:

- Are CASE tools really leading to increased productivity?
- How do you determine cost-effectiveness of systems development activities with and without CASE?
- What issues are relevant to different constituencies in systems development activities? How do methodologies address (or not) those issues?
- What should technicians be doing to improve the quality of their work products?
- What should managers be doing to improve the quality of their applications development projects?
- What should researchers be studying to improve the quality of applications development process and product?

Obviously, there are no right answers to these questions. But, by raising and discussing the issues involved, students obtain an appreciation for the problems of managers, technicians, and researchers in the area of application software development. And, by raising student awareness levels about the shortcomings and imperfections in application development knowledge, they may approach their own work with more realistic expectations.

Summary

The development of an Intelligent Test Bed was described in this paper. The ITB provides an automated reference to methodologies and CASE tools across phases of a project life cycle. The ITB design rationale was described along with the actual design of the system. Finally, a serendipitous use of the ITB for software engineering education facilitates learning, supports expansion of the course topics, increases the depth of topic coverage, and changes the type of assignments for students. The ITB allows educators to present organizing schemas for methodologically related information, and facilitates a richer discussion of both declarative and procedural knowledge associated with teaching software engineering. All of these changes appear to positively affect the learning environment.

Bibliography

1. Boehm, B. W. (1981), Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, Inc.

2. Booch, G. (1983), Software Engineering with Ada, Menlo Park, CA:Benjamin Cummings, Inc.

3. Checkland, P. B. (1981), Systems Thinking, Systems Practice, Chichester, England: J. Wiley.

4. Conger, Sue A., and Judy L. Wynekoop (1989), "A Framework for Evaluating Computer Aided Software Engineering Research and Tools", in Proceedings of the ACM Southeast Regional Conference, April, 1989.

5. Davis, G. B., and M. H. Olson (1985), Management Information Systems: Conceptual Foundations, Structure and Development, NY:McGraw-Hill, Inc.

6. DeMarco, T. (1978), Structured Analysis and System Specification, NY:Yourdon, Inc.

7. Finkelstein, C., and J. Martin (197.), Information Engineering

8. IEEE (1983), IEEE Software Engineering Dictionary, Piscataway, NJ:IEEE Press.

9. Jackson, M. (1983), System Development, Englewood Cliffs, NJ:Prentice-Hall, Inc.

10. Pressman, R., Software Engineering: A Practitioner's Approach, Englewood Cliffs, NJ: Prentice-Hall, 1988.

11. Yourdon, E., and L. L. Constantine (1978), Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, NY: Yourdon, Inc.

Appendix 1

Georgia State Development Life Cycle

Systems Life Cycle

The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The Georgia State Development Life Cycle (GSDLC), developed under the Georgia State University Development Methodology (GSDM), includes phases for initation, problem definition, feasibility, requirements analysis, conceptual design, design, implementation, testing, installation and checkout, operations and maintenance, and retirement.

0.0 Initiation

That period of time in the software life cycle during which the need for an application is identified and the problem is sufficiently defined to assemble a team to begin problem evaluation.

Initiation is motivated differently for MIS and embedded systems. Embedded systems needs are externally defined, frequently in the public arena, (e.g. national defense), which are initiated and determined to be feasible by a source outside the developing organization. MIS systems are typically initiated by a user department requesting that a system be built by an MIS department. The need for MIS systems is typically motivated by some business situation: a change in the method of business, in the legal environment, in the staffing and support environment, or by a strategic goal such as improving competitiveness in the market.

1.0 Problem Definition

The period of time in the software life cycle during which the general statement of the problem which initiated project activity is refined and made specific. All users are identified and their participation on the development team is determined.

2.0 Feasibility

The period of time in the software life cycle during which the fully defined problem is evaluated to determine its economic, technical and organizational feasibility, a preferred concept for the software product is defined, and its superiority to alternative concepts is determined.

Software Development Cycle

The period of time that begins with the decision to develop a software product and ends when the product is delivered. This cycle typically includes a requirement phase, design phase, implementation phase, test phase and sometimes, installation and checkout phase.

3.0 Analysis
Synonyms: Functional Analysis, Definition, Software Requirements Analysis

The period of time in the software life cycle during which the requirements for a software product, such as functional and performance capabilities, are defined and documented, includes the following definitional activities:

- Functional requirements
- Performance requirements
- Interface(s) requirements
- Design requirements (Data storage, response time, constraints, timing, volume, conversion, human-machine interaction etc.)
- Development standards

Outputs of this phase include a description of the current physical system (optional) , current logical system (optional), and a high-level conceptual view of the proposed logical system (e.g. context, Lo and L1 DFD's). Other outputs include descriptions of requirements for the items above specified in text, structured English, decision tables, decision trees, etc.

4.0 Conceptual Design

Synonyms: Preliminary Design, Logical Design, External Design, Software Requirements Specifications

The period of time in the software life cycle during which a user-accepted, proposed logical system is refined and detailed. Outputs of this phase include details of contents of all items defined in the previous step, including but not limited to screen designs, report designs, detailed function descriptions, normalized logical data base design, data entry requirements, etc.

5.0 Design
Synonyms: Detailed Design, Physical Design, Internal Design, Product Design

The period of time in the software life cycle during which designs for the following are created, documented and verified as satisfying requirements:

- Software architecture
- Software components and modules
- Interfaces
- Testing
- Data

Sub-system Design, Program Design (Optional Sub-Phase)

The period of time in the software life cycle during which designs for the following are created, documented and verified as satisfying requirements:

- A complete, verified specification of the control structure,
- Data structure and physical implementation scheme,
- Interface relations,
- Sizing,
- Key algorithms,
- Identification and assumptions of each program component (routine with ≤ 100 source instructions).

6.0 Coding and Unit Testing
Synonym: Computer Software Unit Testing

The period of time in the software life cycle during which the low-level elements of the software product are created from design documentation and debugged.

7.0 Testing
Synonyms: Computer Software Component (CSC), Integration, and Testing

The period of time in the software life cycle during which the components of a software product are evaluated and integrated, and the software product is evaluated to determine whether or not requirements have been satisfied. Different levels of testing might include the following:

- Unit
  Sub-routine
  Module
  Group of modules
  Program
- Sub-system
- System
  Single user
  Multiple user
- Performance
  Volume
  Response time
  CPU time
  Cost
- Quality Assurance
  Meets requirements
  Performs correctly

8.0 Implementation
Synonyms: Installation and Checkout

The period of time in the software life cycle during which a software product is integrated into its operational environement, is tested in the environment to ensure that it performs as required, and is phased into production use. This includes the completion of such objectives as data conversion, installation, training, and total program/system implementation.

Software Development Cycle End

9.0 Operations and Maintenance

The period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements.

10.0 Retirement
Synonym: Phaseout

The period of time in the software life cycle during which support for a software product is terminated, and a complete transition of the functions performed by the product is tranferred to its successor system.

Universal Phase Activities

Verification and Validation

An integral part of the achievement of each life-cycle subgoal is the verification and validation that the intermediate software products do indeed satisfy their objectives.

Verification: To establish the truth of correspondence between a software product and its specification.

Validation: To establish the fitness or worth of a software product for its operational mission.

Configuration Management

The project team librarian is able at any time to provide a definitive version (i.e., baseline) of a document or phase product, such as the requirement specification. These baselines form a unifying link between management and control of the software process, and the management and control of the software product. Configuration management refers to the management of change after an agreed upon version is reached. Only the Project Librarian maintains the definitive version of each product.

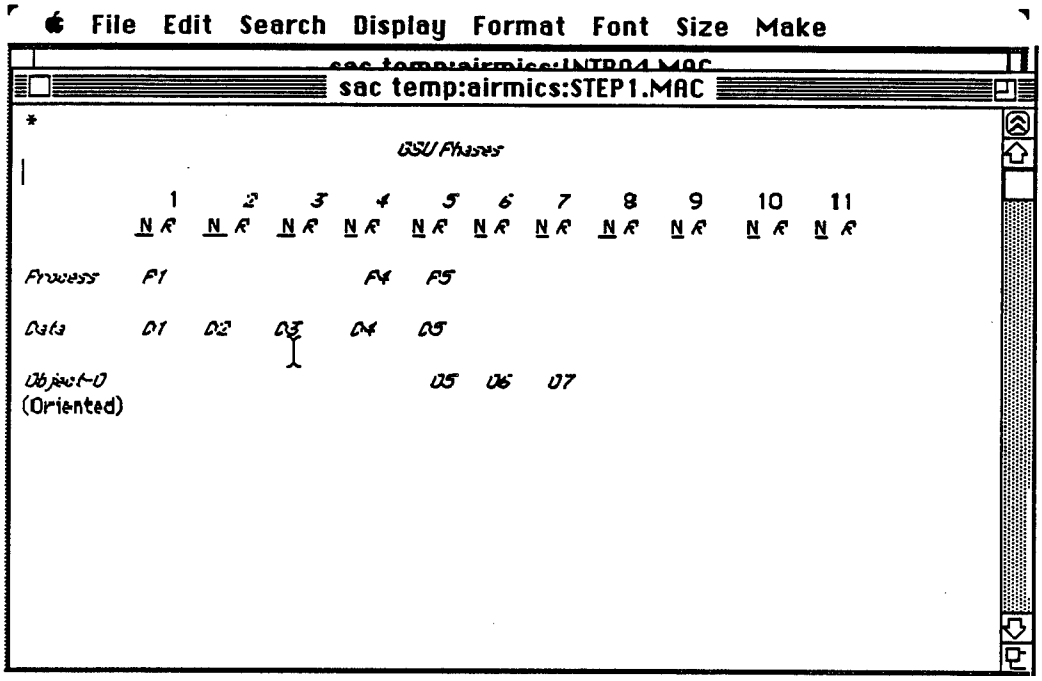


Figure 1.
Intelligent Test Bed Main Menu

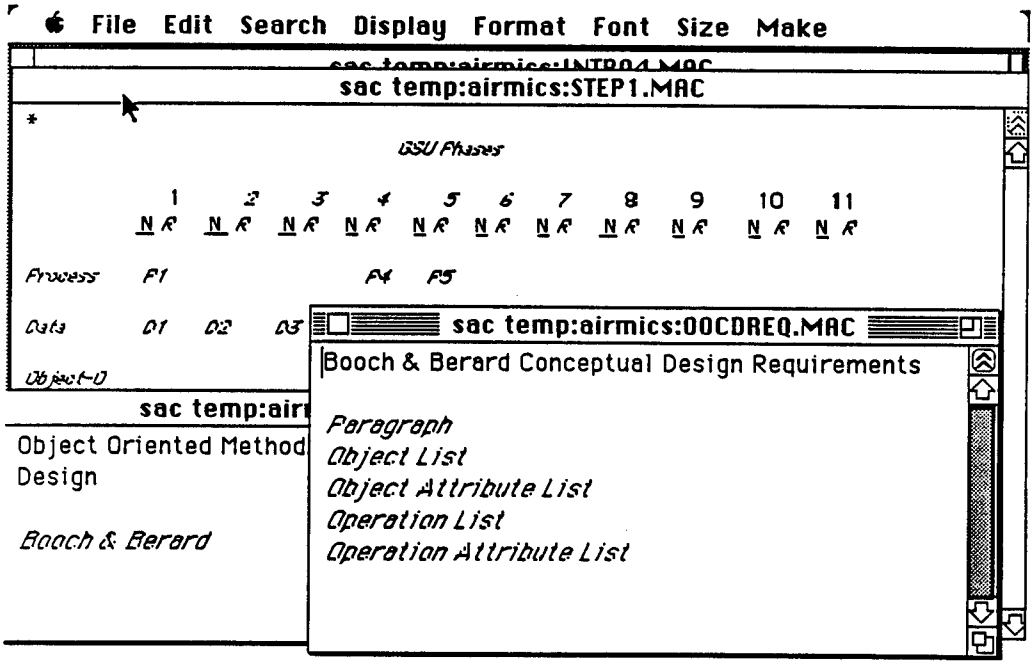


Figure 2.
Sample Requirements List from Object-Orientation

GSDM Phase	DOD 2167a*	Boehm	IEEE**	Davis and Olson ***
System Life Cycle Initiation			SLC-Requirements	
Problem Definition			SLC-Requirements	Definition-Proposal
Feasibility		Feasibility	SLC-Requirements	Definition-Feasibility
Development Life Cycle Analysis	Software Requirements Analysis	Requirements	DLC-Requirements	Definition-Requirements Analysis
Conceptual Design	Preliminary Design	Product Design	DLC-Design	Definition-Conceptual Design
Design	Detail Design	Detail Design Physical Data Base Design	DLC-Design	Development-Physical System
Code/Unit Test	Code/CSU Test	Code	DLC-Implement	Development-Programs
Testing	CSC Integration CSCI Test	Integration	DLC-Test	Development-Procedures
Installation and Checkout	System Integration	Implementatio	DLC-Implementation	Installation-Conversion and Testing
Operation/Maintenance		Maintenance	SLC-Operations/ Maintenance	Installatin-Operations/ Maintenance
Retirement		Phaseout	SLC-Retire	Installation-Post Audit
Continuing Activities -- All Phases				
Validation/Verification	Review	Validation & Verification		
Configuration Management	Not clear	Configuration Management		

Table 1. Georgia State Development Life Cycle

* CSU=Computer Software Unit [DOD #2167a,1985]
CSC=Computer Software Component [DOD #2167a,1985]
CSCI=Computer Software Component Integration [DOD #2167a,1985]

** SLC = System Life Cycle [IEEE, 1983]
DLC = Development Life Cycle [IEEE, 1983]

*** Davis and Olson [1985] have three major subdivisions in their methodology Definition (Def), Development (Dev), and Installation (Install)



Sue A. Conger is an Assistant Professor of Computer Information Systems at Georgia State University. She received her Ph.D. from new York University where was also served on the faculty. She has worked in management information positions for Mobil, Lambda Technology, Chase Manhattan Bank, Educational Testing Service, and the U. S. Department of Agriculture. She has a B.S. in Psychology from Ohio State University, and an MBA in Finance from Rutgers University. Her research interests include systems analysis, design methodologies, and CASE tools.



Martin D. Fraser is an Associate Professor in the Department of Mathematics and Computer Science at Georgia State University. He has worked as Statistical Analysis Manager, American Greetings, Corp., Cleveland, an Information Systems Member at Western Electric, St. Louis, and a Software Analyst (Captain, USAF) at the Satellite Test Center, Sunnyvale. He has a B. S. and M. S. degrees in Mathematics, and a Ph.D. in Mathematics, major in Statistics, from St. Louis University. His research interests are in computer science and statistics.



Ross A. Gagliano is an Associate Professor of Computer Science at Georgia State University. He received his B.S. degree from the U.S. Military Academy, an M.S. in Physics from the U. S. Naval Postgraduate School, and an M.S. and Ph.D. in Information and Computer Science from the Georgia Institute of Technology. Prior to coming to GSU, he was a Senior Research Scientist at the Georgia Tech Research Institute. His major areas of interest include discrete element modeling, simulation, and software engineering.



Kuldeep Kumar is an Assistant Professor of Computer Information Systems at Georgia State University. He received the Ph.D. degree from McMaster University. His research interests include function point analysis, CASE tools, and MIS life cycle analysis. He has a recent article in the Communications of the ACM and two more accepted for publication in that journal.



Ephraim R. McLean is Professor and Georgia E. Smith Chair in Information Systems in the College of Business Administration at Georgia State University. Prior to joingin the Georgia State faculty, he was the Chairman of the Computers and Information Systems Area and the Director of the Computers and Information Systems Research Program at the UCLA Graduate School of Management. He has an undergraduate degree in engineering from Cornell University, and a Ph.D. from M.I.T.'s Sloan School of Management.



G. Scott Owen is Professor of Mathematics and Computer Science at Georgia State University. His research interests are in computer graphics, visualization techniques, software engineering, Ada technology, computer science education, and the application of computers to science and mathematics education. He received his B.S. from Harvey Mudd College in California, and his Ph.D. from the University of Washington in Seattle.

An Implementation of Deterministic Execution Testing and Debugging Tools For Concurrent Ada Programs

Wayne N. Taylor[1], Richard H. Carver[2], and K. C. Tai[3]

Abstract

An execution of a concurrent program P non-deterministically exercises a sequence of synchronization events called a synchronization sequence, or SYN-sequence. This non-deterministic execution behavior creates the following problems during the testing and debugging phases of P: (1) When testing P with input X, a single execution is insufficient to determine the correctness of P with input X, and (2) when debugging an erroneous execution of P with input X, there is no guarantee that this execution will be repeated by executing P with input X.

These problems can be solved by forcing a deterministic execution of P according to a given SYN-sequence; this is referred to as deterministic execution testing and debugging. In this paper, we present the design and implementation of several tools that we have developed to support deterministic execution testing and debugging of concurrent Ada programs. We discuss the features provided by these tools and their limitations, and we describe how to use these tools to test and debug concurrent Ada programs.

1. Introduction

Let P be a concurrent Ada program. An execution of P non-deterministically exercises a sequence of synchronization events called a feasible synchronization sequence, or SYN-sequence. This non-deterministic execution behavior creates the following problems during the testing and debugging phases of P: (1) When testing P with input X, a single execution is insufficient to determine the correctness of P with input X, and (2) when debugging an erroneous execution of P with input X, there is no guarantee that this execution will be repeated by executing P with input X. These problems can be solved by forcing a deterministic execution of P according to a given SYN-sequence; this is referred to as deterministic execution testing and debugging.

[1] IBM, P.O. Box 12195, B501, D120, Research Triangle Park, North Carolina 27709, (919) 254-6041.

[2] Dept. of Computer Science, Box 8206, North Carolina State University, Raleigh, NC 27695-8206, (919) 737-2858.

[3] Software Engineering Program, Rm 304, National Science Foundation, 1800 G. St., N.W., Washington D.C. 20550, (202) 357-7345.

Deterministic Execution Debugging

The purpose of deterministic execution debugging is to allow an erroneous execution to be replayed so that debugging information can be collected. Such information can be collected as it is needed by using a traditional interactive debugger in conjunction with the replay facility.

Deterministic execution debugging of concurrent Ada programs involves the following three problems:

(a) **SYN-sequence definition**: How to define a SYN-sequence of a concurrent Ada program?

(b) **SYN-sequence collection**: How to collect the SYN-sequence of an execution of a concurrent Ada program?

(c) **SYN-sequence replay**: How to replay the SYN-sequence of a previous execution of a concurrent Ada program?

Deterministic Execution Testing

As mentioned above, a single execution of P with input X is insufficient to determine the correctness of P with input X. (A single execution of P with input X exercises only one of possibly many feasible SYN-sequences of P with input X.) A commonly used approach, called multiple execution testing, is to execute P with input X many times. However, multiple executions of P with input X may leave some critical paths uncovered. Also, this approach is ineffective for verifying corrections of errors and for regression testing. To increase our confidence in the correctness of P with input X, it is necessary to test P with a set of IN_SYN tests of the form (X,S), where X and S are an input and a SYN-sequence, respectively, of P. For a selected test (X,S) for P, we determine whether S is feasible for P with input X and if so, we examine the results produced by P with input X and SYN-sequence S. Such testing is referred to as deterministic

execution testing.

Deterministic execution testing is needed not only for detecting errors in P but also for verifying corrections made to P. Assume that an execution of P with input X has exercised a SYN-sequence S which is invalid according to P's specification. After an attempt has been made to correct the error(s) detected by this erroneous execution, we need to verify that S is infeasible for the corrected version of P with input X. Deterministic execution (regression) testing is also needed to verify that corrections made to P do not produce unexpected effects for previous successful executions of P.

Deterministic execution testing involves the following problems in addition to the SYN-sequence definition and collection problems discussed earlier:

(d) **SYN-sequence selection:** How to select SYN-sequences that are effective for error detection.

(e) **SYN-sequence feasibility:** How to determine whether or not a SYN-sequence is feasible for a concurrent program with a given input.

Although deterministic execution testing has advantages over multiple execution testing, it requires additional effort for selecting SYN-sequences and determining the feasibility of SYN-sequences. The amount of such effort can be reduced by combining deterministic execution and multiple execution testing. More discussion of the deterministic execution testing approach and its combination with multiple execution testing is given in [Tai85,Tai89b].

The organization of the paper is as follows. In section 2, we discuss a language-based approach to deterministic execution testing and debugging. In section 3, we present the design and implementation of several tools that we have developed to support deterministic execution testing and debugging of concurrent Ada programs. We discuss the features provided by these tools and their limitations. In section 4, we describe how to use these tools to create an environment for testing and debugging concurrent Ada programs. Section 5 concludes this paper.

2. A Language-Based Approach to Deterministic Execution Testing and Debugging

A language-based approach to deterministic execution testing and debugging of concurrent Ada programs is as follows. First, we define the format of a SYN-sequence of an Ada program **in terms of the synchronization constructs available in Ada** so that a SYN-sequence provides sufficient information for deterministic execution. In [Car89a,Car89b,Tai89a], a SYN-sequence of a concurrent Ada program is defined as a sequence of rendezvous events called a rendezvous sequence or R-sequence. Next, we develop R-sequence collection, replay, and feasibility tools which are based on **source transformation**. The transformation tools are illustrated by the diagram in Fig. 1. A concurrent Ada program P is transformed into another Ada program P', P", or P*, so that during an execution of P', P", or P*, an R-sequence of P can be collected or replayed, or its feasibility can be determined. (Note that it is possible to combine these three tools into a single tool that transforms program P into a program that can serve as P', P" or P* depending on the programmer's choice.)

The transformation of P involves two parts; one is to transform each "rendezvous statement" which performs a rendezvous event (e.g. accept statement, entry call statement, select statement) and the other is to insert a "control" package into P, which either collects, replays, or determines the feasibility of an R-sequence. The definition of an R-sequence for concurrent Ada programs and the source transformations for R-sequence collection, replay, and feasibility are discussed in [Car89a,Car89b,Tai89a].

3. The Transformation Tools

The transformation tools are comprised of three major components: the lexical analyzer, the parser, and the transformation routines.

Lexical analyzer: The lexical analyzer reads as input a concurrent Ada program P character by character and groups individual characters into tokens (identifiers, reserve words, etc.). The lexical analyzer used in the transformation tool was generated by a UNIX tool called LEX [Les75]. The input into LEX is a specification file that contains a description of the tokens to be recognized and the actions to be performed when each token is found. The tokens are described using regular expression notation. The output of LEX is a C program that models a finite automaton constructed from the regular expressions. This C program performs the lexical analysis.

Parser: Given a formal Ada syntax specification, usually called an Ada grammar, the parser reads tokens in P (passed from the lexical analyzer) and groups them into units as specified by the grammar. When a statement is recognized by the parser, the transformation routines associated with this statement are executed. The parser

used in the transformation tools was generated by a UNIX tool called YACC [Joh75,Sul86]. YACC is a sophisticated software tool for the generation of shift-reduce LALR(1) parsers. YACC accepts a specification file representing a syntax specification and produces a C program that will parse an input stream according to the syntax specification. The specification files used as input to YACC and LEX were obtained from the SIMTEL20 Ada Software Repository [Con87]. Some minor changes were made to the Ada grammar to facilitate the transformation routines.

Transformation routines: When the parser recognizes a statement in P, it calls the transformation routines to provide the appropriate statement transformation. This process continues until P is exhausted. Each transformation tool has a separate set of transformation routines which are based on the source transformations described in [Car89a,Car89b,Tai89a]. Combining the lexical analyzer and parser with the transformation routines produces a transformation tool for a given set of source transformations. These transformation tools run in a UNIX environment; currently, we are running the tools on a Sequent BALANCE 8000 multiprocessor system. Each tool is approximately 56k bytes in size.

The concurrent Ada programs submitted to the transformation tools must be free from compilation errors. In addition, there are several restrictions that the tools impose on the programs. Failure to adhere to these restrictions will cause the transformation tools to fail.

Restrictions on Program Format: There are several restrictions on the format of the programs submitted to the tools. The major restrictions on program format are
- separate compilation is not supported,
- the main procedure must be explicitly identified by the programmer, and
- certain identifiers are reserved for use by the tools.

Restrictions on Ada's Tasking Facilities: Ada allows tasking and non-tasking constructs to be freely combined. The tools only support the basic tasking and non-tasking facilities. The major restrictions are
- tasks may be declared as single tasks or singly dimensioned task arrays only,
- bounds in task array declarations must be integer constants,
- a task array index must be a simple integer variable or constant,
- the following are not supported:
  - abort statements
  - records with task components,
  - access types for tasks,
  - dynamic task arrays, and
  - overloading of procedures and entries.

Efforts are underway to remove some of these restrictions.

4. An Environment for Testing and Debugging Concurrent Ada Programs

Fig. 2 shows the use of the R-sequence replay, collection, and feasibility tools to create an environment that supports multiple execution testing and deterministic execution testing and debugging of concurrent Ada programs. A complete description of the environment is given in [Tai89b]. Here, we describe a simplified three step approach to testing and debugging a concurrent Ada program P in this environment.

Step 1: Transform P into P' for R-sequence collection. Using multiple execution testing, execute P' repeatedly in order to detect errors in P. If an error is detected then go to Step 2.

Step 2: Transform P into P'' for R-sequence replay. Execute P'' with the R-sequence(s) collected during Step 1 and collect debugging information in order to locate the cause of the error. After the error has been located, make the necessary correction to P and call the resulting program Q. Go to Step 3.

Step 3: Transform Q into Q* for R-sequence feasibility. In order to verify the correction made to P, execute Q* with the R-sequences collected in Step 1. Valid R-sequences that were collected in Step 1 should remain feasible and invalid R-sequences should be infeasible.

As we discussed in section 2, we can combine multiple execution testing and deterministic execution testing in Step 1. This approach is also supported by the environment.

5. Conclusion

In this paper, we have described our implementation of three deterministic execution testing and debugging tools for concurrent Ada programs. These tools can be used to create an environment that supports multiple execution testing and deterministic execution testing and debugging of concurrent Ada programs. We described a simplified approach to testing and debugging concurrent Ada programs in this environment.

The tools impose several restrictions on program format and structure. Work is in progress on removing some of these restrictions. We plan to make the tools

available to the Ada community.

Deterministic execution testing and debugging can be accomplished by using either a language-based approach, which was discussed in section 2, or an implementation-based approach. The implementation of a concurrent language L usually consists of three components: the compiler, the run-time system, and the operating system. An implementation-based approach to deterministic execution testing and debugging is to modify some or all of the three implementation components of language L so that during an execution of a program P (written in L) with input X, the sequence of synchronization events can be controlled according to a given SYN-sequence of P. An implementation-based solution uses less run-time overhead, but it is usually more difficult and costly to implement and it may be difficult if not impossible to port such tools from one implementation to another. Furthermore, due to the complexity of the implementation components and the level of detail, it may be difficult to express the basic design of an implementation-based tool in such a way as to be useful for developing the tool on possibly radically different implementations. Thus, an implementation-based approach may force tool developers to reinvent the wheel for each new implementation. The source transformations developed using a language-based approach can be used as a high-level design for implementation-based tools.

References

[Car89a] Carver, R. H., "Testing, Debugging, and Analysis of Concurrent Software," Ph.D. thesis, North Carolina State University, May, 1989.

[Car89b] Carver, R. H., and Tai, K. C., "Deterministic Execution Testing of Concurrent Ada Programs," Proc. TriAda89, October, 1989, 528-544.

[Con87] Conn, R., "The Ada software repository and software reusability," Proc. of the Joint Ada Conference, 1987, 45-53.

[Joh75] Johnson, S. E., "Yacc: Yet Another Compiler-Compiler," Computing Science Technical Report No. 32, Bell Laboratories, 1975.

[Les75] Lesk, M. E., "Lex - A lexical analyzer Generator," Computing Science Technical Report No. 39, Bell Laboratories, 1975.

[Sul86] Sullivan, B. M., "Yacc Reference Manual," Technical Report No. TR-86-12, The Wang Institute of Graduate Studies, 1986.

[Tai85] Tai, K. C., "On testing concurrent programs," Proc. COMPSAC 85, Oct. 1985, 310-317.

[Tai89b] Tai, K. C., and Carver, R. H., "Deterministic Execution Testing and Debugging of Concurrent Programs," Proc. 7th Annual Pacific Northwest Software Quality Conference, September, 1989, 170-182.

[Tay88] Taylor, W. N., "Debugging Concurrent Ada Programs," M.S. thesis, North Carolina State University, 1988.

Wayne Taylor received his M.S. in computer science from North Carolina State University. Currently, he is working on the development of local area networks.

Richard Carver received his Ph.D. in computer science from North Carolina State University. His research interests are in the area of software specification and validation, especially for concurrent software.

K. C. Tai received his Ph.D. in computer science from Cornell University. He is on leave from North Carolina State University where he is a Full Professor of Computer Science.

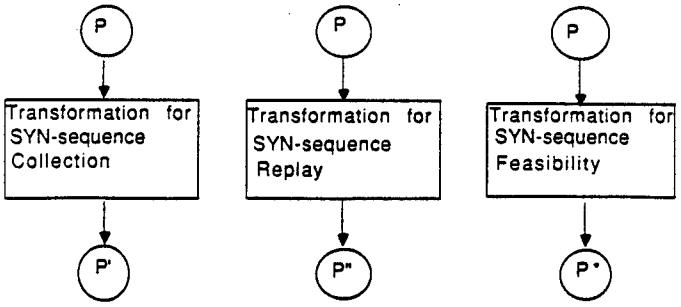


Figure 1. Transformation of a Concurrent Ada Program

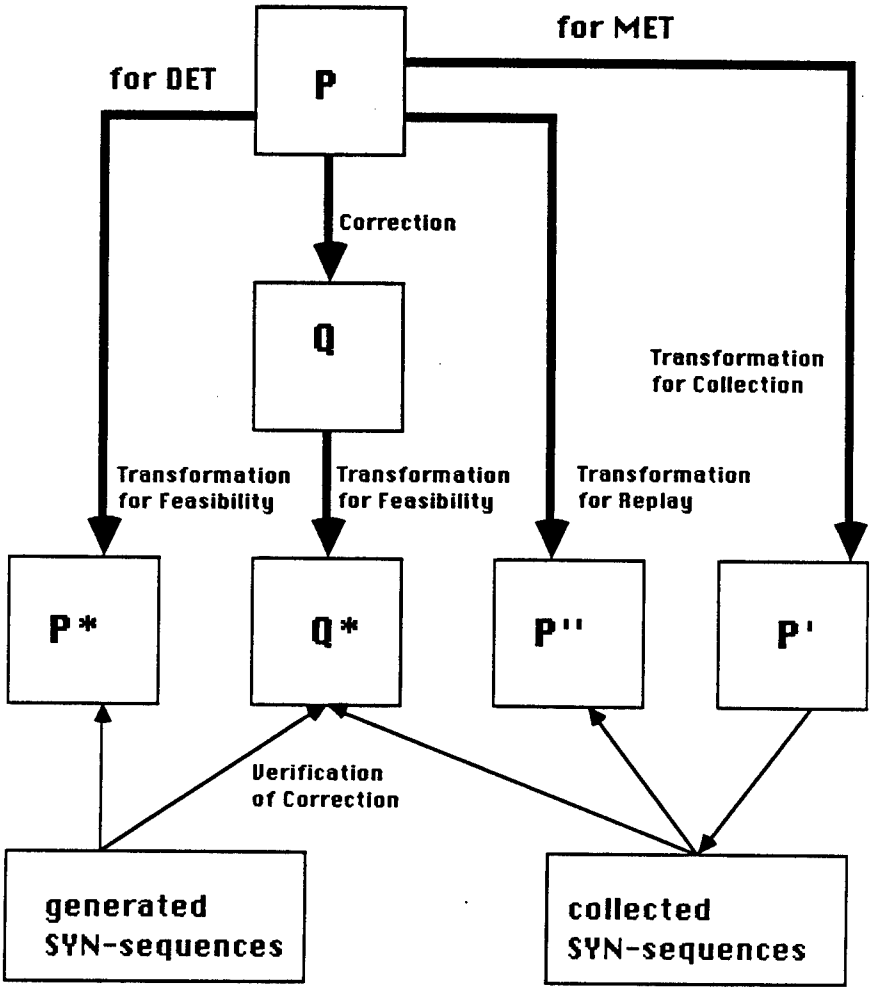


Figure 2. A Testing and Debugging Environment for Concurrent Ada Programs

SOFTWARE REUSE: MANAGERIAL AND TECHNICAL GUIDELINES

James W. Hooper
Computer Science Department
The University of Alabama in Huntsville
Huntsville, Alabama 35899

Rowena O. Chester
Martin Marietta Energy Systems, Inc.
P.O. Box 2003
Oak Ridge, Tennessee 37831

ABSTRACT

It is evident from experience that only incidental software reuse will occur unless an organization plans and institutes specific policies to achieve reuse. We have considered many instances of reuse practice and research findings, and from them have distilled a set of recommendations to help in establishing and carrying out software reuse practices. The recommendations emphasize reuse as an integral part of an effective software engineering development and maintenance process.

Within the category of managerial guidelines, top-level guidelines are offered for establishing a reuse program. In support of these guidelines, consideration is given to some existing impediments to reuse along with approaches for resolution, and to the creation of positive incentives for reuse. Emphasis is placed on instituting a software development and maintenance process incorporating reuse.

Within the category of technical guidelines, discussion is provided on such specific issues as domain analysis, preparing reusable components (spanning the entire software life cycle), assuring component quality, and classifying and storing components. Guidelines are suggested for developing Ada code modules for reuse. Operational issues in the reuse of software are considered, including searching, retrieving, understanding, assessing, adapting, and assembling components.

INTRODUCTION

The critical problems of low software productivity and poor software quality have led to research and experimentation in software reuse to help alleviate the problems. Sufficient progress has now been made that specific recommendations can be made for integrating reuse into the process of software development and maintenance. Availability of the Ada language has served as a strong motivator for reuse, and as a primary mechanism for reusability.

Software reuse has been characterized as "an act of synthesizing a solution to a problem based on predefined solutions to subproblems" (Kang 1987). This definition emphasizes the reuse of predeveloped solutions across the entire life cycle--with the greatest leverage expected to come from the reuse of products from earlier life cycle phases. The implications of this definition include: recognizing the desirability of retaining certain predefined solutions for reuse, codifying and retaining the solutions, recognizing the availability of potentially-applicable components during solution of a problem, and adapting and composing components into software which provides a valid problem solution.

Considerable research and experimentation have now been achieved in all these aspects of reuse. Important research is now underway at the Software Engineering Institute (SEI), the Software Productivity Consortium (SPC), the Microelectronics and Computer Technology Corporation (MCC), government and industrial laboratories, and universities. Now in place are such support facilities as the DoD Ada Software Repository and AdaNET. Numerous corporations and government agencies have achieved considerable success in reuse, across a wide spectrum of applications.

The guidelines presented in this paper, as well as much of the discussion, are based on the research report (Hooper and Chester 1989), which may be consulted for further background and status information, and more complete discussion of the issues. The next section provides guidelines addressing managerial issues of reuse. Following that is a section addressing technical issues. In the Conclusions section we offer some thoughts on the "bootstrapping" process.

MANAGERIAL GUIDELINES

In the following subsection we pursue some of the managerial issues that must be dealt with in order to achieve success in software reuse, discussing approaches for overcoming existing disincentives to reuse

and for providing positive incentives. In the final subsection of this section we focus on the important issue of integrating reuse with the software development and maintenance process.

Managerial Issues and Approaches

There are many issues to resolve to achieve a successful software reuse program. There are organizational, economic, legal, and sociological issues (Fairley et al 1989), and some technical reuse issues as well. The exact nature of issues will naturally vary from one organization to another, and thus means to resolve issues must be tailored to each organization. For example, U.S. Department of Defense (DoD) agencies have different legal issues to face than do their contractors. Thus we consider a broad spectrum of issues, with suggestions for eliminating (or alleviating) disincentives and for introducing positive incentives.

Management and Organization: Top-level management must take positive action to make software reuse a reality. This means much more than just issuing an edict that software reuse will occur. It means committing necessary resources to bring about a different way of approaching software development and maintenance--including a different process and tools, a well-trained support staff, and an adequate initial library of reusable components. It means spending money "up front", for later gains. Management must be prepared to wait for some period of time for the investment to begin to pay off. Realistic goals must be set, and risks must be accepted by management. Technical personnel must know that management is firmly committed to reuse, that perfunctory efforts are not acceptable, and that success in reuse will bring positive career rewards.

Too many good new ideas die within organizations because management expects technical personnel to carry all the burden--extra work over and above usual duties, with little or no support from management, and sometimes only negative incentives. One can guarantee that software reuse will fail if the burden is on technical personnel alone--they simply cannot accommodate the necessary efforts to achieve success in reuse unless management provides resources--including full-time support staff for reuse, and realistic assignments that take into account the expectation of reuse activities.

Top-level management must effect initial planning and decision-making for reuse, including decisions on scope for reuse (what organizational component(s) will be involved, what life-cycle phase(s) will be supported for reuse, what application areas will be addressed).

Government managers are in an especially strong position to bring about reuse--not only in-house, but on the part of their contractors. There are presently substantial difficulties to overcome, as we will discuss in the following sub-sections, but there is the very important prospect of markedly improved software productivity and quality as the outcome of the undertaking. The best motivator of all, for management and technical personnel, will be witnessing success. To that end it is very important that management carefully plan their reuse program before initiating it, including choice of early projects that seem likely to have a good payoff from reuse.

Fairley (1989) has conducted research to determine how best to organize to accommodate software reuse. He discusses functional, matrix and project approaches, and concludes that the best approach is a "reuse matrix" structure, with domain-specific reuse groups. One or more individuals from domain reuse groups would be assigned to each project as a reuse "facilitator" for software relating to his/her domain of expertise. Each domain reuse group would be responsible to maintain a library of reusable components, to educate and assist in the use of available components for the domain, and to encourage creation of additional reusable software. The following guidelines summarize some recommendations; most are from Fairley.

Upper-level management must set reuse goals, create an organizational infrastructure to support software reuse, establish policies, and provide necessary resources.

Mid-level management must develop procedures, populate the organizational structure, allocate resources, and establish controls and metrics to achieve goals.

Project-level management and technical personnel must carry out the reuse/reusability activities in individual projects.

Establish an organizational entity whose charter is to promote reuse considerations at the corporate level.

Establish reuse matrix structure and domain reuse groups.

Structure software development by domains amenable to reuse considerations.

Establish strong connnection between reuse and maintenance activities.

Provide different types of training for managers, developers, and domain reuse specialists.

Make personnel assignments that take reuse and reusability into account.

Assign reuse facilitators to development groups.

Allow two to three years start-up period to achieve economic advantage from reuse program.

Provide corporate financial "safety net" for projects practicing reuse; provide funding for generation of reusable components.

Government managers should take initiative to influence achievement of reuse within their own organizations and within contracts they direct.

Organizational Behavior: There are a number of disincentives at present for personnel to cooperate in using available existing software and in preparing and supplying their own products for use by others. Fairley (1989) mentions some disincentives to using reusable components: "not invented here" syndrome, more "fun" to build than adapt, differing styles and quality criteria, and technical constraints on the product. He mentions the following as disincentives to contribute reusable components: schedule constraints, stylistic issues, reward structure, peer pressure. Fairley strongly emphasizes the importance of reuse being a corporate concern, with impetus for reuse coming from top-level management. I.E., a corporate culture must be established that emphasizes reuse.

To encourage individuals to participate effectively in reuse activities, one approach is to provide behavioral incentives, such as sharing of cost savings, time off, bonuses, free dinners, reserved parking, public awards, and reuse bonus points. More effective, according to Fairley (1989), is to seek to enhance psychological job satisfaction to motivate willing participation in reuse. Examples he suggests are: provide corporate-level support for reuse as a meaningful endeavor; reuse work products at higher levels of abstraction than code; carefully define reuse roles for all involved personnel; determine job performance through a metrics program; provide for professional growth through job rotation and skills acquisition; practice information hiding and object-oriented development to provide some autonomy; emphasize/train to give and accept constructive criticism. Fairley makes the important point that many of these approaches to enhance participation in reuse also promote good software engineering practices.

The above suggestions are briefly summarized in the following guidelines.

Provide incentive rewards to participate in reuse.

Seek to enhance psychological job satisfaction to motivate willing participation in reuse.

Contractual and Legal Considerations: There are presently substantial disincentives to software reuse relative to contracting arrangements and ownership rights. Cost-Plus contracts at present provide disincentives to contractors to reuse software products--in fact, they provide positive incentives to the contrary, to redevelop rather than reuse. Firm Fixed Price (FFP) contracts are worse as to creating reusable products--at least, for delivery to the customer--since it costs extra money to prepare products for reuse. There might be motivation to reuse available products (if such existed) in a FFP contract, as this could lead to greater profit. It could also be, of course, that effort could be expended under internal company funds to generalize some software developed under the contract for the company's later advantage.

The Government must make it attractive to companies to create reusable software and to reuse software, by providing incentives. A suggestion is to provide extra awards for contributions to, or applications of, reuse. Companies could be required (or encouraged) to address applicable reusable software in submitted proposals. If such changes in government procurement policy are to be made, then accompanying changes must occur in how government projects are funded. At present no funds are allocated to a project to prepare software components for future use.

It appears that a company must always retain some proprietary software, for competitive advantage. To the extent software is placed in a repository for general use, a company must be duly compensated for the expected loss of the revenue that retention of the software would have given them (i.e., "royalties" must be paid). Clearly DoD and other agencies must come to grips with the legal and contracting issues involved, for the national benefit.

There are many legal issues that are as yet unresolved pertaining to software reuse, concerning ownership rights in software (for example, when developed under contract to the government), and

liabilities for errors when software is reused by other organizations. They may either be resolved by legislation, by stipulations of individual contracts, or by the results of litigation over time. More research is needed to formulate specific recommendations for resolving this important issue.

The above observations lead us to the following guidelines.

Seek contractual means to require or encourage contractors to create reusable software and to reuse existing software.

Establish and enforce reuse practices within government development groups.

Require reuse within a group of related contracts (e.g., by a prime contractor and subcontractors).

Seek means to alter project funding approaches to encourage creation of reusable software.

Seek resolution to the legal issues of potential liability and partial ownership.

Financial Considerations: It costs more to prepare software for reuse than for a single use--due to extra effort required to generalize the components, conduct extra testing, provide adequate documentation, and to classify and store them for reuse. One could easily spend more money trying to reuse ill-suited components than in preparing custom software. In order to intelligently assess opportunities for reuse and reusability, cost predictions must be made. Underlying the costing must be a database of financial data on previous projects. Unfortunately most organizations have no such record of project performance data, but it can be built up over time. While there are several cost models in use for the general software process, little work has been done until very recently in cost modeling for reuse/reusability assessment (see Fairley et al 1989 for results of recent research).

Until productized reuse-related cost modeling tools and supporting organizational data are available, organizations must resort to ad hoc methods for determining the likelihood of the cost-effectiveness of specific opportunities for reuse and reusability. Unfortunately this is no different than the way most organizations already operate relative to software cost predictions, since available software cost models are apparently not used in most organizations.

In view of these observations, we offer the following guidelines.

Establish mechanisms to accumulate an organizational database of financial data relative to software production and maintenance, including reuse activities.

Provide and use cost modeling tools in concert with organizational data for reuse/reusability decision assessments.

Software Development and Maintenance Incorporating Reuse

Software Process Concepts and Issues: Effective software reuse cannot occur except as part of an integral framework for software development and maintenance. DOD-STD-2167A guides development of software for the government; and although reuse activities may be accomplished within the 2167A framework, reuse is not directly addressed or encouraged. The problem is no doubt more fundamental than the DoD standard, per se, in that if most government and industrial organizations practice reuse at all, it is in an ad hoc, non-systematic way. Each organization needs a systematic approach to the development and maintenance of software that includes reuse and reusability as important, integral, natural, inescapable elements.

The following definition is from Humphrey (1989):

> The SOFTWARE ENGINEERING PROCESS is the total set of software engineering activities needed to transform a user's requirements into software.

Humphrey comments that this process may include, as appropriate, requirements specification, design, implementation, verification, installation, operational support, and documentation, as well as temporary or long term repair and/or enhancement (i.e., maintenance) to meet continuing needs.

There are many misgivings about the "waterfall" process model, and various alternatives have been suggested. Among them are the Spiral Model (Boehm 1988) and Boehm's Ada Process Model; and the prototyping life cycle paradigm (Yeh and Welch 1987). Simos (1987) observes:

> "What is needed is a process model that allows for iteration between the top-down, "problem-driven" approach and a bottom-up, "parts-driven" perspective ... Such a process model would correspond more closely to the real state of practice in software development than the current model, and would at least initially have a less prescriptive, more descriptive, flavor."

A Generic Reuse/Reusability Model: Kang (1987) suggests a refinement to the DOD-STD-2167A life cycle by identifying reuse activities applicable to each phase. He describes a "generic reuse activity model", developed at the Software Engineering Institute (SEI), as the base model for use in refining each phase. He proposes the following four steps to be performed at each phase:

1. Understanding the problem and identifying a solution structure based on the predefined components.
2. Reconfiguring the solution structure to improve the possibility of using predefined components available at the next phase.
3. Acquiring, instantiating, and modifying predefined components, and
4. Integrating the components into the products for this phase.

Kang includes an example of the application of this generic reuse model to the software requirements phase. The reuse model would be applied to refine the other phases of the 2167A life cycle in much the same way. The SEI is currently evaluating experimentally this approach to integrating reuse into the software process.

As the generic reuse activity model now stands, it does not deal with the contribution of reusable components to a library. The following is suggested as Step 5 (thus creating a five-step "reuse/reusability model"):

5. Evaluating reusability prospects of components that must be developed and components obtained by modifying predefined components, for contribution to the set of predefined components.

This activity would include consideration of the advisability of generalizing the components for improved reusability.

An increasingly-important methodology is object-oriented design (OOD). It is considered by many researchers to be promising relative to software reuse. Booch (1987) combines object-oriented design with component reuse--and has spurred a great deal of interest in the promise of reuse. Another influential advocate of OOD as the basis for reuse is Meyer (1987). His Eiffel language serves as the basis for his reuse research and recommendations.

Now we provide the following guidelines.

Initiate action to establish a software engineering process that includes reuse and reusability as integral elements.

Augment DOD-STD-2167A with refinements that specifically support and encourage reusability and reuse (e.g., the five-step generic reuse/reusability model).

Consider the object-oriented methodology for use within the software process.

Seek to automate activities within the software process as understanding and experience permit.

TECHNICAL GUIDELINES

Domain Analysis

There seem to be basically two categories of software that are good candidates for reuse. These could be referred to as horizontally-reusable and vertically-reusable components. Horizontal reuse refers to reuse across a broad range of application areas (such as data structures, sorting algorithms, user interface mechanisms), while vertical reuse refers to components of software within a given application area that can be reused in similar applications within the same problem domain. Horizontal reuse has, no doubt, been studied the most so far, and such reuse has likely occurred much more frequently than vertical reuse. The main reasons for this likely are that such reuse is better understood and easier to achieve. On the other hand, the greatest potential leverage can come from vertical reuse--by intensive reuse of carefully crafted solutions to problems within an application domain. The CAMP project is an example of vertical reuse. In order to achieve vertical reuse, a "domain analysis" is required. Kang (1989) defines domain analysis as follows:

> "Domain analysis is a phase in the software life-cycle where a domain model, which describes the common functions, data and relationships of a family of systems in the domain, a dictionary, which defines the terminologies used in the domain, and a software architecture, which describes the packaging, control, and interfaces, are produced. The information necessary to produce a domain model, a dictionary, and an architecture is gathered, organized, and represented during the domain analysis."

Cohen (1989) summarizes a domain analysis methodology in eight steps:

1. Select specific functions/objects;
2. Abstract functions/objects;
3. Define taxonomy;
4. Identify common features;
5. Identify specific relationships;
6. Abstract the relationships;
7. Derive a functional model; and
8. Define a domain language.

We conclude this section with the following guidelines.

Select domain(s) carefully for analysis, based on the maturity of the organization's activities within each domain, and the planned emphasis the domain is to receive.

Determine and apply a systematic approach to domain analysis, yielding a domain model, a set of domain terminology, and a domain architecture.

Use domain analysis results as a basis for classifying, storing, and retrieving reusable components.

Use domain analysis results as a basis for decisions about the advisability of investing in specific instances of reusable software.

Use domain analysis results to help understand how existing domain-specific reusable software may be applied.

Creating Reusable Components

Reusable Components Spanning the Life Cycle: The goal of practitioners of software reuse should be to make use of existing knowledge and software artifacts throughout the software life cycle ("wide-spectrum reuse"), with the expectation that, the earlier the reuse occurs in the life cycle, the greater should be the payoff.

Many kinds of software components have potential for reuse, including domain-related knowledge, requirements, abstract designs, abstract algorithms, design and program transformations, code, test plans, test cases, and test results. Research is being conducted to seek to generate code from requirements through automatic transformations. Some success has been achieved in very narrow domains, yet in the foreseeable future manual transformations will be necessary in moving from phase to phase in the software life cycle. The greatest occurrence of high-level reuse presently is probably in "personnel reuse"--individuals who previously have analyzed requirements for similar systems, put to use their retained insights and approaches.

Since it costs extra money to prepare software components for reuse, it is important that a careful assessment be made of the likely payoff of such extra costs, based on expected future use of the components, on estimated cost to prepare reusable components, and on expected cost to custom-build software with similar functionality in the future. The availability of cost modeling tools and databases of organizational performance and cost data, are important resources in achieving such assessments. Note that this decision process concerning the preparation of reusable components, is reflected as Step 5 of the suggested generic reuse/reusability model.

A code component retained "in isolation" in a reuse library is likely to be of little value. Thus it is extremely important that design and requirements associated with code modules be retained, along with test plans, test cases, test results, prototypes (perhaps), and other related life cycle products. The associated set of components can be useful in many ways. For example, determining a match (or near-match) with reusable requirements, may lead to reuse of the associated design and code, or at least to reuse of the high-level design. Clearly, the high-level design should be easier to understand and adapt than the associated code would be. And the high-level and detailed designs should be very valuable in understanding code.

We end this subsection with the following guidelines.

Provide domain analysis results within the reuse framework--explicitly and/or implicitly.

Make careful assessments, including financial predictions, in deciding whether to develop a reusable component.

Prepare for reuse all more-abstract life cycle representations of a reusable component.

Record and supply adaptation suggestions with a reusable component.

Generalize a reusable component to the extent practical during its preparation.

State as a requirement the reuse of software and/or the creation of reusable software.

During each life cycle phase (Requirements Determination, High-Level Design, Detailed Design, Coding and Unit Testing, Integration Testing) conduct the activities suggested in the five-step reuse/reusability model.

Recognizing that the maintenance phase contains as sub-phases the software development phases, apply to maintenance similar reuse/reusability activities as for the development phases.

Code Components: There are two different clearly distinguishable categories of code components for reuse. The first may be called "passive" components, or "building blocks", which are used essentially unchanged. Another approach--very effective when feasible--is "dynamic components" (i.e., "generators"), which generate a product for reuse. This approach is also spoken of as reusable "patterns". Generators are of two fundamental types: (a) application generators (employing reusable patterns of code), and (b) transformation systems (which generate a product by successive application of transformation rules). Generators by their nature tend to lift the abstraction level for reuse above that of code building blocks. Fourth generation languages constitute another kind of reusable pattern, and also substantially lift the reuse abstraction level.

The recent past has brought better programming languages for reusability support--especially Ada. Modula-2 and various other available languages also have some good features. But the DoD mandate for use of Ada doubtless makes Ada the language of choice for code reuse.

The Ada language is at once a primary motivator of reuse, and a primary mechanism for reuse. Ada provides a rich set of features supporting reusability, including (St. Dennis 1986) packages, separate compilation and checking across program units, separate specifications and bodies for program units, information hiding constructs (e.g., private types), generics, and strong typing. Ada was designed to promote reliability and maintainability, programming as a human activity, and efficiency. As St. Dennis (1986) notes, "Software reliability, maintainability and efficiency also contribute positively to reusability. Reliability contributes to user confidence in a software component, maintainability to understandability, and efficiency to feasability for reuse."

Basili et al (1989) have provided a number of guidelines for developing Ada code, based on their research into the impacts for reuse of the strength of data bindings of a module with its environment, and on their assessment of effort necessary to transform existing Ada code for reuse in contexts other than the original one. They suggest the following:

* Avoid excessive multiple nesting in any language constructs.
* Usage of the USE clause is to be avoided if possible.
* Components should not interact with their outer environment.
* Appropriate use of packaging could greatly accommodate reusability.
* Avoid mingling resource units with application specific context.
* Avoid literal values.
* Keep interfaces abstract.

There are a number of reuse guidebooks now available (e.g., Wald 1986, ISEC 1985, ESD 1985, St. Dennis 1986). Of these, St. Dennis (1986) provides by far the most extensive set of guidelines for writing reusable Ada source code. Most of the guidelines in (Hooper and Chester 1989) are from St. Dennis--and, most of St. Dennis' guidelines were included. The "Exceptions" guidelines, and a few others, were from ISEC 1985. Most of the guidelines suggested by Basili et al (1989) are included in the list.

The following guidelines briefly summarize our observations above.

Supply reusable code in the form of a generator or a transformation system for greater reuse leverage, when practical.

In preparing code blocks, use Ada generics, parameterized procedures, and code templates for greater reuse generality, as appropriate.

Emphasize good programming style in developing reusable code, creating code exhibiting understandability, reliability and maintainability.

Emphasize Ada as a suitable programming language for reusable code.

Establish a set of organizational guidelines for code development (e.g., see St. Dennis 1986, Hooper and Chester 1989, for Ada guidelines).

Component Quality: If software reuse is to be successful, the available reusable components must be worthy of user confidence. There are numerous aspects that bear upon the achievement and maintenance of component quality, including the understandability and completeness of requirements and designs, the quality of documentation, programming style, and ultimately the correct performance of the reused software. Two underlying issues that require special attention are (a) component validation and verification (V&V), and (b) configuration management.

The importance of the configuration management issue is well known to software engineers. Mechanisms for configuration management are practiced effectively by many organizations. It is worthy of emphasis, however, that since having the trust and confidence of "reusers" is critically important, great emphasis should be placed on workable configuration management procedures.

The issue of component verification and validation is especially significant

for reusable software. Bullard et al (1989) observe that in the past the focus of V&V was to determine whether software met its specification within a specific environment. In the case of reusable software, however, the software may be reused in numerous different environments. They give a good discussion of approaches for detecting reuse errors.

Another important aspect of software quality assessment is the recorded experiences of users of library components. We consider this in a later subsection.

Quality assurance activities are considered essential in organizations practicing software engineering. Due to the critical importance of the correctness of reusable software, even greater emphasis should be placed on the scrutiny of reusable components by a quality assurance group. ISEC (1985) suggests the use of a "reusability checklist" by the quality assurance group.

We conclude this subsection with the following guidelines.

Set standards to be met by all library components.

Emphasize stringent V&V for reusable components, stressing portability and adaptability.

Emphasize enforcement of standards and practices by the Quality Assurance group; employ a "reusability checklist".

Establish and operate an effective configuration managment program for the reuse library.

Classifying and Storing Components: Having determined that components will be made available for reuse, it is necessary to classify each component according to some taxonomy, and store the component in an on-line library for reuse. Considerable research and experimentation have been conducted in this aspect of reuse repository operation. Prieto-Diaz and Freeman (1987) have developed the "faceted" classification method, based on ideas from library science. Each component is characterized by a sextuple consisting of:

<function, objects, medium, system type, functional area, setting>.

They incorporate the idea of "conceptual closeness", to give a user a measure of how closely an available component corresponds to a specified facet during retrieval. Several other possible technical approaches to classifying and storing components are discussed by Hooper and Chester (1989).

For effective retrieval, the library should represent relationship information between components (e.g., between a design component and the corresponding code, between a system specification and a subsystem specification, between two components that are related by reuse potential within a given application domain, and between two components such that one is a specialization of the other).

It is critically important that good documentation be provided with reusable components. See (Wald 1986) for a good discussion (also summarized in (Hooper and Chester 1989)).

Now we provide the following guidelines.

Determine approach(es) for classifying and storing components based, e.g., on domain analysis.

Represent relationships between a component and its more (and less) abstract representations (as to life cycle phase and generalization/specialization), and between a component and others that may collectively solve a given problem or class of problems.

Document each component thoroughly on-line, including user documentation and programming (i.e., maintenance) documentation.

Reusing Components

A number of operational issues must be satisfactorily addressed in order to effectively make use of available reusable components. Included are (a) classifying and storing components, (b) identifying components that meet specific needs, (c) understanding and assessing identified components, (d) adapting components as necessary, and (e) assembling components into a complete software system. We consider these issues briefly in the following subsections. A much more complete discussion may be found in (Hooper and Chester 1989).

Searching and Retrieving: The search and retrieval process is closely tied to the classification approach used. For example, in the case of the Faceted Classification approach, the search is based on specification of a sextuple of descriptive keywords.

Three important aspects of the retrieval process are concepts from information retrieval. They are recall, precision, and ranking. Recall pertains to the percentage of relevant components that are identified; precision pertains to the percentage of identified components that

are relevant; and ranking orders identified components by quality of match (to address the "information overload" issue). These are all substantive issues, and will be increasingly important as reuse libraries become larger.

User-friendly interfaces should be emphasized for component search and retrieval. Burton et al (1987) discuss and depict their approach to an interactive interface in their RSL system, supporting all aspects of reuse. The ease of use of the reuse system is a very important issue, and deserves special emphasis in planning for and implementing reuse within an organization.

The following guidelines pertain to the above discussion.

Devise and implement a mechanism for search and retrieval supporting query and browsing modes.

Emphasize user-friendly interface for search and retrieval.

Provide indication of "goodness of match" of components to a query.

Understanding and Assessing Components: Standish (1984) estimates that software maintenance costs 70-90 percent of the life cycle, and understanding accounts for 50-90 percent of maintenance cost. This would mean that understanding accounts for 35-80 percent of life cycle cost. Understanding is absolutely critical to software reuse--especially if a component must be adapted.

There are many aspects to component understanding, including the creation of the component within the framework of a well-understood, consistent software engineering process, and good approaches to development, testing and maintenance. The existence of effective domain analysis, and knowledge of how a component fits within the domain, can be most helpful in understanding how the component may be used. In short, if good practices are followed in developing, classifying and storing reusable components, good understanding should be a natural by-product.

On the other hand, we often face the need to understand and seek to reuse existing software that was not developed for reuse, and is not well understood by prospective reusers (nor perhaps by anyone available to the prospective reusers). Active research is addressing this issue. For example, Chen and Ramamoorthy (1986) have developed the C Information Abstractor, which scans C programs and stores information into a database concerning identifier types and environments. An Information Viewer permits interrogation of the database. Structured comments are used within C code to supply additional information, such as purpose, assumptions, and preconditions. Research in "reverse engineering" has promise for reuse. Biggerstaff (1989) describes work at MCC in reverse engineering, including their experimental reverse engineering system.

DeMillo et al (1989) emphasize the use of operational history information in assessing candidate reusable components. They developed an experimental system to log some types of observations on component use, for query by a prospective user. They suggest the use of such experience information as: security observations, extent of use, reported failures and faults, and performance observations (e.g., execution efficiency, memory utilization). Some of this data can be collected automatically, while some must be supplied explicitly by users.

The importance of accumulating such usage data should be stressed, and an organization should implement procedures to acquire the necessary feedback.

Now we offer summary guidelines for this subsection.

Understanding of reusable components chiefly ensues from effective domain analysis, good software development practices, and good classification and storage mechanisms.

Seek approaches/tools to help understand software not specifically prepared for reuse (i.e., reverse engineering).

Use operational history of components in assessing their suitability. Obtain feedback from users of components, including number of uses, degree of satisfaction, and errors.

Adapting Components: The ideal situation is that a component (or components) will be identified which exactly meets the need. That will often not be the case, however. Understanding the component is the key to the decision process. "Goodness of fit" of an available component might well be measured by the effort required for adaptation. This would provide guidance when multiple components are identified which are candidates for selection. If a code component is functionally adequate--i.e., it performs a needed role in an acceptable way, then there should be little or no adaptation required if the component is highly cohesive and has no side effects. Cost modeling tools should be used in the

decision process if the expected adaptation is very extensive.

If code requires adaptation, the design and/or specifications corresponding to the code component (and hopefully retained in the library) will likely prove to be very important. If the component must be rewritten in a different programming language, the high-level design should serve as the basis--which is also true if the number of code "patches" required for adaptation is large.

Parameterized code is developed with the intent that input parameters cause "adaptation" of the code, as pre-planned. And, generators are driven by input directives to "adapt" within a pre-planned range. Ada generic procedures provide a mechanism for developing a "family" of procedures for which data types may be specified--and thus a specific "adaptation" created.

Basili et al (1989) are undertaking research to evaluate the reuse implications of Ada modules based on the explicit and implicit bindings of the module with its environment. They are experimenting with tools to assess the effort necessary to transform Ada code for reuse in different contexts.

The following guidelines are provided.

Use higher-abstraction representations in adapting a component (e.g., use design when adapting code).

Emphasize the use of available metrics/tools to assess adaptation effort.

Composition of Code Components: "Composition" refers to interconnecting components to form software systems. The most straightforward approach may be used when the component is a procedure which perfectly meets the need; then composition results from a procedure call, coupled with the action of the "linker". The same is true, of course, if we can successfully adapt a procedure. This is at present the primary mechanism for composition.

Other important mechanisms for composition are UNIX pipes, and inheritance in object-oriented languages. Both of these have considerable benefit in shielding the user from the need to understand code, per se; in the best case the code can be treated as a "black box".

The following guidelines ensue from the discussion of this subsection.

Use existing mechanisms for composition to the extent possible (e.g., procedure linking, unix pipes, inheritance in object-oriented languages).

Seek automated approaches to composition as understanding permits.

CONCLUSIONS

There is every reason to believe that the well-planned and executed practice of software reuse will significantly increase software productivity and quality. It is recognized that it will take time to implement a reuse program and see it paying for itself. It will be necessary to implement reuse activities in a phased manner to some extent. A good beginning point for initiating reuse would be to undertake an assessment within the organization of current status and potential for reuse. For example, Fairley (1989) provides a suggested questionnaire for assessing reuse status.

We offer now some suggested implementation steps, stated in approximately the order in which they should be undertaken. Two assumptions underlying these steps is that the organization is already employing systematic, uniform software engineering practices, and that top-level management is firmly committed to achieving a successful software reuse program.

1. Determine scope for reuse, relative to:
   a. organizational extent of reuse (e.g., command/ division/ regional center/ project);
   b. life-cycle products to reuse (e.g., requirements, design, code, test cases); and
   c. application-domain relatedness (domain-specific, domain-independent, or both).
2. Establish a software engineering process incorporating reuse and the development of reusable products.
3. Adjust the organizational structure as necessary and provide staffing to support software reuse.
4. Institute management policies and practices to foster reuse, including the means to overcome disincentives and to provide positive incentives for individuals and projects.
5. Implement reuse library mechanisms.
6. Perform an initial domain analysis, and develop and/or acquire enough reusable components to begin reuse.
7. Begin the practice of reuse, first perhaps in a single domain; then expand to other application areas as experience is gained.
8. Continually assess effectiveness of the reuse-based process; alter/augment as appropriate.

As a part of the "bootstrapping" process it will be important to inventory

the organization's existing software assets for potential reuse. A reasonable strategy seems to be to determine and focus on a single application domain first, chosen for its future importance to the organization, and its potential for reuse success (including, for example, availability of existing software, thorough understanding of the application domain, and likelihood of recurring requirements for similar software within the domain). Emphasis should be placed from the beginning on developing library mechanisms that can grow to accommodate the expected growth of the organization's reuse program, and on having demanding standards for library submissions, that are clearly formulated and strictly enforced. Over time additional domains can be added, and the organizational structure can evolve to accommodate the increased activity.

ACKNOWLEDGMENT

This work was performed under the auspices of the Ada Reuse and Metrics Project, conducted by Martin Marietta Energy Systems, Inc. for the U. S. Army Institute for Research in Management Information, Communications, and Computer Sciences (AIRMICS), under Contract No. DE-AC05-840R21400. The work at The University of Alabama in Huntsville was performed under Martin Marietta Subcontract 19K-CR613C. Appreciation is expressed to Dan Hocking, AIRMICS Project Manager.

The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

REFERENCES

Basili, V.R., H.D. Rombach, and A. Delis. 1989. Ada Reusability Metrics. In PROCEEDINGS OF THE ADA REUSE AND METRICS WORKSHOP, Atlanta (June).

Biggerstaff, T. 1989. Design Recovery for Maintenance and Reuse. COMPUTER, vol. 22, no. 7 (July), p. 36-49.

Boehm, B.W. 1988. A Spiral Model of Software Development and Enhancement. COMPUTER, vol. 21, no. 5 (May), pp. 61-72.

Booch, G. 1987. SOFTWARE COMPONENTS WITH ADA. Benjamin /Cummings, Menlo Park, CA.

Bullard, C.K., D.S. Guindi, W.B. Ligon, W.M. McCracken, S. Rugaber. 1989. Verification and Validation of Reusable Ada Components. In Lesslie et al 1989, pp. 31-53.

Burton, B.A., R.W. Aragon, S.A. Bailey, K.D.Koehler, and L.A. Mayes. 1987. The Reusable Software Library. IEEE SOFTWARE, vol. 4, no. 4 (July), pp. 25-33.

Chen, Y.F. and C.V. Ramamoorthy. 1986. The C Information Abstractor. In PROCEEDINGS OF COMPSAC 86, Chicago (October), pp. 291-298.

Cohen, J. 1989. GTE Software Reuse for Information Management Systems. In PROCEEDINGS OF THE REUSE IN PRACTICE WORKSHOP, SEI, Pittsburgh (July).

DeMillo, R., W. Du and R. Stansifer. 1989. Reuse Oriented Ada Programming Environment: A Prototype System. in PROCEEDINGS OF THE ADA REUSE AND METRICS WORKSHOP, Atlanta, GA (June).

ESD 1985. ADA REUSABILITY GUIDELINES, 3285-2-208/2.1. SofTech (for U.S.A.F. ESD; April; edited by C.L. Braun, J.B. Goodenough, R.S. Eaves).

Fairley, R., S. Pfleeger, T. Bollinger, A. Davis, A.J. Incorvaia, B. Springsteen. 1989. FINAL REPORT: INCENTIVES FOR REUSE OF ADA COMPONENTS (Volumes I through V). (Sept.). Ada Reuse and Metrics Project (Martin Marietta/AIRMICS).

Hooper, J.W. and R.O. Chester. 1989. DRAFT REPORT: SOFTWARE REUSE GUIDELINES (Sept.). Ada Reuse and Metrics Project (Martin Marietta/AIRMICS).

Humphrey, W.S. 1989. The Software Engineering Process: Definition and Scope. PROCEEDINGS OF THE 4TH INTERNATIONAL SOFTWARE PROCESS WORKSHOP, Devon, UK (held May 1988); SOFTWARE ENGINEERING NOTES, vol. 14, no. 4 (June), pp. 82-83.

ISEC 1985. ISEC REUSABILITY GUIDELINES, 3285-4-247/2. SofTech (for U.S. Army Information Systems Engineering Command; Dec.; editors not stated).

Kang, K.C. 1989. Features Analysis: An Approach to Domain Analysis. In PROCEEDINGS OF THE REUSE IN PRACTICE WORKSHOP, SEI, Pittsburgh (July).

Kang, K.C. 1987. A Reuse-Based Software Development Methodology. In PROCEEDINGS OF THE WORKSHOP ON SOFTWARE REUSE (RMISE, SEI, MCC, SPC), Boulder (October).

Lesslie, P.A., R.O. Chester and M.F. Theofanos. 1989. GUIDELINES DOCUMENT FOR ADA REUSE AND METRICS (DRAFT, March 1, 1989). Martin Marietta Energy Systems, Inc., Oak Ridge, Tennessee (under contract to U.S. Army AIRMICS).

Meyer, B. 1987. Reusability: The Case for Object-Oriented Design. IEEE SOFTWARE, vol. 4, no. 2 (March), pp. 50-64.

Prieto-Diaz, R. and P. Freeman. 1987. Classifying Software for Reusability. IEEE SOFTWARE, vol. 4, no. 1 (Jan.),pp.6-16.

Simos, M.A. 1987. The Domain-Oriented Software Life Cycle: Towards an Extended Process Model for Reusability. In PROCEEDINGS OF THE WORKSHOP ON SOFTWARE REUSE (RMISE, SEI, MCC, SPC), Boulder (October).

St. Dennis, R.J. 1986. A GUIDEBOOK FOR WRITING REUSABLE SOURCE CODE IN ADA (R). Version 1.1. CSC-86-3:8213. Honeywell, Golden Valley, Minnesota (May).

Standish, T.A. 1984. An Essay on Software Reuse. IEEE TRANS. ON S.E., vol. SE-10, no. 5 (Sept.), pp. 494-497.

Wald, B. 1986. STARS REUSABILITY GUDEBOOK. V4.0. DoD STARS.

Yeh. R.T. and T.A. Welch. 1987. Software Evolution: Forging a Paradigm. In PROCEEDINGS OF THE 1987 FALL JOINT COMPUTER CONFERENCE, Dallas (October).

BIOGRAPHIES

JAMES W. HOOPER is Professor of Computer Science at the University of Alabama in Huntsville (UAH), where he teaches and conducts research in programming languages and software engineering. He holds B.S. and M.S. degrees in mathematics, and M.S. and Ph.D. degrees in computer science. Prior to joining UAH in 1980, he was employed by NASA Marshall Space Flight Center, where he conducted research in simulation approaches for NASA missions.

ROWENA O. CHESTER heads the Information Security Research Section and the Advanced Languages Research Section in the Data Systems Research and Development Program of Martin Marietta Energy Systems, Oak Ridge, Tennessee. Recent research activities include operating system and application software certification, database security, PC vulnerability assessments, portability, and Ada software reuse. She holds the B. Eng. Physics degree, and the Ph.D. in physics and electrical engineering.

THE ECONOMICS OF REUSE: ISSUES AND ALTERNATIVES

Terry B. Bollinger
Shari Lawrence Pfleeger

Contel Technology Center
Chantilly, Virginia

Abstract: A major reason for increasing the level of reuse in the software development process is to lower costs. By replacing expensive development activities with the adoption of relatively inexpensive reused parts, software reuse has significant potential as a technique for cutting costs and helping to free developers for work on the more complex or novel components of a software system. Unfortunately, the level of reuse currently in place in the software industry is limited by an inability to predict how and where resources should be invested. This report describes a software cost estimation model developed specifically to deal with decisions about software reuse.

1. INTRODUCTION

Software reuse is the process by which existing software work products (which may include not only source code, but also products such as documentation, designs, test data, tools, and specifications) are carried over and used in a new development effort, preferably with minimal modification. An obvious advantage of software reuse is its potential for significantly reducing the cost of development. Additionally, reuse of well-proven components can increase the reliability of a new product by using thoroughly reviewed or tested work products.

It is important to note that software reuse is neither new nor rare in software development projects. However, it often goes unrecognized, primarily because it tends to be viewed in other contexts. For example, any purchase of off-the-shelf software is clearly a form of reuse, since it involves incorporation of existing software work products to reduce the total development costs of a new project. The notion is so pervasive that very few project managers would seriously consider developing an in-house relational database these days; it is far more cost effective simply to buy a commercially available database product. This type of software integration is actually reuse-by-procurement and is a vital part of modern software engineering.

Similarly, reliance on a commercial operating system is so fundamental to most computer environments that users seldom think of it in terms of software reuse. Yet the operating system serves as a repository of critical, readily reusable software modules providing access to computer hardware. The economic importance of an operating system as a suite of reusable components can be seen readily in terms of the cost of additional development required if those modules were not available.

Software maintenance is another example of "hidden" reuse. Maintenance is reuse-by-replacement, in which a new, functionally similar system is constructed via extensive reuse of a predecessor system; the resulting revision is then used as a replacement for its predecessor. The relationship of maintenance to reuse becomes more obvious as the complexity of a maintenance update approaches the level of a complete rebuild. In such cases, only selected parts of the original system are reused in the new version, and the style of development becomes increasingly similar to a structured, analytical form of reuse.

Even the lowly compiler is strongly dependent on software reuse to accomplish its tasks. As a source language processor, it represents a fully automated mechanism for accessing the software modules contained in its run-time library. Many of its direct expansions of source code into assembly language expansions are equivalent to highly parameterized forms of reuse of frequently used assembly language code sequences.

An important theme runs through the examples above. In each case, significant effort has been expended during design to make the products readily reusable. Commercial products can be constructed not only for ease of use but also for ease of integration. Likewise, maintainable software is often constructed to assure that likely future changes are relatively inexpensive to implement. Finally, both operating systems and compilers are systems in which access to reusable components has been made automatic. In each case, explicit effort during the design and implementation of the software increased reusability.

Such up-front planning greatly enhances the chances that work products will be reused.

Although many developers acknowledge the benefits of reuse, a great many projects are completed without the design and development of reusable components. A fundamental reason for this lack of planned effort to make software reusable is the lack of a good predictive model of the costs and benefits associated with software reuse. Were the developer able to assess the impact of creating reusable parts on a current project and then predict the value of the reuse of those components on future projects, the production of reusable components could become a more standard activity during software development.

It is clear that reuse spans and affects a set of projects, rather than one project in isolation. This report defines a cost estimation methodology that encompasses the multi-project cost issues involved in software reuse. To begin, a number of key concepts and cost relationships are introduced and explored. Then, an analytical approach to assessing the economics of new projects that both create and use reusable parts is described. Finally, the general design of a cost estimation tool based on these methods is outlined. Additionally, future directions are addressed by which reuse cost estimation may be integrated more closely with reuse technology selection and other aspects of the software development process.

2. CONCEPTS

Any cost estimation methodology that acknowledges software reuse must address the following set of foundation concepts:

- *Baseline Projects*
- *Producer/Consumer View of Reuse*
- *Broad Spectrum Reuse*
- *Cost Factors*
- *Reuse Cost Amortization*

These concepts are described here in detail.

2.1 Baseline Projects

Often, a set of projects all of which deal with a particular application area will involve the same or similar development activities, use similar development environments, and involve development teams with similar performance and experience histories. Thus, the productivity of the development team and the resulting costs tend to be similar as well. Managers experienced in the application area often use intuition to make cost estimates for a new project in that application area simply by noting how the project is likely to differ from past projects.

This idea of estimation based on changes to a known development effort is central to evaluation of reuse, since the benefits of reuse must be measured in terms of an equivalent project without reuse. The known development effort is termed a *baseline project*, and it acts as reference point for the estimation. The baseline project is an ordered description of the various activities and costs which may be affected by reuse or other significant changes in a new development effort. Use of a baseline explicitly represents cost in a form that is amenable to automated support and analysis.

Figure 1 shows an example of a baseline project. The baseline consists of a directed, PERT-like network of components or activities, each of which has an associated cost and is likely to occur in a new project within that problem domain. The baseline may be derived either from actual experience with one or more such projects, if such data are available, or through analysis and project planning for a project in a completely new application domain.

The model contains two major types of cost components:

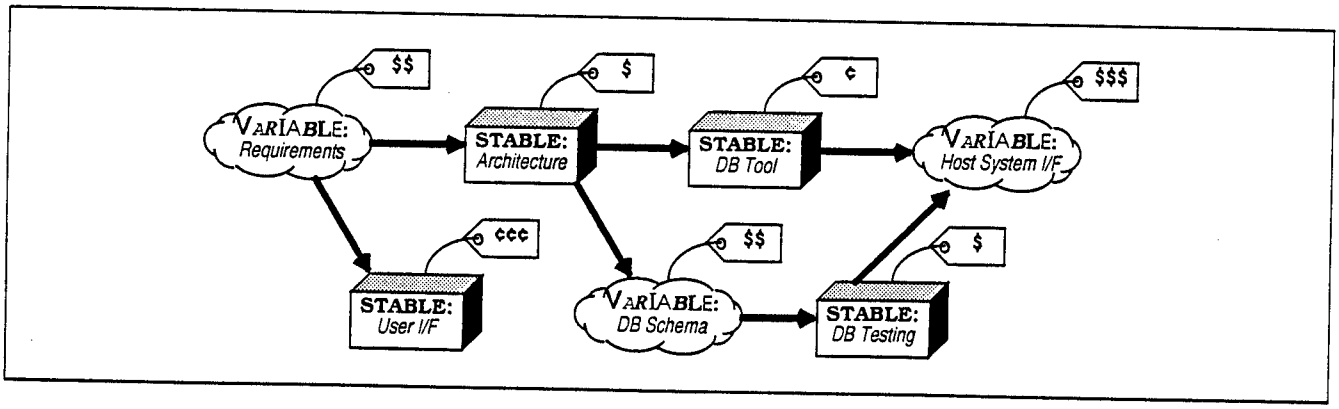


Figure 1 – A Baseline Project Cost Framework

- *Stable development components,* represented by blocks in Figure 1, correspond intuitively to fixed price "parts" that must be procured or developed to meet the needs of the new project. Associated with the stable development components are nominal costs, which are generally expressed as fixed prices. Stable components may be "pure" procurement activities. However, they may also be development activities whose costs are well-understood and predictable with great accuracy.

- *Variable development components,* represented by clouds in Figure 1, correspond to labor-related activities in which something completely new is being developed. Variable development components also have associated nominal costs, but here the components costs are represented in terms of *cost rates* and *production rates,* rather than as fixed prices. Just as stable components need not be pure procurement activities, variable development components need not be pure labor activities; they need only be characterized predominantly by labor costs.

The baseline project can be described by a nested set of such diagrams, each providing more detail than its predecessor. In the high-level diagrams, stable components are characterized predominantly by fixed-price procurement costs. As variable components are refined at lower levels of a model, they eventually come to represent isolated labor activities, at which point their production rates become equivalent to developer productivity rates.

Figure 2 shows an example of such decomposition. At the top, the high-level diagram shows a gross decomposition of components. One of the variable components is then described with finer granularity at the bottom, indicated by the elaboration of one cloud into several components.

The ability to decompose high-level development components into lower, more precise levels of detail is a valuable estimation technique, since it allows the amount of effort applied to estimating a particular development cost to be proportional to the level of perceived risk in the estimate. Components with cost behaviors that are very well understood may be estimated using a single level

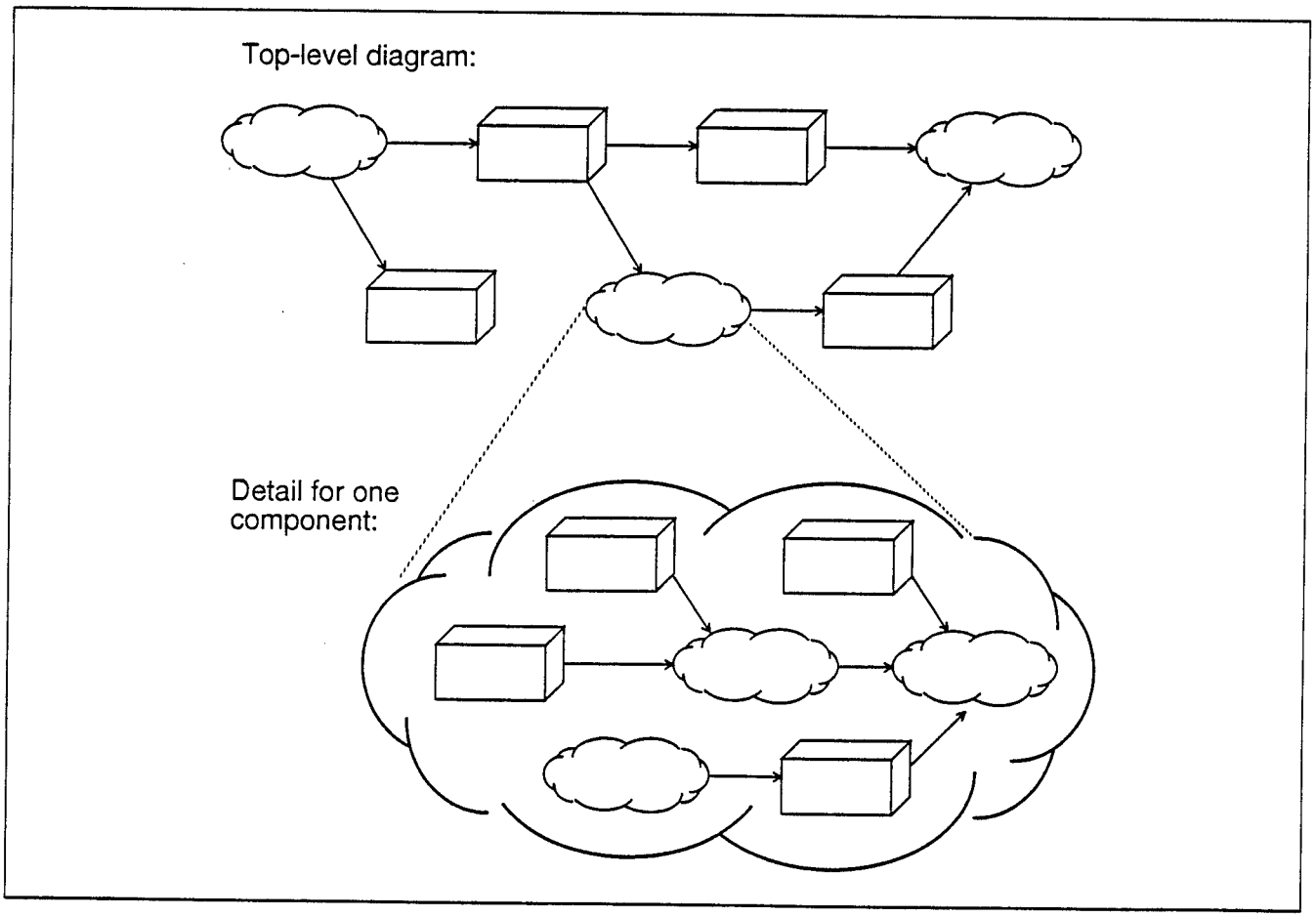


Figure 2 – Decomposition of Development

diagram, while other components that are perceived as being much more unpredictable can be decomposed to permit a finer, more precise level of granularity in the cost estimation process.

2.2 Producer/Consumer View of Reuse

Cost estimation is by its nature a consideration of the economics of software development. Thus, it is natural to assess reuse and its cost impact in an economic framework. This section addresses reuse in light of who produces reusable parts, who uses them, and who pays for investment in reuse.

Producers and consumers. There are two major types of reuse activities: the development of work products for use in subsequent projects, and the use of those work products. Those developers building reusable work products are termed *reuse producers*, while those who retrieve and adapt the products are *reuse consumers*. The distinction between producers and consumers is necessary for understanding the multi-project nature of reuse. It is the producers who incur the initial cost and who receive little or no benefit from their reuse efforts. The consumers, though required to expend effort in understanding, retrieving, and adapting the work products, are the ones who can take advantage of the benefits of reuse. Thus, the producer/consumer view of reuse admits consideration of issues relating to producer investment in reuse, the amortization of cost over the consumers of the work products, and the balancing of cost and benefit to make reuse economically feasible.

The use of terminology with strong commercial implications to describe reuse activities is intentional and useful in understanding the underlying processes. Vendors of commercial software achieve a net economic benefit only when their products are "reused" (purchased) enough times to cover development costs; customers achieve a net benefit by avoiding the usually greater cost of developing a tool or package.

The producer/consumer model of reuse is rich in both managerial and technical implications. For example, organizations that provide no payback incentives to producer projects are unlikely to succeed at integrating reuse into their development processes. In this case, reuse producers are penalized for making additional expenditures that do not directly benefit their development needs. Indeed, one of the most significant inhibitors against extensive reuse in the software industry may be a general lack of explicit incentive strategies to encourage coordinated reuse investments. Without such incentives, reuse degenerates into "scavenging", where each new reuse customer must bear the full cost of finding, understanding, and modifying the work product to meet the project's needs. Code-level scavenging, in which those retrieving the code are unfamiliar with the design or original use of the parts, is an extremely inefficient form of reuse, since costs of finding, adapting, and modifying a part are duplicated each time that part is reused. In contrast, well-planned up-front investments in the producer activity, including the development of a formal repository of reusable parts, can lead to several long-term benefits. By making a part much easier to find and use later, analysis of the part and adaptation for use in the new project will generally require much less effort and involve much less risk than development from scratch.

Reuse investments and benefits. In light of the interaction between reuse producers and consumers, it is clear that management must play an active role in encouraging and investing in reuse. Figure 3 represents a basic cost relationship that must be met if reuse is to result in a net economic benefit.

The left side of this graphical equation shows the sum of *reuse investments* made to increase the reusability of software work products. Here, a reuse investment is simply a cost that does not contribute directly to the completion of the primary development goals of a current project. For example, any labor hours that a developer devotes specifically to classifying and placing code

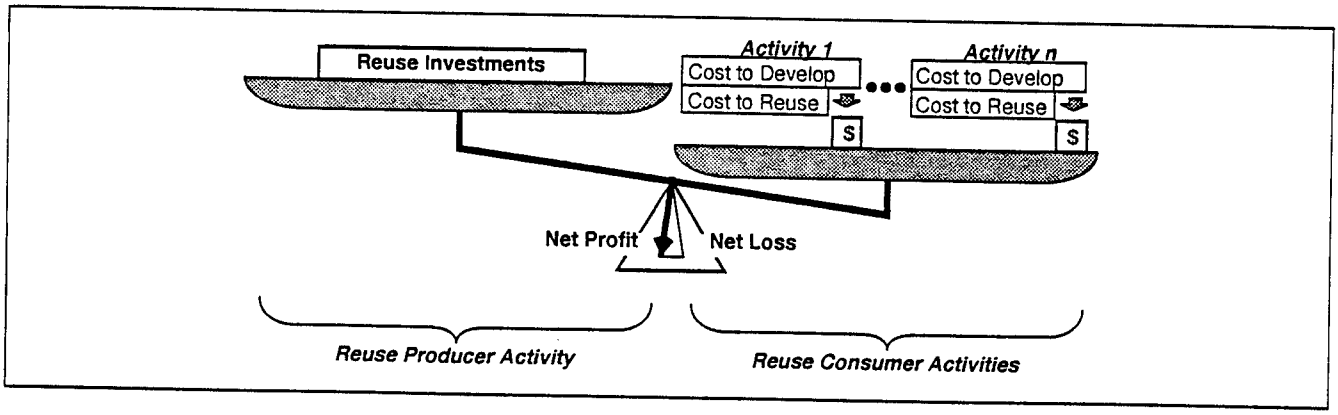


Figure 3 – Software Reuse as an Investment

components into a reuse library is a reuse investment, since those hours are intended primarily to benefit subsequent projects. Like financial investments, such costs represent outlays of current resources (the money spent to make work products more reusable) in hope of future returns (the cost reductions accrued by subsequent projects through replacement of expensive development activities with less expensive existing work products).

The right side of the graphical equation represents the sum of the *reuse benefits* that accrue from the creation of highly reusable parts. The calculation of these benefits is simple in concept, although it can be rather complex in practice. For any collection of reusable parts, there will be some set of later development activities or projects that could potentially benefit by reusing some or all of the parts. The activities that could benefit by reusing parts are labeled *Activity 1* through *Activity n* in the diagram. Note that *n* could range from being very small (e.g., for highly specialized software) to very large (e.g., for very general software such as math packages and operating systems).

For each *Activity i* there will be an anticipated cost of developing and maintaining software without reusable parts, and an actual cost of developing and maintaining software with reusable parts. In the diagram, each expected cost of developing and maintaining without reuse is labeled a *Cost to Develop,* and each actual cost of developing and maintaining with reuse is labeled a *Cost to Reuse.* Obviously, the financial benefit (if any) that *Activity i* achieves through reuse will simply be the difference between these two figures. Added together, the benefits of all of the reuse activities will represent the total benefit gained from making the parts reusable. To be cost effective, this total benefit clearly must be larger than the cost originally invested in making the software reusable.

The reuse investment paradox. A strong theme of the previous discussion is that reuse in general involves a trade-off of costs between an initial producer activity which invests resources in making software reusable, and later consumer activities which receive direct cost benefits from reusing software. Unfortunately, this scenario produces a bit of a paradox, since the most important cost benefits of making software reusable are unlikely to be seen by the producer activity that is attempting to make the software reusable. In particular, a manager who is interested in making reuse investments for the benefit of his company is faced with two problems:

- *Reuse investments are excess costs.* By definition, reuse investments do not usually contribute directly to the development objectives of a project. A manager who wishes to invest resources to make software more reusable is thus faced with the problem of finding a way to fund the work that is acceptable both to the customer and to higher level management.
- *Reuse investments are optional costs.* Reuse investments are also optional costs. They are not required for the successful completion of a project, and they may in fact slow the development of deliverable software.

Because reuse investments are both excess costs and optional costs, a manager who is interested in the overall benefits of reuse to an organization may have a difficult time justifying such expenditures. The customer may see no direct benefits at all, and higher level management will frequently evaluate such decisions in terms of local, project-level costs, rather than in terms of global, company-level costs. Again, this points out the need for good reuse cost estimation, since the first step in justifying such reuse investments will be to identify where and how the cost benefits of reuse investments will be accrued.

Reuse investment equations. The investment/benefits cost relationship of Figure 3 can be expressed mathematically in the following way. Let "Investment" be the total cost of resources applied specifically to making the product or set of products reusable. For each of n subsequent projects or development activities that might take advantage by reusing the products, "Development$_i$" is the associated cost to the i^{th} activity of building the product or products without any use of the equivalent reusable products. "Adaptation$_i$" is the cost to the i^{th} reuse activity of finding, customizing, and integrating the reusable part, as opposed to developing it. Then the cost to the group of consumers of the reusable parts should include the costs of creating and reusing the work products. This cost is represented as

$$\text{Investment} + \sum_{i=1}^{n} \text{Adaptation}_i$$

Since the cost to the consumers of performing the activities without reuse is:

$$\sum_{i=1}^{n} \text{Development}_i$$

it is clear that a net benefit is obtained only when the cost of reuse is less than or equal to the cost without reuse:

$$\text{Investment} + \sum_{i=1}^{n} \text{Adaptation}_i \leq \sum_{i=1}^{n} \text{Development}_i$$

Thus, the net benefit of reusing a specific work product or set of work products is:

$$\left[\sum_{i=1}^{n} \left(\text{Development}_i - \text{Adaptation}_i \right) \right] - \text{Investment}$$

Figure 4 provides a graphical representation of a simplified form of the equation. Here, a single *baseline development cost* is estimated during the construction of the reusable product or products, and is then used throughout the right hand side of the equation in which the benefits of individual instances of reuse are calculated.

This approach simplifies the estimation of cost benefits by localizing the estimation of development costs to the producer activity, where the best information for such estimates is available. Expressing "Development" as the baseline development cost, the simplified reuse benefits equation is:

$$\text{Benefit} = \left[\sum_{i=1}^{n} (\text{Development} - \text{Adaptation}_i) \right] - \text{Investment}$$

This equation can be used to support decisions about when and where reuse is appropriate.

Note that there are several ways to generate a negative benefit (so that reuse is not economically feasible). One way is when the investment cost exceeds the sum of the differences between development and adaptation. Another reflects the situation where adaptation costs exceed the development costs; then, it is cheaper to build a new part than to modify an old one.

2.3 Broad Spectrum Reuse

The Broad Spectrum Reuse Perspective. Broad spectrum reuse refers to the idea that any type of work product (such as code) or characteristic (such as work experience) that can affect and benefit later projects is potentially reusable. Examples of such reusable items include requirements specifications, architectures, detailed designs, source code, documentation, test data, tools, and environments. Characteristics are non-physical reusable factors, including work experience or training in special development skills.

Some technologies have been developed specifically to support various forms of broad spectrum reuse. They can be considered in three categories, depending on the primary aspect of the reuse process supported. *Retrieval* technologies deal primarily with the problem of how to find and retrieve reusable parts. *Adaptation* technologies assist developers in configuring or modifying a reusable part to meet the specific requirements of their effort; Ada generics are a good example of such a technology. *Integration* technologies provide mechanisms for validating whether the design, behavior, and supporting documentation of a reused part is consist with the standards and needs of the project using it. Integration has often been overlooked as a reuse issue, but it nonetheless can be crucial for building a system that is consistent and maintainable in the long term.

For many products or activities there is a conspicuous lack of any technology intended specifically to support reuse. In some instances, this absence reflects a fundamental lack of understanding about how such products can even be represented. For example, there is no standard way to represent a development process or a set of requirements. This representational framework is necessary not only for storage of the products but also for understanding and retrieval. However, in other cases (such as retrieval of architectures and detailed designs), the lack of specific tools is due to a lack of recognition of the need for such tools, rather than to any serious technical impediments.

The Inclusion Effect. To model the cost impact of broad spectrum reuse, it is important to note that the economic benefits of reusing various types of work products and characteristics are not necessarily linear. In particular, the reuse of work products very early in the software development process can provide benefits far greater than the development cost of those items. This additional benefit is attributable to the *inclusion effect,* whereby the use of a work product explicitly mandates or implicitly includes the use of subsequent work products. For example, the reuse of a set of design modules may allow the reuse of the consequent code.

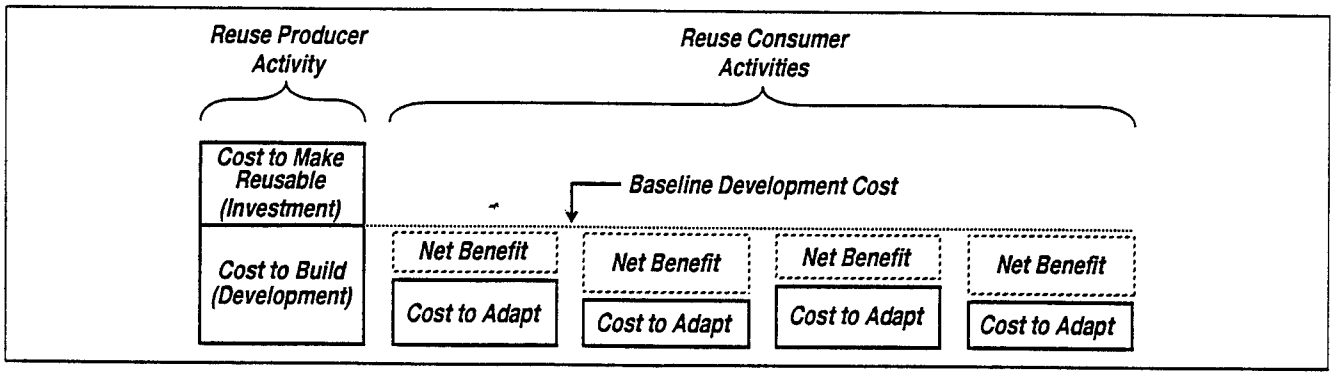


Figure 4 – Simplified Estimation of Reuse Benefits

Figure 5 shows in more detail how the inclusion effect can lead to disproportionate benefits when early work products are reused. In the diagram, reuse of a cluster of closely related requirements leads to inclusion of an entire tree structure of other work products which were derived directly from that cluster. For example, a collection of user interface requirements specifying a windows-and-mouse interaction can lead to inclusion of an entire suite of work products ranging from the architecture of a windows-and-mouse software package through actual code and documentation.

Obviously, a reuse strategy that encourages the inclusion effect can have significant economic advantages over methods focusing exclusively on techniques such as code-level retrieval and reuse. In terms of supporting technology, this observation implies that development of better tools support for reuse of early-phase work products holds great promise for increasing levels of software reuse. In terms of modeling costs, the inclusion effect suggests that a good cost model should accommodate "ripple effects" resulting from reuse at various stages in the software development process.

2.4 Cost Factors

The preceding sections have described the economics of reuse in terms of its difference in cost from a baseline project. To assess the cost of a proposed project with reference to the baseline, terminology must be introduced with which to describe the differences. To that end, a *cost factor* is defined to be any identifiable feature of a new project that is likely to change significantly one or more of the nominal cost values of the baseline project. Cost factors thus may affect fixed price purchases (e.g., procurement of a database) as well as the productivity of software developers. Because reuse decisions often involve the purchase of parts weighed against the development of those parts, a single cost factor can have both fixed price and productivity impacts.

Figure 6 illustrates the impact of a cost factor on the baseline project. The *scope* of a cost factor is the full set of development components affected by the ripple effect of the cost factor on subsequent development components. Note that the scope of a cost factor will encompass any inclusion effects of reuse, thus making cost estimation of such effects explicit.

2.5 Reuse Cost Amortization

As illustrated earlier, making software reusable is a form of investment whose benefits may not be accrued until the completion of subsequent new development activities. However, this situation generates an interesting economics problem: if the customer is not the developer, then the customer has no incentive to pay for the added costs of making that software highly reusable.

The most general solution to this problem is to use an *amortization schedule* to distribute costs in some equitable way over a collection of projects. The schedule allows the developer to cover some part of the original development effort with internal funds, and then be reimbursed in some way by later projects for the earlier reuse investment work.

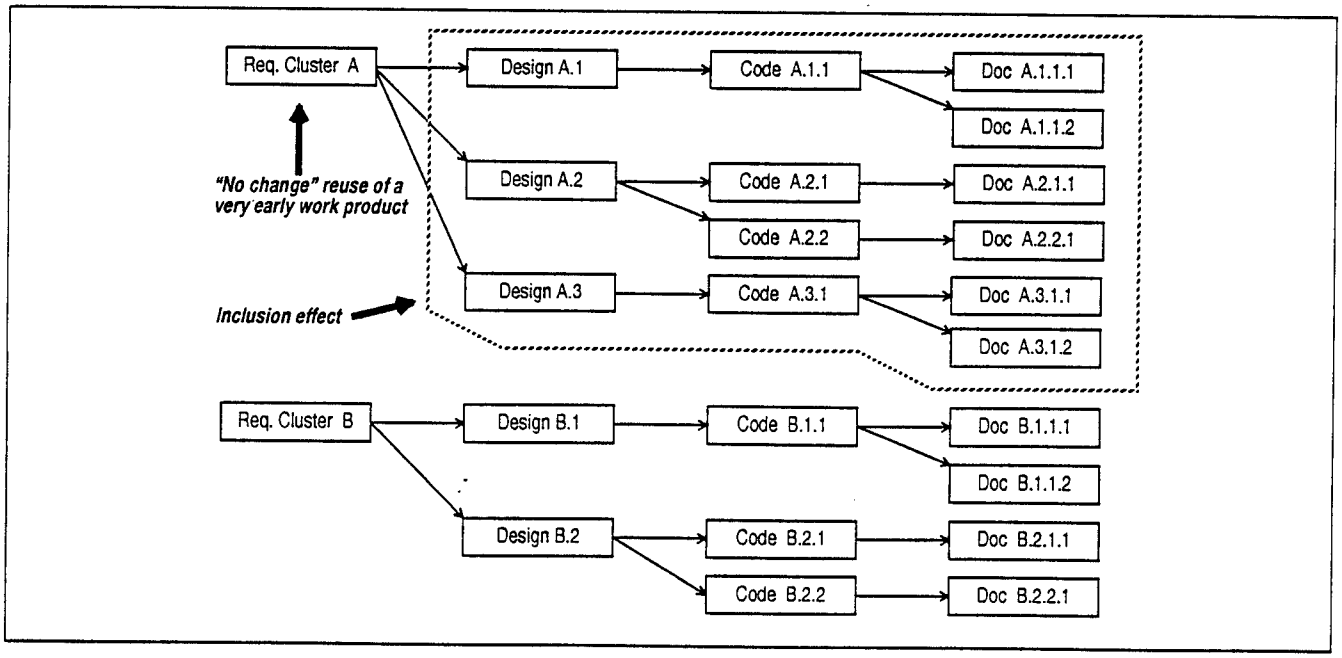


Figure 5 – The Inclusion Effect

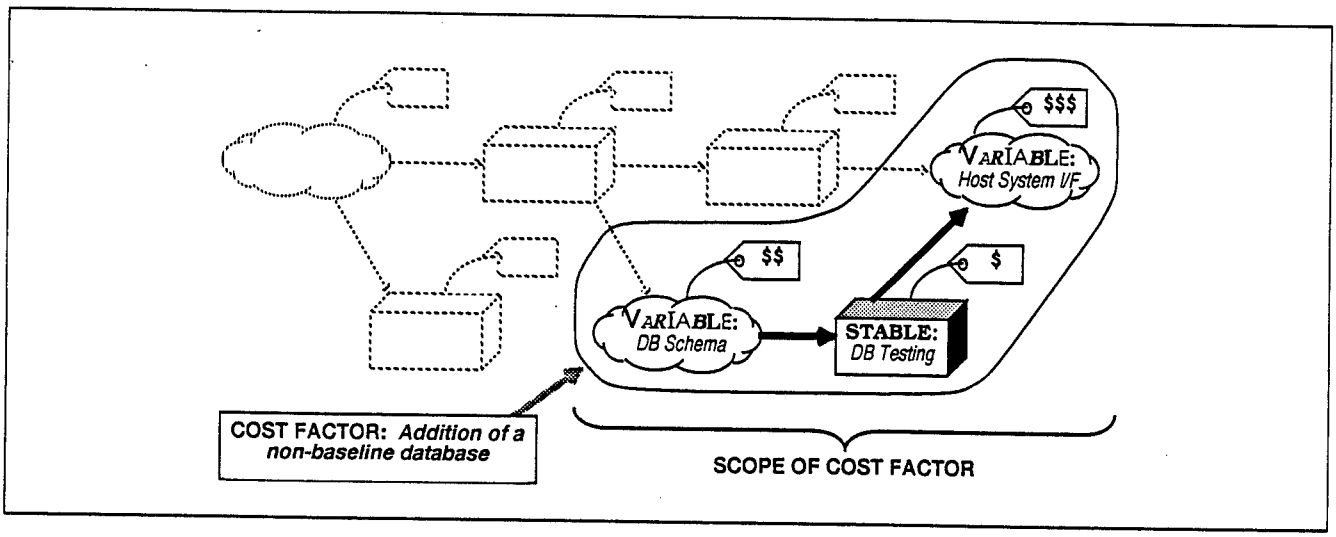


Figure 6 – Cost Factors

It is important to note that an organization may not explicitly charge later projects for the reuse of software. However, the initial investment in making the software reusable is a real allocation of organizational resources, and its costs will affect later projects through the removal of resources or funds that might otherwise have been available. An explicit amortization charge is simply an accounting mechanism to ensure the proper notation and repayment of the reuse investment cost. The repayment is requested from the entities that benefit most from the reuse, namely, the projects that reuse the work products.

Equally important is the realization that amortization schedules are arbitrary to some degree; that is, there is no single amortization schedule that is best. Research is required to identify candidate amortization schemes and the situations in which they would be most appropriate.

Amortization issues are vital to reuse cost modeling, since the cost of a component to a reuse consumer will be determined primarily by such amortization schedules, rather than by measures of component functionality or size. The total cost of reusing software thus depends in part on the earlier selection of pricing schemes determined by one or more reuse producer projects. For these reasons, consumer reuse is most readily modeled as a procurement activity that is similar to the purchase of commercial software packages, rather than simply as a productivity issue.

3. DEVELOPMENT OF A GENERAL REUSE COST MODEL

3.1 Cost Sharing Domains (CSDs)

A fully generalized modeling of the costs and benefits of reuse cannot be accomplished with a single-project perspective. Without sharing or amortization of reuse investment costs across multiple projects, justification of the costs of producer reuse is not readily possible.

One approach to making amortization and pricing decisions involves the concept of *Cost Sharing Domains (CSDs)*. A CSD is simply any group of present and future projects for which costs can be added together and treated as a unit value. That is, a CSD represents the set of development organizations likely to use a particular collection of reusable parts. However, CSDs are not arbitrary collections of projects, since explicit *cost sharing mechanisms* must exist to define CSD membership and facilitate the equitable distribution of cost. Examples of such CSD cost sharing mechanisms include:

- Commercial sale of software, in which cost sharing is achieved via sales revenues from buyers of the products.
- Use of a joint capital expense pool by several projects.
- Consortia, in which multiple companies create a mechanism for sharing the costs of developing tools or technologies that can be reused by all of the member companies.

CSDs provide a convenient framework for discussing the prerequisite conditions for successful reuse. They are also useful for defining a number of cost issues that relate producer costs to consumer costs. For example, Figure 7 compares costs at the project level with those of the CSD. A factor is termed an *investment cost factor* if it results in (producer) project costs that are higher and CSD costs that are lower. Similarly, factors can be considered to be *assets, exploitative,* or *liabilities,* depending on their cost impacts.

Cost Factor Type	Project Costs	CSD Costs
Asset	Lower	Lower
Investment	Higher	Lower
Exploitative	Lower	Higher
Liability	Higher	Higher

Figure 7 – Major Types of CSD Cost Factors

3.2 CSD Banks

Thus far, amortization has been presented in terms of coverage of the cost of reuse investment at the time that the work product is originally constructed. A customer may occasionally be willing to pay the cost if plans are firm to purchase additional future systems that will reuse those parts. However, the risk in such a situation may be high, since the purchases may not actually take place. Thus, a more general solution is preferable, requiring that the organization pay for the investment cost with independent, non-customer funds.

The appropriate source of these funds is the CSD, since by definition the CSD offers the mechanism by which early development costs are amortized over subsequent instances of reuse. To effect payment, the CSD must have an actual pool of funds or resources available to it at the time of the reuse investment.

This need for an initial pool of reuse investment funds or resources leads naturally to the idea of a *CSD Bank*, an entity within the CSD that provides the initial capital for building reusability into products. The strong financial implications are intentional, since they point out that the investment in reusability is a "real" investment of actual capital, one which must be monitored and controlled with some degree of precision. Indeed, a company could make use of an actual commercial bank to play the role of the CSD Bank, provided that the company can provide sound estimates as part of a convincing argument for why such an investment would be mutually beneficial. For example, a company with limited funds could approach a commercial bank to cover the cost of generalizing an internally developed tool; the tool may show significant promise as a marketable product, or even as an aid that will never be distributed outside of the company. Such situations highlight the need for effective techniques both for evaluating reuse potential and for determining likely investment costs. Such estimates are vital in obtaining "hard cash" reuse investment loans from external agencies.

The allocation of funds by a CSD to a project is an example of a *Reuse Investment Loan (RIL)*. As with the CSD Bank, RILs must exist in some form in any organization that is investing in reuse; however, they may be hard to recognize if accounting procedures do not acknowledge them explicitly. For example, projects using overhead funds to build reusability are in effect acquiring RILs from their company, but the cost behavior and net payoff of the RILs is likely to be hidden by the lack explicit reuse-oriented accounting procedures.

3.3 CSD Resource Pools

Given that the mechanism for making reuse investments within a CSD is similar to a loan, the next step in building a structure to support reuse is determining how the funds are to be repaid. The source of revenue for repayment derives from the amortization charges assigned to later consumer projects. Thus, the collection of funds must be addressed.

Producer projects are not normally responsible for recovering reuse amortization charges from later activities, since there is no guarantee of the continued existence of those projects. Instead, amortization charges must be handled at the CSD level, and thus are the responsibility of the CSD Bank. To charge reuse customers for products, the CSD Bank must in effect be the owner of all of the reusable products which it has helped to fund. Collectively, this pool of reusable resources owned by the CSD Bank is the *CSD Resource Pool.*

A RIL is paid in full by a producer project when the reusable product is delivered to the CSD Bank. The bank then places the product in the CSD Resource Pool. Thus, a RIL may be viewed in one of two ways: as a loan that is repaid by turning over the reusable product to the CSD Bank, or as a form of internal contract for the development and delivery of a reusable product to the CSD Bank.

3.4 The Reuse Cost Cycle

The *Reuse Cost Cycle* is a general model of reuse that results when the concepts of reuse investments, reuse producers, reuse consumers CSDs, CSD Banks, RILs, and CSD Resource Pools are combined. Figure 8 shows the overall flow of funds and products that define this cycle.

The components of the reuse cost cycle can be viewed as variable factors that can take many forms. Thus, the CSD Bank can (and in many cases perhaps should) represent an actual staffed function that requires RIL application forms to be filled out and verified before funds are allocated for reuse. Alternatively, the Bank can be embedded in a distributed barter economy in which employees invest free overtime in hopes of later cash payment for each use of their components. The CSD itself may use various types of cost sharing mechanisms; its boundaries may fall within a single large project, across several related projects within one company, across an entire company, over a number of companies via a consortium, or over a very large base of companies and personnel when costs are shared by offering a reusable product for sale on the commercial market.

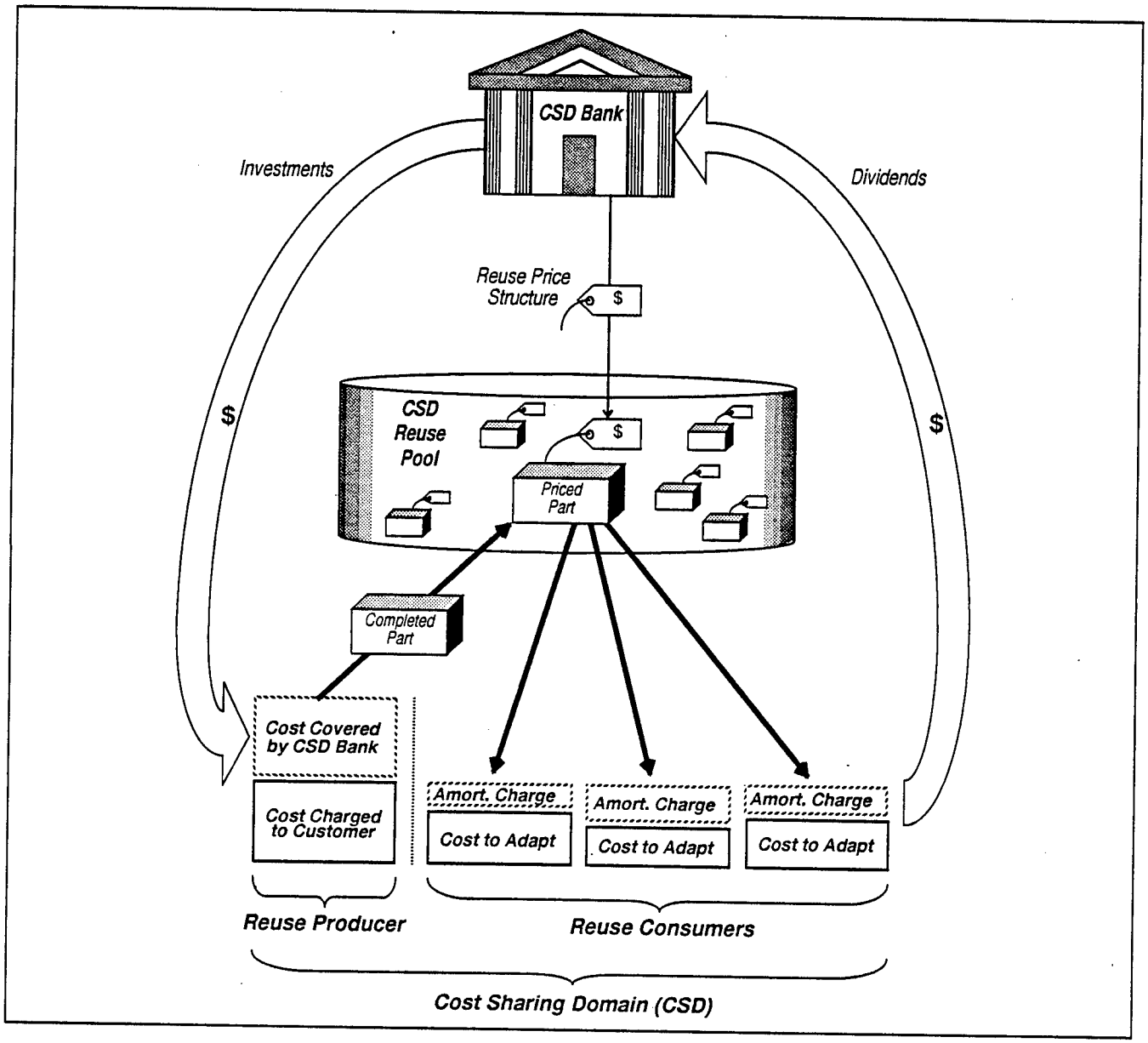


Figure 8 – The Reuse Cost Cycle

Viewed from a cost modeling perspective, each instance of consumer reuse in the reuse cost cycle has two distinct components. The *amortization cost* that is incurred when a project uses a reusable part is modeled as a fixed price procurement. The *adaptation cost* is the labor effort needed to find, adapt, and integrate a reusable part. In some cases, the adaptation cost may be very small, resulting in procurement activity with no new development activities. In most cases, however, the reuse instance will result in a mix of the two types of cost.

A cost estimation method, incorporating the concepts presented here, has been described as *delta cost estimation* in the work of Bollinger and Pfleeger[5]. The "delta" refers to the measurement of impact with respect to the baseline project. By adding this method to the reuse cost cycle, a tool can be developed that will allow an organization to evaluate the trade-offs involved in building and reusing components. Figure 9 illustrates a possible design for such a tool.

4. CONCLUSIONS

Reuse is a technology that needs strong, well-defined support at the organizational level in order to succeed. Unlike many of the more traditional software technologies, software reuse and decisions about when to use it cannot readily be separated from associated cost issues. Failure to address clearly the cost considerations of the reuse cost cycle more often than not will result in a breakdown of that cycle, resulting in significant loss of potential savings.

An important step in supporting reuse at the organizational level is the ability to describe and evaluate long-term reuse investment cost issues clearly. Software purchase is often regarded as a financial investment, subject to many of the considerations introduced in this report. However, viewing reusability as a financial investment is an outlook that has rarely been taken in the past. Moreover, the estimation of cost at the multi-project level is far from being a common practice. This paper urges software developers to consider reuse in a broader framework, one that encompasses many aspects of producer and consumer interactions. The idea of a baseline project, the concepts of cost sharing domains and banks, the reuse benefits inequality, and the delta cost estimation method (described elsewhere) lay the groundwork for a cost estimation tool that will enable management to make informed decisions about the economics of reuse.

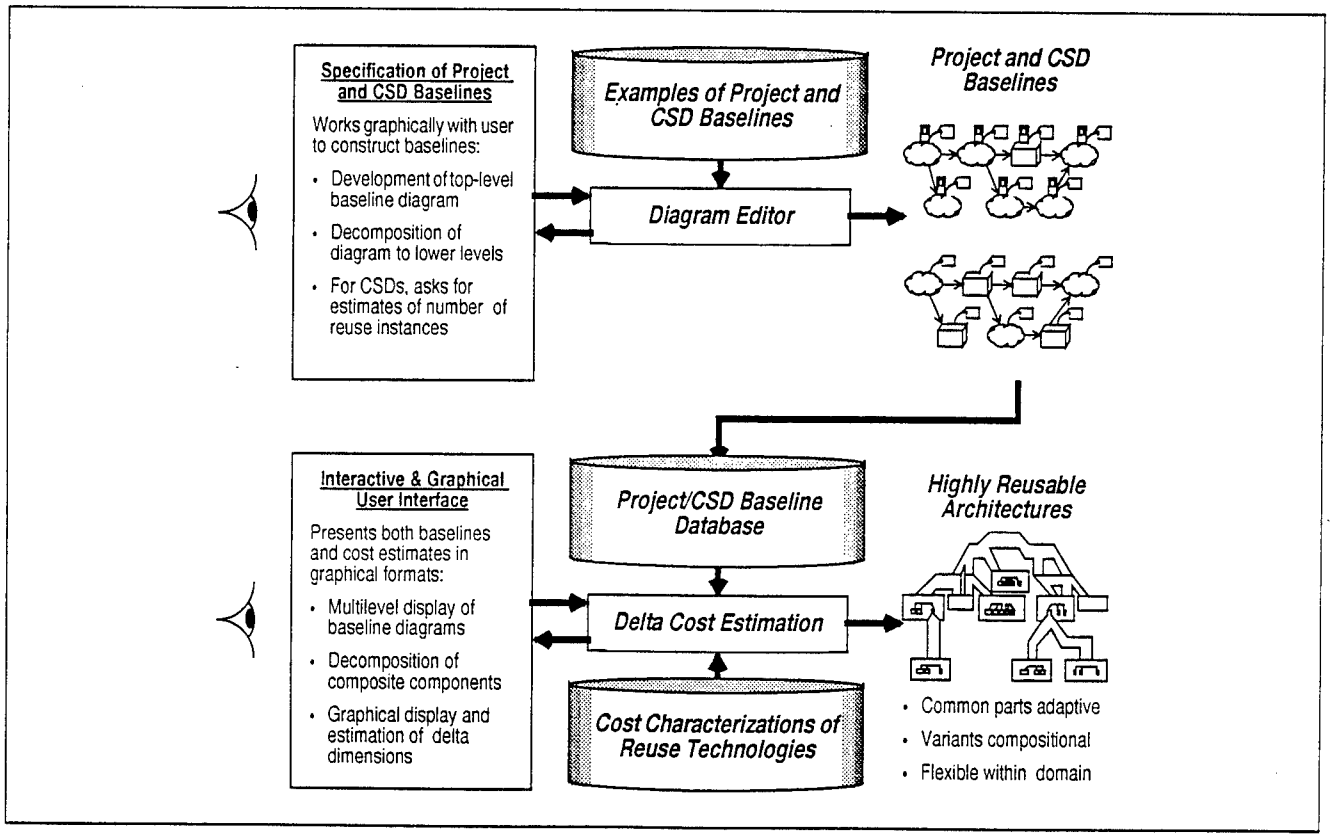


Figure 9 – A Delta Cost Estimation Tool

5. REFERENCES

1. Sidney C. Bailin, "Informal Rigor: A Practical Approach to Software Reuse," *RMISE 1987 Workshop on Software Reuse*, October 1987.

2. Paul G. Bassett, "Frame-Based Software Engineering," *IEEE Software*, July 1987, pp. 9-16.

3. Ted Biggerstaff and Charles Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, March 1987, pp. 41-49.

4. Terry Bollinger and Bruce H. Barnes, "Reuse Rules: An Adaptive Approach to Reusing Ada Software," *Proceedings of AIDA-88*, George Mason University, Fairfax, Va., November 1988, pp. 14-1 - 14-8.

5. Terry Bollinger and Shari Lawrence Pfleeger, "The Economics of Software Reuse," Contel Technology Center Technical Report CTC-TR-89-014, Chantilly, Virginia, 1989.

6. Bruce A. Burton et al., "The Reusable Software Library," *IEEE Software*, July 1987, pp. 25-33.

7. CAMP Program Description brochure, 1986.

8. Anthony Gargaro, "Reusability Issues and Ada," *IEEE Software*, July 1987, pp. 43-51.

9. Kyo C. Kang and Leon S. Levy, "Software Methodology in the Harsh Light of Economics," *Journal of Information and Software Technology*, June 1989.

10. Brian W. Kernighan, "Reusability in the Smalltalk-80 Programming System, *Proceedings of the Workshop on Reusability in Programming*, ITT, Shelton, Conn., Sept. 1983, pp. 235-239.

11. Manfred Lenz, Hans Albrecht Schmid, and Peter F. Wolf, "Software Reuse through Building Blocks," *IEEE Software*, July 1987, pp. 34-42.

12. Bertrand Meyer, "Reusability: The Case for Object-Oriented Design," *IEEE Software*, March 1987, pp. 50-64.

13. Bertrand Meyer, "Eiffel*: A Language and Environment for Software Engineering," *The Journal of Systems and Software 8*, 1988,pp. 199-246.

    *Eiffel is a trademark of Interactive Software Engineering, Inc.

14. Roland T. Mittermeir and Marcus Opus, "Software Bases for Flexible Composition of Application Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 4, April 1987, pp. 440-460.

15. James M. Neighbors, "The Draco Approach to Constructing Software From Reusable Components," *Proc. Workshop on Reusability in Programming*, ITT, Shelton, Conn., Sept. 1983, pp. 167-197.

16. Shari Lawrence Pfleeger, *An Investigation of Cost and Productivity for Object-Oriented Development*, Doctoral Dissertation, George Mason University, May 1989.

17. Shari Lawrence Pfleeger, "A Reuse-Oriented History of Cost Estimation Models," Contel Technology Center Technical Report CTC-TR-89-020, Chantilly, Virginia, 1989.

18. Ruben Prieto-Diaz and Peter Freeman, "Classifying Software for Reusability," *IEEE Software*, Jan. 1987, pp. 6-17.

19. Ruben Prieto-Diaz, "Domain Analysis for Reusability," *Proceedings of COMPSAC 87*, Tokyo, Japan, Oct. 7-9, 1987, pp. 23-29.

20. Will Tracz, "Reusability Comes of Age," *IEEE Software*, July 1987, pp. 6-8.

21. *Proceedings of the Workshop on Reusability in Programming*, ITT, Shelton, Conn., 1983.



Terry B. Bollinger has over 12 years of experience in systems analysis, artificial intelligence and software methods. His current research at the Contel Technology Center focuses on the economics of software reuse and the overall characteristics of reuse-intensive software development processes. Mr. Bollinger is the coauthor of several papers on reuse economics, and he and Shari Lawrence Pfleeger have recently coauthored several research reports for AIRMICS, an Army software research organization. Mr. Bollinger holds B.S. and M.S. degrees in computer science from University of Missouri at Rolla, and is a member of the IEEE Computer Society.



Shari Lawrence Pfleeger has over 15 years of experience in software development and software engineering research. She has designed, built and tested systems, managed software development projects, and provided consulting expertise to major corporations and government agencies. Her textbooks on software engineering and discrete mathematics are being used at universities both at home and abroad. As head of the software metrics program at Contel Technology Center, Dr. Pfleeger's current research focuses on requirements metrics, cost modeling, and the embedding of appropriate metrics in the process maturity hierarchy. She holds a B.A. from Harpur College, an M.A. in mathematics and M.S. in planning from the Pennsylvania State University, and a Ph.D. in information technology from George Mason University. Dr. Pfleeger is a member of ACM, IEEE Computer Society, and Computer Professionals for Social Responsibility.

ECONOMIC ANALYSIS OF SOFTWARE REUSE
A MODEL STUDY

Dr. Harry F. Joiner

Telos Corporation, 55 N. Gilbert Street, Shrewsbury, NJ 07702

Abstract

There has been significant interest recently in the advantages of software reuse from both a programatic and economic perspective. In this study, a cost estimation model based on the Intermediate Constructive Cost Model (COCOMO) [1] is used to analyze the cost and schedule benefits of incorporating increasing amounts of reused software design and code. If as much as 90 percent of the design and code are unmodified, the level of effort estimated for the development of a 100,000 delivered source instruction (DSI) project will be reduced by over 60 percent, and the estimated schedule will be reduced by 26 percent from the corresponding values if all of the code and design were newly developed.

The REVIC Model

Ray's Enhanced Version of Intermediate COCOMO (REVIC) has been developed and calibrated by Raymond L. Kile, U.S. Air Force Headquarters Command, Albuquerque, New Mexico, and includes the capability to estimate projects which incorporate reuse. This study was performed using REVIC version 8.7.2 (15 June 1989) [2] and an assumed project size of 100,000 DSI.

The Environmental Factors for REVIC are only slightly different from the Intermediate COCOMO and are given (with the values used for this study) in Table 1. The Ada development mode was chosen, and all other Factors were set to Nominal except Required Security (unclassified) and Management Reserve for Risk (very low), resulting in an Environmental modifier value of 1.00. Changing the Environmental Factors will result in a constant multiplier of these results, but would not change the percentage results unless a different environment were expected for the reuse environment than for the non-reuse one.

The Required Reuse Factor refers to the development of reusable software components and not the reuse of components. This study assumed that this requirement was nominal.

The Reuse Parameters

REVIC allows for the input of a number of modules each with or without adapted code and design as specified by the user. Three factors are assigned by the user in the Adapted Code portion of the model: Design Modified (DM) percentage (range from 0 to 100), Code Modified (CM) percentage (range from 0 to 100), and Integration Modified (IM) percentage (range greater than or equal to 0).

Table 1. The REVIC Environmental Factors Used for the Reuse Study

ENVIRONMENTAL FACTOR	RATING	VALUE
Analyst Capability	NM	1.00
Programmer Capability	NM	1.00
Applications Experience	NM	1.00
Virtual Machine Experience	NM	1.00
Prog. Language Experience	NM	1.00
Execution Time Constraint	NM	1.00
Main Storage Constraint	NM	1.00
Virtual Machine Volatility	NM	1.00
Computer Turnaround Time	NM	1.00
Requirements Volatility	NM	1.00
Product Reliability	NM	1.00
Data Base Size	NM	1.00
Product Complexity	NM	1.00
Required Reuse	NM	1.00
Modern Programming Practices	NM	1.00
Use of Software Tools	NM	1.00
Required Security	UN	1.00
Mgmt. Reserve For Risk	VL	1.00
Required Schedule	NM	1.00
Software Development Mode	ADA	1.00

Table 2. Schedule, Level of Effort, and Cost Comparison for Various Levels of Reuse in a 100,000 DSI Project

ITEM	DM (%)	CM (%)	NEW CODE (KDSI)	EFFORT (MM)	SCHEDULE (MONTHS)	COST ($MILLION)
A	10	10	37.0	272.5	37.9	$3.023
B	10	30	43.0	313.8	39.6	$3.482
C	10	60	52.0	375.3	42.0	$4.164
D	10	90	61.0	436.1	44.0	$4.839
E	30	10	45.0	327.6	40.2	$3.635
F	30	30	51.0	368.5	41.7	$4.089
G	30	60	60.0	429.4	43.8	$4.765
H	30	90	69.0	489.7	45.7	$5.434
I	60	10	57.0	409.2	43.1	$4.540
J	60	30	63.0	449.6	44.5	$4.988
K	60	60	72.0	509.8	46.3	$5.656
L	60	90	81.0	569.5	48.0	$6.319
M	90	10	69.0	489.7	45.7	$5.434
N	90	30	75.0	529.7	46.9	$5.878
0	90	60	84.0	589.3	48.5	$6.539
P	90	90	93.0	648.6	50.0	$7.197
Q	100	100	100.0	694.4	51.1	$7.705

DM and CM represent the percent of the adapted software's design and code, respectively, that is modified in order to adapt it to the new objectives and environment. The number of DSI includes executable statements, format statements, and data declaration, including those in supporting software such as test drivers, etc. DSI excludes comments, blank lines, and unmodified utility software.

The values of DM and CM were varied independently using the values of 10, 30, 60, and 90. The standard case of 100 for both DM and CM is used as the baseline. Table 2 contains a summary of the results. Since the percentages in DM and CM are for the amount of code modified, they are inversely related to the amount of reuse; e.g., 10% DM represents 90% reuse of design, and 100% represents no reuse.

IM assesses the impact upon integration and test of the reuse and represents the percentage of the normal development effort in this area that is expected because of the reuse. More effort could be required if the reused components were inadequately understood or tested; or less effort could be expected if well tested design and code were being incorporated in the project. For the study, a value of 100 was used for IM.

Results

Table 3 presents the results for the effort and schedule in increasing order. The savings of over 60% on the level of effort for the 10-10 case shows the dramatic savings available in projects for which substantial reuse is practical. Add to that a schedule reduction (unconstrained) of more than 25% and the economic benefits of software reuse become obvious. A project that might cost $7.7 million, in this instance, could be done for approximately $3.0 million by maximizing reuse. The benefits contained in this study reflect only the development portion of the life cycle and are, therefore, only partial. Significant additional savings can be obtained during the life cycle by reduced maintenance costs, improved testing, and greater reliability that can be expected from more widely used software components.

Another aspect of the results in Table 3 is that the design reuse is more significant than the code reuse in reducing the level of effort and schedule. In essence, the earlier in the development process the reuse takes place, the more benefit is gained.

Table 3. Schedule and Level of Effort Improvement Through Reuse

ITEM	PERCENT MODIFIED DM	PERCENT MODIFIED CM	SCHEDULE MTHS	SCHEDULE %	EFFORT MTHS	EFFORT %
A	10	10	37.9	74.2	272.5	39.2
B	10	30	96.6	77.5	313.8	45.2
E	30	10	40.2	78.7	327.6	47.2
F	30	30	41.7	81.6	368.5	53.1
C	10	60	42.0	82.2	375.3	54.0
I	60	10	43.1	84.3	409.2	58.9
G	30	60	43.8	85.7	429.4	61.8
D	10	90	44.0	86.1	436.1	62.8
J	60	30	44.5	87.1	449.2	64.7
H	30	90	45.7	89.4	489.7	70.5
M	90	10	45.7	89.4	489.7	70.5
K	60	60	46.3	90.6	509.8	73.4
N	90	30	46.9	91.8	529.7	76.3
K	60	90	48.0	93.9	569.5	82.0
O	90	60	48.5	94.9	589.3	84.9
P	90	90	50.0	97.8	648.6	93.4
Q	100	100	51.1	100.00	694.4	100.0

That these estimates represent a realistic goal is illustrated by one recently developed system for the Navy. The prime developer (Magnavox Electronic Systems Company) for the Advanced Field Artillery Tactical Data System, an Army Command and Control (C^2) system, used that system as basis for the development of a similar system for the Navy [3]. The Navy Force Fusion System Prototype (FFSP) Project was delivered at approximately 20% of the estimated cost had the system been developed from scratch. The FFSP project contained approximately 450,000 lines of code of which 92% was unmodified or slightly modified code. The project was completed in 12 months (see Table 4). Some of the modifications resulted in improved performance.

Conclusions

This study, or similar analysis, might be used to support a value engineering recommendation by a Government contractor or to support the requirement of reuse in new development projects by the Government. The benefits of reuse in the development process can be immense and the Government must receive those benefits in order to accomplish its long-term software requirements in a constrained budgetary environment.

REFERENCES

[1] Software Engineering Economics, Barry W. Boehm, Prentice-Hall, Inc., 1981.

[2] REVIC version 8.7.2 (15 June 1989), Raymond L. Kile, USAF Systems Command HQ.

[3] Ada Software Reusability Demonstration/Project Report of Force Fusion System Prototype, Magnavox Electronics Systems Company, 1989.

Table 4. Navy FFS Prototype Code Reuse Estimate
(all line counts in 1000's)

CPCI	AFATDS TOTAL	REUSE AS IS	REUSE MOD	REUSE TOTAL	NEW	FFS DEMO TOTAL	REUSE TOTAL %
Operating System	47	45	2	47	1	48	98%
Communications Support	43	43	0	43	0	43	100%
Data Management	106	106	0	106	0	106	100%
Information Management*	151	147	4	151	0	151	100%
Communication Interface**	40	38	2	40	7	47	85%
Message Manager	90	9	5	14	8	22	64%
Status Utility	85	4	4	8	12	20	40%
Data Base Display	15	1	1	2	10	12	17%
TOTALS	577	393	18	411	38	449	92%

* Majority of effort in IM involves format library changes for Navy Symbols.
** Navy interfaces added to current Army interfaces.

BIOGRAPHY



Dr. Harry F. Joiner

Manager for Software Metrics
Telos Corporation
Shrewsbury, New Jersey

Dr. Joiner joined Telos Corporation as a software engineer in August 1986, working on the Firefinder field artillery locating radar. He assumed his current position as Manager of Software Metrics in March 1988. Telos is the largest software engineering support contractor to CECOM CSE.

Before joining Telos, Dr. Joiner served as a consultant on project management and digital signal processing to various oil companies. He has over 20 years of experience in mathematical modeling, digital signal processing, engineering, and project management.

Dr. Joiner has a BA in Mathematics and Chemistry and an MS and Ph.D. in Mathematics. He has served on the faculties of the University of Massachusetts and Texas Christian University.

REUSABLE Ada PRODUCTS FOR INFORMATION SYSTEMS DEVELOPMENT (RAPID)
Lessons Learned During Pilot Operations

Terry Vogelsong & Jack Rothrock

U.S. Army Information Systems Software Development Center - Washington
ATTN: ASQBI-WSS-R STOP H-4
Fort Belvoir, VA 22060-5456
AV 356-6202 COMM (703) 285-6202

ABSTRACT

RAPID Phase I concluded in Janurary 1989 with the delivery of the RAPID Center Library system, initial policies and procedures, and guidelines and standards for reusability and portability. Currently, the RAPID Center is in a 24-month Pilot Operation - Phase II. This paper addresses some of the lessons learned to this point in the Pilot Operation.

INTRODUCTION

In these times of fiscal and manpower constraints, innovative ways to "do more with less" must be discovered. Ada, sound software engineering practices, and reuse are ways to accomplish this goal. To effectively put the technology of software reusability of Ada code into practice, Reusable Ada Products for Information Systems Development (RAPID) was initiated within the U.S. Army Information Systems Software Development Center - Washington (SDC-W), located in Falls Church, Virginia. The RAPID Project has resulted in a reusability program within SDC-W as well as the automated tools required to operate such a project.

SDC-W is one of five Software Development Centers within the U.S. Army Information Systems Software Center (ISSC). The RAPID Project's role in the Command is to promote the reuse of Ada components, and further, to reduce the cost of system development and maintenance. Reusable Software Components (RSCs) can include design, documentation, test data, code -- virtually anything considered "reusable." The RAPID mission: "Ensure the Department of Defense (DoD) objective of reusable, maintainable, reliable Ada software is achieved within ISSC by developing, implementing, maintaining and administering a total Ada software reuse program."

Overview

RAPID was initiated at SDC-W as a pilot prototype reuse program in July of 1987 when the Phase I contract was awarded to SofTech, Incorporated, in Waltham, Massachusetts. Phase I, completed in January 1989, included developing the RAPID Center Library (RCL) system, authoring RAPID Center Operation and Procedure documents for both reusable code and the operation of the RAPID Center, and establishing Guidelines and Standards for reusability and portability.

The RAPID Center is currently in a 24-month Pilot Operation - Phase II. During the first twelve months, a single Standard Army Management Information System (STAMIS) is being supported extensively. The STAMIS partner selected is Standard Installation / Division Personnel System - 3 (SIDPERS-3). The first twelve months of the Pilot Operation is a RAPID "Proof of Concept." The remaining twelve months of the Pilot Operation will test the feasibility of servicing all five Software Development Centers within ISSC. Phase III will expand RAPID Center services to all of the U.S. Army Information Systems Engineering Command (ISEC), the Department of the Army (DA), and beyond, as needed and as funding allows.

RAPID has found many issues during the Pilot Operation which impact the successful reuse of components. RAPID has resolved some of these issues, but several more remain before reuse can truly be successful. This paper will discuss RAPID solutions and some remaining reuse issues in the form of lessons learned. With a limited scale of operation such as the RAPID Pilot Project, mistakes and reusability issues can be detected early and corrected or resolved with minimal disruption to both developers and the RAPID Project.

Why Reuse

The first lesson learned during the Pilot Operation was the need to justify a reuse program. Several extreme views of reuse exist from strong advocacy to skepticism. Below are some opinions on reuse from selected leaders in the field of Ada:

Consultant Ralph E. Crafts, President of Software Strategies and Tactics Inc., in Harpers Ferry, West Virginia argues that, "Congress could meet the Gramm-Rudman deficit reduction target simply by establishing a national software engineering policy with Ada as the standard language. The 75 percent of the government's software budget that goes to maintain old code would drop drastically under an Ada standard. And if reusability claims hold true, Ada could alleviate the overwhelming demand for software and the dearth of qualified programmers."[1]

Christine M. Anderson, Manager of the Ada 9X Project states, "The biggest obstacle to software reuse is a lack of software components. If government helps fund the start-up costs, corporations will develop more software."[2]

According to John P. Solomond, director of DoD's Ada Joint Program Office, reusability may be overstated. "Software vendors are reluctant to use programs they did not develop themselves. They fear being held liable if something goes wrong with someone else's program."[3]

RAPID has collected numerous studies and visited projects such as Common Ada Missile Packages (CAMP), all showing reuse to be technically feasible when using Ada. Reuse of low level utility routines (sorts, windows, etc.), stand-alone systems (compilers, databases, etc.), and total systems (SIDPERS, Standard Finance System (STANFINS), are STAMISs used at hundreds of sites) is common. Even ad hoc reuse is more common than engineered reuse. One of the factors contributing to the success of new development efforts is the "reuse" or moving of experienced personnel from project to project. RAPID is attempting to formalize and standardize reuse efforts. Reuse stops the continual re-invention of components for common functions and improves the quality of the product through the use of thoroughly tested and debugged modules.

It has been proven, with all other factors being equal, that reuse in conjunction with sound software engineering results in a project being completed more quickly, cheaper, and with less effort than normal software development processes. Additionally, reuse allows for better utilization of personnel. Personnel are not expended to re-write common components. Experienced programmers / analysts can concentrate on the difficult, unique problems of the development project.

The most overwhelming fact supporting reuse is the decreased cost of maintenance. Not only are reused components less likely to need "fixes," but with several users using the same components, maintenance and enhancement costs are spread over several projects. During the Pilot Operation, it was learned that a strong Configuration Management (CM) Plan that controls and releases fixes to extracted components and provides notification of enhancements to components was needed. Built into the system was a tracking mechanism that controls extracted RSCs both by individual and project. Strong CM Procedures have been written and the RAPID project has selected Change and Configuration Control (CCC) by Softool, Incorporated, as the CM tool of choice to control and record changes / enhancements.

RAPID Center -- NOT Just a Repository

The RAPID Center is a **TOTAL** reuse program -- not just a repository or library. Initially, RAPID had a reputation for being merely a library. However, the RAPID Center is a complete reuse program that supports the entire software development life-cycle. Reuse will not happen by itself even though studies have shown significant savings due to reuse. An organization like RAPID is needed to establish guidelines and procedures that encourage reuse and sound software engineering. As a result, RAPID learned that in order to support a reuse program, published guidelines and standards were needed. These guidelines and standards are available upon request and have been provided to over 500 individuals and companies. RAPID has the following documents available to aid in the development of Reusable Software Components (RSCs):

a. ISEC Reusability Guidelines, December 1985
b. ISEC Portability Guidelines, December 1985
c. RAPID Center Standards for Reusable Software, January 1989

To further support the RAPID reuse effort, it was determined that an ISSC Command policy was needed. Therefore, RAPID is staffing a Software Reuse Policy and Procedures document that defines the usage of the RAPID Center, responsibilities of software developers and/or their contractor representatives, and lists source documents. This Policy establishes directives, goals, and procedures that effectively promote reuse within the Command.

RAPID Staffing

To aid the evolution of a TOTAL reuse effort, personnel duties and responsibilities were defined. RAPID has a staff tailored to perform and train Ada reuse, encouraging design methods and architectures built from reusable components. The staff consists of:

RAPID Center Manager: Oversees all RAPID Center activities, continually monitors the success of the RAPID program, and keeps extensive records that aid in determining the cost of the program and savings provided through cost avoidance to developers and the Army.

Technical Consultants: Perform domain analysis, attend design reviews and other pertinent reviews, stay abreast of projects, advise project staffs, assist the developer to identify potential areas of reuse, assist in the search for RSCs, and provide guidance and support to programmers integrating reusable components and supporting documentation into applications.

System Analysts and Software Engineers: Identify high value RSCs to be added to the library; evaluate, thoroughly test, and document RSCs before being classified and placed in the library; and, provide maintenance and enhancements to the RCL software system.

Configuration Management Specialist: Ensure all configuration activities are performed for RCL and RSCs including problem report tracking, controlling changes, and releasing new version or enhancements.

Quality Assurance Specialist: Ensure RCL and RSCs are of high quality through frequent reviews, developing and administer testing, and establishing and enforcing metrics.

Administrative Assistant: Prepares all RAPID Center Library System reports and performs follow-up interviews with users.

RAPID Center Librarian: Maintains the RCL data base and performs normal computer operator functions.

Domain Analysis

As part of the research done to establish RAPID, an initial high-level domain analysis was done in 1987. This domain analysis covered only information management systems, i.e., financial, logistical, tactical management information, communication, personnel/force accounting, and miscellaneous "other software" systems. During the Pilot Operation, RAPID realized the need to support multiple domains. Therefore, policies, procedures, and guidelines developed in support of RAPID are generic and evolutionary, and are applicable to any domain or collection. Additionally, the RCL was designed to be easily adjusted to supported domains through deletions, modifications, or additions to search taxonomy.

RAPID has learned that performing a domain analysis should be the starting point for life-cycle support of software engineering efforts. A domain analysis should be performed prior to actual development to identify potential areas for reuse. However, performing a domain analysis is an iterative process throughout the project's life-cycle. As more development projects are supported by RAPID, the domain analysis effort expands evolving to a true MIS domain. Not only does preforming a domain analysis identify components that may be reused, but also directs developers to areas where reuse emphasis should be placed so that new components can be captured as part of post deployment system support.

RAPID support of SIDPERS-3 includes performing iterative domain analysis, attending design reviews, assisting in the search for applicable reusable components, training managers and programmers on reuse methodologies, and advising on sound software engineering practices that result in quality, easily maintainable design and code. Independent analysis of potential reuse opportunities performed prior to the RAPID / SIDPERS-3 partnership resulted in the identification of similar areas of potential reuse. A combination of the first phase of formal domain analysis and the informal SIDPERS-3 research of reuse resulted in identifying SIDPERS-3 High-Demand RSC Categories. These categories are: Screen / Window Management, Interfaces for Common Platforms,

Communications / Networking, Form, Table, and Report Generators, Access Security / Validation, Suspense / Schedule Management, Data Dictionaries, Relational Data Base Management, Floating Point Arithmetic, Ada-SQL Binding, Searching and Sorting, Date / Time Conversion, File Maintenance, External Entity Interface Management, and Basic Software Engineering Entities.[4]

RAPID Center Library Usage

The purpose of the RAPID Library tool confused several users as actual components are never shown on the screen. This is because the RCL is similar to a card catalog in a library. Just as a card catalog provides potential "books" to be read, the RCL provides a list of "candidate" RSCs to be potentially extracted. The user identifies the requirements of a component needed through a faceted classification scheme. The RCL then takes the description, searches for, and displays a list of candidate RSCs. The system has several internal tools that aid in reducing the list to two or three RSCs to be extracted. Just as when using a card catalog, a user checks out the books to be read, the RCL user can download the components identified as potentially solving the user's needs. Just as books are read after using a card catalog, RCL users, after downloading, must off-line analyze in detail the components, looking at the code, reuser's manual, and test drivers to determine which single component is actually reused.

Information captured about an RSC, as part of the RAPID staff analysis and testing of a component, is available to the user during a "search session." Some of the information displayed to the user to aid in the selection of components which will be extracted is as follows:

a. Number of times an RSC is extracted,
b. Number of reported problems and a detailed description of the problems,
c. A reusability and a complexity measure,
d. RSC abstract, how the RSC is classified, and supporting documents available, and,
e. Related RSCs.

Classification Scheme Effectiveness

RCL uses a faceted classification scheme to store and retrieve RSCs. This scheme is based upon a study by Ruben Prieto-Diaz.[5] Facets represent different ways of looking at a component. For each facet, specific descriptive terms, called facet terms, classify an RSC within that facet. Terms with the same meaning are known as synonyms. Not every facet need be employed in classifying an RSC. More than one facet term may be given for a single facet. Component classification can be changed or augmented as required.

A limitation of the present RCL structure for supporting more than one domain / collection was discovered during the RAPID Pilot Operation. The existing terminology and taxonomy is effective for classifying components in the MIS domain. Layers will be added to expand the support to new domains. Through layering, a user will be asked to identify a domain / collection, followed by the type of component desired. The system will then list the facets that support that domain and component type.

Also discovered during the Pilot Operation was the trade-off of adding a new term versus applying that new term to stored RSCs. Adding a synonym has minimal impact on both classification of RSCs and the terminology used. However, when adding a new term, should all stored RSCs be analyzed to determine if the new term equally applies? Through actual classification of RSCs, it was discovered that new terms identified were applicable to only the new RSC.

Reusable Software Components

During the RAPID Center Pilot Operation, the staff has been populating the library with RSCs. However, it became evident that Reusability Standards and Guidelines were not enough to standardize the evaluation and analysis of components. A Reusable Software Components Procedures document was created describing procedures to follow from "cradle to grave" for RSCs. The RSC Procedures are supplemented by RAPID Configuration Management Procedures and a RAPID Evaluation Plan. These documents capture "how" RSCs are handled, give general "rule of thumb" principles, and lay the groundwork to determine cost savings. A standardized way of doing business was created which resulted in procedures being followed not only by contractors, but by government personnel as well.

Sources of RSCs include, but are not limited to, RAPID developed components, selected components from fielded systems, commercial-off-the-shelf components, reviews of ongoing development projects, and public domain repositories. RAPID attempted to gain access to ongoing projects' components and encountered justifiable resistance. Just as RAPID is unwilling to release uncertified components, developers are unwilling to release their systems prior to final delivery. This has hindered the population of the library with "government" generated code. In the interim, RAPID is attempting to develop procedures on handling large amounts of code that become available after system delivery.

RAPID requires that all RSCs be evaluated using the checklist below:

a. Review for portability in accordance with ISEC Portability Guidelines.
b. Review for reusability in accordance with ISEC Reusability Guidelines.
c. Review for reliability.
d. Review for maintainability.
e. Review for proper testing and test data.
f. Review for complete documentation - abstract, reuser's manual, function, interfaces, etc.

This evaluation can be extremely labor intensive. Working closely with Dynamic Research Corporation, creators of AdaMAT, a spreadsheet was created that maps AdaMat metrics to paragraphs in the RAPID Reusability Standards. As a result, RAPID is now using an automated tool to measure and evaluate the quality and software factors of reusability, reliability, portability, and maintainability of components. This tool, along with visual evaluation using the checklist above, promotes the goal of including components that are of high quality, documented, and tested.

User Communications

The RCL system logs a variety of information for the purpose of tracking its performance, evaluating possible changes to enhance the system, and triggering RAPID Center activities. During the Pilot Operation, users have provided suggestions through the suggestion box that have been instrumental in determining corrections and enhancements not only to the RCL but to components stored in the library. The search failures report is used to determine the need for new terms or synonyms and in some cases, the need for a new RSC. These, and other internal tracking mechanisms, have generated unique communication with the users. As a direct result, the RCL is being tailored to support the user's desires and is becoming a tool that users are requesting access to.

A critical procedure implemented during the Pilot Operation was obtaining user "feedback" on extracted components. When a user extracts a RSC from the library, the system automatically generates a suspense "feedback" date of 90 days. Within that timeframe, users are required to provide information about successes or problems with the RSCs. User information is analyzed and several actions may be taken, as appropriate: updating the RSC's use history in the library, initiating a problem report, recommending enhancements to the RSC, recommending new RSCs, or recommending changes to the library search apparatus. Feedback is solicited about functional fit, cost savings, ease of installation, actual versus expected performance, problems, recommendations for improvement, and any other user comments. Subsequent RAPID users find recorded feedback helpful when analyzing a candidate list for selection of components to be extracted.

Entire Life-Cycle Support

It is a common misunderstanding that reuse includes only reuse of code. However, RSCs can include requirements, design, documentation, test data, code -- virtually anything considered "reusable." Significant gains can be obtained through the reuse or "salvaging" of parts necessary for building complete systems. Technically, code is easy to reuse, but other components have the potential for the greatest pay-back. This lesson learned caused a change in the name of the project from Reusable Ada Packages for Information Systems Development to Reusable Ada **Products** for Information Systems Development.

Present problems with components other than code are the lack of standards or policy on what is considered a "good" design or requirement specification. Design analysis tools are being developed as a part of the U.S Army Institute for Research in Management Information, Communications and Computer Science (AIRMICS) project at George Mason University in Virginia. Once tools are available and standards are established, RAPID will further exploit the reuse of designs and requirements.

In the interim, RAPID is targeting the reuse of code, documentation, and test data. This is being done primarily in the Management Information Systems (MIS) domain, with other domains to follow. Components are available through the RAPID Center Library (RCL) System. The RCL is an operational, interactive library system used for the identification, analysis, and retrieval of reusable components. The system was designed to be dynamic or modifiable to adjust to the supported domain. Modifications are performed through internal system library functions via a menu or keypad keys. The RCL was designed and developed to accommodate all components of the development life cycle. Components, be they requirements, design, or code, can be classified, stored and retrieved using the RCL. When extracting, a window appears that displays "related" RSCs. Through this window, all associated components are referenced.

The RAPID system will evolve to storing requirements. A potential search scenario for a requirement would be as follows.

a. User "describes" a requirement.
b. RCL search generates a list of "candidate" requirement components.
c. User analyzes, browses, and selects a requirement.
d. User extracts the requirement.
e. RCL displays, in the "related" RSC window, the requirement's supporting design and code.
f. User has the option of extracting all the information or only what is desired. However, the most rewarding retrieval focuses on requirements that are accompanied by the design and code.

Reuse Training

It became apparent during the Pilot Operation that reuse is misunderstood by a majority of individuals. The mind set that many programmers / analysts have of pride of authorship, of never using another's code for fear of poor quality, and the moral and ethical aspects of responsibility and accountability are all myths, perceptions, and attitudes that RAPID is attempting to dispel. Technically, reuse has been proven to work. RAPID is attempting to alter this "human" factor perception through training that explains reuse and its potential. Training by RAPID staff on RAPID and Ada reuse is performed at three levels and includes the following course topics:

Executive Level, 2 Hours - A high level overview of the RAPID Software Reuse Program. An overview of reuse throughout the life cycle - from conception through maintenance.

Managerial Level, 8 Hours - All of the Executive Level, plus a detailed overview of the services provided by the RAPID Center and the responsibilities of the software developer.

Programmer/User Level, 40 Hours - An intensive review of:

a. Principles of Software Reuse,
b. Principles of Domain Analysis,
c. Designing for Reuse,
d. RAPID Center Library System User Interface,
e. Overview of RAPID Center Guidelines and Standards for Reusable components,
f. How to Write Well Engineered, Reusable Components, and
g. How to Insert RSCs into the development life-cycle.

Beta Program

As a result of RAPID's success supporting Army MIS and the "generic" Ada development of the RCL, it was determined that an evaluation of RAPID's capability to support non-MIS domains / collections was needed. Therefore, the RAPID Beta Program was established in an attempt to study the ability of the RCL to support other significant software development efforts. RAPID's first Beta site is NASA's Space Station Freedom Project at the Johnson Space Center. There is an on-going effort to establish a Beta relationship with the Joint Integrated Avionics Working Group (JIAWG), a DOD chartered organization supporting development of common avionics baseline specifications and standards for the Navy A-12, the Air Force Advanced Tactical Fighter (ATF), and the Army Light Helicopter Experimental (LHX). A likely candidate as a final Beta site is a development effort within U.S. Army Communications and Electronics Command (CECOM). These Beta partners are providing RAPID valuable feedback, enabling the RAPID Project to evolve and support an expanding baseline of customers. The sites also provide input on reuse cost savings and cost avoidance, validating the need for reuse and a project such as RAPID.

Legal Impediments

RAPID has learned that the legal impediments to reuse are the greatest obstacle to successful reuse. Issues ranging from Intellectual Property Law to the liability of the software component developer have been widely publicized and difficult to resolve. The impact of legal issues is global as the Federal Acquisition Regulation (FAR), the Defense Federal Acquisition Regulation (DFAR), Intellectual Property Law (which involves data rights, copyright law, and patent law), and others are all affected. Compounding the problem is the issue of derivative rights where customers have unlimited rights to the software but do not have rights to make derivatives of any "part" of the software.

The tremendous scope of the legal issues problem left RAPID in a difficult position. How can a viable reuse program be implemented while many of these legal issues remain unresolved? RAPID took two distinct approaches. First, a DoD team was organized, headed by the ISSC's General Counsel, working jointly to resolve legal and data rights issues. Second, RAPID established an interim solution to the most critical issues impacting the Pilot Operation: data rights, liability, and warranty. In order to begin supporting contracted and government software development efforts, RAPID established the following interim criteria for components:

1. Developed components are required to have unlimited government rights or they will not be available in the library.

2. Commercial-off-the-shelf (COTS) components will contain copyright notices on the component's abstract. In addition, copyright notices will be imbedded in both the actual code and documentation.

3. Components are provided without a warranty or expressed liability.

Because components are provided without an expressed warranty, it was paramount that RAPID achieve a high level of user confidence. This was accomplished through a comprehensive set of quality control procedures and the use of automated analysis tools. As a result, RAPID's components are as thoroughly tested as current technology allows. They are reviewed by a technician and an analyst, and receive final approval by a RAPID Configuration Control Board before they are placed in the library. In addition, metrics obtained from AdaMAT and test data on each component are provided to the user as part of the extraction or as part of the analysis screens built into the RCL. As all stored components follow RAPID's quality guidelines and standards, user confidence in the components is very high.

As the RAPID program matures, these interim criteria will evolve to support accepted legal positions. Until each legal issue is resolved, RAPID has taken a proactive approach which potentially can provide the concrete data necessary to resolve many of the current legal impediments to reuse.

Incentives

Many of the basic "quality" incentives that apply to individual programmers and developers have already been discussed. A more difficult issue that the RAPID Program faced is that of incentives for the contractor. The Pilot Operation has yet to resolve the contractor incentives issue; however, several possibilities are being explored.

Most contractor incentives revolve around dollars. It must be in the contractor's competitive self-interest to develop reusable components and/or reuse components. Potential contractual arrangements include:

1. Cost plus award fee contract with a major part of the award fee based on the percentage of reuse.

2. A cost type contract based on productivity. Contractors would "need" to use reusable components to meet productivity requirements. A reimbursement policy can be established which would share savings resulting from increased productivity between the government and the contractor.

3. Contract a fixed license fee which includes contractor maintenance for reusable components. High quality components would be developed, resulting in lower maintenance costs. Reuse of these high quality components in subsequent development efforts would occur because of third party maintenance and dependability of components. This provides long term financial reward to both the contractor and the government.

Current acquisition standards, DoD Standards 2167A and 7935A, are ineffective when addressing incentive issues of reuse. The Software Engineering Institute at Carnegie Mellon University has done significant research in this area. As new software engineering technologies evolve with Ada, the Army Acquisition Executives will be required to establish policies which support emerging technologies. This should result in a more robust software engineering environment which supports the contractor and government goal of developing high quality software that is on-time and within budget. RAPID will continue to participate in the efforts to resolve the issues associated with both legal impediments and contractor incentives.

Summary

This paper discussed several reuse issues presented in the format of "Lessons Learned" during the RAPID Pilot Operation. Interim solutions and/or compromises had to be reached in an effort to accomplish RAPID "Proof of Concept." Many issues remain to be resolved and permanent solutions discovered. But with sound policy, cooperation with other reuse efforts, and attention to the needs and perceptions of the supported development staffs, the RAPID program will continue to provide solutions for reuse issues. Until software reuse becomes a "way of life," the RAPID Center must lead the way. The RAPID Center is one vehicle propelling the Army to new levels of increased productivity at reduced costs. RAPID is continuing to "lead the way" into the future being a leader of software reuse and solving many of the reuse issues of today.

About the Authors

Terry Vogelsong is currently the technical team lead on the RAPID project. He holds a Masters in Business Administration with a field of study of Information Systems Management from The George Washington University and a Bachelor of Science in Business Administration from Bowling Green State University. He previously served in the U.S. Army as a commissioned officer from 1980 to 1988 as a Finance and Automated Data Processing Officer. His current interests include exploring opportunities for reusing Ada code, design, and methodologies in Management Information System environments.

Captain Jack Rothrock is an Army Intelligence Officer currently serving as a RAPID Project Officer. He was assigned to SDC-W in 1988 following completion of a Master of Science degree in Information Systems Management from Eastern Washington University. His Bachelor of Science degree is in Business Administration from the University of Maryland. He has been actively involved in development of a maintenance capability for the Army World Wide Military Command and Control System Information System (AWIS) and establishing the RAPID Beta Program in support of the RAPID Pilot Operation. His current interests include research on reuse implementation methodologies and reuse cost/savings analysis.

Endnotes

1. Bass, Brad, "Complexity Keeps Ada From Reaching Its Potential," Government Computer News, November 13, 1989, page 67.

2. Silver, Judith, "Ada 9X Manager Works Toward Revised Standards," Government Computer News, November 13, 1989, page 65.

3. Bass, Brad, "Complexity Keeps Ada From Reaching Its Potential," Government Computer News, November 13, 1989, page 69.

4. Vitaletti, Bill and Ernesto Guerrieri, "Domain Analysis Within The ISEC RAPID Center," Eighth Annual National Conference on Ada Technology Proceedings, March 5-8, 1990.

5. Prieto-Diaz, Ruben and Peter Freeman, "Classifying Software for Reusability," IEEE Software, January 1987, pages 6-16.

Domain Analysis within the ISEC RAPID Center

William Vitaletti
SofTech, Inc.

Ernesto Guerrieri, Ph.D.
SofTech, Inc. and Boston University

Abstract

One of the main activities of the RAPID Center in supporting a development effort is the identification of reuse possibilities within the system being developed as well as across similar systems. This activity is accomplished by performing a domain analysis. This paper describes the domain analysis process proposed for the RAPID Center, the expected domain model, and the experiences gained from applying this process to the currently supported development effort.

1 Introduction

The "Reusable Ada Products for Information system Development" (RAPID) Center is a software reuse support organization for the U.S. Army Information Systems Software Command (ISSC) to promote the effective software reuse of Army software and to reduce the cost of system development and maintenance.

Initially, emphasis was placed on establishing the RAPID Center as an experimental laboratory for evaluating and refining reuse methods and techniques, accumulating reuse experience, and refining RAPID guidelines and procedures. This included the development of the RAPID Center Library (RCL) system [GUERRIERI88], a sophisticated tool for cataloging, searching, retrieving, and managing Reusable Software Components (RSCs), the development of prototype RSCs, and the authoring of initial policy and procedure documents. Currently, the RAPID Center is in a pilot phase in which it extensively supports a single development effort in order to prove reusability concepts, refine the library software, and resolve legal and management issues.

One of the main activities in supporting a development effort is the identification of reuse possibilities within the system being developed as well as across similar systems. This activity is accomplished by performing a domain analysis. This paper describes the domain analysis process proposed for the RAPID Center during the pilot operation and the expected domain model in Section 3. The experiences gained from applying this process to the currently supported development effort is described in Section 4. The paper concludes, in Section 5, with a comparison of the RAPID Domain Analysis process with two other domain analysis methodologies.

2 Definition of Domain Analysis

Domain Analysis (DA) is a process of intense examination of the domain of interest (i.e., management information systems, flight control systems, satellite systems, etc.). The process consists of identifying, collecting, organizing, analyzing, and representing the domain based on the existing information of the domain. We are interested in using domain analysis to identify the requirements for Reusable Software Components (RSCs) that satisfy the common needs in the development of software systems. The process consists of identifying the domain to be considered; developing a model of the domain (e.g., a software architecture) and a taxonomy for the domain; and identifying common objects, structures, and functions which are candidates for RSCs. It involves the examination and study of existing systems, underlying theory, emerging technology, and development histories within the domain of interest.

While variations of domain analysis may differ in focus, each domain analysis approach attempts to understand an application area (or classes of applications) by identifying unique, domain-specific attributes. The process of identifying common application characteristics leads to increased levels of software reuse across the domain.

3 The RAPID Domain Analysis Process

The RAPID Domain Analysis process consists of the following iterative phases (see also Figure 1):

1. The Domain Analysis Preparation phase,
2. The Domain Analysis phase, and
3. The Domain Analysis Product Generation phase.

Each iteration produces a refinement of the domain analysis products produced during the third phase. This allows us to begin our domain analysis with available information and continue to cycle through the process as we acquire additional domain information.

The process is terminated when the domain analysis products have been refined to the desired level or when additional domain information generates insignificant contribution to the analysis. This is known as the halting condition.

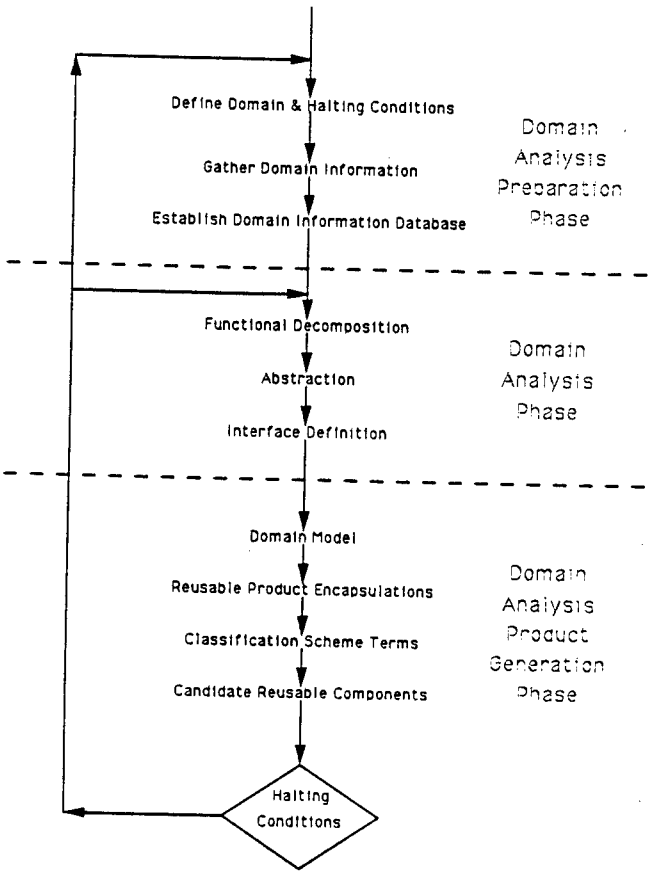


Figure 1: RAPID Domain Analysis Process.

The process can be re-started if there is sufficient new domain information to contribute to a refinement of the analysis, thus, performing a maintenance of the domain analysis products.

3.1 Domain Analysis Preparation Phase

The initial phase of the RAPID Domain Analysis process is a preparatory phase. In general, preparing for domain analysis is much the same as preparing for software development. In fact, the two are tightly coupled.

The Domain Analysis Preparation phase consists of the following activities:

1. Define the domain,
2. Gather domain-specific information, and
3. Establish a domain information database.

3.1.1 Define the Domain

We need to define the domain to be analyzed. This includes defining the boundaries of the domain and, as a consequence, the interfaces to other domains (or subdomains). This will aid us to determine the scope of the domain analysis.

In addition, we need to define the halting conditions for the process. This is achieved by defining the purpose and goals of the domain analysis for the selected domain. Since domain analysis is an expensive effort due to the knowledge-intensive and involved analytical activity, this step is important for evaluating the success of the process. The halting condition will depend on issues such as the level of completeness of the analysis, the planned use of the domain analysis products, etc.

The scope of the application domain will be determined, at a high level, by a sponsor's directive or by a corporation's business area, or, at a low level, through the realization of common needs or parts available in an application area. In either case, we need to justify the cost-effectiveness of a domain analysis by the potential existence of valuable domain analysis products. As a consequence, the domain should be:

- a mature application area where many existing systems are available for examination,
- a stable, well understood technology (i.e., the technical innovation should not render the domain analysis products prematurely obsolete), and
- an area that is well constrained in scope.

3.1.2 Gather the Domain-Specific Information

Once we have identified the domain of interest, the gathering of domain-specific information mainly identifies the sources of information needed for the later phases of the process. These information sources could be experts, documents, analogous systems, etc.

An interview process is used in gathering the domain-specific information from experts. The interview process is a systematic exchange of information with the domain experts, who are questioned regarding aspects of the domain (such as common functionality, etc.). The results of this process identify areas of reuse and may provide conclusive evidence of the effectiveness of the domain analysis.

In an environment using "DOD Automated Information System Documentation Standard DOD-STD-7935A" (such as ISSC), the documents that would be analyzed would be the System/Subsystem Specification (SSS), the Logical Data Model (LDM), and the Functional Description (FD) documents.

Within the FD, the graphical data flow diagrams (DFDs) serve as the primary source of our domain information. In general, DFDs depict the flow of data through the application domain. DFDs identify:

- external entities representing a source of the data,
- the processes that transform the data, and
- the destination of the data.

Level_0 DFDs (or System Data Flow Diagrams, SDFDs) represent the highest level of data flow within the domain. Each SDFD process block provides a processing abstraction

for a Logical Application Group (LAG). Each LAG is further decomposed into Level_1 DFDs called *work categories.*

Level_1 DFDs describe the functional system of the application domain in more detail than the Level_0 DFDs. Design Units (DUs) are derived from Level_1 DFDs as a result of the design process. Each process block from the design unit DFD may be exploded into a more detailed Level_2 DFD which is more physical in nature. Interface Control Data Flow Diagrams (ICDFDs) are a specific type of DFD used to illustrate the relationship between interfacing systems.

3.1.3 Establish the Domain Information Database

It is important in the domain analysis iterative process that we be able to store as much of the information about our domain (i.e, sources, interviews, analysis, and products) as possible in a database. This will aid in automating part of the process as well as making the information available across the various steps, phases, and iterations of the process. Furthermore, since this process will be refined over time, it will allow us to regenerate the analysis automatically with the revised process.

3.2 Domain Analysis Phase

The Domain Analysis Phase consists of three basic activities:

1. Functional decomposition,
2. Abstraction, and
3. Interface definition.

The functional decomposition activity consists in identifying specific objects and their characteristics. These are then grouped into classes/subclasses of objects. The abstraction activity consists in forming object abstractions from the object classes. The interface definition activity consists in identifying and classifying the relationships or interfaces between abstractions.

Specific relationships between objects and functions are extracted. The relationships are then abstracted in order to determine how descriptions can be generalized within the framework of common features. In the process, the specific relationships are mapped onto the common features. The specific relationships and the abstracted relationships are then used in conjunction with the object abstractions to derive a domain model and taxonomy [PRIETO-DIAZ87A].

We implement these activities in an iterative manner where, at each iteration, we will analyze the domain to a higher level of abstraction, thus, refining the domain model and taxonomy.

3.2.1 Functional Decomposition

The functional decomposition activity consists in identifying specific objects and their characteristics (such as operations involving the objects). We then group related objects and operations.

First, we focus on identifying rudimentary objects. In our approach, objects are selected based on the knowledge and understanding of the system being developed. We accomplish this by analyzing each relevant document, interviewing the domain experts, and extracting a list of inputs, outputs, and functions performed.

After we identify an object, we attempt to describe it at a high level. This is important because it reduces repetition and ambiguities during later stages of the process.

Next, we identify and group basic operations performed on objects. This is accomplished in much the same manner used to identify objects. These functions process inputs and/or generate outputs. The resulting list of operations can be extensive, so we "massage" the operations into *functional families.* Operations in a functional family are considered analogous.

To complete this phase of our domain analysis, we catalog each group of objects with the operations performed on that object group, respectively. We condense our initial catalog by employing the title of the functional family to catalog a related operation. The catalog is then used to group objects according to similar operations that are performed on objects within the group.

3.2.2 Abstraction

The second essential activity in our domain analysis phase consists of developing abstractions from groups (or classes) of objects and operations. In forming abstractions, it is important that the characteristics or attributes selected to describe the abstraction are relevant to the intended use of the expected abstraction.

The initial attempts to form abstractions focus on selecting objects with similar functional attributes. These common attributes allow us to form higher level abstractions [PERRY89].

Abstractions may be formed at various levels. In fact, it may appear in some cases that an abstraction can be applied equally at different levels. To eliminate unwanted confusion during the abstraction phase, it is extremely important that we do not attempt to assign both high-level and low-level operations to the same object.

3.2.3 Interface Definition

The third activity in our domain analysis phase centers on the interfaces between abstractions. The conceptual and logical interfaces are examined for commonalities based on data flow and timed events causing data exchange.

3.3 Domain Analysis Product Generation Phase

Our domain analysis is an information-based, cyclical process that matures as more domain information is acquired and manipulated. Each domain analysis activity results in a set of organized, information products; valuable output for advancing through our analysis cycle.

In general, the products of domain analysis are used to define the context in which reusable components can be de-

signed, developed and/or procured. Because reuse is the focus of our analysis, the resulting products ensure the reusability of candidate components.

The Domain Analysis Product Generation phase generates the following products:

- a version of the domain model,
- reusable product encapsulations,
- recommended classification scheme terms, and
- recommended candidate reusable components.

In addition, we hope to produce other products from our domain analysis, such as reusable templates, system requirements, and reusable software designs [PETERSON89]. Each domain analysis product is briefly described in the following paragraphs.

3.3.1 Domain Model

In general, a domain model provides a method to illustrate architecture common to systems within a specific domain. It provides a common structure of activities performed within systems with a minimum set of components, requiring a minimum number of component interfaces. Consequently, domain models reduce domain complexity because all incidents of the same activity pattern are resolved using the same model structure. Applying a good model to a range of domain applications should not alter the working structure of the application.

Since the model presents a more understandable depiction of the system architecture (i.e., limited components and inter-dependencies), it allows us to focus on the generic high-level and subsidiary services, and hide the details of implementation.

After additional domain information has been acquired and the model refined, the focus of our efforts will shift to applying the model in the domain application. By applying this model, generic components will be developed or procured and reused in other systems/subsystems in our domain.

3.3.2 Reusable Product Encapsulation

To achieve the greatest reuse payback, domain analysis must identify recurring abstractions within the domain. For each instance of recurring abstraction, a model solution must be developed and verified, after which code templates can be developed to implement the more general abstraction on the solution model.

During domain analysis, we will attempt to map system requirements to appropriate abstractions for which code already exists. If need be, the code can be generalized to a higher level of abstraction and instantiated for a particular application of the abstraction. If the template is carefully designed and the parameters to the template clearly documented, then instantiating the template will require significantly less effort than modifying the program code [FUTATSUGI88].

3.3.3 Recommended Classification Scheme Terms

A classification scheme is the vocabulary used to identify and describe software components. Except for some common algorithms and basic data structures, there is no standard vocabulary used in the software industry. This is a dilemma. However, domain analysis offers a solution to this predicament [GUERRIERI87].

During domain analysis, sets of fundamental, descriptive terms are collected as objects and operations are identified and abstractions are created. These terms represent different ways of looking at, or classifying, an object or abstraction.

More specifically, as commonalities are recognized between objects, the objects are grouped. The groups of objects are then classified based on attributes shared across objects in the group. The attributes of the group may, in turn, be used to describe any object within the group. As more groups of objects are formed, "groups" of groups are classified according to similarities between classes of objects. Common attributes shared between these groups, may then be used to characterize or describe the groups of objects at a higher level. In domain analysis, this "grouping" and classification scenario continues at multiple levels.

The terms and resulting classification structure developed during the domain analysis process, can be used to accurately identify most objects and/or abstractions within the domain. Subsequently, the classification scheme may also be used to specify reusable components which are represented by objects and abstractions within the domain.

As a final note, our classification scheme may consist of an unlimited number of terms; some with the same meaning. Terms with the similar meanings are grouped together and called synonyms. One term from each synonym group is selected as the representative term used in the actual classification. The remaining terms in the synonym group are keyed to the representative term in a list called a thesaurus. Used in a library system, the thesaurus enables component identification with a list of terms that is only limited by the size of the thesaurus being utilized [ARNOLD88].

3.3.4 Recommended Candidate Reusable Components

One of the primary products of our domain analysis is to provide recommended solutions (reusable components) for the abstractions identified during our analysis.

After identifying and prioritizing the abstractions and/or system requirements within our application domain, we examine other applications with analogous requirements. Our efforts focus on evaluating the solutions (and resulting implementations) to similar requirements in other domains. A good reusable component candidate may exist if its attributes are similar to those of the abstraction identified in our domain, and the solutions (or implementations) are acceptable [PRIETO-DIAZ88].

3.3.5 Miscellaneous Products

There are other important products produced during domain analysis, such as reusable system requirements. Each of these products will save valuable time and effort in system development. For example, system requirements, reused across systems, would also reuse corresponding software designs, implementation code and documentation. In this case, requirement reuse would save additional development costs for software design, code and unit testing. Reuse is most cost effective at this level.

4 A RAPID Domain Analysis Example

In the previous section, we have described the RAPID Domain Analysis process. We are currently applying the process to a development effort that is being supported by the RAPID Center. It is the initial application of the process. As we gain experience with the process, we will refine it appropriately. The application of the RAPID Domain Analysis process started in the fall of 1989, so the results collected to this date (i.e., December, 1989) are preliminary.

4.1 Domain Analysis Preparation

The domain chosen for the initial application of the RAPID Domain Analysis process during the RAPID pilot operation was the Standard Installation/Division Personnel System - version 3 (SIDPERS-3) application. It is a Standard Army Management Information System (STAMIS) application. SIDPERS-3 is an integral part of the Army's personnel support modernization plan. It will provide commanders and their staff with the necessary personnel information to make timely decisions and manage personnel resources effectively in the total spectrum of operations (i.e., warfighting, deployment, mobilization, preparation (in times of peace), and demobilization).

The SIDPERS-3 application was chosen for several reasons:

- The RAPID Center directly supports the U.S. Army Information System Software Center (ISSC) Software Development Center, Washington (SDC-W) which manages STAMIS applications (including the SIDPERS-3 application);
- SIDPERS-3, a system redesign and expansion, is a large development effort, comprising many subdomains that cover the STAMIS domain; and
- The development of SIDPERS-3 is in the requirements analysis phase which is an ideal time for a domain analysis.

The SIDPERS-3 project uses DOD-STD-7935A for functional requirements (and DOD-STD-2167A for software requirements). Furthermore, SIDPERS-3 is using a project-specific adaptation of McAuto's Structured Analysis, Design, and Implementation of Information Systems (STRADIS). STRADIS is a total system development methodology utilizing structured techniques and procedures that include: structured analysis, design, programming, top-down development, and structured walkthroughs.

During the domain analysis preparation phase, our efforts to accumulate the required domain-specific information include both manual and automated processes. Manual procedures include reviewing system and functional specifications, interviewing Army STAMIS and SIDPERS-3 experts, and participating in meetings with the development contractor. Knowledgeable SIDPERS-3 experts are readily available on site at SDC-W.

Automated tools are being prototyped for storage and manipulation of domain-specific data. Semi-automated prototyped tools decompose and store functional specifications and other details (illustrated in Data Flow Diagrams, DFDs), in an information database. Fully automated prototyped tools analyze and search the information database for commonalities based on interfaces, objects, and operations across SIDPERS-3 functional subsystems. As the process is refined, these tools will be formally developed.

4.2 Domain Analysis

The initial activity in the RAPID domain analysis concentrated on decomposing available domain-specific documentation. In the RAPID Domain Analysis method, we decompose each level DFD in the domain application into entities, processes, and data (by manually composing lists at each DFD level). These are stored in a domain information database. Currently, our domain consists of approximately 18 Logical Application Groups (LAGs) and 61 work categories. During decomposition, we are able to identify fundamental objects and operations. Table 1 shows some of the objects that have been identified so far.

Table 1: Objects.

Action	File	Report	Security
Access	List	Request	Status
Assignment	Message	Roster	Suspense
Command	Notice	Requirement	Table
Check	Orders	Schedule	
Data	Record	Screen	

We described the various objects so that, during later stages, we can identify analogous objects. Table 2 provides the descriptions of a few of the objects identified in Table 1.

The basic operations performed on the objects are identified and grouped into functional families. Table 3 illustrates the current set of functional families and the respective operations in each functional family. To complete the functional decomposition step, we associate the operations to the respective object group (See Table 4).

Table 2: Sample Object Descriptions.

Object	Description
Command	Information input by the user via the keyboard to control processing.
File	Structure used to house static tables of information.
Message	Formatted or free-text block of information that can be transmitted to or from an external system, system users, or between tasks.
Request	Indication by the user that a certain display, message, or service is desired.
Screen	Medium used to present formatted information to the user, such as user prompts.

During this phase, we also prototyped an automated tool, the Common Interface Tool (CIT), that analyzes the domain information database and identifies common interfaces. This tool identifies system functions with similar interfaces to the same external entities (in this case, databases and a data table). We supply the tool with the external entities and it returns the functions that interface to those external entities. Results generated by this tool are generally used as a starting point for a more rigorous, manual analysis to confirm interface and possible functional commonalities. Output generated by CIT is illustrated in Figure 2.

The sample output generated in Figure 2 illustrates that there are five operations (functions) that interface in the same manner with the three different data stores (external entities). Specifically, the output indicates that the operations all use data input from the same three data stores. Consequently, these operations become the starting point for a more in-depth, manual analysis for functional commonality.

Next in our analysis, we focus on forming abstractions. We identified that the following operations were performed

Table 3: Functional Families.

Functional Family	Operations
UPDATE	maintain, complete, flag, amend, finalize, add, change, clear, monitor, purge
FORWARD	deploy, pass, send, transmit, furnish, distribute, dispatch
VERIFY	review, endorse, validate, qualify, authorize
REQUEST	select, query
PROCESS	compare, calculate, schedule
ARCHIVE	file, reproduce, restore
RECEIVE	get
CREATE	generate, provide
REPORT	cut, post
DETERMINE	prepare, identify

```
External Entities...
D41          INDIV PERS DATA
D57          ORG DATA
T2           T-MOS ENLD

Functions...
02.01.02.02 GET CLASS/RECLASS DATA        (input)
02.01.03.02 GET CLASS/RECLASS RQST DATA   (input)
06.02.01.02 GET PER/UNIT/CRIT DATA        (input)
06.02.07.02 GET PROM SEL DATA             (input)
12.02.02.02 GET SURVEY DATA               (input)
```

Figure 2: Sample CIT Output.

on the REPORT object:

- accept and verify report request;
- gather report data;
- validate report data;
- update database;
- format report data; and
- output report.

Further analysis (by manually reviewing DFD requirements) resulted in a class of objects with operations similar to those of the REPORT object. Specifically, the NOTICE and ANNEX objects were identified and found to have the operations illustrated in Table 5. These common attributes allowed us to form a higher level abstraction, FORMATTED_FILE, of which REPORT, NOTICE, and ANNEX are instances.

During this phase, we prototyped another automated tool, Functional Identification Tool (FIT), to help us identify functional commonalities within the application domain. This tool was developed to search the information database

Table 4: Specific Operations Performed on Objects.

Object	Operations Performed on the Objects
DATA	generate, forward, update, process, request, verify
FILE	generate, update, forward, archive
MESSAGE	generate, process, verify, forward, receive, archive
NOTICE	determine, generate, archive, update, receive, forward
ORDERS	determine, generate, verify, archive, update, forward, request
RECORD	generate, update, forward, report, archive, validate, process
REPORT	determine, generate, forward, update
REQUEST	determine, generate, verify, forward, process, receive, update

Table 5: NOTICE and ANNEX Operations.

NOTICE	ANNEX
Format_notice_information	Format_annex_information
	Generate_annex_information
Get_notice_information	
Output_notice	Output_annex
Update_notice_database	Update_annex_information
Validate_notice_information	
Verify_user_rqst	Verify_user_rqst

and identify common functions comprised of similar operations or sub-functions. Figure 3 illustrates the FIT output.

Based on the information provided in the Functional Description (FD) and the System/Subsystem Specification (SSS) documents, the sample output in Figure 3 identifies two disjoint application functions that seem to perform the same operations, only with different data. It appears that similar operations are performed to accomplish both the *soldier readiness check* function (denoted by task number 05.03.02) and the *develop a promotion list* function (denoted by task number 06.08.03). Further analysis, performed manually by reviewing the FD and the preliminary Program Design Language (PDL), concludes that the operations performed on the PROMOTION_LIST abstract object and the SOLDIER_READINESS_LIST abstract object are, indeed, similar.

Consequently, these results indicate that a higher-level abstraction (possibly MAINTAIN_LIST) may be formed, of which PROMOTION_LIST and SOLDIER_READINESS_LIST would be instances. In this case, the MAINTAIN_LIST abstraction would have the following operations:

- Receive/process a MAINTAIN_LIST request from a system user;
- Gather initial MAINTAIN_LIST data from local data stores;
- Query additional data stores to verify MAINTAIN_LIST data;
- Update MAINTAIN_LIST data records; and
- Output resulting MAINTAIN_LIST data to the system user.

Possibly, the highest level abstraction we can attempt to form, based on our information database, is the SYSTEM INFORMATION MANAGEMENT abstraction. This particular high-level abstraction is actually composed of lower-level abstractions (See Table 6).

This high level abstraction is developed based on the functional requirements of our domain. In fact, we applied the information management model, developed from our initial domain analysis (refer to Figure 4), to create the abstraction.

For the interface definition activity, we have not yet completed the interface definition. This reflects the current point

```
----> 05.03.02*     CONDUCT READINESS CHECK

RECV READINESS CHECK ROST              05.03.02.01
GET INDIV READINESS DATA               05.03.02.02
VERIFY INDIV READINESS DATA            05.03.02.03
UPDATE READINESS DATA                  05.03.02.04
PRINT READINESS DATA                   05.03.02.05

----> 06.08.03*     CMP CREATE DISTR CONS/SEL DATA

RECV/ACCEPT RQST                       06.08.03.01
GET ELIG DATA                          06.08.03.02
VERIFY CONS/SEL INDIV                  06.08.03.03
UPDATE FILES                           06.08.03.04
PRINT CONS/SEL LISTS                   06.08.03.05
```

Figure 3: Sample FIT Output.

that we have reached in our first iteration of the RAPID Domain Analysis process.

4.3 Domain Analysis Product Generation

Although we have not completed our domain analysis, a preliminary version of the domain model is beginning to take shape.

The preliminary model we developed as a general solution for our problem domain is illustrated in Figure 4 and 5. Our primary goal in developing this model is to express it in an understandable, adaptable format. Adaptability is accomplished by parameterizing components. Hence, they could be scaled to fit a particular application of the model.

We looked at our application domain from two viewpoints (or domain views [LEE89]) and documented each view in a formal diagram. The first view examines the data flow between components within our application domain. The data flow diagram (DFD) in Figure 4 illustrates the data flow in our model.

Table 6: System Information Management Low-level Abstractions.

Abstraction	Description
Display Manager	Accept commands and maintain system user display.
Message Handler	Receive, validate, and process messages.
Report Generator	Generate formatted and unformatted messages.
Specific Process	Perform application-specific processing.
Data Manager	Parse, decode, compress, manipulate, and store data.

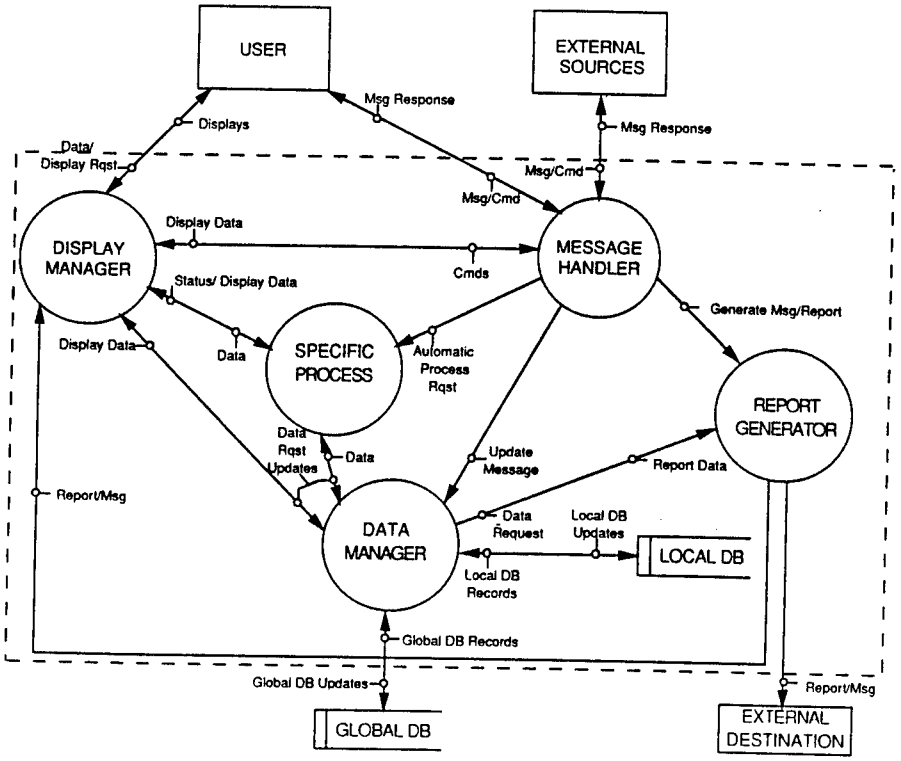


Figure 4: Preliminary Domain Model (DFD).

This model is representative of the commonality inherent in the functional descriptions of most "work categories" within our domain application. At a high level, data can be abstracted to allow it to be handled in similar ways. At lower levels, data characteristics are more explicit and conform to the requirements of the databases of each work category.

As Figure 4 illustrates, the domain model is composed of five abstract components. Connecting arrows denote data flows between the components and data sources and destinations (local and external). The general attributes of each component are:

Display Manager Interprets requests for displays, maintains display formats, gathers display data from database or task, and maps display data and outputs display to the user.

Message Handler Receives and validates messages. Parses and decodes data/messages, performs range of value and duplication checks, prioritizes messages, and forwards data for storage, special processing, or to be queued. Also, maintains a service message/data queue, displays formatted/unformatted information, and forwards data to other tasks.

Report Generator Interprets message requests, gathers related data from database/task, formats message, outputs formatted message to user, and routes messages to external destinations or other subprocesses.

Specific Process Performs application dependent functions. May be initiated by automatic operator subtasks to perform specialized, detailed algorithms.

Data Manager Accepts, parses, updates, compresses, maps and stores data to a specific database format.

The second view focuses on the relationships between entities in the model [MOORE89]. Figure 5 illustrates the component relationships in an entity relationship diagram (ERD). Here, we attempt to express the communication between components concentrating on the entity or component initiating an action or data exchange. To some degree, this allows us to view the control flow within the application domain.

Referenced in conjunction with the data flow diagram, the ERD adds another dimension to our model. The domain model is represented in a form comprehensible to the domain experts (i.e., DFDs and ERDs). Infact, the domain model should be flexible so as to adapt it to the needs of different domains [GUERRIERI89].

Although we have just started work with reusable product encapsulation, we anticipate that our plans to develop abstract components (i.e, developing reusable components by object abstraction, modularization and standardization, based on our domain model) will result in the greatest possible reuse benefit.

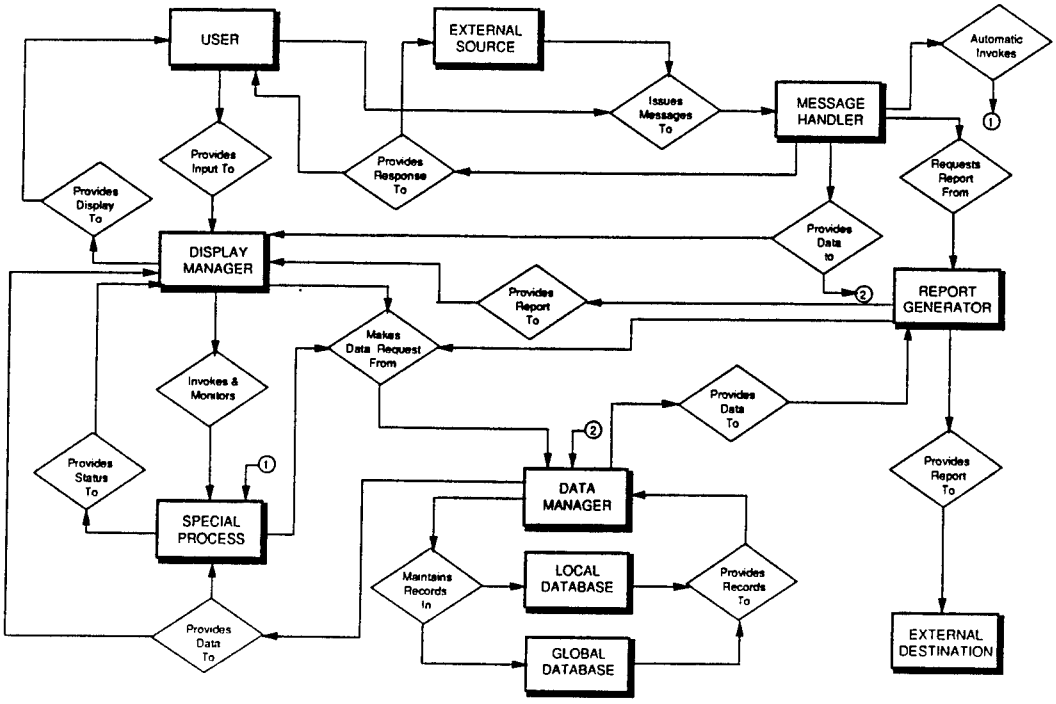


Figure 5: Preliminary Domain Model (ERD).

Because our initial development funds are limited, our efforts on reusable components will concentrate on identifying software components that already exist as solutions or implementations for the abstractions.

5 High-Level Comparison to Other Domain Analysis Approaches

In this section, we will compare the RAPID Domain Analysis process to two other domain analysis models. This comparison is only at a high level in order to show how the RAPID Domain Analysis process is similar to or differs from these other models.

Prieto-Diaz proposed a model domain analysis process [PRIETODIAZ87b, COHEN89] which consists of the following steps:

1. Prepare Domain Information
   (a) Define Domain Analysis Approach
   (b) Bound Domain
   (c) Define Domain
   (d) Select Knowledge Sources
   (e) Define Domain Requirements
2. Analyze Domain
   (a) Identify Common Features
   (b) Select Specific Functions/Objects
   (c) Abstract Functions/Objects
   (d) Define Specific Relationships
   (e) Abstract Relationships
   (f) Do Classification
   (g) Define Domain Language
3. Produce Reusable Workproducts
   (a) Select Reusable Candidates
   (b) Encapsulate Reusable Workproducts
   (c) Define Reusability Guidelines
   (d) Create Domain Standards

The RAPID Domain Analysis process is very similar to Prieto-Diaz's model. Currently, the RAPID Domain Analysis process does not include steps 1.a, 3.c, and 3.d. These are activities performed by the RAPID Center and are considered as a higher level activity to promote software reuse [GUERRIERI88].

The Pittsburgh Workshop Model of Domain Analysis (PWMDA) proposes that a domain consists of three parts:

1. a problem space,
2. a solution space, and

3. a mapping between the two.

The problem space is a network of target products features, underlying principles, analogies, and relationships. The solution space contains design issues or criteria, specific decision alternatives associated with each issue, architectural components, constraints, and implementation components. The mapping is the relationship between features and principles in the problem domain with the issues, decisions, and architectural components in the solution domain [PWMDA89]. The RAPID Domain Analysis process can be adapted to this model. The Domain Information Database would represent the information of the problem space, the Domain Analysis Products would represent the solution space information, and the process would represent the mapping between the two. Further work needs to be done to incorporate the PWMDA model into the RAPID Domain Analysis process.

6 Conclusions

We have described the domain analysis process proposed for the RAPID Center during pilot operation. This included the expected domain model, the experiences gained from applying this process to the currently supported development effort, and a brief comparison to two other domain analysis models.

There is an issue that has already been identified in the literature and that we feel will have an impact in the near future: namely a method to validate the model produced by the domain analysis. This is important for certifying the correctness of the model, understanding the halting conditions for the process, and understanding when and how to maintain the domain analysis products. Currently, we have not addressed this issue, but intend to prior to finalizing our domain analysis process.

Our application of the process to SIDPERS-3 is at an early stage. As we gain experience with our first application through several iterations of the process, we are refining the process. Tools to aid in the process will be proposed and later developed. Finally, we aim to have the final version of the process be adopted by the RAPID Center.

References

ARNOLD88 "The Reuse System: Cataloging and Retrieval of Reusable Software," Arnold, S.P., and Stepoway, S.L., in **Tutorial: Software Reuse - Emerging Technology**, Tracz, W. (Ed.), IEEE Computer Society Press, Washington, D.C., pp. 138-141, 1988

COHEN89 "GTE Software Reuse for Information Management Systems," Cohen, J., and Hignite, B., Position paper for the Reuse in Practice Workshop, Pittsburgh, PA, July 11-13, 1989

FUTATSUGI88 "Parameterized Programming in OBJ2," Futatsugi, K., Goguen, J., Meseguer, J., and Okada, K., in **Tutorial: Software Reuse - Emerging Technology**, Tracz, W. (Ed.), IEEE Computer Society Press, Washington, D.C., pp. 337-346, 1988

GUERRIERI87 "On Classification Schemes and Reusability Measurements for Reusable Software Components," Guerrieri, E., Proceedings of the Workshop on Software Reuse, Rocky Mountain Institute of Software Engineering, Boulder, CO, October 14-16, 1987

GUERRIERI88 "Searching for Reusable Software Components with the RAPID Center Library System," Guerrieri, E., Proceedings of the Sixth National Conference on Ada Technology, Alexandria, VA, pp. 395-406, March, 1988

GUERRIERI89 "The Role of SADT in Domain Analysis for Software Reuse," Guerrieri, E., Position paper for the Reuse in Practice Workshop, Pittsburgh, PA, July 11-13, 1989

LEE89 "Application of Domain-Specific Software Architectures to Aircraft Flight Simulators and Training Devices," Lee, K.J., and Rissman, M., Position paper for the Reuse in Practice Workshop, Pittsburgh, PA, July 11-13, 1989

MOORE89 "Domain Analysis: Framework for Reuse," Moore, J.M., and Bailin, S.C., (preliminary copy; to be published).

PERRY89 "The Role of Domain Independence in Promoting Software Reuse," Perry, J.M., and Shaw, M., Position paper for the Reuse in Practice Workshop, Pittsburgh, PA, July 11-13, 1989

PETERSON89 "Coming to Terms with Terminology for Software Reuse," Peterson, A.S., Position paper for the Reuse in Practice Workshop, Pittsburgh, PA, July 11-13, 1989

PRIETO-DIAZ87a "Classifying Software for Reusability," Prieto-Diaz, R., and P. Freeman, **IEEE Software**, Vol.4, No.1, pp.6-16, January 1987

PRIETO-DIAZ87b "Domain Analysis for Reusability," Prieto-Diaz, R., Proceedings of COMPSAC 87, Tokyo, Japan, pp. 23-29, October 1987

PRIETO-DIAZ88 "Breathing New Life into Old Software," Prieto-Diaz, R., and Jones, G.A., in **Tutorial: Software Reuse - Emerging Technology**, Tracz, W. (Ed.), IEEE Computer Society Press, Washington, D.C., pp. 152-160, 1988

PWMDA89 "Results from Domain Analysis Working Group," Working Group Members, Reuse in Practice Workshop Proceedings, Pittsburgh, PA, July 11-13, 1989

Biography

Ernesto Guerrieri, Ph.D.
SofTech, Inc.,
460 Totten Pond Road,
Waltham, MA 02154-1960
Arpanet: ernesto@ajpo.sei.cmu.edu

Ernesto Guerrieri is a System Consultant at SofTech, Inc. He is one of the principal technical leaders of the RAPID project and led the development effort of the RAPID Center Library system. He is also a Distinguished Reviewer for the Ada 9X project and the technical leader for the testing environment in the BIDDS Management System project. He has managed several contracts among which are the support effort for the Ada Language Maintanence and the maintenance of the ACVC Implementers' Guide. Ernesto Guerrieri is an Adjunct Associate Professor in the Electrical, Computer and Systems Engineering department at Boston University where he teaches "Advanced Data Structures." He holds a Ph.D. in Computer and Systems Engineering from Rensselaer Politechnic Institute, Troy, New York and the equivalent of a Masters of Science and Bachelor of Science degrees from the University of Pisa, Italy. His interests include software portability, software reusability, software engineering, and programming languages. He is a member of ACM, IEEE, AAAI, and Sigma Xi.

William Vitaletti
SofTech, Inc.,
2000 N. Beauregard Street,
Alexandria, VA 22311-1794

William Vitaletti is a System Consultant at SofTech, Inc. He is currently a systems analyst for the RAPID project, responsible for performing domain analysis. He holds a Masters of Science degree in Computer Science from the State University of New York at Binghamton and a Bachelor of Science degree in Business Economics from the State University of New York at Binghamton. His current interests focus on domain analysis modeling and automated tools development for use in the domain analysis arena.

REPOSITORY SUPPORT FOR A REUSE PROCESS

Beverly J. Kitaoka

Science Applications International Corporation
Sarasota, Florida

Abstract

To realize the full potential of reuse, we must look beyond the reuse of building block components to an expansive set of software lifecycle workproducts which includes software architectures, specifications, designs, data, code, tests, and other useful information. Reuse of this information will only occur if reuse is incorporated into all activities of the software lifecycle process with ready access to the workproducts. A software reuse repository capable of supporting a reuse process is essential.

Introduction

In determining the required capabilities of a reuse process-supportive repository, we must investigate the application of reuse in each activity of the software lifecycle process. Since reuse can be incorporated into most software lifecycle processes, we have not selected any particular process, but instead address activities common to most of these lifecycles:

- analysis
- prototyping
- design
- implementation
- configuration management
- quality assurance
- maintenance

We will initially discuss the establishment of a repository for reuse, focussing on the role of domain analysis in tailoring repository capabilities to support the users as they perform lifecycle activities to develop systems in the domain. This tailoring will allow the repository to become more effective for these users. Following this discussion, we will examine the use of this repository in s ıpport of the software reuse lifecycle process.

Repository Establishment

Understanding repository establishment issues is crucial to minimizing the repository investment and maximizing its usefulness to the reuse process. Content management, user capabilities, and repository administration issues must be addressed in the context of reuse process support. This section will discuss a domain-oriented approach to addressing these establishment issues.

Domain Analysis. Domain analysis is the process of examining similar systems for commonality in structure, objects, functions, characteristics and relationships to produce a generalization of all the systems in a given domain. The analysis of these characteristics is used to create standard designs and standard interfaces for systems and subsystems. Systems analysts can then use these standard designs to specify new systems in the domain.

Prieto-Diaz defines a domain analysis process with the following basic activities: 1) knowledge acquisition, 2) domain definition (defining, scoping, and setting domain boundaries), 3) model formation and definition to support domain understanding and information organization, 4) model evolution which refines the model as new information is acquired, and 5) model verification and validation. Domain models can include classification taxonomies, domain languages, standards (e.g., standard interfaces), and functional models (e.g. generic architectures) [1].

Content Management. Content management activities include the collection, classification, evaluation, tracking, and cataloging of repository contents. Domain analysis products which define standards for

design, coding, testing, and interfaces [1] will be used to tailor the repository supply/collection and evaluation procedures for the domain. Products which define the types of components in a domain and how they are organized, such as a taxonomy, will be used to tailor the classification and cataloging/procedures for the domain. Component tracking, discussed in the configuration management section of this paper, does not appear to be affected by the domain analysis process.

User Capabilities. The repository user capabilities are heavily influenced by domain analysis. The repository user profiles (who uses the repository and the activities they perform) will be derived from the domain model. We can then tailor the library interface to reduce the effort of these users by providing repository capabilities which support the activities defined for their respective roles in domain terminology. To provide a user-friendly and efficient interactive interface, for example, the library retrieval mechanism should be based on the classification taxonomy produced during domain analysis, and present the access (supply, retrieval, extraction) capabilities in the relevant terminology supplied by the domain analysis process. Use of the faceted classification approach developed by Dr. Prieto-Diaz, for example, allows the domain user to specify the kind of package desired by providing values for several facets, such as function, object, functional area, etc., as follows [2]:

function: calculate
object: trajectory

functional
area: anti-tank missile

The strength of the faceted scheme is the controlled vocabulary derived during the domain analysis process for each type of domain user (developer, maintainer, librarian, etc.). The controlled vocabulary limits the number of facets needed to describe a component for matching in the repository. Some of the facets are relevant to all domain users, but those that are not will not be presented, reducing the choices required of the user. The Venn diagram in Figure 1 shows the reduction in a potentially extensive collection of facets to the relevant facets for a particular type of user in a domain [2].

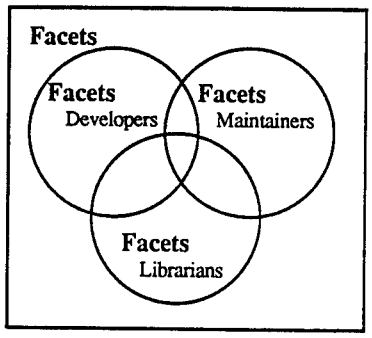


Figure 1. Sample Domain Facet Sets

Repository Administration. Activities of repository administration include:

- Component Database Maintenance
- Catalog Updates/Distribution
- New Components Classification
- Component Order Processing
- User Registration
- Usage Monitoring
- Accounting

The administration of a repository is indirectly affected by the domain analysis activity, to the extent that the organization of the repository being administered is affected by domain analysis. For example, the organization of the catalog and the content of its entries is tailored to service the needs of the domain users.

Ongoing domain analysis activities and experience in using the repository will provide opportunities for the tuning and the enhancement of the repository to keep it viable and ensure its long-term usefulness.

Reuse Process Support

With an established repository, we can now develop software systems which will not only benefit from its existence, but will contribute to its enhancement. As each system is built, the repository becomes more bountiful, reducing the resources needed to build the next system. This section will discuss the role of the repository in support of the reuse process activities in development of these systems.

Analysis

The domain model derived from the domain analysis activities is stored in the repository along with a framework set of reusable domain components. The ideal repository will reflect the results of this domain analysis to the extent of constraining the systems analyst in such a way that use of the domain model is inherent.

The systems analysis activity will derive the initial architecture and requirements for a particular system from the domain model. Techniques such as prototyping will be applied during this acivity to select alternatives, understand performance requirements and specify parameters which configure the generalized components of the domain model. Instantiation of the domain model for a given system will then evolve as information is exchanged between the users and designers of the system.

Prototyping

Prototyping can be a useful activity for many software lifecycle activities to analyze alternatives. It is particularly useful in determining system requirements.

The composition of prototypes from reusable components is an expedient method of testing alternative specifications, designs, and implementations. Prototype software may differ from production software in generality and robustness. Guidelines for reusing prototype components can be stored in the repository to assist in their use.

When selecting characteristics for a system, it is useful to have parameterized components where alternative system configurations can be tested with just the modification of data. This data can be stored in the repository for reuse with the component. The generality of these components may restrict their usage in production, however, due to performance requirements. Regardless, they can be useful for determining performance requirements for the production system and helpful in specifying production components.

The repository needs to have provisions for distinguishing prototype components from production components. In the case of incremental builds where prototype components can evolve into more robust prototype components or production components, the repository needs to provide a traceability feature for versions which are retained for reuse. In cases where production components already exist for the prototype, this feature will reduce the efforts required to build the production software.

Design

The domain model workproduct of the domain analysis activity will provide direction in designing and implementing a system in the domain. This model can be used to produce a domain-directed repository interface which will direct the design activity so the developer makes maximum use of available components in the repository [3]. This capability will allow reuse-directed design analogous to that of the hardware world where major subfunctions of electronic systems are available on integrated circuits. Unlike today's software process where we design down to the detailed level and then try to match the design to existing components, the hardware designers know what chips are available and direct their designs towards these chips [4]. Clearly, this approach leads to greater reuse, with consequent savings in development cost.

Repository issues that affect support of reusable design workproducts include the application of graphics standards which will allow the exchange of graphic design representations among design tools. Information about the graphics design representation must be stored in the repository so the user can select the proper tools for analyzing the design.

Implementation

The standard designs and interfaces stored in the repository for a domain will allow implementation work to proceed in parallel and shorten the development schedule [5].

A reuse process will include the identification of opportunities for producing reusable components as well as using existing components on the project. The process should, therefore, include activities for supplying these new components to the repository. In the search for existing components, repository tools and data

collected about the component can assist the software engineer with the analysis of potential candidates for reuse. As components are supplied to the repository, tools such as metrics analysis tools can be run by the librarian as part of the certification process and the data can be stored with the component. These tools can be made available to the user who may need more information than that which is selected for storage with the component. An evaluation capability can also be provided by the repository to allow the sharing of users' experience with the component.

The repository should have a capability which monitors the use of facets in matching components to the user's needs. The absence of matches in a facet search can be analyzed to determine whether accurate facets have been selected to match the component or if there is a need for a component envisioned by the faceted classification which does not yet exist in the repository.

When identified components meet the user's requirements, the component can be extracted and used as is. In cases where the candidates are close to meeting the requirements, the user can make trade-off decisions as to whether the design should be adapted to reuse the potential candidate or the component should be adapted to meet the specification. If the component retains its reuse characteristics after adaptation, it should be supplied to the repository according to the criteria described in the configuration management section of this paper.

The data selected to configure a particular instance of a system, subsystem, or component in the domain can also be reused and should be stored as such. Other relevant reusable information includes test procedures, test data, and test results. The repository will provide access to these reusable workproducts with information about their relationships to other assets in the system.

Re-engineering. The reuse process provides other alternatives for implementing reusable components for the repository. The knowledge acquisition activity of domain analysis includes the analysis of information such as source code and documentation from existing systems. Potential reuse candidates may be discovered during this activity. Since it is unlikely that these existing workproducts were built for reuse, they will need to be re-engineered to adopt reuse characteristics. These re-engineered reusable workproducts can then be used to build new systems or re-engineer the existing ones. Repository tools which support re-engineering techniques will be useful in moving components from a "depository" state to a "certified" state for reuse [6].

Configuration Management

Configuration management capabilities will control the workproducts so they can be used with confidence. Versions of components which are written in different languages, affect changes to other parts of the system, or provide different functionality/performance from the original component should be stored in the repository as new components. The repository should provide traceability between the versioned component and the original component, allowing the versioned component to inherit applicable attributes from the original component for repository efficiency. The descriptions, version, dates, authors and other relevant information stored with a component along with this version tracking capability assists the configuration manager in controlling the changes made to software in a system. If the changes made to a component are significant enough to qualify as a new version, the configuration manager can determine whether the system should be modified to use the new version or whether it should remain with the existing version.

The repository needs to have a good configuration control system to track component revisions and insure the appropriate construction of compound components. The usage traceability feature of a reuse library tracks the extraction of a component for reuse. This capability then works in conjuction with the repository configuration management function to provide version/revision control information, as well as defect notification to users. The configuration manager can use this feature to maintain the accuracy and consistency of project baselines, as well to track bugs in the components.

Repository access controls can protect the ownership rights of reusable workproducts

and assist in the prevention of improper workproduct usage. The repository system ownership and data rights information stored with the components is also useful information for the configuration manager who needs to be aware of the implication of data rights as they apply to the software in a system.

Quality Assurance

As components are supplied to the repository, they are subjected to an evaluation process to determine their acceptability. This process includes the application of standards checking and metrics analysis tools, the inspection of information supplied with the component, and execution of tests with test data. The results of the evaluation process are stored in the repository with the component and used to determine a "certification" metric to assist the user in understanding the degree of trustworthiness assigned to the component. This information is useful input to the quality assurance activities of the lifecycle process where the extent to which the code component has been "certified" can reduce the efforts of the software quality assurance engineer.

Traceability features of components from requirements to implementation in a repository can also reduce the quality assurance efforts.

Maintenance

Systems built from repository components will not suffer from the maintenance problems that plague many of today's systems. The requirements, designs, and much of the development information will be captured during the reuse process and accessible to the maintenance engineers through the repository.

Traceability of workproducts from specification through design to implementation is an important feature for the maintenance user of a repository. It is useful in locating the implementation which is inconsistent with the specification or in isolating the software that needs to be modified as systems requirements change.

The common systems designs produced during systems analysis can be used in the process of re-engineering existing systems to improve their consistency and maintainability, and to reduce the learning time for the maintenance engineers. The re-engineering process can be simplified by the existence of domain models and reusable components.

Conclusions

The repository is essential to the support of a software reuse process and must become an integral part of the software support environment. The issues involved in supporting reuse in software systems development are applicable to many of the software lifecycle processes in use today. The success of a particular repository implementation is the extent to which the repository supports the reuse activities in a variety of lifecycles. The depiction of repository activities in each activity of the reuse process will reinforce software reuse activities as the software engineer uses the environment to create and support the software in a system.

Bibliography

[1] Ruben Prieto-Diaz. "Integrating Domain Analysis in Software Development Process," Contel Technical Note #CTC-TN-81-011, October 1989

[2] Ruben Prieto-Diaz. "Implementing Faceted Classification for Software Reuse". To be presented at the International Conference on Software Engineering, March 1990.

[3] Christine Braun and Ruben Prieto-Diaz. "Domain-Directed Reuse," presented at the 14th Annual Software Engineering Workshop, NASA/Goddard Flight Center, Greenbelt, MD., November 1989.

[4] Christine L. Braun and Mamie Liu. "Software Reuse in Networking Applications," proceedings of the Washington Ada Symposium-'89, June 1989.

[5] Robert Holibaugh. "Reuse: Where to Begin and Why," proceedings of the Reuse In Practice Workshop, Software

Engineering Institute, Pittsburgh, PA., July 1989.

[6] Beverly J. Kitaoka. "Establishing Ada Repositories for Reuse," TRI-Ada '89 Proceedings, October 1989.

[7] Gerald Jones and Ruben Prieto-Diaz. "Building and Managing Software Libraries," in IEEE Software, 1987, pp.228-235.

Biography

Mrs. Kitaoka is the Division Manager for the Ada Software Division of SAIC. The Ada Software Division has developed Ada software under a number of contracts which have provided a basis for its repository of reusable Ada software. As the SAIC STARS program manager, Mrs. Kitaoka is involved in the development of software technology to increase productivity through research and development in technology areas which include software process, reuse, repositories, standard interfaces, software environments, and the Ada language.

In her role as Deputy Chairman for SAIC's Software Engineering Working Group, she is involved in the generation of company standards for software development, reuse, software quality assurance, and software configuration management. She is also Chairman of SAIC's Reuse Working Group, directing corporate repository and SIG-Ada's Reuse Working Group Repository/Library subgroup chair.

Mrs. Kitaoka received the BS in mathematics and the MS in computers, information and control engineering from the University of Michigan.



SOFTWARE RECLAMATION: Improving Post-Development Reusability

John W. Bailey and Victor R. Basili

The University of Maryland Department of Computer Science
College Park, Maryland 20742

Abstract

This paper describes part of a multi-year study of software reuse being performed at the University of Maryland. The part of the study which is reported here explores techniques for the transformation of Ada programs which preserve function but which result in program components that are more independent, and presumably therefore, more reusable. Goals for the larger study include a precise specification of the transformation technique and its application in a large development organization. Expected results of the larger study, which are partially covered here, are the identification of reuse promoters and inhibitors both in the problem space and in the solution space, the development of a set of metrics which can be applied to both developing and completed software to reveal the degree of reusability which can be expected of that software, and the development of guidelines for both developers and reviewers of software which can help assure that the developed software will be as reusable as desired.

The advantages of transforming existing software into reusable components, rather than creating reusable components as an independent activity, include: 1) software development organizations often have an archive of previous projects which can yield reusable components, 2) developers of ongoing projects do not need to adjust to new and possibly unproven methods in an attempt to develop reusable components, so no risk or development overhead is introduced, 3) transformation work can be accomplished in parallel with line developments but be separately funded (this is particularly applicable when software is being developed for an outside customer who may not be willing to sustain the additional costs and risks of developing reusable code), 4) the resulting components are guaranteed to be relevant to the application area, and 5) the cost is low and controllable.

Introduction

Broadly defined, software reuse includes more than the repeated use of particular code modules. Other life cycle products such as specifications or test plans can be reused, software development processes such as verification techniques or cost modeling methods are reusable, and even intangible products such as ideas and experience contribute to the total picture of reuse [1,2]. Although process and tool reuse is common practice, life cycle product reuse is still in its infancy. Ultimately, reuse of early lifecycle products might provide the largest payoff. For the near term, however, gains can be realized and further work can be guided by understanding how software can be developed with a minimum of newly-generated source lines of code.

The work covered in this paper includes a feasibility study and some examples of generalizing, by transforming, software source code after it has been initially developed, in order to improve its reusability. The term software reclamation has been chosen for this activity since it does not amount to the development of but rather to the distillation of existing software. (Reclamation is defined in the dictionary as obtaining something from used products or restoring something to usefulness [3].) By exploring the ability to modify and generalize existing software, characterizations of that software can be expressed which relate to its reusability, which in turn is related to its maintainability and portability. This study includes applying these generalizations to several small example programs, to medium sized programs from different organizations, and to several fairly large programs from a single organization.

Earlier work has examined the principle of software reclamation through generic extraction with small examples. This has revealed the various levels of difficulty which are associated with generalizing various kinds of Ada dependencies. For example, it is easier to generalize a dependency that exists on encapsulated data than on visible data, and it is easier to generalize a dependency on a visible array type than on a visible record type. Following that work, some medium-sized examples of existing software were analyzed for potential generalization. The limited success of these efforts revealed additional guidelines for development as well as limitations of the technique. Summaries of this preceding work appear in the following sections.

Used as data for the current research is Ada software from the NASA Goddard Space Flight Center which was written over the past three years to perform spacecraft simulations. Three programs, each on the order of 100,000 (editor) lines, were studied. Software code reuse at NASA/GSFC has been practiced for many years, originally with Fortran developments, and more recently with Ada. Since transitioning to Ada, management has observed a steadily increasing amount of software reuse. One goal which is introduced here but which will be addressed in more detail in the larger study is the understanding of the nature of the reuse being practiced there and to examine the reasons for the improvement seen with Ada. Another goal of this as well as the larger study is to compare the guidelines derived from the examination of how different programs yield to or resist generalization. Several questions

are considered through this comparison, including the universality of guidelines derived from a single program and whether the effect of the application domain, or problem space, on software reusability can be distinguished from the effect of the implementation, or solution space.

Superficially, therefore, this paper describes a technique for generalizing existing Ada software through the use of the generic feature. However, the success and practicality of this technique is greatly affected by the style of the software being transformed. The examination of what characterizations of software are correlated with transformability has led to the derivation of software development and review guidelines. It appears that most, if not all, of the guidelines suggested by this examination are consistent with good programming practices as suggested by other studies.

The Basic Technique

By studying the dependencies among software elements at the code level, a determination can be made of the reusability of those elements in other contexts. For example, if a component of a program uses or depends upon another component, then it would not normally be reusable in another program where that other component was not also present. On the other hand, a component of a software program which does not depend on any other software can be used, in theory at least, in any arbitrary context. This study concentrates only on the theoretical reusability of a component of software, which is defined here as the amount of dependence that exists between that component and other software components. Thus, it is concerned only with the syntax of reusable software. It does not directly address issues of practical reusability, such as whether a reusable component is useful enough to encourage other developers to reuse it instead of redeveloping its function. The goal of the process is to identify and extract the essential functionality from a program so that this extracted essence is not dependent on external declarations, information, or other knowledge. Transformations are needed to derive such components from existing software systems since inter-component dependencies arise naturally from the customary design decomposition and implementation processes used for software development.

Ideal examples of reusable software code components can be defined as those which have no dependencies on other software. Short of complete independence, any dependencies which do exist provide a way of quantifying the reusability of the components. In other words, the reusability of a component can be thought of as inversely proportional to the amount of external dependence required by that component. However, some or all of that dependence may be removable through transformation by generalizing the component. A measure of a component's dependence on its externals which quantifies the difficulty of removing that dependence through transformation and generalization is slightly different from simply measuring the dependence directly, and is more specifically appropriate to this study. The amount of such transformation constitutes a useful indication of the effort to reuse a body of software.

Both the transformation effort and the degree of success with performing the transforms can vary from one example to the next. The identification of guidelines for developers and reviewers was made possible by observing what promoted or impeded the transformations. These guidelines can also help in the selection of reusable or transformable parts from existing software. Since dependencies among software components can typically be determined from the software design, many of the guidelines apply to the design phase of the life cycle, allowing earlier analysis of reusability and enabling possible corrective action to be taken before a design is implemented. Although the guidelines are written with respect to the development and reuse of systems written in the Ada language, since Ada is the medium for this study, most apply in general to software development in any language.

One measure of the extent of the transformation required is the number of lines of code that need to be added, altered, or deleted [4]. However, some modifications require new constructs to be added to the software while others merely require syntactic adjustments that could be performed automatically. For this reason, a more accurate measure weighs the changes by their difficulty. A component can contain dependencies on externals that are so intractable that removing them would mean also removing all of the useful functionality of the component. Such transformations are not cost-effective. In these cases, either the component in question must be reused in conjunction with one or more of the components on which it depends, or it cannot be generalized into an independently reusable one. Therefore, for any given component, there is a possibility that it contains some dependencies on externals which can be eliminated through transformation and also a possibility that it contains some dependencies which cannot be eliminated.

To guide the transformations, a model is used which distinguishes between software function and the declarations on which that function is performed. In an object-oriented program (for here, a program which uses data abstraction), data declarations and associated functionality are grouped into the same component. This component itself becomes the declaration of another object. This means the function / declaration distinction can be thought of as occurring on multiple levels. The internal data declarations of an object can be distinguished from the construction and access operations supplied to external users of the object, and the object as a whole can be distinguished from its external use which applies additional function (possibly establishing yet another, higher level object). The distinction between functions and objects is more obvious where a program is not object-oriented since declarations are not grouped with their associated functionality, but rather are established globally within the program.

At each level, declarations are seen as application-specific while the functions performed on them are seen as the potentially generalizable and reusable parts of a program. This may appear backwards initially, since data abstractions composed of both declarations and functions are often seen as reusable components. However, for consistency here, functions and declarations within a data abstraction are viewed as separable in the same way as functions which depend on declarations contained in external components are separable from those declarations. In use, the reusable, independent functional components are composed with application-specific declarations to form objects, which can further be composed with other independent functional components to implement an even larger portion of the overall program.

Figure 1 shows one way of representing this. All the ovals are objects. The dark ones are primitives which have predefined operations, such as integer or Boolean. The white ovals represent program-supplied functionality which is composed with their contained objects to form a higher level

object. The intent of the model is to distinguish this program-specific functionality and to attempt to represent it independently of the objects upon which it acts.

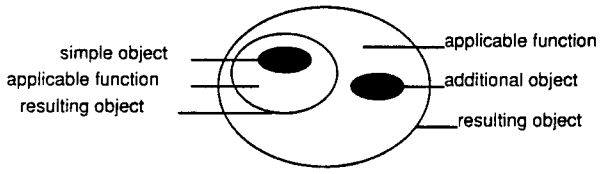


Figure 1.

Some Ada which might be represented as in the above figure might be:

```
package Counter is          -- resulting object
  procedure Reset;          -- applicable function ...
  procedure Increment;
  function Current_Value return Natural;
end Counter;

package body Counter is
  Count : Natural := 0;     -- simple object
  procedure Reset is
  begin
    Count := 0;
  end Reset;
  procedure Increment is
  begin
    Count := Count + 1;
  end Increment;
  function Current_Value return Natural is
  begin
    return Count;
  end Current_Value;
end Counter;


package Max_Count is        -- resulting object
  procedure Reset;          -- applicable function ...
  procedure Increment;
  function Current_Value return Natural;
  function Max return Natural;
end Max_Count;

with Counter;
package body Max_Count is
  Max_Val : Natural := 0;   -- additional object
  procedure Reset is
  begin
    Counter.Reset;
  end Reset;
  procedure Increment is
  begin
    Counter.Increment;
    if Max_Val < Counter.Current_Value then
      Max_Val := Counter.Current_Value;
    end if;
  end Increment;
  function Current_Value return Natural is
  begin
    return Counter.Current_Value;
  end Current_Value;
  function Max return Natural is
  begin
    return Max_Val;
  end Max;
end Max_Count;
```

In this example, the objects are properly encapsulated, though, they might not have been. If, for example, the simple objects were declared in separate components from their applicable functions, the result could have been the same (although the diagram might look different). In actual practice, Ada programs are developed with a combination of encapsulated object-operation groups as well as separately declared object-operation groups. Often the lowest levels are encapsulated while the higher level and larger objects tend to be separate from their applicable function. Perhaps in the ideal case, all objects would be encapsulated with their applied function since encapsulation usually makes the process of extracting the functionality at a later time easier. This, therefore, becomes one of the guidelines revealed by this model.

If the above example were transformed to separate the functionality from each object, the following set of components might be derived:

```
generic
  type Count_Object is (<>);
package Gen_Counter is      -- resulting object
  procedure Reset;          -- applicable function ...
  procedure Increment;
  function Current_Value return Count_Object;
end Gen_Counter;

package body Gen_Counter is
  Count : Count_Object      -- simple object
        := Count_Object'First;
  procedure Reset is
  begin
    Count := Count_Object'First;
  end Reset;
  procedure Increment is
  begin
    Count := Count_Object'Succ (Count);
  end Increment;
  function Current_Value return Count_Object is
  begin
    return Count;
  end Current_Value;
end Gen_Counter;

generic
  type Count_Object is (<>);
package Gen_Max_Count is    -- resulting object
  procedure Reset;          -- applicable function ...
  procedure Increment;
  function Current_Value return Count_Object;
  function Max return Count_Object;
end Gen_Max_Count;

with Gen_Counter;
package body Gen_Max_Count is
  Max_Val : Count_Object    -- additional object
            := Count_Object'First;
  package Counter is
            new Gen_Counter (Count_Object);
```

```
    procedure Reset is
    begin
      Counter.Reset;
    end Reset;
    procedure Increment is
    begin
      Counter.Increment;
      if Max_Val < Counter.Current_Value then
        Max_Val := Counter.Current_Value;
      end if;
    end Increment;
    function Current_Value return Natural is
    begin
      return Counter.Current_Value;
    end Current_Value;
    function Max return Natural is
    begin
      return Max_Val;
    end Max;
  end Gen_Max_Count;

  with Gen_Max_Count;
  procedure Max_Count_User is
    package Max_Count is
                    new Gen_Max_Count (Natural);
  begin
    Max_Count.Reset;
    Max_Count.Increment;
    ...
  end Max_Count_User;
```

Note that the end user obtains the same functionality that a user of Max_Count has, but the software now allows the primitive object Natural to be supplied externally to the algorithms that will apply to it. Further, the user could have obtained analogous functionality for any discrete type simply by pairing the general object with a different type (using a different generic instantiation).

This model is somewhat analogous to the one used in Smalltalk programming where objects are assembled from other objects plus programmer-supplied specifics. However, it is meant to apply more generally to Ada and other languages that do not have support for dynamic binding and full inheritance, features that are in general unavailable when strong static type checking is required. Instead, Ada offers the generic feature which can be used as shown here to partially offset the constraints imposed by static checking.

Applying this model to existing software means that any lines of code which represent reusable functionality must be parameterized with generic formal parameters in order to make them independent from their surrounding declaration space (if they are not already independent). Generics that are extracted by generalizing existing program units, through the removal of their dependence on external declarations, can then be offered as independently reusable components for other applications.

Unfortunately, declarative dependence is only one of the ways that a program unit can depend on its external environment. Removing the compiler-detectable declarative dependencies by producing a generic unit is no guarantee that the new unit will actually be independent. There can be dependencies on data values that are related to values in neighboring software, or even dependencies on protocols of operation that are followed at the point where a resource was originally used but which could be violated at a point of later reuse. (An example of this kind of dependency is described in the Measurement section.) To be complete, the transformation process would need to identify and remove these other types of dependence as well as the declarative dependence. Although guidelines have been identified by this study which can reduce the possibility for these other types of dependencies to enter a system, this work only concentrates on mechanisms to measure and remove declarative dependence.

More Examples

In a language with strong static type checking, such as Ada, any information exchanged between communicating program units must be of some type which is available to both units. Since Ada enforces name equivalence of types, where a type name and not just the underlying structure of a type introduces a new and distinct type, the declaration of the type used to pass information between units must be visible to both of those units. The user of a resource, therefore, is constrained to be in the scope of all type declarations used in the interface of that resource. In a language with a fixed set of types this is not a problem since all possible types will be globally available to both the resource and its users. However, in a language which allows user-declared types and enforces strong static type checking of those types, any inter-component communication with such types must be performed in the scope of those programmer-defined declarations. This means that the coupling between two communicating components increases from data coupling to external coupling (or from level two to level five on the traditional seven-point scale of Myers, where level one is the lowest level of coupling) [5].

Consider, for example, project-specific type declarations which often appear at low, commonly visible levels in a system. Resources which build upon those declarations can then be used in turn by higher level application-specific components. If a programmer attempts to reuse those intermediate-level resources in a new context, it is necessary to also reuse the low-level declarations on which they are built. This may not be acceptable, since combining several resources from different original contexts means that the set of low-level type declarations needed can be extensive and not generally compatible. This situation can occur whether or not data is encapsulated with its applicable function, but for clarity, and to contrast with the previous examples, it is shown here with the data and its operations declared separately.

For example, imagine that two existing programs each contain one of the following pairs of compilation units:

```
-- First program contains first pair:
package Vs_1 is
  type Variable_String is
    record
      Data    : String (1..80);
      Length  : Natural;
    end record;
  function Variable_String_From_User
                        return Variable_String;
end Vs_1;
```

PAGES 481-488 OMITTED

```
with Vs_1;
package Pm_1 is
  type Phone_Message is
    record
      From : Vs_1.Variable_String;
      To   : Vs_1.Variable_String;
      Data : Vs_1.Variable_String;
    end record;
  function Phone_Message_From_User
                        return Phone_Message;
end Pm_1;


-- Second program contains second pair:
package Vs_2 is
  type Variable_String is
    record
    - Data   : String (1..250) := (others=>' ');
      Length: Natural := 0;
    end record;
  function Variable_String_From_User
                        return Variable_String;
end Vs_2;

with Vs_2;
package Mm_2 is
  type Mail_Message is
    record
      From    : Vs_2.Variable_String;
      To      : Vs_2.Variable_String;
      Subject : Vs_2.Variable_String;
      Text    : Vs_2.Variable_String;
    end record;
  function Mail_Message_From_User
                        return Mail_Message;
end Mm_2;
```

Now, consider the programmer who is trying to reuse the above declarations in the same program. A reasonable way to combine the use of Mail_Messages with the use of Phone_Messages might seem to be as follows:

```
with Vs_1;
with Pm_1;
with Mm_2;
procedure User is
  Name : Vs_1.Variable_String;
  Pm : Pm_1.Phone_Message :=
              Pm_1.Phone_Message_From_User;
  Mm : Mm_2.Mail_Message :=
              Mm_2.Mail_Message_From_User;
begin
  Name := Pm.To;
  Mm.From := Name;    -- illegal
end User;
```

This will fail to compile, however, since the types Vs_1. Variable_String and Vs_2.Variable_String are distinct and therefore values of one are not assignable to objects of the other (the value of Name is of type Vs_1.Variable_String and the record component Mm.From is of type Vs_2. Variable_String).

In the above example, note that the variable string types were left visible rather than made private to make it seem even more plausible for a programmer to expect that, at least logically, the assignment attempted is reasonable. However, the incompatibility between the underlying type declarations used by Mail_Message and Phone_Message becomes a problem. One solution might be to use type conversion. However, employing type conversion between elements of the low level variable string types destroys the abstraction for the higher-level units. For instance, the user procedure above could be written as shown below, but exposing the detail of the implementation of the variable strings represents a poor, and possibly dangerous, programming style.

```
with Vs_1;
with Pm_1;
with Mm_2;
procedure Type_Conversion_User is
  Name : Vs_1.Variable_String;
  Pm : Pm_1.Phone_Message :=
            Pm_1.Phone_Message_From_User;
  Mm : Mm_2.Mail_Message :=
            Mm_2.Mail_Message_From_User;
begin
  Name := Pm.To;
  Mm.From.Data (1..80) := Name.Data;
  Mm.From.Length := Name.Length;
end Type_Conversion_User;
```

Notice that we had to be careful to avoid a constraint error at the point of the data assignment. This is one example of how attempts to combine the use of resources which rely on different context declarations is difficult in Ada.

Static type checking, therefore, is a mixed blessing. It prevents many errors from entering a software system which might not otherwise be detected until run time. However, it limits the possible reuse of a module if a specific declaration environment must also be reused. Not only must the reused module be in the scope of those declarations, but so must its users. Further, those users are forced to communicate with that module using the shared external types rather than their own, making the resource master over its users instead of the other way around. The set of types which facilitates communication among the components of a program, therefore, ultimately prevents most, if not all, of the developed algorithms from being easily used in any other program.

This study refers to declarations such as those of the above variable string types as *contexts*, and to components which build upon those declarations and which are in turn used by other components, such as the above Mail_Message and Phone_Message packages, as *resources*. Components which depend on resources are referred to as *users*. The above illustrates the general case of a context-resource-user relationship. It is possible for a component to be both a resource at one level and also a context for a still higher-level resource. The dependencies among these three basic categories of components can be illustrated with a directed graph. Figure 2 shows a graph of the kind of dependency illustrated in the example above.

A resource does not always need full type information about the data it must access in order to accomplish its task. In the above examples, it would be possible for the Mail and Phone message resources to implement their functions via the functions exported from the variable string packages without any further information about the structures of those lower level variable string types. Sometimes, even less knowledge

of the structure or functionality of the types being manipulated by a resource is required by that resource for it to accomplish its function.

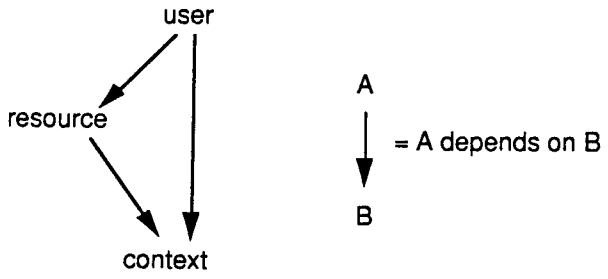


Figure 2.

A common example of a situation where a resource needs no structural or operational information about the objects it manipulates is a simple data base which stores and retrieves data but which does not take advantage of the information contained by that data. It is possible to write or transform such a resource so that the context it requires (i.e., the type of the object to be stored and retrieved) is supplied by the users of that resource. Then, only the essential work of the module needs to remain. This "essence only" principle is the key to the transformations sought. Only the purpose of a module remains, with any details needed to produce the executing code, such as actual type declarations or specific operations on those types, being provided later by the users of the resource. In languages such as Smalltalk which allow dynamic binding, this information is bound at run time. In Ada, where the compiler is obligated to perform all type checking, generics are bound at compilation time, eliminating a major source of run time errors caused by attempting to perform inappropriate operations on an object. Even though they are statically checked, however, Ada generics can often allow a resource to be written so as to free it from depending upon external type definitions.

Using the following arbitrary type declaration and a simplified data store package, one possible transformation is illustrated. First the example is shown before any transformation is applied:

```
-- context:
package Decls is
   type Typ is ...  -- anything but limited private
end Decls;

-- resource:
with Decls;
package Store is
  procedure Put (Obj : in Decls.Typ);
  procedure Get_Last (Obj : out Decls.Typ);
end Store;

package body Store is
  Local : Decls.Typ;
  procedure Put (Obj : in Decls.Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Decls.Typ) is
  begin
    Obj := Local;
  end Get_Last;
end Store;
```

The above resource can be transformed into the following one which has no dependencies on external declarations:

```
-- generalized resource:
generic
  type Typ is private;
package General_Store is
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

package body General_Store is
  Local : Typ;
  procedure Put (Obj : in Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Typ) is
  begin
    Obj := Local;
  end Get_Last;
end General_Store;
```

Note that, by naming the generic formal parameter appropriately, none of the identifiers in the code needed to change, and the expanded names were merely shortened to their simple names. This minimizes the handling required to perform the transformation (although automating the process would make this an unimportant issue). This transformation required the removal of the context clause, the addition of two lines (the generic part) and the shortening of the expanded names. The modification required to convert the package to a theoretically independent one constitutes a reusability measure. A user of the resource in the original form would need to add the following declaration in order to obtain an appropriate instance of the resource:

```
package Store is new General_Store (Decls.Typ);
```

Formal rules for counting program changes have already been proposed and validated [4], and adaptations of these counting rules (such as using a lower handling value for shortening expanded names and a higher one for adding generic formals) are being considered as part of this work.

The earlier example with the variable string types can also be generalized to remove the dependencies between the mail and phone message packages (resources) and the variable string packages (contexts). For example, ignoring the implementations (bodies) of the resources, the following would functionally be equivalent to those examples:

```
-- Contexts, as before:
package Vs_1 is
  type Variable_String is
    record
      Data : String (1..80);
      Len  : Natural;
    end record;
  function Variable_String_From_User
                        return Variable_String;
end Vs_1;

package Vs_2 is
  type Variable_String is
    record
      Data : String (1..250) := (others=>' ');
      Len  : Natural := 0;
    end record;
  function Variable_String_From_User
                        return Variable_String;
end Vs_2;


-- Resources, which no longer depend upon
-- the above context declarations:
generic
  type Component is private;
package Gen_Pm_1 is
  type Phone_Message is
    record
      From : Component;
      To   : Component;
      Data : Component;
    end record;
  function Phone_Message_From_User
                          return Phone_Message;
end Gen_Pm_1;

generic
  type Component is private;
package Gen_Mm_2 is
  type Mail_Message is
    record
      From : Component;
      To   : Component;
      Subj : Component;
      Text : Component;
    end record;
  function Mail_Message_From_User
                          return Mail_Message;
end Gen_Mm_2;
```

Now, the programmer who is trying to reuse the above declarations by combining the use of Mail_Messages with the use of Phone_Messages has another option. Instead of trying to combine both contexts, just one can be chosen (in this case, Vs_2):

```
with Vs_2;
with Gen_Pm_1;
with Gen_Mm_2;
procedure User is
  package Pm_1 is new
            Gen_Pm_1 (Vs_2.Variable_String);
  package Mm_2 is new
            Gen_Mm_2 (Vs_2.Variable_String);
  Name : Vs_2.Variable_String;
  Pm : Pm_1.Phone_Message :=
                Pm_1.Phone_Message_From_User;
  Mm : Mm_2.Mail_Message :=
                Mm_2.Mail_Message_From_User;
begin
  Name := Mm.From;
  Pm.To := Name;    -- now OK
end User;
```

An additional complexity is required for this example. The resources must be able to obtain component type values from which to construct mail and phone messages. Although this is not obvious from the specifications only, it can be assumed that such functionality must be available in the body. This can be done by adding a generic formal function parameter to the generic parts, requiring the user to supply an additional parameter to the instantiations as well:

```
generic
  type Component is private;
  with function Component_From_User
          return Component;
                        -- parameterless for simplicity
package Gen_Pm_1 is
  type Phone_Message is
    record
      From : Component;
      To   : Component;
      Data : Component;
    end record;
  function Phone_Message_From_User
                          return Phone_Message;
end Gen_Pm_1;
```

Although the above examples show the context, the resource, and the user as library level units, declaration dependence can occur, and transformations can be applied, in situations where the three components are nested. For example, the resource and user can be co-resident in a declarative area, or the user can contain the resource or vice versa.

This reiterates the earlier claim that, at least for the purpose of this model, it does not matter if the data is encapsulated with its applicable function, it just makes it easier to find if it is. In the programs studied, the lowest level data types, which were often properly encapsulated with their immediately available operations, were used to construct higher level resources specific to the problem being solved. It was unusual for those resources to be written with the same level of encapsulation and independence as the lower level types, and this resulted in the kind of context-resource-user dependencies illustrated above.

For example, in the case of the generalized simple data base, the functionality of the data appears in the resource while the declaration of it appears in the context. The only place where the higher-level object comes into existence is inside the user component, at the point where the instantiation is declared. If desired, an additional transformation can be applied to rectify this problem of the apparent separation of the object from its operations. Instead of leaving the instantiation of the new generic resource up to the client

software, an intermediate package can be created which combines the visibility of the context declarations with instantiations of the generic resource. This package, then, becomes the direct resource for the client software, introducing a layer of abstraction that was not present in the original (non-general) structure.

For example, the following transformation to the second example above combines the resource General_Store with the context of choice, type Typ from package Decls. The declaration of the package Object performs this service.

```
generic
  type Typ is private;
package General_Store is
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

package Decls is
  type Typ is ...
end Decls;

with Decls;
with General_Store;
package Object is
  subtype Typ is Decls.Typ;
  package Store is new General_Store (Typ);
  procedure Put (Obj : in Typ)
          renames Store.Put;
  procedure Get_Last (Obj : out Typ)
          renames Store.Get_Last;
end Object;

with Object;
procedure Client is
  Item : Object.Typ;
begin
  Object.Put (Item);
  Object.Get_Last (Item);
end Client;
```

Note that no body for package Object is required using the style shown. If it were preferable to leave the implementation of Object flexible, so that users would not need to be recompiled if the context used by the instantiation were to change, the context clauses and the instantiation could be made to appear only in the body of Object. An alternate, admittedly more complex, example is shown here which accomplishes this flexibility:

```
package Object is
  type Typ is  private;
  function Initial return Typ;
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : in Typ);
private
  type Designated;
  type Typ is access Designated;
end Object;

with Decls;
with General_Store;
package body Object is
  type Typ is new Decls.Typ;
  function Initial return Typ is
  begin
    return new Designated;
  end Initial;
  package Store is new General_Store (Typ);
  procedure Put (Obj : in Typ) is
  begin
     Store.Put (Obj.all);
  end Put;
  procedure Get_Last (Obj : in Typ) is
  begin
     Store.Get_Last (Obj.all);
  end Get_Last;
end Object;
```

In the alternate example, note that the parameter mode for the Get_Last procedure needed to be changed to allow the reading of the designated object of the actual access parameter. Also, a simple initialization function was supplied to provide the client with a way of passing a non-null access object to the Put and Get_Last procedures. Normally, there would already be initialization and constructor operations, so this additional operation would not be needed. The advantage of this alternative is that the implementation of the type and operations can change without disturbing the client software. However, the first alternative could be changed in a compilation-compatible way, such that any client software would need recompilation but no modification.

It is also possible to provide just an instantiation as a library unit by itself, but this requires the user to acquire independently the visibility to the same context as that instantiation. This solution results in the reconstruction of the original situation, where the instantiation becomes the resource dependent on a context, and the user depends on both. The important difference, however, is that now the resource (the instantiation) is not viewed as a reusable component. It becomes application-specific and can be routinely (potentially automatically) generated from both the generalized reusable resource and the context of choice, while the generic from which the instantiation is produced remains the independent, reusable component. The advantage of this structure lies in the abstraction provided for the user component which is insulated from the complexities of the instantiation of the reusable generic. Since the result is similar to the initial architecture, the overall software architecture can be preserved while utilizing generic resources. The following illustrates this.

```
package Decls is
  type Typ is ...
end Decls;

generic
  type Typ is private;
package General_Store is
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

with Decls;
with General_Store;
package Object is new General_Store(Decls.Typ);
```

```
with Decls;
with Object;
procedure Client is
  Item : Decls.Typ;
begin
  Object.Put (Item);
  Object.Get_Last (Item);
end Client;
```

By modifying the generic resource to "pass through" the generic formal types, the user's reliance on the context can be removed:

```
generic
  type Gen_Typ is private;
package General_Store is
  subtype Typ is Gen_Typ;   -- pass the type through
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

package Decls is
  type Typ is ...
end Decls;

with Decls;
with General_Store;
package Object is new General_Store(Decls.Typ);

with Object;
procedure Client is
  Item : Object.Typ;
begin
  Object.Put (Item);
  Object.Get_Last (Item);
end Client;
```

Measurement

In the above examples, the context components were never modified. Resource components were modified to eliminate their dependence on context components. User components were modified in order to maintain their functionality given the now general resource components, typically by defining generic actual parameter objects and adding an instantiation. In the case of the encapsulated instantiations, an intermediate component was introduced to free the user component of the complexity of the instantiation. It is the ease or difficulty of modifying the resource components that is of primary interest here, and the measurement of this modification effort constitutes a measurement of the reusability of the components. The usability of the generalized resources is also of interest, since some may be difficult to instantiate.

Considering the above examples again, the simple data base resource Store required the removal of the context clause and the creation of a generic part (these being typical modifications for almost all transformations of this kind). In addition, the formal parameter types for the two subprograms were changed to the generic formal private type, causing a change to both the subprogram specification and body. No further changes were required.

```
-- original:
with Decls;
package Store is
  procedure Put (Obj : in Decls.Typ);
  procedure Get_Last (Obj : out Decls.Typ);
end Store;

package body Store is
  Local : Decls.Typ;
  procedure Put (Obj : in Decls.Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Decls.Typ) is
  begin
    Obj := Local;
  end Get_Last;
end Store;

-- transformed:
generic
  type Typ is private;                        -- change
package General_Store is
  procedure Put (Obj : in Typ);               -- change
  procedure Get_Last (Obj: out Typ);          -- change
end General_Store;

package body General_Store is
  Local : Typ;
  procedure Put (Obj : in Typ) is             -- change
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj: out Typ) is        -- change
  begin
    Obj := Local;
  end Get_Last;
end General_Store;
```

The Phone_Message and Mail_Message resources required the deletion of the context clause, the addition of a generic part consisting of a formal private type parameter and a formal subprogram parameter, and the replacement of three occurrences (or four, in the case of Mail_Message) of the type mark Vs_1.Variable_String with the generic formal type Component.

```
-- original:
with Vs_1;
package Pm_1 is
  type Phone_Message is
    record
      From : Vs_1.Variable_String;
      To   : Vs_1.Variable_String;
      Data : Vs_1.Variable_String;
    end record;
  function Phone_Message_From_User
                          return Phone_Message;
end Pm_1;

-- transformed:
generic
  type Component is private;      -- change
  with function Component_From_User
              return Component;   -- change
```

```
package Gen_Pm_1 is
  type Phone_Message is
    record
      From : Component;          -- change
      To   : Component;          -- change
      Data : Component;          -- change
    end record;
  function Phone_Message_From_User
                        return Phone_Message;
end Gen_Pm_1;
```

Generalizing the bodies of Gen_Pm_1 and Gen_Mm_2 would involve replacing any calls to the Variable_String_From_User functions with calls to the generic formal Component_From_User function. In the case of the simple bodies shown before, this would require three and four simple substitutions, for Gen_Pm_1 and Gen_Mm_2, respectively.

In addition to measuring the reusability of a unit by the amount of transformation required to maximize its independence, reusability can also be gauged by the amount of residual dependency on other units which cannot be eliminated, or which is unreasonably difficult to eliminate, by any of the proposed transformations. For any given unit, therefore, two values can be obtained. The first reveals the number of program changes which would be required to perform any applicable transformations. The second indicates the amount of dependence which would remain in the unit even after it was transformed. The original units in the examples above would score high on the first scale since the handling required for its conversion was negligible, implying that its reusability was already good (i.e., it was already independent or was easy to make independent of external declarations). After the transformation, there remain no latent dependencies, so the transformed generic would receive a perfect reusability score.

Note that the object of any reusability measurement, and therefore, of any transformations, need not be a single Ada unit. If a set of library units were intended to be reused together then the metrics as well as the transformations could be applied to the entire set. Whereas there might be substantial interdependence among the units within the set, it still might be possible to eliminate all dependencies on external declarations.

In the above examples, one reason that the transformation was trivial was that the only operation performed on objects of the external type was assignment (except for the mail and phone message examples). Therefore, it was possible to replace direct visibility to the external type definition with a generic formal private type. A second example illustrates a slightly more difficult transformation which includes more assumptions about the externally declared type. In the following example, indexing and component assignment are used by the resource.

Before transformation:

```
-- context
package Arr is
  type Item_Array is
        array (Integer range <>) of Natural;
end Arr;

-- resource
with Arr;
procedure Clear (Item : out Arr.Item_Array) is
begin
  for I in Item'Range loop
    Item (I) := 0;
  end loop;
end Clear;

-- user
with Arr, Clear;
procedure Client is
  X : Arr.Item_Array (1..10);
begin
  Clear (X);
end Client;
```

After transformation:

```
-- context (same)
package Arr is
  type Item_Array is
        array (Integer range <>) of Natural;
end Arr;

-- generalized resource
generic
   type Component is range <>;
   type Index is range <>;
   type Gen_Array is
         array (Index range <>) of Component;
procedure Gen_Clear (Item : out Gen_Array);
procedure Gen_Clear (Item : out Gen_Array) is
begin
  for I in Item'Range loop
    Item (I) := 0;
  end loop;
end Gen_Clear;

-- user
with Arr, Gen_Clear;
procedure Client is
  X : Arr.Item_Array (1..10);
  procedure Clear is new Gen_Clear
                          (Natural,
                           Integer,
                           Arr.Item_Array);
begin
  Clear (X);
end Client;
```

The above transformation removes compilation dependencies, and allows the generic procedure to describe its essential function without the visibility of external declarations. As before, an intermediate object could be created to free the user procedure from the chore of instantiating a Clear procedure, which requires visibility to both the context and the resource. However, it also illustrates an important additional kind of dependence which can exist between a resource and its users, namely information dependence.

In the previous example, the literal value 0 is a clue to the presence of information that is not general. Therefore, the following would be an improvement over the transformation shown above:

```
generic
   type Component is range <>;
   type Index is range <>;
   type Gen_Array is
         array (Index range <>) of Component;
   Init_Val : Component := Component'First;
procedure Gen_Clear (Item : out Gen_Array);
procedure Gen_Clear (Item : out Gen_Array) is
begin
  for I in Item'Range loop
    Item (I) := Init_Val;
  end loop;
end Gen_Clear;
```

Note that the last transformation allows the user to supply an initial value, but also provides the lowest value of the component type as a default. An additional refinement would be to make the component type private which would mean that Init_Val could not have a default value. Information dependencies such as the one illustrated here are harder to detect than compilation dependencies. The appearance of literal values in a resource is often an indication of an information dependence.

A third form of dependence, called protocol dependence, has also been identified. This occurs when the user of a resource must obey certain rules to ensure that the resource behaves properly. For example, a stack which is used to buffer information between other users could be implemented in a not-so-abstract fashion by exposing the stack array and top pointer directly. In this case, all users of the stack must follow the same protocol of decrementing the pointer before popping and incrementing after pushing, and not the other way around. Beyond the recognition of it, no additional treatment of this form of dependence between components will appear in this study.

Formalizing the Transformations

The following is a formalization of the objectives of transformations which are needed to remove declaration dependence.

1. Let P represent a program unit.

2. Let D represent the set of n object declarations, d_1 .. d_n, directly referenced by P such that d_i is of a type declared externally to P.

3. Let O_1 .. O_n be sets of operations where O_i is the set of operations applied to d_i inside P.

4. P is completely transformable if each operation in each of the sets, O_1 .. O_n can be replaced with a predefined or generic formal operation.

The earlier example transformation is reviewed in the context of these definitions:

1. Let P represent a program unit.
   P = **procedure** Clear (Item : **out** Arr.Item_Array) is ...

2. Let D represent the set of n object declarations, d_1 .. d_n, directly referenced by P such that d_i is of a type declared externally to P.
   D = { Arr.Item_Array }

3. Let O_1 .. O_n be sets of operations where O_i is the set of operations applied to d_i inside P.
   O_1 =
{ indexing by integers, integer assignment to components }

4. P is completely transformable if each operation in each of the sets, O_1 .. O_n can be replaced with a predefined or generic formal operation.

Indexing can be obtained through a generic formal array type. Although no constraining operation was used, the formal type could be either constrained or unconstrained since the only declared object is a formal subprogram parameter. Since component assignment is required, the component type must not be limited. Therefore, the following generic formal parts are possible:

```
type Component is range <>;
type Index is range <>;
```

followed by either:

```
type Gen_Array is array (Index) of Component;
```

or:

```
type Gen_Array is
         array (Index range <>) of Component;
```

Notice that some operations can be replaced with generic formal operations more easily than others. For example, direct access of array structures can generally be replaced by making the array type a generic formal type. However, direct access into record structures (using "dot" notation) complicates transformations since this operation must be replaced with a user-defined access function.

Application to External Software

Medium-Sized Projects

To test the feasibility of the transformations proposed, a 6,000-line Ada program written by seven professional programmers was examined for reuse transformation possibilities. The program consisted of six library units, ranging in size from 20 to 2,400 lines. Of the 30 theoretically possible dependencies that could exist among these units, ten were required. Four transformations of the sort described above were made to three of the units. These required an additional 44 lines of code (less than a 1% increase) and reduced the number of dependencies from ten to five, which is the minimum possible with six units. Using one possible program change definition, each transformation required between two and six changes.

A fifth modification was made to detach a nested unit from its parent. This required the addition of 15 lines and resulted in a total of seven units with the minimum six dependencies. Next, two other functions were made independent of the other units. Unlike the previous transformations which were targeted for later reuse, however, these transformations resulted in a net reduction in code since the resulting components were reused at multiple points within this program. Substantial information dependency which would have impaired actual reuse was identified but remained within the units, however.

A second medium-sized project was studied which exhibited such a high degree of mutual dependence between pairs of library units that, instead of selecting smaller units for generalizations, the question of non-hierarchical dependence was studied at a system level. The general conclusion from this was that loops in the dependency structure (where, for example, package A is referenced from package body B and package B is referenced from package body A) make generalization of those components difficult. The program was instead analyzed for possible restructuring to remove as much of the bi-directional dependence as practical. This was partially successful and suggests that this sort of redesign might appropriately precede other reuse analyses.

The NASA Projects

Currently, the research project is examining several spacecraft flight simulation programs from the NASA Goddard Space Flight Center. These programs are each more than 100,000 editor lines of Ada. They have been developed by an organization that originally developed such simulators in Fortran and has been transitioning to the use of Ada over the past several years. Because all the programs are in the same application domain and were developed by the same organization there is considerable opportunity for reuse. In the past, the development organization reported the ability to reuse about 20% of earlier programs when a new program was being developed in Fortran. However, since becoming familiar with Ada, the same organization is now reporting a 70% reuse rate, or better.

After gaining an understanding of the nature of the reuse accomplished in Fortran and later in Ada, and how similar or different reuse in the two languages was, we would like to test several theories about why the Ada reuse has been so much greater. We already know that the reuse is accomplished by modifying earlier components as required, and not, in general, by using existing software verbatim. Because of this reuse mode, one theory we will be testing is that the Ada programs are more reusable simply because they are more understandable.

For the current study, the programs were studied to reveal opportunities to extract generic components which, had they been available when the programs were being developed originally, could have been reused without modification. There is an additional advantage to working with this data, however, since, as mentioned above, the several programs already exhibit significant functional similarities which can be studied for possible generalization. In other words, whereas the initial discussion of generic extraction has focussed on attempts to completely free the essential function of a component from its static declaration context, this data gives examples of similar components in two or more different program contexts and therefore allows us to study the possibility of freeing a component from only its program-specific context and not from any context which remains constant across programs.

This gives rise to the notion of domain-specific generic extraction as opposed to domain-independent generic extraction. Given the problems associated with extracting a completely general component, as examined earlier, a case can be made to generalize away only some of the dependence, leaving the rest in place. The additional problem, then, becomes how to determine what dependence is permissible and what should be removed. The permissible dependence would be common across projects in a certain domain, and would therefore be domain-specific while the dependence to be removed would be the problem-specific context. When reused, then, these components would have their problem-specific context supplied as generic actual parameters.

This is currently a largely manual task, since the programs must be compared to find corresponding functionality and then examined to determine the intersection of that functionality. Interestingly, on the last project the developers themselves have also been devising generic components which are instantiated only one time within that program. This implied to us that some effort was being spent to make components which might be reusable with no, or perhaps only very little, modification in the next project. We have confirmed with the developers that this is in fact the case. By comparing the results of our generalizations with those done by the developers, we find that ours have much more complex generic parts but correspondingly much less dependence on other software. This is a reasonable result, since the developers already have some idea about the context for each reuse of a given generic; what aspects of that context are likely to change from project to project and what aspects are expected to remain constant across several programs. The program-specific context, only, appears in the generic parts of the generics written by the developers, while our generalizations have generic parts which contain declarations of types and operations which apparently do not need to change as long as the problem domain remains the same. In other words, when our generic parts are devised by analyzing only a single instance of a component, we cannot distinguish between program-specific and domain-specific generalizations.

One interesting question we would like to answer is whether we can derive the generic part that makes the most sense within this domain by comparing similar components from different programs and generalizing only on their differences, leaving the software in the intersection of the components unchanged. In this way, a component would be derived which would not be completely independent but, like the developer-written generics, would be sufficiently independent for reuse in the domain. Then, a comparison with the generics developed within the organization would be revealing. If the generics are similar then our process might be useful on other parts of the software that have not yet been generalized by the developers. However, if they differ greatly, it would be useful to characterize that difference and

understand what additional knowledge must be used in generalizing the repeated software. Unfortunately, there is not enough reuse of the developer's generics yet to make this final comparison but a project is currently in progress which should supply some of this data.

The following example illustrates the complexity of the generic parts which were required to completely isolate a typical unit from its context. Here, the procedure Check_Header was removed from a package body and generalized to be able to stand alone as a library level generic procedure.

```
generic
  type Time is private;
  type Duration is digits <>;
  with function Enable return Boolean;
  type Hd_Rec_Type is private;
  with procedure Set_Start
       (H : in out Hd_Rec_Type; To : Duration);
  with function Get_Start
       (H : Hd_Rec_Type) return Duration;
  with procedure Set_Stop
       (H : in out Hd_Rec_Type; To : Duration);
  with function Get_Stop
       (H : Hd_Rec_Type) return Duration;
  type Real is digits <>;
  with function Get_Att_Int
       (H : Hd_Rec_Type) return Real;
  with function Conv_Time
       (D_Float : Duration) return Duration;
  Header_Rec : in out Hd_Rec_Type;
  Goesim_Time_Step : in out Duration;
  with function Seconds_Since_1957
       (T : in Time) return Duration;
  with procedure Debug_Write (Output : String);
  with procedure Debug_End_Line;
  type Direct_File_Type is limited private;
  with procedure Direct_Read
       (File : Direct_File_Type);
  with procedure Direct_Get
       (File : in  Direct_File_Type;
        Item : out Hd_Rec_Type);
  with function Image_Of_Base_10
       (Item : Duration) return String;
  with procedure Header_Data_Error;
procedure Check_Header_Generic
  (Simulation_Start_Time : in Time;
   Simulation_Stop_Time  : in Time;
   Simulation_Time_Step  : in Duration;
   History_File : in out Direct_File_Type);
```

The instantiation of this generic part is correspondingly complex:

```
procedure Check_Header_Instance is new
                          Check_Header_Generic
  (Abstract_Calendar.Time,
   Abstract_Calendar.Duration,
   Debug_Enable,
   Attitude_History_Types.Header_Record,
   Set_Start,
   Get_Start,
   Set_Stop,
   Get_Stop,
   Utilities.Read,
   Get_Att_Hist_Out_Int,
   Converted_Time,
   History_Data.Header_Rec,
   History_Data.Goesim_Time_Step,
   Timer.Seconds_Since_1957,
   Error_Collector.Write,
   Error_Collector.End_Line,
   Direct_Mixed_Io.File_Type,
   Direct_Mixed_Io.Read,
   Get_From_Buffer,
   Image_Of_Base_10,
   Raise_Header_Data_Error);
```

In contrast, a typical generic part on a unit which was developed and delivered as part of the most recent completed project by the developers themselves is shown here:

```
with Css_Types;
generic
  Number_Of_Sensors : Natural :=
       Css_Types.Number_Of_Sensors;
  with function Initialize_Sensor
     return Css_Types.Css_Database_Type is <>;
package Generic_Coarse_Sun_Sensor is
  ...
```

Note that by allowing the visibility of Css_Types, the generic part was simplified. Being unfamiliar with the domain, had we attempted to generalize Coarse_Sun_Sensor by examining only the non-generic version of a corresponding component in another program we would not be able to tell whether the dependence on Css_Types was program-specific or domain-specific. Here, however, the developer leads us to believe that Css_Types is domain-specific while the number of sensors and sensor initialization is program specific.

Guidelines

The manual application of the principles and techniques of generic transformation and extraction has revealed several interesting and intuitively reasonable guidelines relative to the creation and reuse of Ada software. In general, these guidelines appear to be applicable to programs of any size. However, the last guideline in the list, concerning program structure, was the most obvious when dealing with medium to large programs.

• Avoid direct access into record components except in the same declarative region as the record type declaration.

Since there is no generic formal record type in Ada (without dynamic binding such a feature would be impractical) there is no straightforward way to replace record component access with a generic operation. Instead, user-supplied access functions are needed to access the components and the type must be passed as a private type. This is unlike array types for which there are two generic formal types (constrained and unconstrained). This supports the findings of others which assert that direct referencing of non-local record components adversely affects maintainability [6].

• Minimize non-local access to array components.

Although not as difficult in general as removing dependence

on a record type, removing dependence on an array type can be cumbersome.

• Keep direct access to data structures local to their declarations.

This is a stronger conclusion than the previous two, and reinforces the philosophy of using abstract data types in all situations where a data type is available outside its local declarative region. Encapsulated types are far easier to separate as resources than globally declared types since the operations are localized and contained.

• Avoid the use of literal values except as constant value assignments.

Information dependence is almost always associated with the use of a literal value in one unit of software that has some hidden relationship to a literal value in a different unit. If a unit is generalized and extracted for reuse but contains a literal value which indicates a dependence on some assumption about its original context, that unit can fail in unpredictable ways when reused. Conventional wisdom applies here, and it might be reasonable to relax the restriction to allow the use of 0 and 1. However, experience with a considerable amount of software which makes the erroneous assumption that the first index of any string is 1 has shown that even this can lead to problems.

• Avoid mingling resources with application specific contexts.

Although the purpose of the transformations is to separate resources from application specific software regardless of the program structure, certain styles of programming result in programs which can be transformed more easily and completely. By staying conscious of the ultimate goal of separating reusable function from application declarations, whether or not the functionality is initially programmed to be generic, programmers can simplify the eventual transformation of the code.

• Keep interfaces abstract.

Protocol dependencies arise from the exportation of implementation details that should not be present in the interface to a resource. Such an interface is vulnerable because it assumes a usage protocol which does not have to be followed by its users. The bad stack example illustrates what can happen when a resource interface requires the use of implementation details, however even resources with an appropriately abstract interface can export unwanted additional detail which can lead to protocol dependence.

• Avoid direct reference to package Standard.Float

Even when used to define other floating point types, direct reference to Float establishes an implementation dependence that does not occur with anonymous floating point declarations. Especially dangerous is a direct reference to Standard.Long_Float, Standard.Long_Integer, etc., since they may not even compile on different implementations. Some care must also be taken with Integer, Positive, and Natural, though in general they were not associated with as much dependence as Float. Note that fixed point types in Ada are constructed as needed by the compiler. Perhaps the same philosophy should have been adopted for Float and Integer. Reference to Character and Boolean is not a problem since they are the same on all implementations.

• Avoid the use of 'Address

Even though it is not necessary to be in the scope of package System to use this attribute, it sets up a dependency on System.Address that makes the software non-portable. If this attribute is needed for some low-level programming than it should be encapsulated and never be exposed in the interface to that level.

• Consider the inter-component dependence of a design

By understanding how functionally-equivalent programs can vary in their degree of inter-component dependence, designers and developers can make decisions about how much dependence will be permitted in an evolving system, and how much effort will be applied to limit that dependence. For system developments which are expected to yield reusable components directly, a decision can be made to minimize dependencies from the outset. For developments which are not able to make such an investment in reusability, a decision can be made to allow certain kinds of dependencies to occur. In particular, dependencies which are removable through subsequent transformation might be allowed while those that would be too difficult to remove later might be avoided. A particularly cumbersome type of dependence occurs when two library units reference each other, either directly or indirectly. This should be avoided if at all possible. By making structural decisions explicitly, surprises can be avoided which might otherwise result in unwanted limitations of the developed software.

Acknowledgements

This work was supported in part by the U.S. Army Institute for Research in Management Information and Computer Science under grant AIRMICS-01-4-33267, and NASA under grant NSG-5123. Some of the software analysis was performed using a Rational computer at Rational's eastern regional office in Calverton, Maryland.

References

1. Basili, V. R. and Rombach, H. D. Software Reuse: A Framework. In preparation.

2. Basili, V. R. and Rombach, H. D. The TAME Project: Towards Improvement- Oriented Software Environments. IEEE Transactions on Software Engineering, SE-14, June 1988.

3 Funk & Wagnalls, Standard College Dictionary, New York, 1977.

4. Myers, G. Composite/Structured Design, Van Nostrand Reinhold, New York, 1978.

5. Dunsmore, H.E. and Gannon, J.D. Experimental Investigation of Programming Complexity. In Proceedings ACM/NBS 16th Annual Tech. Symposium: Systems and Software, Washington D.C., June 1977.

6. Gannon, J.D., Katz, E. and Basili, V.R. Characterizing Ada Programs: Packages. In Proceedings Workshop on Software Performance, Los Alamos National Laboratory, Los Alamos, New Mexico, August 1983.



John W. Bailey is a Ph.D. candidate at the University of Maryland Computer Science Department. He is a part-time employee of Rational and has been consulting and teaching in the areas of Ada and software measurement for seven years. In addition to Ada and software reuse, his interests include music, photography, motorcycling and horse support. Bailey received his M.S. in computer science from the University of Maryland, where he also earned bachelor's and master's degrees in cello performance. He is a member of the ACM.



Victor R. Basili is a professor at the University of Maryland, College Park's Institute for Advanced Computer Studies and Computer Science Department. His research interests include measuring and evaluating software development. He is a founder and principal of the Software Engineering Laboratory, which is a joint venture among NASA, the University of Maryland and Computer Sciences Corporation. Basili received his B.S. in mathematics from Fordham College, an M.S. in mathematics from Syracuse University and a Ph.D. in computer science from the university of Texas at Austin. He is a fellow of the the IEEE Computer Society and is editor-in-chief of IEEE Transactions on Software Engineering.

Literate Programming for Reusability:
A Queue Package Example

T. L. (Frank) Pappas
Intermetrics, Inc.
607 Louis Drive
Warminster, Pa 18974

Abstract

Writing reusable software components requires more than just following coding guidelines. It also requires that potential clients of a component can easily understand the documentation associated with the component. Literate programming, as suggested by Donald Knuth [7], is concerned with combining code and documentation in a form that is more easily read by humans, rather than by compilers. This paper illustrates the benefits of a literate programming approach to writing reusable software by presenting a generic package written in a literate programming style. ADAWEB, a language for literate programming that combines Ada and TEX [10], was used to prepare this paper.

Introduction. Programmers can write reusable Ada software if they follow a reasonable set of guidelines [6]. For example, the guidelines in [11] are based on four categories of reusability issues: readability, composability, portability, and RunTime System (RTS) independence.

Briefly, *readability* is concerned with the presentation of the software. If potential clients (reusers) of the software cannot understand the software, they will not reuse it. *Composability* is concerned with the ease of combining Ada units. For example, if a unit depends on its context, it cannot be combined with other units, unless its context is also included. *Portability* is concerned with ensuring equivalent execution when the software is reused in other applications or environments. *RTS independence* is concerned with avoiding inadvertent dependence on a particular RTS.

While writing reusable software is important, it is not enough. Even if we can identify and retrieve reusable software [5], there are other factors that impede software reuse. An important factor, is the lack of trust programmers have in someone else's software. An insightful analogy is offered in [13]. "People are leery about buying a used car for many of the same reasons programmers are reluctant to reuse someone else's work."

If a software component is used successfully over a long period of time, programmers will learn to trust the component, and reuse it without hesitation. But the first few times a component is considered for reuse, it will be reused only after a programmer reads and tests the software, and decides that the component is useful, reliable, and meets efficiency requirements.

Accepting the definition of software given in [4], software includes both code and documentation; so if we are concerned with writing reusable software, we should be concerned with the documentation as well as the code. If a software component gives a programmer the impression that it will take almost as much time to understand the component as it will to write an equivalent one from scratch, the component will not be reused!

Unfortunately, software documentation from a reusability viewpoint has not received the attention it deserves. When documentation is provided as a separate document, it may adhere to a standard, but it frequently satisfies only the letter of the standard, not its intent. When the documentation is provided as source commentary, it is usually written after the fact; and if the commentary adequately defines how the code works and also provides the information needed to easily reuse or modify the code, then the code will be difficult to read, understand, and maintain. In either case, adaptations to the software frequently result in documentation that does not agree with the code.

Literate programming. Donald Knuth's literate programming approach offers a different perspective on writing software [1,2,8,12]. Literate programming focuses on software as a message from its author to its readers, i.e., the software is written so that it can be read by people rather than organized for processing by a compiler. The translation of a literate program into a form that can be processed by a compiler is left to computers.

Software written in a literate programming style closely reflects Dijkstra's virtual machine approach to software development. Recalling Dijkstra's approach,

the author of a component initially hypothesizes the existence of a virtual machine with an instruction set that provides a solution to a simple algorithmic statement of the problem that the component is to solve. Successful refinements of the solution replace the high-level statements of the virtual machine with statements from a lower-level virtual machine, that again solve the problem. These refinements continue until the virtual machine actually exists, i.e., the programming language that the solution is written in along with external components, executive services, or operating system services.

Unfortunately, the sequence of refinements, along with any commentary or graphics describing a refinement, is usually not part of the component, nor is it usually keep as part of the component's documentation. This means that someone who wants to reuse or modify this component does not have access to the insights the author had while developing the component.

With literate programming, the refinements, text, and graphics are the component. The refinements are the "sections" of the component, but rather than replacing each refinement with the instructions of the lower-level virtual machine, a higher-level instruction is refined in its own section, with its own text and graphics. This captures the essence of the virtual machine approach, while providing literate software.

A language for literate programming. As an aid in writing literate programs, Knuth combined Pascal and his TEX typesetting system [10] into a literate programming documentation language named WEB [7]. Then, with the aid of two WEB processors, WEAVE and TANGLE, Knuth used WEB to rewrite TEX as a literate program, which he subsequently published as a 594-page book [9]. Recently Ada versions of WEB, WEAVE, and TANGLE have been developed [3,14]. (Versions for other languages are also available [3].)

Figure 1 illustrates how the Ada version of WEB was used to produce this paper. The ADAWEB file, `queue.web`, was processed by ADAWEAVE to produce the TEX file, `queue.tex`. The TEX compiler was used to produce a device independent file, `queue.dvi`, that can be printed using a printer specific driver. The file was then printed on a laser printer. To test the component, the file `queue.web` was processed by ADATANGLE to produce the Ada source file `queue.ada`, which was then compiled and tested.

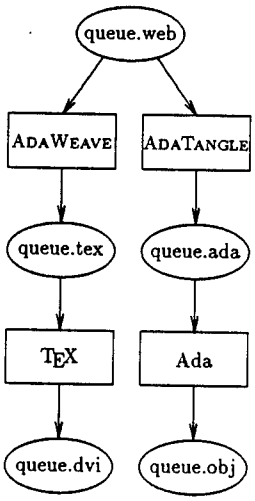


Figure 1: Processing an ADAWEB file

Using ADAWEB, programmers can write software so that the documentation and code complement each other. From Ada, ADAWEB users get a powerful programming language that supports modern software engineering concepts and encourages the creation of reusable code [11]. From TEX, ADAWEB users get a powerful typesetting language that simplifies the tasks of explaining how the software works, what system dependencies exist, and how the software is used. ADAWEB users can even use TEX's limited graphics. Combined with its macro capability, ADAWEB provides an effective tool for writing reusable software.

ADAWEB also supports system maintenance through its change-file capability. Software changes can be specified in a separate input file, so variants for different systems or enhanced versions still use the original version as a base. When a typeset version of the software is printed, changed sections are marked. The use of a changefile is illustrated later.

The remainder of this paper describes the implementation of a generic queue package using ADAWEB and following the guidelines in [11]. The figures, graphics, and special effects were produced using only features of ADAWEB, TEX, METAFONT, and TEX macro packages running on an PC AT/386 with MSDOS. TEX and its macro packages are available on a wide range of computers, from PC's to mainframe computers, to supercomputers. Since ADAWEB is written in C, it should also execute on a wide range of computer systems.

Currently, the only usable version of ADAWEB [3] is

exclusively available for research purposes. A commercial version of AdaWeb is under development by an Ada product vendor, and should be available in the summer of 1990.

With the exception of the commentary added for this paper [which has been inserted within these double brackets.], the material in the generic queue example would be part of the literate program. Note that the table of contents and index are generated automatically from the AdaWeb file by AdaWeave. However, the AdaWeb user does have some control over what is placed in both the table of contents and the index.

References

[1] Jon Bentley and Donald E. Knuth. Programming Pearls: Literate Programming. *CACM*, 364–369, May 1986.

[2] Jon Bentley, Donald E. Knuth, and Douglas McIlroy, Programming Pearls: A Literate Program. *CACM* 471–483, June 1986.

[3] Jon Bentley and Norman Ramsey. Weaving a language independent WEB. *CACM*, 1051–1055, September 1989.

[4] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, Inc., 1981.

[5] Bruce A. Burton, Rhonda W. Aragon, Stephen A. Bailey, Kenneth D. Koehler, and Lauren A. Mayes. The Reusable Software Library. *Software*, 25–33, July 1987.

[6] Anthony Gargaro and T. L. (Frank) Pappas. Reusability Issues and Ada. *Software*, 43–51, July 1987.

[7] Donald E. Knuth. *The Web System of Structured Documentation.* Technical Report STAN-CS-83-980 Stanford University, September 1983.

[8] Donald E. Knuth. Literate Programming. *Computer Journal*, 97-111, May 1984.

[9] Donald E. Knuth. *TEX: The Program*. Addison-Wesley, 1986.

[10] T. L. (Frank) Pappas. TEXnology on the IBM PC. *Computer*, 111-120, August 1989.

[11] T. L. (Frank) Pappas and Anthony Gargaro. *Fundamentals of Ada Reusability.* Cambridge University Press, Summer 1990.

[12] Sewell, E. Wayne *Weaving a Program: Literate Programming in* Web. Van Nostrand Reinhold, 1989

[13] Will Tracz. Reusability Comes of Age. *Software*, 6–8, July 1987.

[14] Y. C Wu and T. P. Baker. A Source Code Documentation System for Ada. *Ada Letters*, ix(5):84-88, July/August 1989.



Frank Pappas is an Ada consultant and senior computer scientist for the Defense Systems Group at Intermetrics. He has been involved with Ada since 1982, when he became a coauthor of the U.S. Army Ada Curriculum. Since 1984, Frank has had a major interest in Ada software reuse, and he is currently coauthoring a book on that topic. He also teaches a number of Ada programming courses.

Frank has an M.S. in computer science from Villanova University, and is a member of the ACM, SIGAda, and the IEEE Computer Society.

1* A Bounded Generic Queue Package. A Queue is a basic data structure. As illustrated in Figure 2, it is a sequence of items in which the items can only be added at one end of the sequence, called the *back* of the queue, and can only be removed from the other end, called the *front* of the queue. The number of items in the queue (sequence) is the *length* of the queue. If there are no items in the queue, the queue is *empty*. A queue is bounded if there is a maximum size to any particular sequence. A bounded queue is *full* if it contains the maximum number of items.

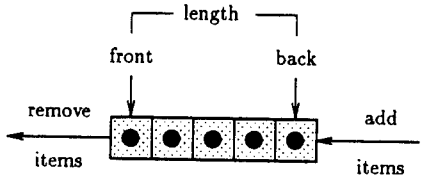


Figure 2: The structure of a queue

The following table gives an overview of the bounded queue abstraction provided by this component.

Abstraction	Bounded queue type–manager.	
Generic parameters	Item_Type:	The private type of items in the queue.
Visible data definitions	Queue_Type:	The limited private type of the queue.
Constructor procedures	Clear: Assign: Add: Remove:	Make the queue empty. Make a copy of the queue. Add an item to the queue. Remove an item from the queue.
Selector functions	Equal: Empty: Full: Length: Front:	Are two queues equal? Is the queue empty? Is the queue full? Length of the queue. Item at the front of the queue.
Exceptions	Queue_Empty: Queue_Full: Queue_Too_Small:	Illegal operation on an empty queue. Illegal operation on a full queue. Target queue is too small.

Figure 3: Overview of this queue abstraction

[An AdaWeb program is always broken into small pieces, called sections. The sections are numbered serially. The present section number is 1. To keep the size of this paper within reasonable limits, some sections have been omitted.]

[Every AdaWeb section begins with optional *commentary* about that section. The commentary is written in TEX. You are now reading the commentary for §1. The commentary is followed by optional *definitions* which can be used to make the section more readable. While this section does not have any definitions, several later ones do. Finally, the section ends with optional *program text.* In this section, the program text for §1 directly follows this paragraph. Program text is either written directly in Ada or also includes angle brackets to represent Ada code that appears in later sections. For example, you use the angle brackets notation '⟨ Bounded generic queue package 2 ⟩' to say "The Ada text to be inserted here is called 'Bounded generic queue package', and you can find out about it by looking at section 2."]

⟨ Bounded generic queue package. 2* ⟩

2* The generic package consists of a generic package specification and a package body. These units will be compiled together.

☞ Some Ada compilers may require a generic package specification and its package body to be submitted together, so don't separate them. [The pointing finger is not part of AdaWeb. It was added to emphasize key points about the use of the component. The graphic itself, was created using one of TEX's companion programs, METAFONT, TEX's font generator.]

[AdaWeb inserts a line after the program text that informs you in which sections the program text is used. In this section, the program text was used only in section 1.]

⟨ Bounded generic queue package. 2* ⟩ ≡
  ⟨ Generic queue package specification. 3* ⟩
  ⟨ Queue package body. 19* ⟩
This code is used in section 1*.

3*. Generic queue package specification. As an abstract data type, the queue has constructors that change the state of queue objects and selectors that yield information about the state.

☞ Naming conventions for packages vary widely, so a macro, *Queue_Package_Template*, is used for the package name in the specification and body. To change the name of this package to one that is consistent with the guidelines in [11], add the following lines to the change file:

```
@x
@d Queue_Package_Template = Bounded_Queue
@y
@d Queue_Package_Template = Bounded_Queue_Package_Template
@z
```

[If the indicated change is made, the longer name will appear as the replacement text in the TEX file produced by AdaWeave. The Ada file produced by AdaTangle will use the longer name as the package name in both the specification and the body. The change file contains a collection of changes, each of which has the form '@x⟨old lines⟩@y⟨new lines⟩@z'. The new lines are used to replace the old lines when the AdaWeb file is processed. An '*' is appended to the section numbers of those sections changed. Optionally, only the changed sections are printed. This feature was used to avoid printing all of the sections of the AdaWeb component for this paper.]

[A parameterless macro has the form:

```
@d macro_name = replacement_text
```

where each occurrence of `macro_name` is replaced by the `replacement_text`. These macros should only be used when either the desired effect cannot be achieved in Ada, or their use increases the readability of the Aweb program.]

define *Queue_Package_Template* ≡ *Bounded_Queue*

⟨ Generic queue package specification. 3* ⟩ ≡
```
generic
  ⟨ Generic parameters. 4* ⟩
package Queue_Package_Template is
  ⟨ Visible data definitions. 5* ⟩
  ⟨ Constructor specifications. 6* ⟩
  ⟨ Selector specifications. 11* ⟩
  ⟨ Exceptions. 17 ⟩
private
  ⟨ Private part. 18* ⟩
end Queue_Package_Template;
```
This code is used in section 2*.

4* The generic parameters. This package only has one generic parameter, namely the type of the item to be placed on the queue. The type is **private** to provide the most flexibility in using the queue.

☞ The subtype passed as the generic actual parameter must not be a limited type, an unconstrained array type, or an unconstrained type with discriminants.

Since the package requires assignment and equality operations, it doesn't make sense to use a limited private type and require the user to pass an assignment operation as a generic parameter. [The dangerous bend sign, which is not part of AdaWeb, is used to caution against potentially unwise or dangerous changes to the component. Again, it was created using METAFONT.]

⟨ Generic parameters. 4* ⟩ ≡
type *Item_Type* **is private**;
This code is used in section 3*.

5* The visible data definitions. Only one data definition is visible, namely the queue type. It is limited private, rather than just private, so the operations of assignment (*Assign*) and equality (*Equal*) are operations on abstract queue objects rather than on their representation.

☞ A queue object of size $n > 0$ is declared as:

$$Queue : Queue_Type\ (Size \Longrightarrow n);$$

After this declaration is elaborated, the following conditions will hold:

$$Length\ (Of_The_Queue \Longrightarrow Queue) = 0$$
$$Empty\ (Queue) = True$$
$$Full\ (Queue) = False$$

⟨ Visible data definitions. 5* ⟩ ≡
type *Queue_Type* (*Size* : *Positive*) **is limited private**;
This code is used in section 3*.

6* Constructor specifications.

⟨ Constructor specifications. 6* ⟩ ≡
⟨ *Clear* the items from the queue. 7* ⟩
⟨ *Assign* the items from one queue to another. 8* ⟩
⟨ *Add* the item to the back of the queue. 9 ⟩
⟨ *Remove* the item from the front of the queue. 10 ⟩
This code is used in section 3*.

7* Clear constructor specification. This constructor makes the queue empty. In particular, after execution of the statement

$$Clear\ (The_Queue \Longrightarrow Queue);$$

the following conditions will hold:

$$Length\ (Of_The_Queue \Longrightarrow Queue) = 0$$
$$Empty\ (Queue) = True$$
$$Full\ (Queue) = False$$

⟨ *Clear* the items from the queue. 7* ⟩ ≡
procedure *Clear* (*The_Queue* : **in out** *Queue_Type*);
This code is used in section 6*.

8* Assign constructor specification. This constructor assigns the elements of one queue to another. Execution of the statement

$$Assign\ (From \Longrightarrow A,\ To \Longrightarrow B);$$

will complete with the condition

$$Equal\ (A,\ B) = True$$

holding if $Length\ (Of_The_Queue \Longrightarrow A) \leq B.Size$; otherwise the exception *Queue_Too_Small* will be raised.

⟨ *Assign* the items from one queue to another. 8* ⟩ ≡
  procedure *Assign* (*From* : **in** *Queue_Type*; *To* : **in out** *Queue_Type*);
This code is used in section 6*.

11* Selector specifications.

⟨ Selector specifications. 11* ⟩ ≡
  ⟨ Are the two queues *Equal*? 12* ⟩
  ⟨ Is the queue *Empty*? 13 ⟩
  ⟨ Is the queue *Full*? 14 ⟩
  ⟨ What is the *Length* of the queue? 15* ⟩
  ⟨ What is the item at the *Front* of the queue? 16 ⟩
This code is used in section 3*.

12* Equal selector specification. If A and B are queues with elements $a_1, a_2, \cdots, a_n$, and $b_1, b_2, \cdots, b_m$, respectively, then

$$Equal\ (A,\ B) = True$$

holds whenever $n = m$, and $a_i = b_i$, $i = 1, 2, \ldots, n$.

⟨ Are the two queues *Equal*? 12* ⟩ ≡
  function *Equal* (*Left* : *Queue_Type*; *Right* : *Queue_Type*) **return** *Boolean*;
This code is used in section 11*.

15* Length selector specification. For any queue A,

$$Length\ (Of_The_Queue \Longrightarrow A) = n$$

where $a_1, a_2, \cdots, a_n$ is the (possibly null) sequence of elements in A.

⟨ What is the *Length* of the queue? 15* ⟩ ≡
  function *Length* (*Of_The_Queue* : *Queue_Type*) **return** *Natural*;
This code is used in section 11*.

18* The private part. The full type for *Queue_Type* is a record type with a discriminant *Size*. This was apparent from the private type. Note that we deviated from the convention of suffixing record components with *_Part* when naming *Size*, since we want to choose the name from the user's perspective.



A default discriminant expression is not provided since some compilers allocate the largest possible record size for unconstrained record objects. To prevent the unsuspecting user from such compilers, an initial value was not specified. If unconstrained queue objects are really needed, change the discriminant to:

Size : *Positive* := 0



The index subtype of *List_Type* is *List_Index_Subtype*. This subtype seems unnecessary since it is just a renaming of *Natural*. However, it is used to make the definition of *Queue_Type* easier to modify if this component must be compiled under an compiler that attempts to allocate the maximum possible size for *Queue_Type* objects. For such compilers, only the definition of *List_Index_Subtype* needs to be changed to

subtype *List_Index_Subtype* **is** *Natural* **range** 0..*n*;

where *n* is the maximum queue size needed. [This section will appear in the index under **system dependencies.**]

[The program text for this section was typed in as follows:

```
@<Private part.@>=
subtype List_Index_Subtype is Natural;
type    List_Type is array (List_Index_Subtype range <>) of Item_Type;
@# %add a little bit of extra space between the lines
type Queue_Type (Size: Positive) is
record
Front_Part : List_Index_Subtype := 1;
Back_Part  : List_Index_Subtype := 0;
List_Part  : List_Type (0 .. Size);
end record;
```

When ADAWEAVE processed queue.web, it inserted TEX macros to cause the reserved words to be printed in boldface and the identifiers to be printed in italics, and to insert the identifiers into the index.]

[Similarly to get 'if *Left_Item* ≠ *Right_Item* **then**' in the section commentary simply requires typing if *Left_Item* ≠ *Right_Item* then .]

⟨ Private part. 18* ⟩ ≡

```
subtype List_Index_Subtype is Natural;
type List_Type is array (List_Index_Subtype range <>) of Item_Type;
type Queue_Type (Size : Positive) is
  record
    Front_Part : List_Index_Subtype := 1;
    Back_Part : List_Index_Subtype := 0;
    List_Part : List_Type (0..Size);
  end record;
```

This code is used in section 3*.

19* Queue Package Body. A nice way of looking at the representation of a queue is illustrated in Figure 4. Here, the two ends of the *List_Part* array are brought together, so item 0 follows item 7, and item 7 precedes item 0. The front of the queue is marked with *Front_Part* which is the index of the front item. The back of the queue is marked with *Back_Part*, which is the index of the last item added. In Figure 4, the queue consists of the single slice:

$$List_Part\ (3..7)$$

The next item added to the list will be placed in *List_Part* (0). [The figures in this paper were generated using the PICTEX macro package. This macro package will work with any TEX implementation without modification.]

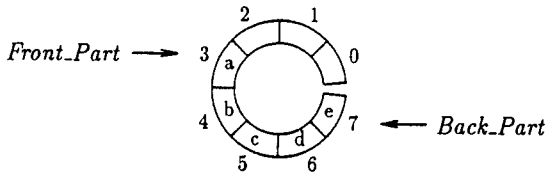


Figure 4: A simple queue with five items

A more complex situation results when the elements in the queue cross from item 7 to item 0, as illustrated in Figure 5. The easiest way to deal with this situation is to consider the list as consisting of a , *List_Part* (*Front_Part*..*Size*), and a *List_Part* (0..*Back_Part*). In Figure 5, this corresponds to *List_Part* (7..7) and *List_Part* (0..3), respectively. The representation has only one piece when *Front_Part* $\leq$ *Back_Part*.

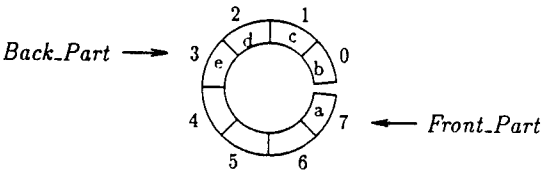


Figure 5: A complex queue with five items

⟨ Queue package body. 19* ⟩ ≡
```
package body Queue_Package_Template is
   ⟨ Constructor bodies. 21* ⟩
   ⟨ Selector bodies. 29* ⟩
end Queue_Package_Template;
```
This code is used in section 2*.

20* This is a useful macro definition that is used in more than one section. [Macros can also have parameters.]
define *There_Is_Only_One_Piece_In*(*q*) ≡ *q.Front_Part* $\leq$ *q.Back_Part*

21* Constructor bodies.

⟨ Constructor bodies. 21* ⟩ ≡
⟨ Body of *Clear* the items from the queue. 22* ⟩
⟨ Body of *Assign* the items from one queue to another. 23* ⟩
⟨ Body of *Add* the item to the back of the queue. 27 ⟩
⟨ Body of *Remove* the item from the front of the queue. 28 ⟩
This code is used in section 19*.

22* Clear constructor body. This procedure simply resets the values of *Front_Part* and *Back_Part* to their initial values, as shown in Figure 6. There's nothing magic about the values 1 and 0. All that's really needed is that *Front_Part* = (*Back_Part* + 1) **mod** (*The_Queue.Size* + 1).

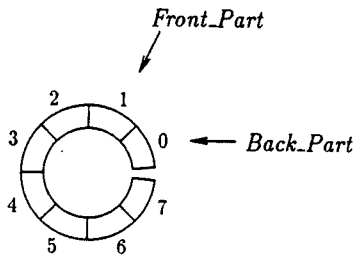


Figure 6: Initializing a queue to empty

⟨ Body of *Clear* the items from the queue. 22* ⟩ ≡
```
procedure Clear (The_Queue : in out Queue_Type) is
begin
   The_Queue.Back_Part := 0;
   The_Queue.Front_Part := 1;
end Clear;
```
This code is used in section 21*.

23* Assign constructor body. First check that the *To* queue is large enough to hold all of the elements in the *From* queue. If it is, then make the assignment.

define *The_Queue_Is_Not_Big_Enough* ≡ *Length* (*Of_The_Queue* ⟹ *From*) > *To.Size*

⟨ Body of *Assign* the items from one queue to another. 23* ⟩ ≡
```
procedure Assign (From : in Queue_Type; To : in out Queue_Type) is
begin
   if The_Queue_Is_Not_Big_Enough then
      raise Queue_Too_Small;
   else
      ⟨ Make the assignment. 24* ⟩
   end if;
end Assign;
```
This code is used in section 21*.

24* Copying the elements of the *To* queue is simple if there is only one piece, but it gets a little bit complicated, when there are two pieces. Therefore, the two cases are considered separately.

⟨ Make the assignment. 24* ⟩ ≡
```
if There_Is_Only_One_Piece_In (From) then
   ⟨ Perform a simple assignment. 25* ⟩
else
   ⟨ Perform a complex assignment. 26* ⟩
end if;
To.Back_Part := Length (Of_The_Queue ⟹ From);
To.Front_Part := 1;
```
This code is used in section 23*.

25.* When there's only one piece, a simple slice assignment can be used. For the queue in Figure 7, we would just assign the slice *(3..7)* from *To* to the slice *(1..5)* in *From*.

[Notice that the macro *The_To_Part_Of_The_List* cannot be replaced by using a renaming declaration, since *List_Part* depends on the value of a discriminant.]

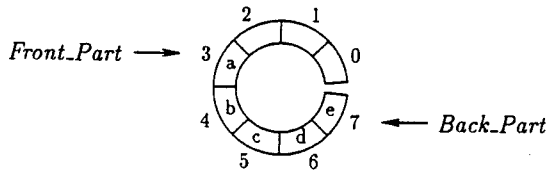


Figure 7: A queue with five items

define *The_To_Part_Of_The_List* ≡ *To.List_Part* (1..*Length* (*Of_The_Queue* ⟹ *From*))

⟨ Perform a simple assignment. 25* ⟩ ≡

The_To_Part_Of_The_List := *From.List_Part* (*From.Front_Part*..*From.Back_Part*);

This code is used in section 24*.

26.* When there are two pieces, we need to form the catenation of the left part and the right part [references to which have been placed in the Index]. In Figure 8, we take the catenation of the two slices *(7..7)* and *(0..3)*.

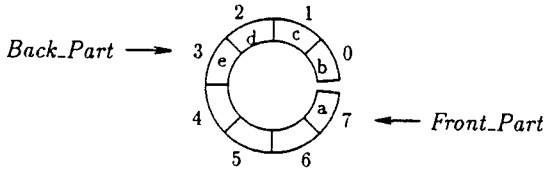


Figure 8: Another queue with five items

define *The_Left_Part_Of_The_From_List* ≡ *From.List_Part* (*From.Front_Part*..*From.Size*)

define *The_Right_Part_Of_The_From_List* ≡ *From.List_Part* (0..*From.Back_Part*)

⟨ Perform a complex assignment. 26* ⟩ ≡

The_To_Part_Of_The_List := *The_Left_Part_Of_The_From_List*&*The_Right_Part_Of_The_From_List*;

This code is used in section 24*.

29.* Selector bodies.

⟨ Selector bodies. 29* ⟩ ≡

⟨ Body of Are the two queues *Equal*? 30* ⟩

⟨ Body of Is the queue *Empty*? 32 ⟩

⟨ Body of Is the queue *Full*? 33 ⟩

⟨ Body of What is the *Length* of the queue? 34 ⟩

⟨ Body of What is the item at the *Front* of the queue? 35 ⟩

This code is used in section 19*.

30*. Equal selector body. Two queues must have the same length if they are equal, so we might as well check for this condition first.

```
define The_Queues_Have_The_Same_Length ≡
        Length (Of_The_Queue ⟹ Left) = Length (Of_The_Queue ⟹ Right)
define The_Queues_Are_Different ≡ False
define The_Queues_Are_Identical ≡ True

⟨ Body of Are the two queues Equal? 30* ⟩ ≡
  function Equal (Left : Queue_Type; Right : Queue_Type) return Boolean is
  begin
    if The_Queues_Have_The_Same_Length then
      ⟨ Test for equality. 31* ⟩
    else
      return The_Queues_Are_Different;
    end if;
  end Equal;
```

This code is used in section 29*.

31*. Now we have to check each element in order, to see if the queues are identical. ⟦*Each_Item_In_The_Queues* is a macro that cannot be defined as a constant or function in Ada, since it consists of a membership test. Also, note the use of the *format* definition for **forloop**. This says that the macro **forloop** should be formatted the same way **loop** is. This let's us use a distinct terminator for a *for loop*.

```
define Each_Item_In_The_Queues ≡ i in 1..Length (Of_The_Queue ⟹ Left)
define An_Item_Of(q) ≡ q.List_Part ((q.Front_Part + i − 1) mod (q.Size + 1))

define Left_Item ≡ An_Item_Of (Left)
define Right_Item ≡ An_Item_Of (Right)

define forloop ≡ loop
  format forloop loop

⟨ Test for equality. 31* ⟩ ≡
  for Each_Item_In_The_Queues loop
    if Left_Item ≠ Right_Item then
      return The_Queues_Are_Different;
    end if;
  end forloop;
  return The_Queues_Are_Identical;
```

This code is used in section 30*.

36.* Index.

Every identifier, of length two or more, used in this component is shown here together with a list of the section numbers where the identifier appears. Underlined entries indicate where the identifier was defined. [In this version of AdaWeb, only macro names are automatically underlined. A production version of AdaWeb should extend this underlining to all entities. Users can add entries to the index, e.g, system dependencies. Users can force an entry to be underlined, prevent an entry from being underlined or even from appearing in the index.] Section names also appear in the index. [The rest of the Index was generated by AdaWeave.]

The following sections were changed by the change file: 1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 15, 18, 19, 20, 21, 22, 23, 24, 25, 26, 29, 30, 31, 36.

Add: 9, 27.
An_Item_Of: 31.*
Assign: 5,* 8,* 23.*
back: 1.*
Back_Part: 18,* 19,* 20,* 22,* 24,* 25,* 26,* 27, 32, 33, 34.
Boolean: 12,* 13, 14, 30,* 32, 33.
Bounded_Queue: 3.*
Clear: 7,* 22.*
Each_Item_In_The_Queues: 31.*
empty: 1.*
Empty: 5,* 7,* 13, 28, 32, 35.
Equal: 5,* 8,* 12,* 30.*
False: 5,* 7,* 30.*
forloop: 31.*
From: 8,* 10, 23,* 24,* 25,* 26,* 28.
front: 1.*
Front: 16, 35.
Front_Part: 18,* 19,* 20,* 22,* 24,* 25,* 26,* 28, 31,* 32, 33, 34, 35.
Full: 5,* 7,* 14, 27, 33.
full: 1.*
Increment_Back_Index_Of: 27.
Increment_Front_Index_Of: 28.
Item: 9, 27.
Item_At_The_Back_Of: 27.
Item_At_The_Front_Of: 28.
Item_Type: 4,* 9, 16, 18,* 27, 28, 35.
Left: 12,* 30,* 31.*
left part: 19,* 26.*
Left_Item: 18,* 31.*
length: 1.*
Length: 5,* 7,* 8,* 15,* 23,* 24,* 25,* 30,* 31,* 32, 33, 34.
List_Index_Subtype: 18.*
List_Part: 18,* 19,* 25,* 26,* 27, 28, 31,* 35.
List_Type: 18.*
loop: 31.*
Natural: 15,* 18,* 34.
Of_The_Queue: 5,* 7,* 8,* 15,* 16, 23,* 24,* 25,* 30,* 31,* 32, 33, 34, 35.
Positive: 5,* 18.*
Queue: 5,* 7,* 13, 14, 32, 33.
queue.web: 18.*
Queue_Empty: 17, 28, 35.
Queue_Full: 17, 27.
Queue_Package_Template: 3,* 19.*
Queue_Too_Small: 8,* 17, 23.*
Queue_Type: 5,* 7,* 8,* 9, 10, 12,* 13, 14, 15,* 16, 18,* 22,* 23,* 27, 28, 30,* 32, 33, 34, 35.
Remove: 10, 28.
Right: 12,* 30,* 31.*
right part: 19,* 26.*
Right_Item: 18,* 31.*
Size: 5,* 8,* 18,* 19,* 22,* 23,* 26,* 27, 28, 31,* 33, 34.
system dependencies: 2,* 18.*
Temp: 28.
The_Left_Part_Of_The_From_List: 26.*
The_Queue: 7,* 22.*
The_Queue_Is_Not_Big_Enough: 23.*
The_Queues_Are_Different: 30,* 31.*
The_Queues_Are_Identical: 30,* 31.*
The_Queues_Have_The_Same_Length: 30.*
The_Right_Part_Of_The_From_List: 26.*
The_To_Part_Of_The_List: 25,* 26.*
There_Is_Only_One_Piece_In: 20,* 24.*
There_Is_Room_In_The_Queue_For_Another_Item: 27
There_Is_Something_At_The_Front_Of_The_Queue: 35
There_Is_Something_In_The_Queue_To_Remove: 28.
To: 8,* 9, 23,* 24,* 25,* 27.
True: 5,* 7,* 8,* 12,* 30.*

⟨ Are the two queues *Equal*? 12* ⟩ Used in section 11*.
⟨ Body of Are the two queues *Equal*? 30* ⟩ Used in section 29*.
⟨ Body of Is the queue *Empty*? 32 ⟩ Used in section 29*.
⟨ Body of Is the queue *Full*? 33 ⟩ Used in section 29*.
⟨ Body of What is the item at the *Front* of the queue? 35 ⟩ Used in section 29*.
⟨ Body of What is the *Length* of the queue? 34 ⟩ Used in section 29*.
⟨ Body of *Add* the item to the back of the queue. 27 ⟩ Used in section 21*.
⟨ Body of *Assign* the items from one queue to another. 23* ⟩ Used in section 21*.
⟨ Body of *Clear* the items from the queue. 22* ⟩ Used in section 21*.
⟨ Body of *Remove* the item from the front of the queue. 28 ⟩ Used in section 21*.
⟨ Bounded generic queue package. 2* ⟩ Used in section 1*.
⟨ Constructor bodies. 21* ⟩ Used in section 19*.
⟨ Constructor specifications. 6* ⟩ Used in section 3*.
⟨ Exceptions. 17 ⟩ Used in section 3*.
⟨ Generic parameters. 4* ⟩ Used in section 3*.
⟨ Generic queue package specification. 3* ⟩ Used in section 2*.
⟨ Is the queue *Empty*? 13 ⟩ Used in section 11*.
⟨ Is the queue *Full*? 14 ⟩ Used in section 11*.
⟨ Make the assignment. 24* ⟩ Used in section 23*.
⟨ Perform a complex assignment. 26* ⟩ Used in section 24*.
⟨ Perform a simple assignment. 25* ⟩ Used in section 24*.
⟨ Private part. 18* ⟩ Used in section 3*.
⟨ Queue package body. 19* ⟩ Used in section 2*.
⟨ Selector bodies. 29* ⟩ Used in section 19*.
⟨ Selector specifications. 11* ⟩ Used in section 3*.
⟨ Test for equality. 31* ⟩ Used in section 30*.
⟨ Visible data definitions. 5* ⟩ Used in section 3*.
⟨ What is the item at the *Front* of the queue? 16 ⟩ Used in section 11*.
⟨ What is the *Length* of the queue? 15* ⟩ Used in section 11*.
⟨ *Add* the item to the back of the queue. 9 ⟩ Used in section 6*.
⟨ *Assign* the items from one queue to another. 8* ⟩ Used in section 6*.
⟨ *Clear* the items from the queue. 7* ⟩ Used in section 6*.
⟨ *Remove* the item from the front of the queue. 10 ⟩ Used in section 6*.

Section

A Bounded Generic Queue Package 1
Generic queue package specification 3
The generic parameters 4
The visible data definitions 5
Constructor specifications 6
Clear constructor specification 7
Assign constructor specification 8
Add constructor specification 9
Remove constructor specification 10
Selector specifications 11
Equal selector specification 12
Empty selector specification 13
Full selector specification 14
Length selector specification 15
Front selector specification 16
Exception handling and propagation 17
The private part 18
Queue Package Body 19
Constructor bodies 21
Clear constructor body 22
Assign constructor body 23
Add constructor body 27
Remove constructor body 28
Selector bodies 29
Equal selector body 30
Empty selector body 32
Full selector body 33
Length selector body 34
Front selector body 35
Index 36

SHOULD DATA ABSTRACTION BE VIOLATED TO ENHANCE SOFTWARE REUSE?

S. Muralidharan
Bruce W. Weide

Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, OH 43210-1277

murali@cis.ohio-state.edu, (614) 292-8234
weide@cis.ohio-state.edu, (614) 292-1517

Abstract

Most object-oriented languages include inheritance mechanisms that are claimed to support or enhance software reuse. Two different uses of these mechanisms are usually possible in such languages: specification inheritance and implementation inheritance. Should Ada have inheritance? Here, we explore one aspect of this question by considering the use of implementation inheritance to extend the services provided by a reusable generic package. While there may be a constant-factor performance advantage to be gained by using inheritance to extend a reusable component, a variety of negative consequences (including reduced reusability in an important sense) overwhelmingly suggest that implementation inheritance should be used carefully, if at all, in the context of designing and implementing reusable software components.

1. Introduction

The effectiveness of data abstraction as an organizing principle for software, and particularly as the basis for designing reusable software components, has been widely acknowledged. However, there is no general agreement on whether data abstraction by itself is sufficient to address many of the subtler problems of reuse [Liskov 87]. Most object-oriented languages complement features for data abstraction by including a mechanism for *inheritance*. The often-heard claim for such languages is that they somehow — largely through their inheritance mechanisms — better support software reuse. Some even argue inheritance is necessary to achieve reusability [Meyer 88].

Supported in part by the National Science Foundation under grant CCR-8802312.

Two Uses of Inheritance

How does inheritance help? There are two distinct views of inheritance: *specification inheritance* and *implementation inheritance* [LaLonde 89]. The former is used to define a hierarchy of abstract program components in which some inherit abstract behaviors from others. The latter is used to define a hierarchy of concrete program components in which some inherit certain implementation details (e.g., object representations) from others. There are connections between the two hierarchies because some concrete components are implementations of other abstract components.

Using his own object-oriented language with inheritance, Eiffel, [Meyer 88] nicely illustrates this distinction. Through the use of "deferred classes" he shows how to define a generic class describing the behavior of an abstract type Stack without providing an implementation for it. He notes that inheritance along with such deferred classes is used "to capture common behaviors." This is the idea of specification inheritance.

At the same time, Eiffel is shown to support implementation inheritance. One view of this use of inheritance is as the extension of a parent module with new services beyond those provided by the parent. Code in the heir is permitted to make direct use of some concrete representation details defined in its parent. Of course, the heir may also invoke the methods (operations) provided by its parent, but inheritance is not necessary for this.

While both hierarchies may be captured by a single inheritance mechanism, as in Eiffel, the distinction in purpose is important. Specification inheritance arises from perceived similarities among components at the *behavioral* level and does not imply violation of abstraction — abstract similarities are precisely what matter. This use of inheritance simplifies client understanding of abstract reusable

components. Implementation inheritance results from apparent similarities among *representational* or *algorithmic* details and therefore does involve some violation of abstraction [Snyder 86]. But it (supposedly) improves some aspects of the implementations of reusable components.

Inheritance and Reusable Software Components

Advocates of inheritance in one form or another cite modifiability, rapid prototyping, and flexibility among its general advantages [Liskov 87, Meyer 88]. These claims seem compelling when software is developed for one-time use or where "accidental" reuse is expected, but it is not obvious they are so significant when software is built largely from components that are *designed* to be reused. It is therefore natural to ask whether the addition of an inheritance mechanism to Ada would enhance the language's already touted suitability for construction of systems built from reusable components — particularly if it is at the expense of possible violation of abstraction principles upon which the rest of the language relies.

The specific question considered in this paper is the following: *In the context of generic reusable software components in Ada, is it necessary or desirable to permit implementation inheritance?* We concentrate on an example involving extension of the functionality of a reusable component by adding a new procedure or function to those already exported by a generic package providing an encapsulated abstract data type (ADT). The arguments and conclusions from this simple illustrative example shed considerable light on the pros and cons of implementation inheritance in general, not just for Ada.

Approaches to Extending a Generic Package

There are at least two general approaches to defining a reusable software component that exports the same types and operations as another component, along with a new operation. One is to define a completely new component that provides all the services of the original one, plus the new operation. Ada has various language constructs that can support this approach under some conditions [Bardin 88]. Object-oriented languages generally support a different technique in which only the new operation is actually provided by the new component. The other operations come from the original component by virtue of inheritance. Reusable component designs based on the second approach — *extension* of the component — are considered in this paper.

One can also imagine two basic modes of extension. The first is a *layered* design. An Ada generic package providing an ADT can be extended by building code for the new operation on top of operations provided by the underlying package that provides the type. The second technique uses an implementation inheritance mechanism (added as a hypothetical Ada feature for purposes of the paper). Here, some representational details of the otherwise encapsulated type are known to the implementer of the new operation. The paper considers these alternative methods of extension in order to explore the pros and cons of implementation inheritance in this context.

Overview

The conclusion from the example in Section 2 and the arguments that follow is fuzzier than we might have liked. There is indeed the possibility for a slight (constant factor) performance advantage if extension is accomplished through implementation inheritance. The price for this potential improvement is harder to quantify but seems overwhelming. It includes greater difficulty in software testing, verification, and maintenance, and reduced reusability in an important sense. There are also potential legal problems lurking in the assumption that implementation inheritance can even be considered as a basis for software reuse. On balance, we must conclude that *if implementation inheritance were possible in Ada, it should be used sparingly if at all in the design of reusable software components.* This conclusion also holds for other languages that already permit implementation inheritance.

The paper is organized as follows. Section 2 provides an example Ada generic package that will be used in the discussion in the rest of the paper. Section 3 details two distinct approaches to extending the package in Section 2 with a new operation: one that does not violate abstraction and another that does. Section 4 discusses reuse advantages that cannot be realized if abstraction is violated while extending component functionality. In Section 5 we outline possible motivations for using implementation inheritance for extension. Section 6 has our conclusions.

2. Example: A Generic Ada Package

Figure 1 shows the specification and body of a generic package providing an encapsulated ADT called Stack. The design of the abstraction and the implementation shown are unusual in a few respects that are important to this paper.

Comments on the Specification

Every possible component type for Stacks is assumed to have associated procedures to initialize (create), finalize (destroy), and swap (exchange) values of that type. The STACKS package also provides these for uniformity, so one could instantiate a package to provide Stacks of Stacks of some type, for example. The package is also designed to permit a

```
generic
    type Item is limited private;
    with procedure Item_Initialize (e: in out Item);
    with procedure Item_Finalize (e: in out Item);
    with procedure Item_Swap (e1: in out Item; e2: in out Item);
    max_size: Natural;

package STACKS is

    type Stack is limited private;
       -- mathematical model of Stack is String of Item

    procedure Initialize (s: in out Stack);
       -- ensures s = empty string

    procedure Finalize (s: in out Stack);

    procedure Swap (s1: in out Stack; s2: in out Stack);
       -- ensures s2 = #s1 and s1 = #s2

    procedure Push (s: in out Stack; e: in out Item);
       -- requires |s| < max_size
       -- ensures s = #s o #e and Item.init(e)

    procedure Pop (s: in out Stack; e: in out Item);
       -- requires |s| > 0
       -- ensures #s = s o e

    procedure Length (s: in Stack) return Natural;
       -- ensures Length = |s|

private
    type Representation;
    type Stack is access Representation;

end STACKS;
```

Fig. 1a — Specification of a Generic Package Providing the Encapsulated Type Stack

variety of correct, efficient, plug-compatible implementations. Initialization and finalization operations should be defined for every new type because in some implementations they will be essential (e.g., to allocate and reclaim memory in a way peculiar to a particular concrete representation of the type). Swapping is used as a general method of data movement because it can be implemented to execute in constant time for any representation of any limited private type that is defined as shown in the example. This is done simply by swapping pointers to the representation data structures. Some uses of these operations are illustrated in the examples in Figures 1b, 3, and 4.

The package provides only a minimal set of operations. We call these the *primary* operations for Stacks — those operations whose implementations need to access the concrete representations of Stacks in order to be implemented efficiently. We consider an operation to be implemented *efficiently* if it suffers no more than a constant factor speed penalty by not having access to the concrete representations of its arguments. By definition, the primary operations for a type should be orthogonal in the sense that it should not be possible to implement one of them efficiently using a combination of the others. Other operations involving Stacks are called *secondary* operations. The example in the next section involves extension of STACKS by adding a secondary operation: Copy.

A client of STACKS should need to see only the package specification in order to use it. Here, embedded semi-formal comments constitute a description of the abstract behavior of Stacks and the primary operations. In the notation, "requires" and "ensures" introduce pre- and post-conditions for operations, respectively. A post-condition references the incoming value of an argument by a prefixed "#".

Notice that this package is *designed* to be reused. Reuse is not accidental or incidental, i.e., someone did not discover the package hiding in a larger system and notice it could be used elsewhere. Careful consideration was given to what operations are associated with the new type, and to which operations on Stacks are primary. Throughout this paper, when we say "reusable software component" we mean a component that was designed with reuse in mind.

```
package body STACKS is

    type Store is array (1..max_size) of Item;
    type Representation is record
          top: Natural;
          contents: Store;
    end record;

    procedure Initialize (s: in out Stack) is
    begin
       s := new Representation;
       s.top := 0;
    end;

    procedure Finalize (s: in out Stack) is
    begin
       while (s.top <> 0) loop
          Item_Finalize (s.contents(s.top));
          s.top := s.top - 1;
       end loop;
       -- storage de-allocation for s is implicit here
    end;

    procedure Swap (s1: in out Stack; s2: in out Stack) is
       temp: Stack;
    begin
       temp := s1;
       s1 := s2;
       s2 := temp;
    end;

    procedure Push (s: in out Stack; e: in out Item) is
    begin
       s.top := s.top + 1;
       Item_Initialize (s.contents(s.top));
       Item_Swap (s.contents(s.top), e);
    end;

    procedure Pop (s: in out Stack; e: in out Stack) is
    begin
       Item_Swap (s.contents(s.top), e);
       Item_Finalize (s.contents(s.top));
       s.top := s.top - 1;
    end;

    procedure Length (s: in Stack) return Natural is
    begin
       return (s.top);
    end;

end STACKS;
```

Fig. 1b — Package Body of STACKS

Comments on the Implementation

The STACKS package body is coded as shown partly to illustrate some differences between the two approaches to extension. It is somewhat atypical of components written in this style because the representation of a Stack directly uses built-in Ada types. Most such packages — especially ones that are slightly more complicated — are built on top of other reusable components and make consistent use of initialization and finalization (as in Figure 3). Ordinarily initialization of local variables is done only at the beginning of a block and finalization only at the end, which can be automatic in C++ [Strous 86] but unfortunately not in Ada.

In the STACK package body, the choice of built-in arrays as the representation is actually a complicating factor because the values in Ada arrays are not automatically initialized or finalized. The code therefore calls initialization and finalization procedures for type Item not just as the first and last actions in a block, but at other times as well, in order to maintain an implementation convention discussed in Section 4. This convention forces the code for Push, Pop, and Finalize to be more complex (and

```
generic
     type Item is limited private;
     with procedure Item_Initialize (e: in out Item);
     with procedure Item_Finalize (e: in out Item);
     with procedure Item_Swap (e1: in out Item; e2: in out Item);
     with procedure Item_Copy (e1: in out Item; e2: in out Item);

     type Stack is limited private;
     with procedure Initialize (s: in out Stack);
     with procedure Finalize (s: in out Stack);
     with procedure Swap (s1: in out Stack; s2: in out Stack);
     with procedure Push (s: in out Stack; e: in out Item);
     with procedure Pop (s: in out Stack; e: in out Item);
     with procedure Length (s: in Stack) return Natural;

package COPYSTACKS is

     procedure Copy (s1: in out Stack; s2: in out Stack);
        -- ensures s2 = #s1 and s1 = #s1

end COPYSTACKS;
```

Fig. 2 — Specification of a Generic Package Providing the Copy Operation for Stacks

in the case of Finalize, less efficient [Harms 89a]) than it would be if built-in arrays were not used. On the other hand, the convention helps illustrate one of the problems with implementation inheritance, as explained in Section 4.

3. Two Approaches to Extension

In this section we discuss two approaches to extending a generic package providing an encapsulated type. The first approach does not violate abstraction, and the second does. The first approach is possible in Ada as it stands. The second involves introduction of a hypothetical inheritance construct.

Suppose a new operation "Copy" is required for the type Stack. Figure 2 shows the specification of a generic package COPYSTACKS that provides this operation for Stacks.

Layered Approach to Software Extension: Abstraction Not Violated

Figure 3 shows a possible body of the COPYSTACKS package for the layered approach to extension. It is easy to notice that the implementation of the Copy procedure does not and cannot access the representations of its parameter Stacks, and will work for any representations of the types Stack and Item. (In fact, by including a second Stack type as a generic parameter, the Copy operation can be written so it can copy a stack with one representation to a stack with another representation.) A client using a STACKS package instance can now create an instance of the package COPYSTACKS, and can treat the type Stack as though it has another operation: Copy.

This approach for enhancing a generic package with new operations in Ada is quite general. Because of all the parameters to the package, it is also a bit cumbersome, which has led to some sentiment for permitting package parameters to generic packages [Belfou 89].

Implementation Inheritance Approach to Software Extension: Abstraction Violated

A different approach to software extension would allow the implementation of the Copy operation to directly access the representation of the Stack. Some object-oriented languages provide mechanisms to make this possible. We add a hypothetical language construct to Ada to illustrate this approach in the Ada framework. In Figure 4, the declaration "inherit package body STACKS" would allow the procedure Copy to see and use the otherwise hidden implementation details in the body of STACKS.

The implementation shown in Figure 4 violates the abstraction of the encapsulated type Stack, because a secondary operation, Copy, has been allowed to access the representational details of the Stacks it deals with.

4. Advantages of Not Violating Abstraction

This section discusses some reasons why abstraction should not be violated to enhance software reuse, i.e., why the Copy code in Figure 3 is preferable to that in Figure 4.

1. Modifications to the package body of STACKS are localized; clients will remain unaffected.

Suppose it is determined that the implementation of the STACKS package is incorrect or inefficient, and must be modified. The layered implementation of the secondary operation Copy in Figure 3 will not be

```
package body COPYSTACKS is

    procedure Copy (s1: in out Stack; s2: in out Stack) is
       x, y: Item;
       temp, garbage: Stack;
    begin

       -- initialize the values of the local variables
       Item_Initialize (x);
       Item_Initialize (y);
       Initialize (temp);
       Initialize (garbage);

       -- empty out destination of copy, i.e., s2
       Swap (s2, garbage);

       -- transfer the contents of s1 to temp (in reverse order)
       -- let s be the value of s1 and let t be the value of temp at this point
       -- loop invariant:  s1 o reverse(temp) = s o reverse(t)
       while (Length (s1) <> 0) loop
          Pop (s1, x);
          Push (temp, x);
       end loop;

       -- transfer the contents back from temp to s1 and s2
       -- let s' be the value of s1 and let t' be the value of temp at this point
       -- loop invariant:  s1 = s2 and s1 o reverse(temp) = s' o reverse(t')
       while (Length (temp) <> 0) loop
          Pop (temp, x);
          Item_Copy (x, y);
          Push (s1, x);
          Push (s2, y);
       end loop;

       -- finalize the local variables
       Item_Finalize (x);
       Item_Finalize (y);
       Finalize (temp);
       Finalize (garbage);

    end Copy;
end COPYSTACKS;
```

Fig. 3 — A Layered Implementation of the Copy Operation for Stacks

affected because it depends only on the *specification* of STACKS. However, if the implementation of Copy is allowed to depend on the details of the STACKS package body as in Figure 4, then it should also be re-coded (and re-compiled and re-validated). In large software systems, it is critical that the effects of modifications be isolated and restricted. Such developmental independence is possible only when client operations rely on just the specification of an abstraction and not on any particular implementation.

In addition, development and especially modification of a client package based on the source code of another package is bound to be error-prone, since the client programmer has to understand obscure implementation details of the inherited package. In this case, for instance, there is a convention in the STACKS package body that for every Stack s, `s.contents[s.top+1..max_size]` have not been initialized or, if they have ever been used, their values have been finalized. The COPYSTACKS code of Figure 4 must understand this convention. To see why, suppose a different convention is used: `s.contents[s.top+1..max_size]` contain initialized Items. The loop in Figure 4 should now finalize `s2.contents[count]` before copying into it. In other words, the Copy code of Figure 4 depends not only on the representation of Stacks, but on (unstated and implicit) conventions about how the representation is used by the primary operations. This sort of coupling is a maintenance nightmare.

2. A single implementation of a secondary operation can be used with different representations of the types of its parameters.

The possibility of having many implementations of the same abstraction and the ability for a client to simultaneously use them in a program are impor-

```
package body COPYSTACKS is

    inherit package body STACKS;

    procedure Copy (s1: in out Stack; s2: in out Stack) is
       garbage: Stack;
       count: integer;
    begin

       -- initialize the values of the local variables
       Initialize (garbage);
       count := 0;

       -- empty out destination of copy, i.e., s2
       Swap (s2, garbage);

       -- copy the contents of s1 to s2
       -- let s be the value of s1 at this point
       -- invariant:
       --     s1.contents(1..count) = s.contents(1..count) and
       --     s1.contents(1..count) = s2.contents(1..count) and
       --     s1.top = s.top
       while (count <> s1.top) loop
          count := count + 1;
          Item_Copy (s1.contents(count), s2.contents(count));
       end loop;
       s2.top = s1.top;

       -- finalize the local variables
       Finalize (garbage);

    end Copy;

end COPYSTACKS;
```

Fig. 4 — An Implementation of the Copy Operation Using Implementation Inheritance

tant for large software projects [Meyer 88]. Many implementations with different performance or hardware characteristics should be possible for the same specification, and a client should have the flexibility to choose one based on his/her requirements. For example, it is possible to construct another package providing the type Stack with the same specification as shown in Figure 1a, but with a different body. This implementation might represent the type Stack differently (e.g., using a linked list instead of an array). Another example would be different sorting algorithms (e.g., quicksort, heapsort) for the same sorting abstraction. We have shown elsewhere how observing certain Ada package design guidelines makes it possible for clients to use such packages interchangeably without having to modify or re-validate client programs [Murali 89b].

Consider the implementation of the Copy operation shown in Figure 3. It will work with another package that represents the type Stack differently as long as the interface of this package (both structural and behavioral) is the same as shown in Figure 1a. This is possible only because the implementation of the Copy operation is independent of the representation of the Stack type. If, however, the Copy operation depends on the representational details of the type Stack as in Figure 4, then it must be re-written. In this sense, not violating abstraction leads to enhanced software reuse.

This argument also suggests that the primary operations on an encapsulated type should be kept to a minimum. When this is done, the same secondary operation will work for many different representations of the type. Of course, a smaller set of primary operations also makes it easier to understand an abstraction and easier to implement it in a different way. The reader may view this remark as so obvious as to go without saying. But many published designs of "reusable components" do not observe the principle and do not recognize which operations are primary and which are secondary [Meyer 88, Isner 89].

3. Verification or validation of clients is independent of representation details.

If the implementation of a client procedure depends only on the specification of an encapsulated type and not its representation or other implementation details, the client procedure can be verified without regard for the implementation of the encapsulated type. The crucial idea here is to model a program type with a mathematical type and then to reason about a client program as though it were dealing purely with mathematical variables. In the exam-

ple here, when validating the Copy code of Figure 3, a Stack variable is treated as though it were a mathematical string; see the package specification in Figure 1a and the loop invariants in Figure 3. [Krone 88] and [Harms 89b] discuss various reasons why this is important for both formal and informal reasoning about client correctness. In Figure 4, on the other hand, the assertions necessarily refer to representational details of Stacks. Not just the code but even the reasoning about correctness of Copy is reusable in the layered approach. Neither is reusable with implementation inheritance.

4. Source code for the package body may not be physically available, even if it could be accessed with a new language construct.

A fundamental problem with reuse of implementation details (or reuse of source code in general) is that the implementation details may not be available in the first place. The state of the art is that reusable software components are often presented in source form, with copious documentation [Berard 87]. When a mature reusable software component industry comes into existence, however, only the specification and object code that implements it are likely to be sold. The trend in other engineering disciplines is one reason for this belief; for example, in the electronics industry only a sealed chip is marketed, not a mask.

There is a powerful non-technical reason for this conclusion, too. The internal data structures and algorithms used by a component developer are not patentable under current law. Therefore, it has become common for companies to register copyrights for object code versions of their components and to maintain that the source code contains separately protected trade secrets [Tomiji 87]. To protect their investment, reusable component developers will do well to keep their source code secret if that is technically feasible — which is the case with Ada.

5. Suggested Reasons for Allowing Implementation Inheritance

This section explores the other side of the coin: some possible motivations for including language mechanisms to reuse implementation details of an abstraction. It is essential to understand that constructs for implementation inheritance are primarily aimed at allowing a client to use that information but not to modify it. In Figure 4, the package body of COPYSTACKS which (hypothetically) inherits the body of STACKS can only *use* the information in it to implement the operation Copy; the representation of the type Stack and the code for the primary operations cannot be *changed* by the inheritor. If modifications were to be allowed, then there would seem to be no need for a new language construct. One might as well create a copy of the original body and edit it.

Why is it ever necessary to inherit the implementation details of an abstraction when the abstraction has to be extended with a new operation? Three possible answers are discussed below.

1. It may not be possible to write the new operation using the primary operations on the abstraction; the component might not provide a "complete" set of primary operations.

While this argument may be valid when software is developed for one-time use, it is less persuasive in the context of reusable software. A reusable component is typically *designed* with much forethought, and must provide at least a minimal set of primary operations needed for useful manipulations. (We are not talking about accidental reuse here, but components designed for reuse.) Since any additional operations should be implementable as secondary operations using the primary operations, enhancements to functionality should be possible without access to the implementation details.

When it is the case that an abstraction was not designed properly, changes to the specification will often involve changing the representations and/or code for some or all of the operations. Inheritance of implementation details using a construct that allows only additions and not modifications will be useless in this case. As noted earlier, no new language mechanism is needed when existing code has to be changed.

2. Only inefficient implementation of the new operation is possible using the primary operations; if the implementation details were available, performance improvement (in order-of-magnitude) might be possible.

This situation is indeed possible, but implementation inheritance generally will not solve the problem if it arises. For example, execution of Copy as a secondary operation as in Figure 3 takes time proportional to the number of elements on the stack. It is possible to write a clever constant-time implementation of the Copy operation if Copy is a primary operation on Stacks [Weide 86]. However, to write such an implementation of Copy, other primary operations on the type Stack also must be re-coded.

The only way to do this is to rewrite a complete implementation of an enhanced version of the STACKS package with the additional operation Copy. Implementation inheritance to access the representational details of type Stack as in Figure 4 will not solve the problem because other operations on Stacks have to be re-coded to make Copy more efficient by more than a constant factor. We know of no example of a well-designed reusable component — one with a complete but not a redundant set of primary operations — where implementation inheritance will permit more than a constant factor speedup of secondary operations over that possible with a layered extension.

3. Even if it is possible to write the new operation efficiently using only the primary operations, the (constant factor) overhead of procedure calls in this operation's code can be avoided if implementation details are directly accessible.

The Copy operation implemented as a secondary operation does indeed have the overhead of some extra procedure calls when compared with the Copy implementation that directly accesses the underlying representation of Stack. The order of the time complexity of both the algorithms in Figure 3 and 4 is the same. The difference in efficiency is only a constant factor. But it is possible that this difference is crucial for some applications.

Even in the case when the constant-factor difference is important enough to matter, though, the efficiency advantage when balanced against the disadvantages of using implementation inheritance leads us to the conclusion that implementation inheritance is *generally not justifiable* in the context of building or extending reusable components. Most object-oriented languages with inheritance involve some form of run-time computation of type information (e.g., dynamic binding in Eiffel [Meyer 88]) in order to take complete advantage of inheritance, and this inefficiency will at least partially offset any constant factor speed-up advantages it might offer for extension. More importantly, rapid advances in hardware technology and parallel computing tend to overcome constant run-time overheads in rather short order. The disadvantages of using implementation inheritance — greater coupling between modules that don't need to be coupled, and probably more costly maintenance as a result — will not be ameliorated by time, only made more significant.

6. Summary and Conclusion

We have discussed two methods to extend an existing reusable component with a new operation. One method — a layered approach — does not violate abstraction. The other — implementation inheritance — permits code for the new operation to access the implementation details of the reused component. We have discussed the advantages of never violating abstraction, and some of the claimed benefits for implementation inheritance. The arguments seem to be overwhelmingly in favor of not violating abstraction when software reuse issues are considered.

The conclusion demonstrated here for one use of inheritance (extension) for a particular class of reusable components (generic packages providing ADTs) does not mean language constructs for implementation inheritance are necessarily a bad idea. But they probably should be used sparingly if at all, as the examples here illustrate.

Acknowledgments

We thank the members of the Reusable Software Research Group for their comments on the content of this paper, and Steve Edwards for discussions about some peculiarities of Ada.

References

[Balfour 89] Balfour, B., "Issues Related to the Creation of Classes in Ada," presented at *The Informal Workshop on Ada Support for Object-Oriented Programming*, Woburn, MA, September 1989.

[Bardin 88] Bardin, B., and Thompson, C., "Using the Re-Export Paradigm to Build Composable Ada Software Components," *Ada Letters*, vol. 8, no. 2, March/April 1988, pp. 39-54.

[Berard 87] Berard, E. V., *Creating Reusable Ada Software*, EVB Software Engineering, Inc., Frederick, MD, 1987.

[DoD 83] Department of Defense, Ada Joint Program Office, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, Washington, DC, Government Printing Office, 1983.

[Harms 89a] Harms, D.E., and Weide, B.W., *Efficient Initialization and Finalization of Data Structures: Why and How*, Department of Computer and Information Science, The Ohio State University, Columbus, OH, March 1989, OSU-CISRC-3/89-TR11.

[Harms 89b] Harms, D. E., and Weide, B. W., *Types, Copying, and Swapping: Their Influences on the Design of Reusable Software Components*, Department of Computer and Information Science, The Ohio State University, Columbus, OH, March 1989, OSU-CISRC-3/89-TR13.

[Isner 89] Isner, J., et al., *C++ Libraries: Manual and Tutorial Release 2*, AT&T Bell Laboratories, Warren, NJ, 1989.

[Krone 88] Krone, J., *The Role of Verification in Software Reusability*, Ph. D. Dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, August 1988.

[LaLonde 89] LaLonde, W. R., "Designing Families of Data Types Using Exemplars," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 2, April 1989, pp. 212-248.

[Liskov 87] Liskov, B. H., "Data Abstraction and Hierarchy," *Addendum to the Proceedings of Object-Oriented Programming: Systems, Languages, and Applications*, Orlando, FL, October 1987, pp. 17-34.

[Meyer 88] Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, Cambridge, UK, 1988.

[Murali 89a] Muralidharan, S., "On Inclusion of the Private Part in Ada Package Specifications," *Proceedings of the Seventh National Conference on Ada Technology*, Atlantic City, NJ, ANCOST, Inc., March 1989, pp. 188-192.

[Murali 89b] Muralidharan, S., and Weide, B. W., "Achieving Multiple-Implementation Extensibility with Ada Generic Packages," presented at *The Informal Workshop on Ada Support for Object-Oriented Programming*, Woburn, MA, September 1989.

[Snyder 86] Snyder, A., "Encapsulation and Inheritance in Object-Oriented Systems," *Conference Proceedings of Object-Oriented Programming: Systems, Languages, and Applications*, Portland, OR, September 1986, pp. 38-45.

[Strous 86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

[Tomiji 87] Tomijima, A. U., "How Japan's Recently Amended Copyright Law Affects Software," *IEEE Software 4*, 1, January 1987, pp. 17-21.

[Weide 86] Weide, B. W., *A New ADT and Its Applications in Implementing "Linked" Structures*, Department of Computer and Information Science, The Ohio State University, Columbus, OH, January 1986, OSU-CISRC-TR-86-3.



Sitaraman Muralidharan is a Ph. D. student and a research assistant in the Department of Computer and Information Science at The Ohio State University. His research interests are in distributed systems, programming languages, and software engineering. Muralidharan got his B.E. (Honors) Degree in Electrical and Electronics Engineering from the Regional Engineering College, Tiruchi, University of Madras, India in 1983. He got his M.E. (Distinction) Degree in Computer Science and Automation from the Indian Institute of Science, Bangalore, India in 1984. His Internet address is murali@cis.ohio-state.edu, and his postal address is Department of Computer and Information Science, 2036 Neil Avenue Mall, Columbus, Ohio 43210.



Bruce W. Weide is Associate Professor of Computer and Information Science at The Ohio State University. He holds a Ph.D. in Computer Science from Carnegie Mellon University and B.S.E.E. from the University of Toledo. Weide's current research interest is in addressing technical problems related to practical reusable software components. His Internet address is weide@cis.ohio-state.edu, and his postal address is the same as shown above.

MEASUREMENT OF ADA THROUGHOUT THE SOFTWARE DEVELOPMENT LIFE CYCLE

Bryan L. Chappell

Software Technology, Inc.
5904 Richmond Highway, Suite 610
Alexandria, Virginia 22303

Sallie Henry
Kevin A. Mayo
Computer Science Department
Virginia Tech
Blacksburg, Virginia 24060

Summary

Quality enhancement has now become a major factor in software production. Software metrics have demonstrated their ability to predict source code complexity at design time and to predict maintainability of a software system from the source code. Obviously metrics can assist software developers in the enhancement of quality. Tools which automatically generate metrics for Ada are increasing in popularity. This paper describes an existing tool which produces software metrics for Ada that may be used throughout the software development life cycle. This tool, while calculating established metrics, also calculates a new structure metric that is designed to capture communication interface complexity. Measuring designs written using Ada as a PDL allows designers early feedback on possible problem areas in addition to giving direction on testing strategies.

This research was funded by Software Productivity Consortium (SPC), and the Center for Innovative Technology (CIT).

KEYWORDS: Software Design, Software Design Tools, Software Metrics, Software Development Life Cycle, PDL Measurement.

Introduction

"You can't control what you can't measure" [DEMAT82], [YOURE89]. Interest in controlling the software development process has always been a concern for management but more recently it has received attention from all levels of software development personnel. The software crisis has made software developers more aware of the need to measure (or even better, to predict) the quality of their software product. The DOD has also taken steps to assure the quality of their contractor's source code. Several tools have been developed to generate metrics for various languages. Software metric researchers have realized that more time, effort, and money can be saved if the measurement is performed throughout the software development life cycle, especially during the design phase. With the growing popularity and use of the Ada programming language, the next extension must be to analyze Ada.

Several studies have been done that support the use of measurement throughout the software life cycle. Ramamoorthy et al. [RAMAC85] suggest the use of metrics throughout the software life cycle, emphasizing that different metrics need to be used during different phases of software development. Kafura and Canning [KAFUD85] suggest the use of structure metrics in the design and development phase. Metrics applied in the design of a software system can be used to determine the quality of a design, and can also be used to compare different designs for the same requirement specifications [HENRS89], [HENRS90b], [HENRS90c].

Using metrics during the testing of a software product helps to determine the reliability of the software. Lew, Dillon, and Forward [LEWK88] used software metrics to help improve software reliability. The software metrics were able to quantify the design and provide a guide for designing reliable software.

As previously noted, maintenance efforts cost the most of any part of the software development life cycle. Kafura and Reddy [KAFUD87] used seven metrics in a software maintenance study. Among other results, they found that the growth in system complexity that was determined by the metric measurements agreed with the general character of the maintenance tasks that were performed. Other research shows that metrics applied during the coding phase can predict the maintainability of the software product [WAKES88], [LEWIJ89].

It is obvious from the preceding paragraphs that software metrics can be a useful guide in the life of a software product. In order to facilitate the use of software metrics, an automatable metric analysis tool must be developed. Gilb, in [GILBT77], stresses the importance of an automated metric analysis tool, citing a TRW study where metrics-guided testing is half the cost of conventional testing.

This research produced a software metric tool for Ada that can be used throughout the entire software development life cycle. Some of the metrics produced are based on the structure of the code, which can be seen early in the software design. This aspect, coupled with the encouragement by the Ada community to use Ada as a PDL, provides for the gathering of metrics at the design stage with the metric analysis tool.

There is a benefit of having a design language that is a subset of a programming language while also having design language features. Many researchers have proposed the use of Ada as a program design language (PDL) [CHASA82], [GABBE83], [SAMMJ82]. Each of the military services under the Department of Defense is currently issuing requests for proposals (RFPs) which require the use of an Ada-based PDL [HOWEB84].

The Software Metric Analyzer that was developed as part of this research can be used to analyze Ada code written with any level of refinement. Therefore, the analyzer supports the use of Ada as a PDL. Design quality can be determined and different designs for the same requirement specifications can be compared. Henry and Selig [HENRS90b] have been able to predict the complexity of the resultant source code from measurement of designs written in an Ada-like PDL. Many metrics are calculated by the Software Metric Analyzer, including the

code and structure metrics that are discussed in the next section. The structure metrics produce more relevant information than code metrics when analyzing design code. This is because most of the details of the design code concerns the system's hierarchy and calling structure. The Software Metric Analyzer should be of great benefit as a life cycle support tool.

Metrics for Ada

Once it is realized that software needs to be measured, it is important that the measuring process be automatable. It would be unreasonable to consider measuring software manually due to the size of most software systems. Also, automatable measuring is more accurate and consistent. Therefore, this research concentrated on only those metrics that are quantitative and automatable. The available quantitative metrics can be grouped into three rather broad categories. These categories are *code metrics*, *structure metrics* and *hybrid metrics*. Code metrics produce a count of some feature of the source code. Structure metrics attempt to measure the logic and interconnectivity of the source code. Hybrid metrics combine one or more code metrics with one or more structure metrics. Code metrics are easier to calculate than the structure and hybrid metrics, but since the structure and hybrid metrics measure different aspects of the original source code, they are very important and worth the effort of computation [HENRS81a], [HENRS90b].

Code Metrics

Code metrics produce a count of some feature of the source code. The code metrics that this study uses are lines of code [CONTS86], Halstead's Software Science [HALSM77], and McCabe's Cyclomatic Complexity [MCCAT76].

Of the many definitions of lines of code (LOC) that are discussed in the literature, the definition that is used in our research is from Conte et al.:

> A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.

Halstead's Software Science Indicators are well-known and used frequently. Halstead's metrics are based on token counts of operators and operands. Therefore, part of this research involved defining what the operators and operands are for Ada. Of the many metrics proposed by Halstead, this research calculates his program length (N), program volume (V) and effort (E).

McCabe's Cyclomatic Complexity (CC) is based on the logic structure of subprograms and has its basis in graph theory. His metric measures the number of basic paths through a subprogram. McCabe's metric is calculated by using the shortcut he presented, i.e., counting the number of simple conditions and adding one.

Structure Metrics

Structure metrics attempt to measure the logic and control flow of a program. The structure metrics that this study incorporates are Henry and Kafura's Information Flow metric [HENRS81b] and interface metrics [MAYOK89].

Henry and Kafura's Information Flow metric (INFO) is based on the information flow connections between a subprogram and its environment. These connections can be in the form of subprogram invocations and accesses to "global" data structures, i.e., those available through a common package specification.

Interface metrics attempt to more fully measure the interfaces among subprograms by recognizing several facts about software: variables of different types have a different complexity inherent in their types (e.g., an integer variable is less complex than a record variable), operations vary in complexity (e.g., addition is less complex than division), and conditionals vary in complexity (e.g., a FOR loop is different in complexity than a WHILE loop). Also, how variables are referenced (either they are read from or written to, or both) contributes to complexity. All of these facts combine to yield a better measure of the complexity of subprogram invocations.

Interface metrics were defined with the Ada programming language in mind. With the possibilities available through the use of packages and generic units, program complexity needs to be viewed in a new way [GANNJ86], [SHATS88].

Hybrid Metrics

Hybrid metrics consist of one or more code metrics combined with one or more structure metrics. The hybrid metrics that this research uses are the hybrid form of Henry and Kafura's Information Flow metric [HENRS81b] and Woodfield's Review Complexity (RC) [WOODS80].

Henry and Kafura's Information Flow metric in its hybrid form is a multiplication of any code metric and the structure form of the information flow metric. Woodfield's metric is associated with programmer time. He states that certain subprograms must be reviewed by a programmer several times due to their interconnections within the program. Woodfield defines three types of connections between subprograms: subprogram invocation, modification of global data, and assumptions made in one subprogram that are used in another subprogram. Woodfield's Review Complexity uses Halstead's effort (E) as the code metric, but suggests that any code metric could be used.

The Software Metric Analyzer

The software metric research group at Virginia Polytechnic Institute has developed a Software Metric Analyzer. A diagram of the Software Metric Analyzer is in Figure 1 (at the end of this paper).

The Software Metric Analyzer can be run using Ada code written at any level of refinement. Therefore, the analyzer supports the use of Ada as a PDL. This enables designers to analyze their design before it is completely implemented. Hopefully, design errors can be caught during the design and will not be allowed to propagate through to the implementation. There is a need for life cycle support in an Ada environment. The Software Metric Analyzer should be of great benefit as a life cycle support tool.

Phase 1

The main thrust of this research effort was in developing the Ada Translator for use in calculating the metrics discussed in the preceding section. The Ada Translator takes as input Ada source code or Ada PDL, as well as information on the pre-defined language environment (e.g., the package STANDARD). The Ada Translator is written in Lex, YACC, and C

[LESKM75], [JOHNS75]. The outputs from the translator are code metrics and relation language code. Phase 1 is the only phase to see the source code, thus, phase 1 generates the code metrics which rely on source code token counts.

The relation language that is produced by phase 1 is our own, in-house object language [HENRS88]. The Ada source code is translated into the equivalent relation language source which contains all the information that is necessary to generate structure and hybrid metrics.

Also, it is not necessary for the software producer to worry about loosing any proprietary information embedded within the Ada source code. The Ada Translator could be run at the producer's location and the code metrics file and the relation language code file could be processed afterwards at Virginia Polytechnic Institute. The translation from Ada to relation language, while producing sufficient information to generate structure and hybrid metrics, removes enough detail that there is no worry of loosing any proprietary information [HENRS88].

The Ada Translator keeps a project library, much like an Ada compiler does. This information is maintained solely by the translator for use in processing later compilation units. This helps minimize problems such as name resolution and symbol table processing across packages.

Phase 2

From the relation language code, interface metrics are calculated in their entirety. Also, relations are computed. Relations are grouped by subprogram and are used in phase 3. A set of relations is computed for each variable within a subprogram. Relations identify flow of control and flow of information throughout the system.

Phase 3

The third phase of the Software Metric Analyzer is where all the information produced thus far is accumulated and assembled into a format accessible to the user. The code metrics from phase 1 and the interface metrics from phase 2 are included in this final phase. With the relations generated in phase 2, the Metric Generator calculates structure metrics. The code metrics, together with these newly calculated structure metrics, allow the hybrid metrics to be calculated. Therefore, all metric values are available to the user from this third and final phase of the Software Metric Analyzer.

The analyzer has two different methods of grouping units of analysis. Modules can be defined in order to group a package together with all of its nested subprograms and packages. In this way, complexity analysis can be performed on larger sections of code. The other grouping mechanism of the analyzer is more flexible and allows any combination of units of analysis to be grouped together, regardless of their lexical nesting level.

The Data

Software Productivity Solutions (SPS) is a small company (less than 40 employees) interested in producing quality software products. Located in Melbourne, Florida, most of their contracts are from the Department of Defense. The data used in the analysis of the Software Metric Generator is an Ada-based Design Language (ADL) Processor produced by SPS. The system was donated to the Virginia Tech Metrics Research Group for use in the validation of the Software Metric Analyzer. The ADL programming system provides for syntax checking and limited semantic checking of ADL source files. Also, it generates a variety of data dictionary and cross reference reports and has an on-line help facility. The system consists of four subsystems: support, database, report, and analyzer. In total, this system has 83,799 textual lines of code and 5,355 subprograms and packages. The support section contains 6,702 lines of code. The database section contains 43,019 lines of code. The report section contains 3,874 lines of code. The analyzer section contains 30,204 lines of code.

Ada Metrics Results

The Software Metric Analyzer processed the ADL Processor's source code and the resultant code, structure, and hybrid metrics were analyzed. All correlations in this paper are Pearson correlations.

After the data was gathered, the metrics had to be validated. Usually this is done by using error history analysis or development time information but none of this was available. Therefore, the validation was performed by a subjective hand inspection of some of the subprograms that were flagged as potential problem areas.

Those subprograms perceived as potential problem areas are the outliers. A subprogram is considered to be an outlier if *one* or more of its metric values falls in the top 1% of that metric's values. Since the data is clustered towards small complexity values, only the largest 1% of the data was removed. The range of metric values would definitely vary from system to system, therefore the definition of an outlier would need to vary as well.

The subprograms that were inspected were those that have high complexity values for all of the metric measurements. Table 1 contains information concerning the overlap of the outlier data among the metrics. See Table 2 for some summary statistics of the data and Table 3 for the intermetric correlations of the data with the outliers removed.

It can be seen from the information in Table 1 that 76% of the outliers are in the top 1% of *only one* metric's value range. Most of these subprograms probably should not be considered problem areas since only one metric considers them to have a high complexity. This is a good example of showing the need to use more than one metric in analyzing a programming system.

On the other end of the spectrum, 1.7% of the outliers are considered outliers based on *all* of their metric values. Surely these subprograms are the ones that should be examined carefully, tested rigorously or perhaps be redesigned.

The intermetric correlations of the data in Table 3 are consistent with previous results for languages other than Ada[HENRS81a], [LEWIJ89], [HENRS90b]. The code metrics correlate highly, which seems to agree with previous metric studies. McCabe's Cyclomatic Complexity metric does not correlate to the other code metrics as well as all of the other code metrics correlate to one another. Part of the reason for this could be the fact that package definitions are not required to have a body of code with them.

Table 1. Overlap of Outliers

Number of Metrics In Overlap	Percentage of the Outlier Data
1	76.0
2	5.5
3	2.5
4	2.9
5	5.6
6	5.8
7	1.7

Table 2. Summary Statistics of the Data

Metric	Mean	STD Dev	Min	Max
LOC	16.3	15	1	150
N	83.0	83	4	645
V	429.2	518	8	4055
E	11152.5	19049	12	184177
CC	2.4	2	1	18
RC	11506.6	19568	12	182989
INFO	12510577.0	67495181	1	900486748

Table 3. Intermetric Correlations of the Data

Metric	LOC	N	V	E	CC	RC
LOC						
N	0.947					
V	0.942	0.997				
E	0.916	0.957	0.965			
CC	0.668	0.726	0.717	0.727		
RC	0.904	0.940	0.946	0.968	0.700	
INFO	0.065	0.090	0.098	0.072	-0.007	0.106

Woodfield's Review Complexity correlates very high (0.968) to the code metric used (Halstead's *E*). This is because most of the units of analysis have a Woodfield *fan-in* value that is less than or equal to 3. This yields a Review Complexity value equal to Halstead's *E*. Packages could be part of the reason for Woodfield's *fan-in* not exceeding 3 that often. A package with only a package specification will never have a Woodfield *fan-in* that exceeds 3 because packages can not be called, nor can they modify a data structure that is declared external to them.

It can be seen in Table 3 that Henry and Kafura's Information Flow metric does not correlate at all to the other metrics. This is because the Information Flow metric measures the flow of information between units of analysis and does not measure size. The Information Flow metric is probably a better indicator of complexity than the size metrics, because although size contributes to complexity, module interactions make programs difficult to write, test and understand. These interactions are also the source of mary errors [HENRS81a].

It should also be noted that since the Information Flow metric does not depend on size but rather on the flow of information, it can produce meaningful results from analyzing a design written in Ada or an Ada-like PDL. This is because designs written in a PDL are broken down by procedures and contain calls to other procedures. With just this calling structure, the Information Flow metric can compute a meaningful complexity measure [HENRS90b].

Interface Complexity Results

The findings from the interface complexity study are promising. There were 20,975 observations of intra-package communication. These complexities were reduced to form eight interface complexity measures associated with each procedure. These measures are:

Table 4: Interface Complexity Measures

AccN	The number of accesses from the current procedure to other procedures.
Acc	The measure representing the environment complexity at the time of the calls.
P1	The measure capturing the effect of parameter variable types on communication.
P2	The measure showing the effect of modification on the parameters.
FAccN	The number of accesses into the current procedure from external sources.
FAcc	The measure representing the environment within external procedures during calls to the current procedure.
FP1	The measure showing the effect of parameter variable types on communication within external calling procedures.
FP2	The measure showing the effect of modification to the parameters that are being sent into the current procedure from within external procedures.

These measures correlate with the established metrics in varying degrees. However, it is the function of these variables that interests this study. The following four functions were examined:

Table 5: Functions of Interface Complexity Measures

F1	(Acc * P1) + (FAcc * FP1)
F2	(Acc * P1) * (FAcc * FP1)
F3	(Acc * P2) + (FAcc * FP2)
F4	(Acc * P2) * (FAcc * FP2)

Of these functions, F2 and F4 correlate very high with the Henry-Kafura Information Flow. Therefore, why reinvent the wheel? However, function F1 showed moderate correlations to both code and hybrid metrics. This is interesting due to the fact that F1 is measuring factors that are similar to both code and hybrid metrics.

Table 6: Correlations among Functions and Established Metrics

Function	LOC	E	INFO
F1	0.28539	0.47745	0.50984
F2	0.03311	0.02996	0.91971
F3	0.34048	0.57726	0.06788
F4	0.03270	0.02950	0.91970

The lack of an error history compromised this study, and to compensate for this deficiency a systematic subjective examination of the code was undertaken. This involved rank ordering the eight interface complexity measures, and pulling the subprograms associated with these highest valued measures.

The metrics correctly measured their intended purposes. Furthermore, this inspection showed that these measures could be used to detect areas of the code (or design) with potential problems. These metrics also pointed out Ada-specific characteristics, such as a package of definitions. This package, while having no code, is full of declarations for variables. Therefore, any reference to one of these variables is a communication to the package. These metrics proved very useful in the detection of questionable programming styles, including data hiding.

Used at the design stage, these metrics could predict areas of abuse, and areas in need of redesign. If used at implementation time, these metrics could identify "hot spots" within the code, i.e., areas of probable errors in interfaces. If used at testing or integration points then these metrics could help to plan a strategy for interface testing. Finally, if used at maintenance these measures could help to effect quality maintenance procedures.

Conclusions

We believe that this metric tool can assist software developers in their effort of quality enhancement. If metrics are applied early in the software life cycle, potential problem areas may be detected and corrected early. Correcting problems early within the design and implementation of software systems can result in a great cost savings. By having designers and programmers use this tool to augment their design, coding and testing process, their final software product will be more robust. This tool can make a large contribution to the software development community towards their goal of achieving a quality software product.

InterNet Contact: henry@vtodie.cs.vt.edu

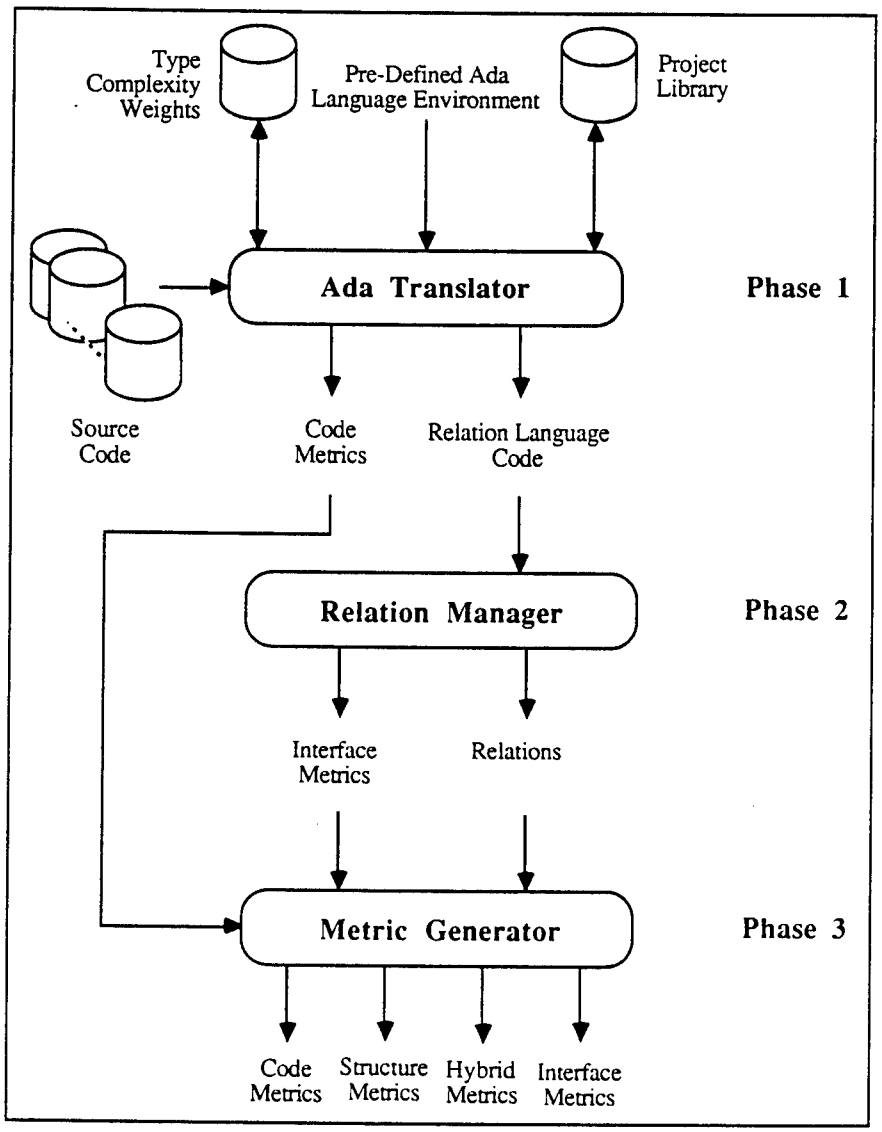


Figure 1. The Software Metric Analyzer

Bibliography

[CHASA82] Chase, Anna I. and Mark S. Gerhardt, "The Case for Full Ada as a Design Language," *Ada Letters*, Vol. 2, No. 3, Nov./Dec. 1982, pp. 51-59.

[CONTS86] Conte, S. D., H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*, Reading, MA: The Benjamin/Cummings Publishing Company, Inc., 1986.

[DEMAT82] DeMarco, Tom, *Controlling Software Projects: Management, Measurement & Estimation*, Englewood Cliffs, NJ: Yourdon Press, 1982.

[GABBE83] Gabber, Eran, "The Middle Way Approach for Ada Based PDL Syntax," *Ada Letters*, Vol. 2, No. 4, Jan./Feb. 1983, pp. 64-67.

[GANNJ86] Gannon, J. D., E. E. Katz, and V. R. Basili, "Metrics for Ada Packages: An Initial Study," *Communications of the ACM*, Vol. 29, No. 7, July 1986, pp. 616-623.

[GILBT77] Gilb, Tom, *Software Metrics*, Cambridge, MA: Winthrop Publishers, Inc., 1977.

[HALSM77] Halstead, Maurice H., *Elements of Software Science*, New York: Elsevier North-Holland, Inc., 1977.

[HENRS81a] Henry, Sallie, Dennis Kafura and Kathy Harris, "On the Relationships Among Three Software Metrics," *Performance Evaluation Review*, Vol. 10, No. 1, Spring 1981, pp. 81-88.

[HENRS81b] Henry, Sallie and Dennis Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, Sept. 1981, pp. 510-518.

[HENRS88] Henry, Sallie, "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers; or, Do You Recognize This Well-Known Algorithm?," *The Journal of Systems and Software*, Vol. 8, No. 1, Jan. 1988, pp. 3-11.

[HENRS89] Henry, Sallie, and Roger Goff, "Complexity Measurement of a Graphical Programming Language," to appear in *Software–Practice and Experience*, 1989.

[HENRS90b] Henry, Sallie, and Calvin Selig, "Design Metrics which Predict Source Code Quality," to appear in *IEEE Software*, 1990.

[HENRS90c] Henry, Sallie, and Roger Goff, "Comparison of a Graphical and a Textual Design Language Using Software Quality Metrics," to appear in *The Journal of Systems and Software*, 1990.

[HOWEB84] Howe, Bob and Jean E. Sammet, "Ada Policy Material from Dallas Meeting in October 1983," *Ada Letters*, Vol. 3, No. 4, Jan./Feb. 1984, pp. 131-136.

[JOHNS75] Johnson, Stephen C., "YACC: Yet Another Compiler-Compiler," *Computing Science Technical Report, No. 32*, Bell Laboratories, Murray Hill, NJ, 1975.

[KAFUD85] Kafura, Dennis and James Canning, "A Validation of Software Metrics Using Many Metrics and Two Resources," *Proceedings: 8th International Conference on Software Engineering*, Aug. 1985, pp. 378-385.

[KAFUD87] Kafura, Dennis and Geereddy R. Reddy, "The Use of Software Complexity Metrics in Software Maintenance," *IEEE Transactions on Software Engineering*, Vol. 13, No. 3, Mar. 1987, pp. 335-343.

[LESKM75] Lesk, M. E. and E. Schmidt, "Lex—A Lexical Analyzer Generator," *Computing Science Technical Report, No. 39*, Bell Laboratories, Murray Hill, NJ, 1975.

[LEWK88] Lew, Ken S., Tharam S. Dillon, and Kevin E. Forward, "Software Complexity and Its Impact on Software Reliability," *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, Nov. 1988, pp. 1645-1655.

[LEWIJ89] Lewis, John, and Sallie Henry, "A Methodology for Integrating Maintainability Using Software Quality Metrics," *Proceedings: IEEE Conference on Software Maintenance*, Oct. 1989, pp. 32-37.

[MAYOK89] Mayo, Kevin A., " Definition and Validation of Interface Complexity Metrics," M.S. Thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, December, 1989.

[MCCAT76] McCabe, Thomas J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, Dec. 1976, pp. 308-320.

[RAMAC85] Ramamoorthy, C. V., et al., "Metrics Guided Methodology," *Proceedings: Computer Software & Applications Conference*, Oct. 1985, pp. 111-120.

[SAMMJ82] Sammet, Jean E., Douglas W. Waugh, and Robert W. Reiter, Jr., "PDL/Ada—A Design Language Based on Ada," *Ada Letters*, Vol. 2, No. 3, Nov./Dec. 1982, pp. 19-31.

[SHATS88] Shatz, Sol M., "Towards Complexity Metrics for Ada Tasking," *IEEE Transactions on Software Engineering*, Vol. 14, No. 8, Aug. 1988, pp. 1122-1127.

[WAKES88] Wake, Steve and Sallie Henry, "A Model Based on Software Quality Factors Which Predicts Maintainability," *Proceedings: Conference on Software Maintenance-1988*, Oct. 1988, pp. 382-387.

[WOODS80] Woodfield, S. N., "Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors," Ph.D. Dissertation, Computer Science Department, Purdue University, Dec. 1980.

[YOURE89] Yourdon, Ed, "Software Metrics: You Can't Control What You Can't Measure," *American Programmer*, Vol. 2, No. 2, Feb. 1989, pp.3-11.

Bryan L. Chappell received his M.S. in Computer Science from Virginia Tech in 1989. Bryan Chappell's interests include: Operating Systems, Compiler Construction, and Software Engineering. Mr. Chappell is currently working for Software Technology, Inc., in Alexandria, Virginia.



Sallie Henry received her B.S. from the University of Wisconsin-LaCrosse in Mathematics. She received her M.S. and Ph.D. in Computer Science from Iowa State University. While attending Iowa State University, Dr. Henry's research interests were in Programming Languages, Operating Systems, and Software Engineering. After completing her Ph.D. when returned to the University of Wisconsin-LaCrosse as an assistant professor and later an associate professor of Computer Science.

Dr. Sallie Henry is currently on the faculty of the Computer Science Department at Virginia Tech. She has been working in the area of Software Engineering for the past eight years. Her primary research interests are in Software Quality Metrics, Evaluation of Methodologies, Software Testing Methodologies, and Cost Modeling. Her most recent work has been in the validation of software quality metrics, with particular focus on applying metrics during the design phase of the software life cycle. The first of two design studies uses an Ada-like PDL for designing software and the other study incorporates quality metrics with a graphical design language. Dr. Henry's research has been supported by funding from NSF, Naval Surface Surface Weapons Center, Digital Equipment Corporation, IBM, Software Productivity Consortium, Virginia's Center for Inovative Technology, and Xerox Corporation. She is a member of IEEE and ACM.

Kevin A. Mayo received his M.S. in Computer Science from Virginia Tech in 1989. Mr. Mayo's interests include: Software Engineering, Compiler Constructure, Human Factors, and selected Artificial Intelligence Topics. Mr. Mayo is currently working toward his Ph.D. in Computer Science from Virginia Tech.

Software Reliability Prediction and Management

Basil Papanicolaou,

Sanders, a Lockheed Company

Abstract

Software Reliability is one of the most critical system attributes, impacting operational performance, development and maintenance cost, and schedule. Even though it is usually manifested during system-level test and integration, it starts to take shape much earlier in the planning stages of a project. A study was performed to investigate a methodology for an early software reliability assessment, so that reliability can be monitored for risk management purposes. Results showed that reliability assessment is possible even from the pre-software development phase and that reliability dependencies can be defined and measured so that reliability can become a driver through the development. Reliability estimates are based on run time system configuration rather than in terms of the traditional static approach.

Basic Software Reliability Models

Software reliability is the probability that a software system operates on the target system for a period of time without software error. Several models that deal with software reliability have been developed [1] and validated [2]. Their objective is to forecast the behavior of the system when it becomes operational. In general they describe a random process that represents the behavior of failures with time. A software program is characterized by the number of inherent faults N_0, that is faults that will occur during the software life-cycle.

The Musa software model [3] is used as the basis of our methodology. It has been chosen because of its reasonableness, validity, and capability to provide results in terms of execution time and calendar time. The output can thus be combined with hardware reliability to yield real-time mean time to failure (MTTF). Calendar time can be used to determine development and test and integration schedules. According to the model the number of errors detected and corrected as a function of the cumulative execution time τ.

$$N = N_0 \left[1 - \exp\left(-\frac{C\tau}{M_0 T_0}\right)\right] \quad (1)$$

where M_0 is the total number of failures occuring during the maintained life of the software, T_0 is the initial MTTF, and C is the test compression factor.

Test compression factor is the ratio of execution time required in the operational phase to execution time required in the test phase to cover the input space of the program. For simplicity we assume that there is one-to-one correspondence between observed errors and software failures. T_0 is a function of N_0, target computer characteristic parameters, and constants, which have been derived by Musa from many projects [2]. MTTF is expressed as an exponential function of execution time. The model also provides reliability value as a function of MTTF, and calendar time required to reach reliability goal, according to planned test resources.

In its simplest form a software reliability prediction model gives the number of remaining errors as a function of test time,

$$N = N_0 e^{-At}$$

where A has been estimated during this study from many projects and has been found to vary slightly from project to project. Figure 1 shows the predicted and actual number of remaining errors during the software implementation phase of a project.

Early assessment

The existing software reliability models can be applied when software errors appear in testing. However, in many cases reliability assessments are needed much earlier. The

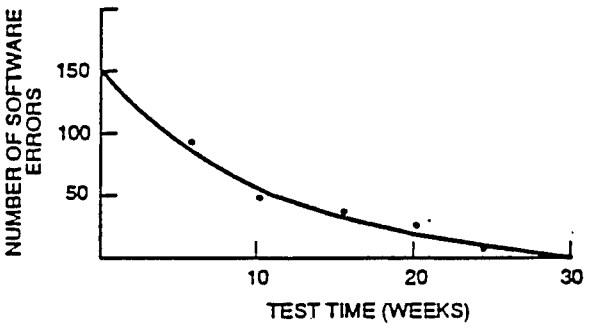


Figure 1. Number of Errors vs Test Time

following describes our approach to solve the problem:

Software reliability depends on the quality of the software support environment and the planning and methodology associated with software development. This basic assumption is the starting point for the software reliability assessment from the pre-software development phase of a project. This reliability dependency is quantitatively expressed by the software production quality factor Q, introduced by this methodology.

The actual number of inherent faults at the beginning of a software activity depends on the size (S) of the software, the average fault density d_f the software production quality factor Q:

$$N_0 = k\, d_f\, S\,(1-Q) \qquad (2)$$

where k is a constant.

The software quality factor is the weighted average of a set of reliability contributing elements that can be easily defined and measured. The weights have been derived through comparative analysis between observed failure rates, system requirements, and development information from several projects. Table 1 shows the software quality factor components and associated weights. Average error densities at specific points in the development cycle, such as the start of the coding phase, and operational test phase, were derived from data collected from several projects.

Table 1.
Software Quality Factor Components

Q Components	Weight
Software Requirements	20
Software Complexity	-10
Modern Progr Practices	10
Language	10
Schedule Variance	-5
Software Eng Environment	15
Personnel Attributes	20
Past Experience	5
Resource Availability	-5

Each software production quality factor component is derived by evaluating a specified set of measurable representative quantities. As example software requirements can be evaluated in terms of the percentage of traceable, testable, complete, consistent, and correct requirements, as observed through software product evaluations, software inspections, or set as goals, if development has not started yet. Derivation of some elements, such as schedule variance, is based on empirically derived algorithms. Software complexity can be determined in terms of software type, or, when available, the software complexity number [4], and so on.

Once the adjusted number of inherent faults has been calculated from (2), we can derive MTTF, reliability and calendar time according to the Musa model. Figure 2 shows predicted MTTF versus operational time for a 250,000-line project during the Operational Test and Evaluation phase. The prediction was made at the concept development phase.

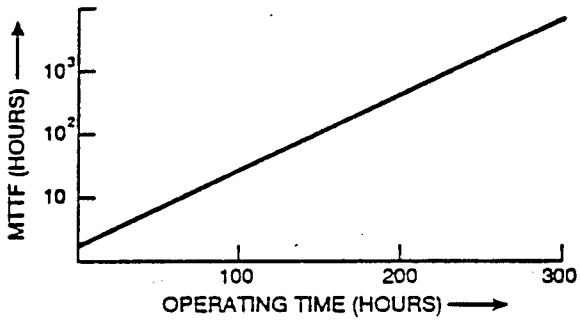


Figure 2. MTTF vs Operational Time

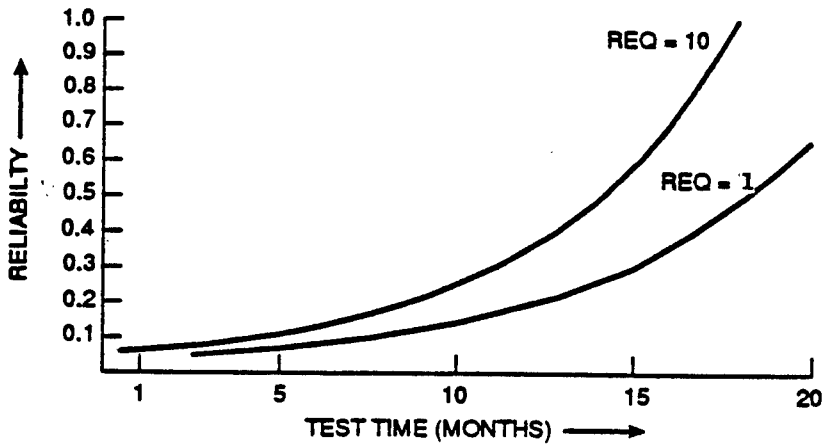


Figure 3. Reliability vs Test Time Sensitivity

A drawback of the Musa model is that it considers the software as a black box, while a typical characteristic of a mission critical, large scale, real-time system is that various software elements have different complexity or utilization rates. A class of software reliability prediction models, known as micro-models, is addressing this aspect; however they have not been found suitable as a project wide reliability monitoring tool. A principle developed by Fiorentino and Soistman [5] is used to complement the Musa model. The reliability of each software component in a system is estimated as described above. Next the total software reliability R_S is estimated by combining all component reliabilities Ri in terms of their mutual transitional probabilities Pij.

$$(Rs)=(Ri)\ (Pij)$$

For Ada based systems Rs is derived from the reliability of Ada subprograms and tasks and their transitional probabilities under run time configuration. It is estimated that at this point the confidence level for derived reliability predictions, assuming adherence to the planned set of model parameters, is over eighty percent.

The ability to derive reliability values for different system and software development environment characteristics, represented by the software production quality factor, allows for detailed analysis and tradeoff studies on how to achieve the desirable software system reliability goal in terms of software design, support, schedule, cost, and resources. Figure 3 shows reliability curves under best-case (REQ=10) and worst-case (REQ=1) software requirements quality.

Tradeoffs assist analysts, designers, and managers in forming the architecture, selecting the methodology, allocating reliability goals for the system and its component subsystems, and achieving them under minimum life-cycle cost. Some of the applications are evaluation of alternative design approaches, system behavior under different operational modes, impact of resource quality, schedule constraints, and software engineering technology [6].

Transition to the implementation phase

When actual data become available during the software development phase N0 and T0 are reestimated using maximum likelihood approximation methods [1]. Data is collected through the test and integration phases, using software problem reports or recorded failure data. The relationship between the associated test compression factors must be determined. Transitional probabilities can be found through system exercise under real-time conditions and event recording. Prediction confidence increases as more actual data are accumulated and used.

The process can be used for the entire software development life-cycle or for any particular software activity. It can be readjusted or reinitialized at any point in the life-cycle. It can also be used to handle independently different classes of errors, if required. Software reliability can be combined with hardware reliability by means of an appropriate reliability network for the system [6,7] to yield total system reliability.

Spreadsheets are suitable for the implementation of reliability models. The can provide user interface, data processing, interface to data bases, and graphics utilities.

Reliability Management

Software reliability can now be measured and related to developmental and system parameters. The seven basic steps of the methodology that allows for software reliability assessment and management throughout the system development life-cycle can be summarized as follows:

1. Software Reliability Model selection: While the Musa Execution Time model has been used in this work, other software reliability prediction models can be used. Different models can be used for different phases of the program, as suitable. It is however recommended that the models consider the target system and the development process and environment and that they have predictive capabilities.

2. Model interface definition: Model parameters and their relationship to software development environment, project plan, and target hardware must be well understood and expressed in measurable quantities.

3. Model calibration: Models need to be calibrated for the intended application and developmental environment, with data collected from similar projects. Calibratable parameters include the elements of the software quality factor, the norm error densities at the beginning of system test and integration or any other phase, test compression factor, etc.

4. Software reliability goal: If software reliability is not specified, it must be derived from the required system reliability through an appropriate decomposition method [6,7]. Reliability goals for each phase of software development activity must also be specified.

5. Model integration into the software development process: The model will be useful only if used to monitor continuously the software development. It must be part of software product evaluation, software quality assurance, and engineering analyses.

6. Reliability assessment and analysis: Measured or predicted reliability values are derived and compared to reliability goals. Tradeoffs between possible adjustments are then performed with the help of the reliability model.

REFERENCES

[1] Amrit L. Goel, A Guidebook for Software Reliability Assessment, Technical Report No. 83-11, RADC, 1983.

[2] J. D. Musa, "Validity of Execution-Time Theory of Software Reliability, IEEE Trans. of Reliability, Vol R-28, Aug 1979, pp 181-191.

[3] J. D. Musa, "A Theory of Software Reliability and its Application", IEEE Trans. Software Engineering, Vol SE-1, Sept 1975, pp 312-327.

[4] McCabe, T. J., "A Complexity Measure", IEEE Transactions on Software Engineering, SE-2(4), 1976, pp 308-320.

[5] E. Fiorentino, E. Soistman, "Combined Hardware/Software Reliability Predictions", Proceedings, 1985 Annual Reliability and Maintainability Symposium, pp 169-176.

[6] J.Musa, A. Iannino, K. Okumoto, Software Reliability, McGraw-Hill, 1987.

[7] Shooman, Martin L., Software Engineering Design, Reliability and Management, McGraw-Hill, 1983.

Effectiveness Measures for the Software Process

Richard Guilfoyle*
Monmouth College
West Long Branch, N. J.

Abstract

Observable items that contribute to software development are potential indicators of success or failure. Standardizing the use of these indicators leads to measures of effectiveness for the software process. Analyzing these measures suggested refined characterizations for requirements and systems that aid in clarifying requirements. The necessary tools and techniques for effectiveness measurement and evaluation also need standardization. In response to such needs, a paradigm encouraging thorough reporting of all significant issues and addressing the problems of evaluation is presented.

1 Introduction

The body of this paper analyzes problems associated with measuring effectiveness within the software process. The development uses a working definition of effectiveness that emphasizes measurable qualities and criteria for effectiveness. Also presented, to support the use of this definition, are new approaches to characterizing requirements and systems along with analyses of compliance evaluation.

Briefly, *effectiveness* connotes results complying with requirements, while *process* relates to the rules and operations that produce the result. The supposition made is that forming effectiveness measures requires tracking observable aspects in the software process that foretell the level of compliance between the intentions and outcomes. Consequently, process measures of effectiveness, were they available, would offer us possibilities like monitoring the development of systems to predict the quality of results. Conceivably, with corrective control, we could optimize progress. So process measures of effectiveness are tools supporting project management that focus on relationships between the process components, requirements and system.

From the perspective of managing development, two key activities that emerge are *monitoring* and *correcting*. Monitoring effectiveness requires representations for *both* the requirements and system states; both of these representations should be analyzed together. There is, however, the unfortunate prospect that each will have a blurry image. Unfinished components and unresolved intentions induce this lack of clarity. So the measures envisioned will have to deal with difficulties like inadequately represented requirements and systems.

Along with representations for requirements and systems, a correction mechanism also would require a process model capable of showing the consequences of applying alternative rules and operations at given stages of development. Potentially, having such a model available would reduce the likelihood of making poor choices during the genuine development.

Thus, the search for process measures of effectiveness reveals several needs; the body of this paper proposes ways to deal with the needs and analyzes what it would take to satisfy them. The needs follow.

- Provide capabilities that foster continual clarification and solidification of requirements.
- Develop improved measures or metrics for characterizing the state of the developing system in light of its emerging requirements.
- Provide methods for developing, using and refining potential compliance metrics.
- Furnish process control models capable of depicting the progress of development.

*This effort received partial support through funds administered by the Battelle Memorial Institute and provided by the Advanced Software Technology branch of the U. S. Army CECOM plus the Army Research Office — Contract No. DAAL03-86-D-0001; D.O. 1265.

2 Assumptions and Definitions

A working definition for effectiveness in the context of the software process is central to the other topics presented here. Accordingly, effectiveness means: *the degree to which the intended result has been achieved.* For the software process, achieving the intended result signifies that the system met expectations — it conforms to its requirements. This definition skirts problems like inadequately depicted requirements and how one gauges the level of conformity between the results and requirements. Instead, it suggests that to measure effectiveness, the essential information originates within *both* the requirements and the results. Adopting the definition and forming process measures of effectiveness thus necessitates the development of representations that aid efforts to evaluate the extent of the conformity between the results and requirements.

Software development, however, uses a set of rules and operations intended to actualize a software system. Taking this view, the rules and operations do not represent requirements or results. On the other hand, these rules and operations certainly do influence the level of conformity mentioned above. So we need to find ways of recasting the combined states of the requirements, developing system and the software process into a *prediction* about the results conforming to the intent. Note that deficient requirements will impair predictive accuracy, i.e., we cannot be accurate when our target is indeterminate. Note also that the current states or states-history is a key ingredient of projections about the future, so the need for suitable ways to represent the key ingredients surfaces again.

While the goal of predicting compliance is optimistic, having the representations supporting such a goal would let us calculate current levels of conformity and so would be valuable by themselves. Thus, a worthwhile fall-back goal is to develop more comprehensive representations integrating intentions and current results as inputs to compliance measures.

Focusing on compliance briefly, it is essential to develop a basis of comparison between the requirements and the emerging system. Recall that generally both are incomplete. Consequently, comparisons between requirements and developing systems have to tolerate ambiguity and allow for incompleteness. Further, the comparisons made during different development projects will differ. So there is a need to find a set of underlying principles that help us to set up bases for comparison that are project independent and robust during development. An evaluation paradigm presented later serves to address this need.

Returning to the topic of prediction, forecasting that a certain landmark will be reached as a result of following a particular sequence of steps, implies an underlying model that relates rules and operations to outcomes. Creating a general model, applicable over a range of different development conditions seems infeasible for several reasons. First, not enough evidence of causal relationships is available to build a comprehensive model. Second, it is bad science to base a model on inexplicable statistical patterns and it is nonsense to make forecasts with them. Third, gathering enough data to estimate a comprehensive model (disregarding the other difficulties) would be a colossal task.

Based on the above arguments, attempts at predicting outcomes must remain small in scope until more is understood about the impact of the rules and operations employed in software development. Limited investigations are feasible but only if they adopt more refined characterizations of the software process, requirements and systems.

2.1 Characterizing Requirements

Requirements (documents) simultaneously represent intentions at differing levels of abstraction since the depth of understanding for each of the system components also differs. Accordingly, there is a need to encourage a level of specificity that aids design and validation efforts without also stipulating the design. The following terminology seeks to refine the way we deal with requirements and thereby make the state of the requirements more explicit. The assumption is that studying how well the requirements address each of these points enables planners to identify and correct weaknesses in the requirements. Considerations such as quantification and modularity influenced the choice of terminology. For example, one of the general goals is easier comparisons; so quantifying properties will support such a goal. Also, there is general agreement that the appropriate use of modularity can help to reduce complexity; so it would be valuable to create requirements with loose coupling between each of the categories.

The terminology proposed below follows usage commonly found in technical articles; still, there is a need to quantify and correlate variations in these characteristics with properties of the resulting system, thereby supporting efforts to optimize effectiveness in general. There is a further need to determine whether this new terminology fully captures the essence of the requirements' state to assure the adequacy of the characterizations.

For now, the categories are: variability, testability,

traceability, ambiguity, completeness and expressivity. The pattern: <bad quality>↔<good quality> seen below signifies that there is a range of possible values for each characteristic.

2.1.1 Variability: Volatile ↔ Steady

This characteristic describes the degree or extent of change occurring in the requirements. Note first that this terminology differs from the notion of stability, i.e., describing the tendency to return to equilibrium. Postmortem studies often cite highly volatile requirements among the most notable factors causing project difficulty or failure. Accepting that volatile requirements are very troublesome to deal with, this refinement highlights requirements variability so that developers might take special steps to reduce changes and their impact. Potential measures of volatility include rates of engineering change requests or lists of items ranked according to their probability of change and corresponding impact.

Several approaches to development use terms like evolutionary, incremental or prototypical development to describe the developer's interactions between the requirements and the partially fabricated system. Implicitly or explicitly, these approaches reinforce the assertion that high volatility is a crucial problem and their response to such problems is to subscribe to a principle of controlled change. Accordingly, proceeding in smaller increments or running a prototype confines the change by transforming the problem into a collection of smaller, related problems with limited scope.

2.1.2 Testability: Obscure ↔ Demonstrable

This characteristic describes the degree or extent of the capacity to show conformity to the requirements. Enhanced testability means easier validation of the developing system while diminished testability means that there is an increased likelihood of no, or possibly erroneous, validation. The assumption here is that focusing on testing is likely to foster earlier clarification of the functionality, structure and behavior aspects of the requirements. This is plausible because a better understanding of the system appears to be a precondition for easing the test effort.

A few development approaches use statistical estimates of reliability to control system quality during development; see [5], [13], and [12] for more details. Customarily, these approaches withhold the release of parts of the system until they attain an appropriate level of reliability. These observed, or predicted, reliability levels are quantities having plausible use as conformity metrics, relating results to requirements. Moreover, these metrics are potential components of process measures of effectiveness.

In the context of this definition, the statistical models described above conceivably are suitable bases for developing scales of testability. For example, the effort required to exploit each of the various reliability models may be a measure of testability. Testability also pertains to the efficacy of the validation process and should be analyzed as a potential adjustment factor for validity claims. That is, assertions about system validity and the measure of system testability should operate collectively. The point here is that demonstrable requirements tend to strengthen assertions of validity while obscure ones tend to weaken such assertions.

Reliability modeling techniques, while powerful, may be muddled by changing requirements and consequently not hold up well in volatile conditions. Interactions of that nature should reinforce our appreciation of the need to separate concerns; e.g., while trying to improve testability, it will be constructive to also reduce the impact of variability. In spite of the sensitivity to variability, ways of employing the reliability level to guide decisions regarding the timing of maintenance changes in existing systems appears to be feasible [12].

2.1.3 Traceability: Indirect ↔ Explicit

This characteristic describes the degree or extent of the coupling between the requirements and implementation. The assumption here is that explicit linkage between requirements and results makes the task of judging compliance easier. One approach might be to elaborate requirements emphasizing the links between each level of detail.

The extent to which formal specification techniques contribute to a development is a potential basis for a traceability measure. Formal, in this context, means employing mathematics-like notation to represent the requirements and applying proof-like reasoning to develop and validate the system. Examples of these are the Vienna Development Method (VDM) [11], [3] and development using the Z notation [14]. Starting from the requirements, these methods use transformations to elaborate immature (undeveloped) representations more completely, sometimes with computer aids. One advantage of using such transformations is that they do not introduce errors into the elaborated representations. Another advantage is that the inputs and outputs to the transformations form a recognizable sequence of links between the requirements and the results, thereby increasing traceability. One further advantage is that formal methods can speed up the

process of making changes in steady systems. Thus, such formal methods should score higher in traceability than informal ones. When the connections between requirements and results are hard to follow, oblique or nonexistent, the scale tips towards indirectness, i.e., lower traceability scores. Such a possibility could certainly occur as a result of volatile requirements with the volatility overwhelming attempts to maintain control.

2.1.4 Ambiguity: Vague ↔ Clear

This characterization describes the degree or extent of certainty about the intent of the requirements. The objective is to develop a gauge that shows when the intentions are too vague and therefore need further clarification. For example, the spiral approach developed by Boehm expects the estimation of certain risks before proceeding to the next step [4]. Estimates like these, which are quantitative, may provide a footing for developing an ambiguity metric. Using such estimates, a high level of risk indicates greater ambiguity, or vagueness, while low risk signifies confidence about both the intent and the prospect of achieving it.

2.1.5 Completeness: Deficient ↔ Absolute

This characterization describes the degree or extent to which the requirements address all the intentions. Completeness is one of the more familiar characteristics of requirements and has received much attention. Is seems feasible to abstract the results of prior studies and construct a check-list of areas for consideration to avert omissions in the requirements. Hence, the more items addressed, the more complete the requirements. One prominent specialist, T. Gilb, advocates something of this nature [7]. His approach towards extensiveness advocates including expectations, costs, quality, priorities, schedule — just about every relevant item — in a measurable way. Gilb argues that anything less may lead to unexpected results. Also, dissatisfaction with those results implies unexpressed, i.e., incomplete, requirements. A potential basis for measuring completeness might use data relating to the set of areas presently covered by the requirements compared to those that the system must cover. Of course, one predicament here is that when building unprecedented systems there will not be any checklist available. The recommendation then is familiar: proceed with caution; use incremental, evolutionary or prototypical approaches. Thus, the recognition of incompleteness suggests that we be especially prudent. A scale for incompleteness may offer further guidance about how prudent to be.

2.1.6 Expressivity: Crude ↔ Powerful

This characterization describes the degree or extent of the capacity to describe the intentions. When the language of expression is unable to represent the intent, i.e., it is crude, the requirements may be incomplete (deficient), ambiguous or worse. The language of expression is the way used to communicate requirements, e.g., textually, iconographically or otherwise. When structures, behavior or functionality can only be represented in cumbersome or indirect ways, the crudeness of the requirements may be excessive. Analyses of language "power", leading to expressivity scales, might start with investigations of the relative merits of visual formalism [9], design representation [15] and textual representations. Assuming that visual and iconographic representations are more lucid, then requirements representations making greater use of them should be regarded as being more powerful, while those employing more text should be regarded as the more crude.

2.1.7 Discussion

The relationships between the characteristics described are nebulous. Hypothetically, requirements could specify the wrong system from the user's viewpoint. In such a case, these requirements end up being both complete and vague, i.e, they are complete for a client who is not the user. Yet, they are vague because their intent (meeting the user's expectations) was wrong. All this might raise questions about the validation procedures used; and so, completeness and testability add intricacy to the problem.

A topic related to reliability prediction involves a question of how to generate fault situations. The strategies mentioned earlier rely heavily on usage profiles but some experienced developers object to that, voicing concern over a lack of other types of testing. Advocates of the reliability approaches generally accept the need for specialized testing to prove or guarantee particular properties of the system. So the fault generation procedure may significantly affect system testability metrics.

In validation, the requirement from which the system departs must be explicit; so testability surfaces in another way. It may be possible to construct testing plans that operate with incomplete or volatile requirements as follows. Every project has unique qualities that distinguish it from others; for those qualities, try to determine the validity measures and thresholds from project to project in a uniform manner. That is, develop a paradigm for setting up project-dependent but serviceable measures of validity. Instead of standardizing the measures, this paradigm

would offer rigorous ways to create measures applicable only for particular projects, thereby addressing testability. Such a paradigm would help both user and developer to agree on the criteria to use for acceptance of the product. By suggesting how to portray the developing system with respect to the original intent, the paradigm helps us to determine how close the development status is to the required condition.

In an area related to the completeness characterization, proof- theoretic approaches have the potential to become the dominant way to achieve the ideal of error-free systems. Such systems are ones whose only vulnerability is from occurrences *mistakenly* assumed to be independent of the product system. Instead of emerging from shoddy checking efforts, operational failures in such systems result from inaccurate suppositions about the links with the external operating environment. Still, formal techniques applied to incremental development suffer from several shortcomings; some of them being similar to those pertaining to the statistical control approaches. For example, given highly volatile requirements, new proofs must accompany changed designs ensuing from changed requirements. Additionally, formal methods tend to require higher levels of mathematical and logical sophistication than other approaches. Additionally, very large projects overpower the current capabilities of formal methods.

When projects reported as failures come to light, we must consider the possibility that the development organization was unfit for the task. To ignore it could be a dangerous oversight; even high quality requirements cannot compensate for unfit development teams. Demonstrating developer competence prior to start-up often appears to be immaterial. Moreover, project data collected during development treats developer competency with indifference. When unfit developers use rules and operations that lead to noncomplying results, the knowledge of their blundering is reinforced, but instead of learning what to do — we are seeing what not to do! The Software Engineering Institute contractor assessment procedure [10] offers a potential way to lessen concerns over competence and clarify data derived from software projects. The institute's efforts to develop such an assessment procedure might provide a standard for rating developers. Of course, such a standard has to be practical, achievable and defendable; reflecting good engineering practices. This problem points out another concern needing further investigation, in particular, we must try to determine the appropriate preconditions required to carry out successful development.

2.2 Characterizing Systems

In answer to the need for terminology supporting the characterization of systems, the next itemization suggests areas for further analysis and development. The items, rephrasing those described in [8, page 159], exemplify qualities associated with *software systems*. Following the recommendations portrayed earlier, designers should incorporate all of these terms as part of the system requirements. Moreover, the measures or project-dependent interpretations for the items should be thoroughly dealt with at the *requirements* stage of development. On the other hand, inability to reach early closure on the meanings of all terms must not cripple a project. Accordingly, the need is to find a way to proceed, converging on unresolved interpretations *as the system matures*. As previously stated, the strategies based on incremental, evolutionary or prototypical approaches are robust with respect to immature requirements.

The separate system qualities follow.

Functionality: capabilities provided, generality, security.

Usability: human factors, aesthetics, consistency, documentation.

Reliability: frequency/severity of failure, recoverability, predictability, accuracy, mean time to failure.

Performance: cost, speed, efficiency, resource consumption, throughput, response time.

Supportability: testability, extensibility, configurability, adaptability, maintainability, compatibility, serviceability, installability, localizability.

To highlight the differing emphasis here, recall that earlier, the discussion of reliability described how the use of run-time failure patterns could be a part of quality control activities. The view there was that being able to employ reliability models evidenced improved testability. Now, in this context, the idea is that the portrayal of admissible failure patterns is a key part of the requirements process. Generalizing to other areas, each of the items just listed should be framed out in a *measurable* way. That is, the acceptable patterns have to be represented in ways that permit comparisons with the actual system patterns, so compliance evaluation becomes reasonable.

For example, cost patterns are notable for their frequent absence from requirements statements. Instead, the cost aspects of the intended system should

assign weights to items such as variability, testability, traceability, ambiguity, completeness and expressivity. Thus for variability, they should describe an admissible pattern of variability confined by cost; or, describe an admissible pattern of costs resulting from variability. The treatment of the other requirements items should be similar. By taking each of the requirements characterization areas and relating them to each of the system items, we provide a suitable foundation on which to construct evaluations of intent versus results.

3 Measuring Effectiveness

A claim made earlier was that linking process measures to results basically means predicting the results of a development effort using fragmentary information from the current state. Two important concerns are the prediction's substance and accuracy. You must make tangible assertions about real events and have a tangible basis for comparing predictions with results. Examples of items typifying what needs to be linked are shown in the subsequent table. Whether they are all tangible is arguable, but, incorporating them in requirements is possible with an appropriate basis of understanding.

EXAMPLES OF MEASURES	
Process	Effectiveness
#team-members	deviation from timetable
#program-lines tested	deviation from budget
#function-points implemented	deviation from performance aims
#person-days consumed	reliability level
% of cpu-capacity used	delivered functionality
unusual patterns in above items	usability
#hours of run-time logged	supportability level

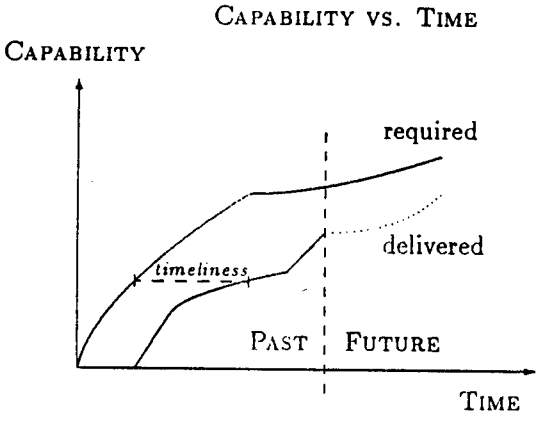
The unsolved problem is to find the blend of items like those in the first column that accurately predicts values for entries like those in the second. The statements listed below show presumed relationships between process and effectiveness measures. They are paraphrased from assertions made in a U.S. Army pamphlet on software management indicators [2]. The assertions made exemplify the causal relationships thought to exist between the rules and operations of software development and the results. The statements are predictive in nature but their predictive accuracy is problematical. Stating outcomes in a more quantified style, e.g., using probability distributions, would be more constructive.

- Consistent near capacity use of computational resources is likely to cause a change in the project. Severe degradation in schedule adherence, cost management and quality standards occur when the available spare resource falls below 10 percent.
- Under-staffing is likely to cause results to fall behind their intended schedule.
- Adding personnel near deadlines suggests potential trouble and is unlikely to compensate for lateness.
- Iligh personnel turnover rates may lead to continuity problems: e.g., later staff may be uncertain about the project's original intent.
- Projects having more than 15 percent of their requirements classified as untestable, or untraceable, also have a medium to high potential of changing in a significant way, often with negative impact on cost and completion schedule.
- Trends in the number of completed units suggests the extent of the developer's understanding of the requirements.
- The availability of needed tools suggests the developer's readiness. When tools are unavailable, delayed results are likely.

Several of these predictive statements refer to timeliness. The apparent implication is that timeliness is one of the more important components of effectiveness. The all-inclusive requirements approach cited earlier would set down a pattern of expectations and assign a weight for timeliness, denoting its relevance to the partially complete results. Doing this necessitates a more thorough understanding of all requirements so that they can be ordered according to priorities. Another explanation for the emphasis on timetables may be that tracking timeliness is easier than measuring other departures from requirements. Even so, there are pitfalls to simply measuring deviations from the planned schedule.

By using standard axes to plot both the required and provided capabilities against time in the figure below, timeliness may be viewed as the *horizontal* directed distance from the required to the provided curve [6]. (The plot of required capabilities represents a timetable.) The distance between the two plots can be shortened at later times in a project by altering one or both of the two curves' slopes.a For

example, changing the times when required capabilities are to be available makes the upper curve either more gradual or more steep. Should the trend of the other curve remain as before, then the future timeliness will change.



Assertions about the provided capabilities curve maintaining its current trend, undoubtedly, are forecasts. Since predicting near-term events is less risky than predicting those far off, we need to find ways to keep the time factor shorter. That is, make relatively more projections about how things will turn out in the near future. Once again, incremental and evolutionary approaches surface. The effect of such approaches on the plots shown above is to shift the delivered capabilities plot leftwards, closer to the required plot. The reason is that the improved predictive accuracy allows the developer to hit the target (timetable for future deliveries) more often. A plot corresponding to a prototyping approach cannot be expected to show such a shift; instead, the slope corresponding to the delivered capabilities will show more dramatic ascents and descents because the delivery of prototypes is earlier but they are discarded later.

Regarding timeliness, consider the following questions focusing on timeliness, deadlines and related process measures:

- What rationale justified the deadlines? Are the deadlines guides, forecasts or absolutes? What was their source — contractor or agency?
- How can we determine whether a missed deadline resulted from unrealistic expectations, faulty requirements or inappropriate efforts?
- Which observables most strongly correlate with the degree of lateness? Do these generalize to most development projects?
- Can process measures reveal unrealistic expectations for deadlines?

It is evident that some of the causes for lateness can come from outside the development process, lessening any process measure's capability to accurately predict deviations from schedule. Knowledge of the preconditions for successful development again becomes necessary. Yet somehow, the process itself should recognize and react to problems; so monitoring and recording the changing state of the process might detect tangible evidence of those problems. Even so, it appears futile to *predict* departures from schedule without awareness of some milestone schedule. Generalizing the reasoning, efforts to predict departures from requirements by watching the current process activities also must refer to the requirements themselves, the current state of the process and the preconditions for success.

3.1 An Evaluation paradigm

This section contains an summary of a paradigm, developed by a colleague, R. Pirchner at Monmouth College, and myself [1], that can be employed to create the basis for evaluating a system's compliance with its requirements. Working with such a basis allows evaluations to take place in a disciplined manner. The evaluation basis not only focuses on the data, but also on the assumptions and reasoning processes underlying the evaluation. Hence, the paradigm specifies a framework considering all components necessary to carry out the evaluation. The data gathering steps come only after instituting a basis for evaluation.

There are two basic phases of the paradigm: designing the evaluation and reviewing its implementation. The concern during design is to clarify assumptions and evaluation goals while the review clarifies the evaluation outcomes. The first five steps shown below, embody the design phase in the paradigm. Certain terms appearing in the list below have specialized meanings in the context of the paradigm. The term *goal-characteristic* denotes a system objective; compliance evaluation must occur for such properties. Collected, recorded, summarized, analyzed, or similarly processed data is termed a *representation*. Such data represents the system, the requirements or both.

1. Identify the goal-characteristic for the evaluation, and select one or several observables from the requirements or system judged to be related to the goal-characteristic.

2. Provide justification for hypothesizing this relationship between the observables and the goal-characteristic.

3. Determine the specific components of the observables which will be the focus of the evaluation and determine the relevant data to gather.

4. Establish a repeatable data-gathering technique and determine criteria by which to judge the admissibility of the data.

5. Determine the appropriate representations of the system or requirements; this includes defining the procedures which will be used to analyze the data and present the results.

6. Perform the data-gathering, and certify the admissibility of the data.

7. Using admissible data, measure the compliance and report results.

8. Provide a critique, addressing any limitations, of the study.

While the paradigm at times may appear to state the obvious, the fact that its authors encountered so many ambiguous studies led them to formulate it. Too often, valuative statements turned out to be subjective declarations lacking supportive evidence. The position taken in the paradigm is that subjective declarations become more credible by fully revealing the basis used for making the declarations.

3.2 Example

Consider an undeveloped system that is expected to ease some human task, be on time and be reliable. Such qualities illustrate goal-characteristics that must be thoroughly described. The six characterizations for requirements help to clarify the meanings of such goals. Nonetheless, preferences among the properties like functionality, performance and reliability must be spelled out. For example, should we favor or sacrifice functionality with regard to timeliness? Continuing, the observables related to each goal must be identified. It is not enough to presume a relationship between observable and characteristic; instead, the involved parties must *accept the explanation* of the relationship.

Carrying on, how do we define easier? The more relevant items under functionality seem to be the capabilities provided and generality. Thus, what capabilities will it take to alter the task into an easier one; how will generality be evidenced? Since we still have not nailed down the meanings, we should introduce a list of observables related to the goal-characteristic. Observables to include might be: work time to complete task, number of steps per task, number of transactions per task or training time. Generality might be seen by showing that the previous tasks performed can now be enlarged, or more complex. At this point, by following the evaluation paradigm, a definition for the goal- characteristic begins to jell. In the process of establishing patterns that signify "easier", we must address the testability component of requirements. That is, the requirements formulation must include measures of the developing system's capability to ease the task. With a planned phase-in of more rigorous tests as a function of project milestones, the project's requirements also can accommodate ambiguity and completeness. Moreover, moving from one form of test to another has the potential to reduce variability. For example, designing for a requirement such as "a reduced number of interactions per task" may be a good first approximation. Later, however, the final proof of "easier" is: compliance to a certain pattern of task completions. If the final pattern required is undefinable near the project's beginning, then proceeding with no pattern allows the other properties to vary unchecked and ignores their eventual impact on the goal-characteristic. Such actions imply lack of development control.

From the perspective of effectiveness, the problem is to determine which observables to track so that we can measure our progress. Progress, it turns out, corresponds to a goal-characteristic such as *acceptable to its receiver.* Once again the paradigm, applied to this new goal-characteristic, provides an orderly way to evaluate progress. Further, it would be valuable to know which rules and operations employed by developers having the greatest impact on progress in most projects. As stated earlier, creating a general model capable of predicting outcomes appears infeasible now. However, presumed relationships like those listed earlier in this section abound. They need to be subjected to the evaluation paradigm, quantified and subjected to careful scrutiny with results meticulously reported, so that the software engineering community can benefit from the knowledge.

4 Summary

The many rules and operations applied during software development endeavor to transform requirements into a working system. Quantifying the impact of these rules offers a way to predict the compliance of software to its requirements. But in the earliest development stages, the requirements them-

selves are immature. The ideas presented first in this paper offer characterizations of requirements which, by careful analysis, can accelerate requirements maturation. The six categories of analysis suggested address major deficiencies that frequently show up in requirements. By focusing on them, we recognize the deficiencies more readily and our understanding improves. The characterization areas also are amenable to quantification, making measures of requirements readiness feasible.

Assuming that better requirements would encourage the use of measurable criteria and observable patterns to describe the intent, they provide more effective information to developers. Thus, effectiveness is a function of the requirement's maturity as well. Measuring compliance involves comparing representations of the requirements and the product system. Such representations should be designed specifically to support the testing, validating and valuative functions. Consequently, the guidelines for setting up bases for comparison and evaluation are a worthwhile asset. The evaluation paradigm presented in this paper offers such a technique, one encouraging objectivity and managing subjectivity.

With rigorous evaluation procedures in place it might be feasible to conduct experiments that determine the influence of the rules and operations used in software development. From this information, knowledge the requirements's maturity and other preconditions, the development of process measures of effectiveness can begin. Then process measures of effectiveness offer a way to predict levels of compliance.

References

[1] "An Approach to Evaluating Software Development Methods." Technical Report MC89-S/W-METII-0001, prepared for the Product Assurance and Test Directorate of USA CECOM, Ft. Monmouth, N.J. March 1989. (unpublished).

[2] Army Materiel Command. *Software Management Indicators.* AMC Pamphlet 70-13, Alexandria, VA, January 31, 1987.

[3] D. B. Bjørner and C. B. Jones. *Formal Specification and Software Development.* Prentice-Hall, New York, 1982.

[4] B. W. Boehm. "A Spiral Model of Software Development." *IEEE Computer*, Vol 21, No. 5, May 1988, pp. 61–72.

[5] P. A. Currit, M. Dyer, and H. Mills. "Certifying the Reliability of Software." *IEEE Transactions on Software Engineering*, Vol. SE-12, January 1986, pp. 3–11.

[6] A. M. Davis, E. H. Bersoff and E. R. Comer. "A Strategy for Comparing Alternate Software Development Life Cycle Models." *IEEE Transactions on Software Engineering*, Vol. SE-14, No. 10, October 1988, pp. 1453–1461.

[7] T. Gilb. "Software Specification and Design — Must Engineer Quality and Cost Iteratively." *Software Engineering Notes*, Vol. 11, No. 1, January 1986, pp. 47–59.

[8] R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a Company-wide Program.* Prentice-Hall, Englewood Cliffs, 1986.

[9] D. Harel. "On Visual Formalisms." *Communications of the ACM*, Vol. 31, No. 5, May 1988, pp. 514–530.

[10] W. S. Humphrey and W. L. Sweet. *A Method for Assessing the Software Capability of Contractors.* Available as ADA183429 from the Defense Tech. Info. Ctr. (DTIC), Alexandria, VA, 1987.

[11] C. B. Jones. *Systematic Software development using VDM.* Prentice-Hall, New York, 1986.

[12] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, New York, 1987.

[13] R. W. Selby, V. R. Basili and F. T. Baker. "Cleanroom Software Development: An Empirical Evaluation." *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 9, September 1987, pp. 1027–1037.

[14] J. M. Spivey *The Z Notation: A Reference Manual.* Prentice-Hall, New York, 1989.

[15] D. E. Webster. "Mapping the Design Information Representation Domain." *IEEE Computer*, Vol. 21, No. 12, December 1988, pp. 8–23.

Author

Richard Guilfoyle is a Professor of Mathematics at Monmouth College. He has taught mathematics and computer science for over two decades as well as worked as a consultant in numerical analysis, simulation and software engineering. He received a Ph.D. in Mathematics from Stevens Institute of Technology. He is affiliated with the ACM and its SIGNUM, SIGPLAN, SIGSIM and SIGSOFT groups as well as with the IEEE Computer Society.

Measuring Coupling of Ada Program Modules

Ronald J. Leach

Keith S. Bagley

School of Engineering
Howard University
Washington, D.C. 20059

David E. Butler

AT&T Bell Laboratories
Naperville, IL 60566

Iva L. Brown

Bell Communications Research
Piscataway, NJ. 08854

Patrick W. Spann

Bell Communications Research
Piscataway, NJ. 08854

ABSTRACT

SANAC, a System for ANalyzing Ada Coupling is an interactive tool designed to measure the level of interconnections among modules of an Ada program which may be contained in multiple files. The underlying structure of SANAC is centered around the syntax and semantics of the Ada programming language. The tool itself is written in Ada and is composed of 26 Ada source files totaling approximately 5261 lines; it also uses 11 text files. The data produced by SANAC can be used to estimate software reliability. Development and validation of reliability is being done as part of the Howard University Metrics Project (HUMP).

1. INTRODUCTION

Since software is a fundamental component of many systems, the reliable operation of such systems critically depends on the reliable operation of their software components. Structured design methodology suggests that the quality of a design is associated with design principles. The degree of adherence to these principles may be objectively measured by one or more of the primitive metrics. One of these primitive metrics includes coupling which is measured by the number of interconnections among modules. For example, coupling increases as the number of calls between modules increases or as the amount of shared data increases. The hypothesis is that designs with high coupling will contain more errors [Conte86].

Our basic design objective is to provide a system to analyze Ada programs, to produce results that can be used to measure the quality of the design of the program, and to provide quantitative information about the degree of coupling to help in the assignment of resources to be used in software maintenance. This system is SANAC.

In an effort to explain exactly what SANAC is and its functionality and development, the following sections are provided: Section 2 provides background information which defines coupling and the different types of coupling, and describes the purpose of the tool. Section 3 describes the overall design of the tool. Section 4 describes the methodology of development which includes environmental data. Section 5 describes the use of the tool. Finally, Section 6 describes future research. Sample runs are included in the appendices.

2. BACKGROUND INFORMATION

2.1. What is coupling?

Coupling is a measure of interconnection among modules in a program's structure. It depends on a number of things including the complexity of the interface between modules, the point at which entry or reference is made to a module, and the data which passes across the interface. In software design, simple interconnections or low coupling among modules encourage modularity and reduce the probability of a "ripple effect" caused when errors that occur in one module propagate throughout the system [Pressman87].

2.2. Types of Coupling

SANAC measures six types of coupling for Ada programs: weak, data, stamp, control, external, and content. Each is listed according to its rating on the coupling spectrum which ranges from low to high coupling. A definition of each, followed by an example, is provided below.

2.2.1. Weak Coupling

One procedure does not use a value passed by another module.

```
MAIN PROCEDURE
--------------

PROCEDURE SANAC IS
  BEGIN
    PUT("HI THERE");
    PROC(7);
  END SANAC;


CALLED PROCEDURE
----------------
PROCEDURE PROC(J:IN) IS
  BEGIN
    FOR I IN 1..7 LOOP
      .
      .
    END LOOP;
  END PROC;
```

2.2.2. Data Coupling

One module uses an "in" parameter passed by the other and the value cannot be changed.

```
MAIN PROCEDURE
--------------

  PROCEDURE SANAC IS
    BEGIN
      PUT("HI THERE");
      PROC(7);
    END SANAC;

CALLED PROCEDURE
----------------
PROCEDURE PROC(I:IN) IS
  Save : INTEGER;
  BEGIN
     Save := I;
  END PROC;
```

2.2.3. Stamp Coupling

One module uses an "in out" parameter passed by other and the value can be changed.

```
MAIN PROCEDURE
--------------

PROCEDURE SANAC IS
  I : INTEGER:=10;
  BEGIN
    PROC(I);
  END SANAC;

CALLED PROCEDURE
----------------
PROCEDURE PROC(J : IN OUT) IS
  BEGIN
    J := 20;
  END PROC;
```

2.2.4. Control Coupling

One module uses values passed by another in its control or conditional statements.

```
MAIN PROCEDURE
--------------

PROCEDURE SANAC IS
    I : INTEGER:=10;
    BEGIN
      PROC(I);
    END SANAC;


CALLED PROCEDURE
----------------
PROCEDURE PROC(J : IN) IS
  BEGIN
     IF J = 20 THEN
          .
          .
     END IF;

     FOR B IN 1..J LOOP
          .
          .
     END LOOP;

     CASE J IS
       .
     END CASE;

     WHILE A < J LOOP
       .
     END LOOP;

     EXIT WHEN A = J;
   END PROC;
```

2.2.5. External Coupling

Two modules access the same internal data area where there is only one copy of the data at a time.

```
MAIN PROCEDURE
--------------

I : INTEGER:=10;
PROCEDURE SANAC IS
  BEGIN
    I := 5 ;
    PROC;
  END SANAC;


CALLED PROCEDURE
----------------

PROCEDURE PROC IS
  BEGIN
    I := 20;
  END PROC;
```

2.2.6. Content Coupling

One module has a branch into the code of another module.

```
MAIN PROCEDURE
--------------

PROCEDURE SANAC IS
  BEGIN
     .
     .
    EXCEPTION
     WHEN AN_EXCEPTION=>
       .
       .
  END SANAC;

CALLED PROCEDURE
----------------

PROCEDURE PROC IS
  BEGIN
       .
       .
    RAISE AN_EXCEPTION;
  END PROC;
```

3. PURPOSE OF TOOL

SANAC, a System for Analyzing Ada Coupling, is an interactive tool designed to measure coupling in an Ada program which may be spread across multiple files. The results of SANAC'S analysis can be used to measure the quality of the design of the program and to provide quantitative information about the degree of coupling to help in the assignment of resources to be used in software maintenance.

3.1. Design description of tool

SANAC's design stems from the methodology used for development, which is data structured system design or DSSD [Pressman 87]. In this section, the design of the tool will be discussed. The design description will be divided as follows: data description, derived program structure, and interfaces within structure.

3.2. Data description

The data in question are Ada source files. These files may contain procedures, functions, package specifications and/or package bodies. There are a few stipulations which should be noted. First, if a main procedure is in a file, there can be no external packages included in this file; the package specification and/or body must be in a different file altogether. Secondly, the files may include tasks; however, task bodies are not analyzed for "entries" or "selects".

3.3. Derived program structure

To maintain cohesion, the operations to be performed are separated into various procedures and functions, where one procedure/function performs one operation. SANAC is divided into three areas: user interface, coupling analysis and coupling results, as indicated by Figure 1.

3.3.1. User interface

The user interface, is responsible for prompting the user for all input which is necessary for operation. Menu screens, and most of the other information which is part of the user interface, are contained in files which are read and printed verbatim. This procedure for displaying information to the user was chosen to simplify any updates to the interface which may be necessary.

3.3.2. Coupling analysis

Coupling Analysis has several duties to perform as Figure 2 illustrates: Source File Evaluation, Main File Analysis, if necessary, and any Additional Analysis (i.e., separates, packages). This section of the SANAC program will handle each of these processes in turn before continuing on to Coupling Results.

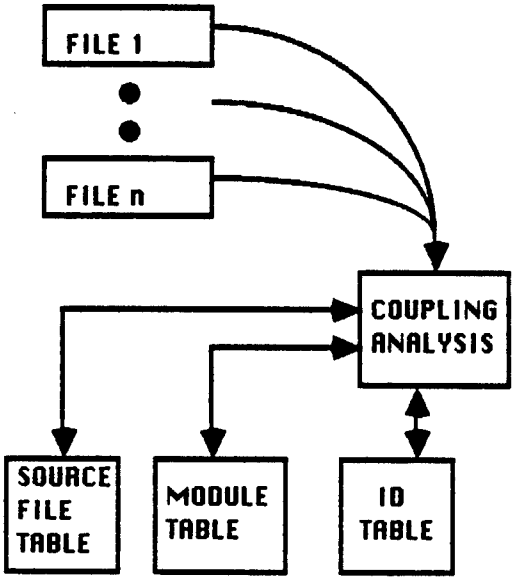


Figure 2: Coupling Analysis

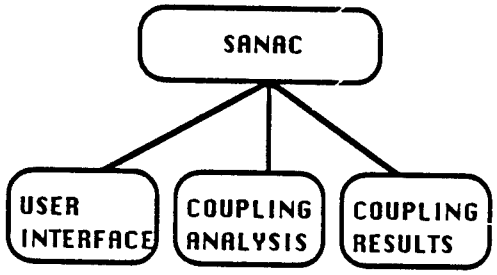


Figure 1: Top-down View

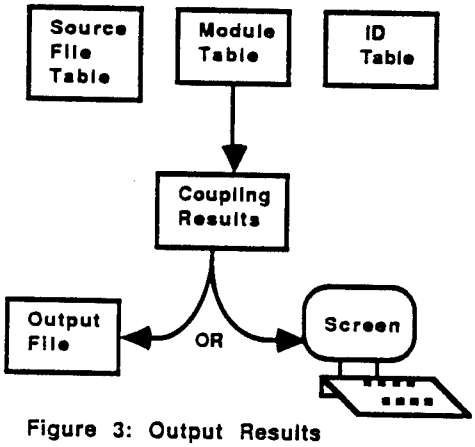


Figure 3: Output Results

After the input files have been received, they are evaluated to determine each file's type (i.e., separate, main, package specification and/or body). If a file does not fall into one of the above categories, then it is labeled as "Not An Ada File".

for distinct reserved words, primarily in the beginning of the files; however some files may require further evaluation. For example, if the reserved word "package" precedes "procedure" -- excluding comments -- then the current file is evaluated to be a package specification or package body, depending upon further searches. Also, if the word "separate" precedes "procedure", then the file is evaluated to be a separate file, and if "procedure" precedes "separate" and "package" then the file is evaluated to be a main file (i.e., it includes the main procedure for the system).

Now that the source file evaluation is complete, and the type of each file is known, the main file will be evaluated first, provided that a single "main" program exists. On the other hand, if multiple main files exist, then the last one entered is the one to be analyzed. In analyzing the main file, if any procedure or function declarations are encountered, then they are analyzed for coupling. This is done until the "begin" block for the main procedure is found, at which time all procedures and/or functions called by the main procedure are labeled so that they too will be evaluated. Eventually, the end of the main procedure will be reached, at which time the Additional Analysis phase will commence.

NOTE: In the special case where a main file is not entered, a "STUB" will be created which will call all modules in all files entered. This feature is included for the case in which the user wishes to determine the coupling that exists among modules which may not include a main procedure. Thus it is possible to analyze Ada code that has not yet been incorporated into a single monolithic program.

During the Additional Analysis phase, all files that are labeled to be analyzed for coupling are analyzed. These files are analyzed similarly to the analysis of the main file, only in more depth. In the main procedure analysis, if any procedures or functions are declared, then they are analyzed in the same manner as in Additional Analysis; however when the begin block of the main procedure is encountered, the only concerns are procedure or function calls.

When a procedure or function is encountered, any parameters and their modes are stored in an Id-Table corresponding to this module. With each parameter, there is a possibility of an instance of weak coupling. For every parameter that is not referenced in the module, there is an instance of weak coupling. Next, the local variables, if any, are stored in the same Id-Table as the parameters corresponding to this module. If any other procedure or function declarations are encountered before the "begin" block of the current module, then they are analyzed for coupling just as the current module is being analyzed.

When the reserved word "begin" is found, signifying the begin block of the current module, the following strategies for determining coupling are used.

Strategy for Data Coupling: If a parameter appears on the right hand side of an assignment statement, then a counter that represents data coupling for the current module is incremented.

Strategy for Stamp Coupling: If a parameter appears on the left hand side of an assignment statement, then a counter that represents stamp coupling for the current module is incremented.

Strategy for Control Coupling: If a parameter appears as part of an "if", "while", "exit", "case", or "for" condition, then a counter that represents control coupling is incremented.

Strategy for External Coupling: If a variable is referenced in a module, and that variable does not appear in the Id-Table for the current module (i.e., not a parameter or local variable), then a counter that represents external coupling is incremented.

Strategy for Content Coupling: If a statement which explicitly raises a user-defined exception is encountered in a module, and the exception handler is not located in the same module, then a counter that represents content coupling is incremented.

The above strategies analyze the coupling in a module until the reserved word "end" is found, and it is followed by the current module's name.

3.3.3. Coupling results

By this time all files have been evaluated, and, the results of the analysis can be printed. If a main procedure file was found, then the coupling that exists between the main procedure and all the modules that it calls is printed. The modules called by the main procedure are then printed along with any coupling involved in modules called by these modules.

3.3.4. Interfaces within structure

The interface which ties this system together is a series of tables which the file names, module names, coupling information, and other pertinent information can be stored. These tables are used throughout execution. First, during the Source File Evaluation, the file names and module names are stored in the source file table and the module table respectively. Secondly, during Main Procedure Analysis coupling information, and more module names are stored in the module table. As parameters and local variables are encountered, they are stored in the Id-Table corresponding to the module. Lastly, when all analysis is complete, the module table which includes the module names and the coupling information is used by Coupling Results to output the results as depicted in Figure 3.

4. METHODOLOGY AND ENVIRONMENT FOR DEVELOPMENT

The design methodology utilized during the design phase in the development of the System for ANalyzing Ada Coupling (SANAC) was Data Structured System Development (DSSD). This method represents the SANAC software requirements by focusing on data structures rather than data flow.

All development was done on an AT&T 3B2/400 computer running AT&T System V UNIX where a pre-release version of an Ada compiler was utilized. The main problem with the environment was that it was not an Ada development environment, and the machine was rather slow in compilation; one particular file took approximately 50 minutes to compile. A UNIX shell script made compiling the 27 Ada source files a relatively easy task. Another problem was the lack of a debugger. The references [Ada83] and [Cohen86] were used extensively. One of the source code files, parse.a, was taken from the Ada Repository.

Throughout design and development phase measures have been taken to insure that SANAC would be system independent, and thus far SANAC has been successfully ported to a Sun work station under Verdix Ada.

SANAC was tested using two methods which aid in providing a systematic approach to testing. The first approach was black box testing which involves conducting tests to demonstrate that each specified function of the product is fully operational. The second method was white box testing which is concerned with whether all internal operations perform to specification and all internal components have been adequately exercised.

5. USE OF TOOL

5.1. System flow

By entering the system command at the prompt: $sanac, the user is then allowed to progress through the system interactively.

SANAC accepts as input the name(s) of Ada source files which may contain a main procedure, a separate procedure or function, or a package specification and/or body. Entry of the file name(s) is allowed only when the Coupling Measures option is selected from the Main Menu. Other available options include a Coupling Description which defines the types of coupling to be measured by SANAC, a Help Session which briefly describes the system and available options, and Exit which returns the user to the underlying operating system.

After the file name(s) are entered, each file is evaluated to determine if it contains one of the following types:

1) main procedure,
2) separate procedure or function, or
3) package specification and/or body.

The results of the evaluation are returned for review. If a file evaluated is not of a type previously described, it is labeled "Not An Ada File".

In the case where all the file(s) entered are not Ada source files, no coupling will be determined and control will be passed to the Main Menu for selection of one of the options mentioned above. In the case where one or more of the files entered are not Ada files, these particular files will be ignored and only the files containing Ada source code will be analyzed. if a main procedure is not found in any of the files entered, an informative message states that a main procedure called "STUB" will be created so that the file(s) entered can be measured for coupling.

Based on the type of file(s) entered, SANAC execution can continue through one of two possible paths. If a main procedure is found in one of the files, this file is opened first for analysis. Any additional files entered will be analyzed only if the main procedure references a package or separate procedure or function which they may contain. During the analysis of the main procedure, two important steps are taken in what is called the coupling preliminaries phase. They include distinguishing the system-defined and user-defined packages which are WITH'ed and storing global variables. The output of the number of WITH'ed packages is based on an idea presented in [Perkins86]. If a main procedure is not found, as mentioned earlier, a "STUB" is created to reference each procedure, function, or package found in the file(s) entered. This phase is called additional analysis.

After completion of one of the above phases, coupling is determined for all procedures, functions, or packages that reference other procedures, functions, or packages. During this process, any parameters or local variables are stored if a procedure or function is found. The strategies used to determine each type of coupling mentioned earlier in the paper are then implemented. If there are no procedures or functions, then coupling cannot be determined.

When all files have been analyzed, a statistical report is available in tabular form. This report lists each module, and the coupling found in each procedure or function referenced by that module. If a module does not reference any other procedure or function, an appropriate message is printed. Also, the number and names of the system-defined and user-defined packages are printed. The user has the option of directing this statistical information to the screen or to an output file. This option is available on the Coupling Measures Output Specification screen.

After the statistical information has been reviewed or directed to an output file, the user is provided with the Main Menu screen with the options described earlier.

5.2. System walkthrough

A system walkthrough will now be provided. Only the file entry, file evaluation, and output specification screens will be presented, along with the resulting output. The files used in this walkthrough can be found in Appendices A - F. References can be made to the information in paper to facilitate understanding the process that takes place during the walkthrough. The files and their contents were kept simple to facilitate the calculation of each type of coupling. They were included to give an example of each type of coupling to be measured by SANAC.

Test Case #1 contains the following files:

in Appendix G. The resulting output for Test Case #1 can be found in Appendix H.

Test Case #2 contains the following files:

three package specification/body,
one separate procedure, and
one separate function.

Some of the screens mentioned above for Test Case #2 are in Appendix I. The resulting output for Test Case #2 can be found in Appendix J.

6. FUTURE RESEARCH

Because simple connectivity and low coupling is desired between modules within a system, SANAC can be used to develop quality standards by measuring coupling metrics. SANAC can be run on various types of existing code, each with different levels of quality already known, with the results for each measured and retained along with the quality of the code. These results and the quality associated with these results can serve as quality measuring standards. Newly developed code can be run through the SANAC system and the measurements retained. These results can then be compared to the standard results obtained from the existing code, for which the quality was known. If the results of the new code are similar to the standard results of a piece of existing code, then the new code can be determined to be of quality similar to that code which its results are similar.

By setting up quality standards of existing code and comparing newly developed code results to these standards, SANAC can be used as an automated tool to develop models of quality code. It will be used together with existing tools to perform metric analysis of code in several different languages as part of a major research project. See [Perlis81] and [Leach89]. The TAME system being developed at the University of Maryland at College Park by V. R. Basili and H. Dieter Rombach [Basili87] has a somewhat different focus; it provides an environment for tailoring automatic metrics collection to the goals of a particular software development installation and is Ada-specific.

REFERENCES

[Ada83] "Reference Manual for the Ada Programming Language", ANSI/MIL- STD-1815A, United States Department of Defense, Washington, D.C., February 1983

[Basili87] Basili, V. R., and H. D. Rombach, "TAME: Tailoring an Ada Metrics Environment", Proceedings of the Joint Ada Conference, Washington, D.C., March 16-19, 1987, 318-325.

[Cohen86] Cohen, N., "Ada as a second language", McGraw-Hill, Inc. (1986), New York

[Conte86] Conte, S. D., Dunsmore, H. E., Shen, V. Y., "Software Engineering Metrics and Models", The Benjamin/Cummings Publishing Company, Inc. (1986), California, pp.102-109

[Leach89] Leach, R. J., "Software Metrics Analysis of the Ada Repository", Proceedings of the Seventh Annual National Conference on Ada Technology, Atlantic City, NJ, March 13-16, 1989, pp. 270-279

[Perkins86] Perkins, J. A., Lease, D. M., and Keller, S.E., "Experience Collecting and Analyzing Automatable Software Quality Metrics for Ada", Proceedings of the Fourth Annual National Conference on Ada Technology, Atlanta, GA, March 19-20, 1986, pp. 67-74

[Perlis81] Perlis, A., Sayward, F., Shaw, M., "Software Metrics: An Analysis and Evaluation", The MIT Press (1981), Massachusetts, pp. 252-259

[Pressman87] Pressman, R., "Software Engineering, A Practitioner's Approach", McGraw-Hill, Inc. (1987), New York, pp. 232-233

BIOGRAPHIES

Ronald J. Leach is a Professor in the Department of Systems and Computer Science at Howard University. He has a Ph.D. in Mathematics from the University of Maryland at College Park and an M.S. in Computer Science from Johns Hopkins University. He is the author of more than 30 technical publications. His research interests include software engineering, computer graphics and concurrent computing.

Keith S. Bagley received his B. S. degree in Mechanical Engineering from Cornell University in Ithaca, N.Y. in 1986. Currently, he is a Graduate Research Assistant at Howard University in Washington, D. C. where he is part of a team developing User Interface Evaluation Systems technology. He will receive a M. S. degree in Computer Science in December, 1989.



David E. Butler received his B.S. Degree in Computer Science from Xavier University of Louisiana in 1988, and his M.S. Degree in Computer Science from Howard University in Washington, D.C. in 1989. Currently he is a Member of Technical Staff at AT&T Bell Laboratories where he is part of a development group in Packet Switching Technology.



Iva L. Brown is a Member of Technical Staff of the LFACS Assignment and Reports Development District at Bell Communications Research in Piscataway, NJ. She received her Bachelor of Science Degree in Computer Science from Southern University in Baton Rouge, LA. and her Master's of Computer Science in July, 1989 from Howard University in Washington, DC.

Patrick W. Spann is a Member of Technical Staff of the Customer Services System Development (CSSD) group at Bell Communications Research (Bellcore) located in Piscataway, NJ. He received his B.S. Degree in Computer Science from Southern University in Baton Rouge, LA. in December 1987 and obtained his M.S. in Computer Science from Howard University in Washington, D.C in August, 1989.

Acknowledgement

The research of Ronald J. Leach and Keith S. Bagley was partially supported by the Maryland Procurement Office. Research of Ronald J. Leach was also partially supported by the Army Research Office and the Naval Surface Warfare Center.

APPENDIX A

```
-----* FILE: sample.A -----*

with Text_IO;
with Toy_Data;        --* file toy.A --*
with Case_Changer;    --* file case.A --*
with Stack_Code;      --* file stk_code.A --*

PROCEDURE Sample_Proc IS

     TYPE Integer_List_Type IS ARRAY (1..10) OF Integer;
     SUBTYPE Buffer IS String(1..80);

     i,x,pt1       : Integer;
     j,k,pt2       : Natural;
     line_size     : Natural;
     temp1,temp2   : Integer;
     val1,val2     : Integer;
     outside       : Integer;
     external      : Integer;

     Top           : Integer := 0;
     Stack_Object  : Integer_List_Type;
     Inline        : Buffer := (others => ' ');

--* file defunct.A --*
Function Defunct (Val, Proc : Integer)
                  Return Integer IS Separate;

--* file seps.A --*
Procedure Seps (Temp  : IN OUT Natural;
                Value : IN Integer)
                        IS Separate;

Procedure Triangle ( x,z : IN Integer;
                        j, l : IN OUT Integer) IS
   i    : Integer;
   y    : Natural;

   BEGIN

      y := z + 5;
      i := l + j;

      Seps(y,i);

      IF y < 7 THEN
         l := j + 3;
      END IF;
      IF l > 6 THEN
         j := 10;
         y := i * j;
         PUT("VALUE OF Y ->");
         PUT(y);
         NEW_LINE;
         raise constraint_error;
      END IF;

END Triangle;
```

```
Procedure Restore (xpoint, num1 : IN Integer;
                   num2 : IN  OUT Integer) IS
      k : Integer;

     BEGIN

       FOR k in xpoint..20 LOOP
           IF num1 = k THEN
              num2 := num1;
              PUT("NUM1 IS EQUAL TO XPOINT");
           END IF;
       END LOOP;

END Restore;

  BEGIN

     GET(pt1,temp1,temp2);
     Toy(i,pt1,j,k,pt2);
     IF pt2 /= 0 THEN
        PUT("POINT #2 = ");
        PUT(pt2);
        Seps(pt2,10);
     END IF;

     Triangle(temp1,temp2,val1,val2);
     IF val1 = 10 THEN
        Defunct(val1,val2);
        Restore(val1,i,temp2);
     END IF;

     FOR I in 1..10 LOOP
         GET(x);
         Push(x,Stack);
     END LOOP;

     FOR I in 1..10 LOOP
         Pop(x,Stack);
         PUT(x);
         NEW_LINE;
     END LOOP;

   WHILE NOT END_OF_FILE LOOP
     GET_LINE(INLINE,Line_Size);
     Convert;
     PUT_LINE(INLINE);
   END LOOP;

EXCEPTION
   WHEN Numeric_Error =>
        PUT(" Numeric Error Occurred -- Program Terminates");

   WHEN Constraint_Error =>
        PUT(" Constraint Error Occurred -- Program Terminates");

END Sample_Proc;
```

APPENDIX B

```
-----* FILE: toy.A -----*

Package Toy_Data IS

  Procedure Toy ( k, lpoint : IN Integer;
                  l,p, fpoint : IN OUT Natural);

End Toy_Data;

Package Body Toy_Data IS

  Procedure Toy ( k, lpoint : IN Integer;
                  l, p, fpoint : IN OUT Natural) IS

           x, y : Natural;

           BEGIN

            WHILE fpoint <= 10 LOOP
               lpoint := fpoint + 5;
               i := p * x;
               y := k + 7;
               fpoint := fpoint + 1;
            END IF;

   END Toy;
END Toy_Data;
```

APPENDIX C

```
-----* FILE: case.A -----*

PACKAGE Case_Changer IS

SUBTYPE Buffer IS String(1..80);

PROCEDURE Convert (Text : in out Buffer; Lastc : in Integer);

END Case_Changer;

PACKAGE BODY Case_Changer IS

PROCEDURE Convert (Text : in out Buffer; Lastc : in Integer) IS

     --****************************************--
     --* L O C A L    D E C L A R A T I O N S *--
     --****************************************--

Offset : CONSTANT := character'pos('a') - character'pos('A');

Char : CHARACTER;
Position : INTEGER := 1;
```

```
BEGIN -- Convert

                    --*****************************************--
                    --* Scans a line of text and changes      *--
                    --* all upper case characters to lower    *--
                    --* for the length of the string.         *--
                    --*****************************************--
                    FOR Position IN 1..Lastc LOOP

                        Char := Text(Position);    -- if uppercase

                        IF Char IN 'A'..'Z' THEN  -- change to lower
                           Text(Position) :=
                           character'val(character'pos(Char) + Offset);
                        END IF;

                     END LOOP;

        RETURN;
    END Convert;
END Case_Changer;
```

APPENDIX D

```
-----* FILE: stk_code.A -----*

Package Stack_Code is
  Procedure Push (Item : IN Integer; Stack_Object : IN OUT Integer_List_Type);
  Procedure Pop (Item : OUT Integer; Stack_Object : IN OUT Integer_List_Type);
End Stack_Code;

Package Body Stack_Code is
  Procedure Push (Item : IN Integer; Stack_Object : IN OUT Integer_List_Type) IS

     BEGIN
       Top := Top + 1;
       Stack_Object(Top) := Item;
     END Push;

  Procedure Pop (Item : OUT Integer; Stack_Object : IN OUT Integer_List_Type) IS

     BEGIN
       Item := Stack_Object(Top);
       Top := Top - 1;
     END Pop;
END Stack_Code;
```

APPENDIX E

```
-----* FILE: defunct.A -----*

Separate (Sample_Proc)

Function Defunct ( Val, Proc : Integer )
                 Return Integer IS
  J : Integer;
```

```
BEGIN

     IF Proc /= 0 then
        J := Val / Proc;
        External := J;
        raise numeric_error;
     END IF;

     return (J);

END Defunct;
```

APPENDIX F

```
-----* FILE: seps.A *-----

Separate (Sample_Proc)

Procedure Seps (Next : IN OUT Natural;
                Other : IN Integer) IS

 First  : Integer;
 Follow : Integer;

 BEGIN

    First := Other;
    Next  := Outside;

    IF Next = First THEN
       Follow := Other;
    END IF;

END Seps;
```

APPENDIX G

Source File Evaluation

We have evaluated your file(s) to be the following:

sample.A	MAIN
toy.A	PACKAGE SPEC & BODY
case.A	PACKAGE SPEC & BODY
stk_code.A	PACKAGE SPEC & BODY
seps.A	SEPARATE
defunct.A	SEPARATE

APPENDIX H

S A N A C

M O D U L E C O U P L I N G

sample_proc

	WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT
toy	2	4	1	1	1	0
seps	0	2	1	1	0	0
triangle	1	5	2	1	0	1
defunct	0	2	0	1	1	1
restore	0	1	1	2	0	0
push	0	1	1	0	1	0
pop	0	1	1	0	1	0
convert	0	1	1	1	0	0

toy

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

convert

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

push

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

pop

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

seps

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

defunct

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

triangle

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT
0	1	1	1	0	0

seps

restore

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

NUMBER OF SYSTEM DEFINED PACKAGES: 1

SYSTEM DEFINED PACKAGES:

    text_io

NUMBER OF USER DEFINED PACKAGES: 3

USER DEFINED PACKAGES:

    toy_data
    case_changer
    stack_code

APPENDIX I

Source File Evaluation

We have evaluated your file(s) to be the following:

toy.A	PACKAGE SPEC & BODY
case.A	PACKAGE SPEC & BODY
stk_code.A	PACKAGE SPEC & BODY
seps.A	SEPARATE
defunct.A	SEPARATE

NOTE: No Main Procedure Entered.
A Stub Has Been Created.
Max 20 Procedures Will Be Analyzed.

PAGES 560-589 OMITTED

APPENDIX J

S A N A C

M O D U L E C O U P L I N G

STUB

	WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT
toy	2	4	1	1	1	0
convert	0	1	1	1	0	0
push	0	1	1	0	1	0
pop	0	1	1	0	1	0
seps	0	1	1	1	0	0
defunct	0	2	0	1	1	1

toy

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

convert

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

push

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

pop

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

seps

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

defunct

WEAK	DATA	STAMP	CONTROL	EXTERNAL	CONTENT

THIS MODULE DOES NOT REFERENCE ANY OTHER MODULES

NUMBER OF SYSTEM DEFINED PACKAGES: 0

SYSTEM DEFINED PACKAGES:

NUMBER OF USER DEFINED PACKAGES: 0

USER DEFINED PACKAGES:

TECHNIQUES FOR THE ANALYSIS OF PORTABILITY
A STUDY OF THE AFATDS CONCEPT EVALUATION CODE

C. Alan Burnham, Peter N. Ho, and Harry F. Joiner

Telos Corporation, 55 N. Gilbert Street, Shrewsbury, NJ 07702

Abstract

This study analyzes the porting of a large Ada system from a bare machine platform to a new target environment that includes support software such as the operating system, graphics kernel, and communications software. The options of porting the existing operating system and support software or modifying the remaining code to accommodate the new support software were evaluated for the level of effort required. Factors for the quality of the code and ease of portability were incorporated. Because of the size of the system, several aspects of the analysis needed to be automated.

INTRODUCTION

In the rapidly developing environment of today's computer industry, the need to retarget software systems to new hardware or to develop software before the final hardware is chosen presents various technical challenges. A principal reason for the Department of Defense reliance on Ada is to increase the portability of software from one hardware/software platform to another. Larger systems will be ported, requiring the analysis of the trade-offs between porting an established system or designing a new system that targets the new platform. One of the factors in this analysis is the level of effort to port a large software system to a new platform. This study illustrates the problems such an analysis can encounter and some techniques that were used to resolve them.

The Advanced Field Artillery Tactical Data System (AFATDS), a large U.S. Army Command and Control (C^2) system, began development with the Concept Evaluation Phase (CEP) on the developer's prototype hardware. The CEP system included development of a Random Access Memory (RAM)-based Operating System (OS), database, Graphical Kernel System (GKS), and communications software for Local Area Network (LAN) and modem support to support the application software. For Segment 1 of the phased development, AFATDS was to be ported or retargeted to the Army Tactical Command and Control System (ATCCS) Common Hardware and Software (CHS). The CHS includes its own virtual memory-based UNIX OS, database, LAN, modem, and GKS.

The porting analysis considered 24 options based on what features of the CEP code are ported. The OS options considered in this study included: (1) porting the CEP OS in RAM-based form (assuming enough RAM would be available), (2) porting the CEP OS, modifying it to provide a virtual memory capability, and (3) using the CHS UNIX OS and modifying the remaining CEP code to run on it. The options for each of the other features (i.e., LAN, GKS, and modem) included porting the CEP feature to CHS or replacing the CEP feature with the corresponding CHS feature and modifying the remaining CEP code. This study will treat the porting of the LAN, GKS, and modem software as a single option to simplify the example. The porting of the database was not considered at the time of the study.

While it is important to obtain as accurate an estimate as possible of the relative levels of effort required, it is just as essential to the decision process to obtain the actual levels of effort required to perform each of the porting options .

METHODS

The standard approach to evaluate software portability involves a direct analysis of the code by cognizant software engineers to ascertain the amount of code and design that must be changed and the difficulty of making those changes. However, this study presented major problems because of the size of the software (1.5 million noncomment, nonblank Lines of Code ([LOC]) continued in approximately 7,000 modules). In order to manage the volume of information, automated software tools were used to augment the human analysis. The following steps were used:

1. A preliminary review of the code was performed to estimate the approach to the analysis.

2. The basic equations were formulated and the required information identified.

3. Tools were obtained or developed to collect the needed information, and additional human analysis was performed.

4. The level of effort was estimated for a full-scale development of the system.

5. The level of effort was estimated for each of the options and recommendations were made.

Although this analysis addresses the effort involved in porting the software, it does not address system-unique features which may impact the actual performance of the ported software.

Background

The effort to analyze AFATDS portability began with an analysis of the options available for the porting task. The Analysis yielded several ways in which the AFATDS CEP code could be ported to the CHS:

1. The CEP OS could be ported to a bare CHS machine.

2. The CEP OS could be ported to a bare CHS machine and modified to provide a virtual memory capability.

3. The CEP software could be modified to run on the CHS UNIX OS.

4. For each of the above OS options:
   a. the CEP features (GKS, LAN, and modem) could be ported to the CHS
   b. The CEP software could be modified to use the CHS features (GKS, LAN, and modem)

The option tree is presented in Figure 1. Although the original analysis actually considered other alternatives, those cited above and illustrated in Figure 1 are sufficient to demonstrate the techniques used.

Analysis of the code indicated that worst-case estimates for the change traffic are (1) 10 percent of the GKS code and (2) 20 percent of the LAN and modem code. The major effort required to modify the AFATDS GKS software appears to be the renaming of the functions and data types since both the CEP code and the CHS GKS are ANSI standard.

Analysis

The basic equations used for the level-of-effort estimates were adapted from Barry Boehm's Constructive Cost Model (COCOMO) [1]. We determined that for porting activities, the Basic COCOMO estimate for maintenance is the most appropriate. In order to fit our application we adapted the equation as:

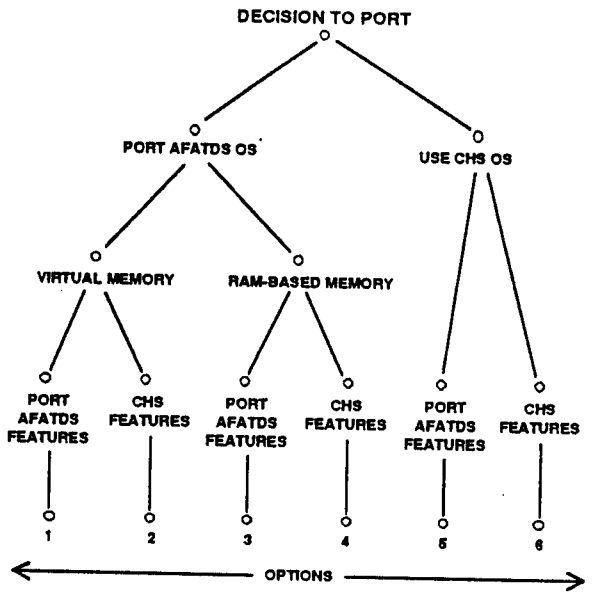


Figure 1. **The AFATDS Portability Option Tree**

(1) $$EFFORT = CT*DEV_EFF$$

where EFFORT is the effort in man-months required to port the software, CT is the change traffic (i.e., the fraction of the Delivered Source Instructions [DSI] in the product that must be modified during the porting process); and DEV_EFF is the effort in man-months required to develop the total system. In a project of this size, CT must be estimated by automated means.

The development effort (DEV_EFF) can be estimated by the standard exponential equation:

(2) $$DEV_EFF = PROD*COEF*(KDSI)^{EXP}$$

where PROD is a constant based on productivity and the development environment, COEF is the COCOMO coefficient which is a function of the development method; KDSI is thousands of DSI; and EXP is the corresponding COCOMO exponent. AFATDS contained both Ada and assembly code. Therefore, the study used the Ada COCOMO model for the Ada Code and the semidetached intermediate COCOMO model for the assembly code [1, pp 78-80]. COEF is 2.8 and EXP is 1.04 for the Ada code. COEF is 3.0 and EXP is 1.12 for the assembly code.The productivity factor PROD will be related to the quality of

the code (QUAL), the porting environmental factors (ENVIR), and an assessment of the portability of the code (PORT), and will be equal to the product of the three factors. The defining equation is:

(3) PROD = QUAL*ENVIR*PORT

The environmental factor ENVIR chosen was a constant value of 0.6, regardless of the option chosen, since we assumed the porting will be an efficient process and the developer well prepared. The values of QUAL and PORT will be discussed later.

TOOLS

In order to assess the areas of the AFATDS CEP code that required modification and to determine the overall quality of the code, two code analysis tools were employed: the Source Code Analysis Tool (SCAT), developed by TELOS; and ADAMAT™, a commercial Ada measurement tool developed by Dynamics Research Corporation (DRC).

SCAT

SCAT was used to search the source code for code that would be affected by each of the porting options. The SCAT results provide the information necessary to estimate the change traffic in each software subsystem by identifying the interfaces between subsystems and calls to affected modules within a subsystem. SCAT also provided an independent count of the noncomment, nonblank LOC. It does not account for software threads that may have architectural implications. Detailed results of the SCAT analysis are contained in [2].

An example of the approach used is provided by the OS analysis. First, the modules in the OS were identified and the LOC counted. Then the calls to those modules in the remainder of the code were identified and counted since each of these calls would be effected by a change in the OS. Additional references to hardware or system-dependent calls were identified and added to the code to be changed during the porting operation.

ADAMAT

ADAMAT scans Ada source code to identify over 400 Ada constructs, and collates these into a set of more than 200 metrics to evaluate the coding practices used to write the program units. The scores range from 0.00 to 1.00 with higher scores preferred. It is a static tool and does not measure the performance of the code. The six criteria at the top of the set of metrics are Anomaly Management, Independence, Modularity, Self-Descriptiveness, Simplicity, and System Clarity. ADAMAT can also be customized to provide specialized reports such as the Machine and System Dependency Rating needed for this analysis. A more complete discussion of the tool and these analyses is provided by [3].

The quality factor QUAL was determined by use of ADAMAT. The ADAMAT scores were very consistent across the AFATDS subsystems and could, therefore, be averaged to provide the following quality factors:

Anomaly Management	0.65
Self-Descriptiveness	0.78
Simplicity	0.62
System Clarity	0.86
Overall Average	0.73

The value of QUAL is the inverse of this average (1.37) since the productivity will be inversely related to the quality of the code.

The remaining criteria averages from the ADAMAT analysis were combined with the specialized dependency report to produce the following portability factors:

Independence	0.99
Modularity	0.84
Dependency Rating	0.99
Overall Average	0.94

The portability factor PORT, like the QUAL factor, is inversely related to this overall average and has a value of 1.06.

RESULTS OF THE AFATDS CASE STUDY

Overall Level of Effort

The first results required are the estimated level of effort for the Ada development and the assembly language development. Equations 2 and 3 were combined and the results appear in Table 1. The SCAT data re-ported 1,570,574 lines of Ada and 14,131 lines of assembly language for a total of 1,584,705 LOC. Table 1 shows the values of the constants used. For the Ada code, the quality and portability

Table 1. **Estimate of Overall Development Level of Effort**

	QUAL	ENVIR	PORT	PROD	COEF	KDSI	EXP	EFFORT M-M
ADA	1.37	0.60	1.06	0.87	2.8	1571	1.04	5161
ASSEMBLY	1.37	0.60	1.06	0.87	3.0	14	1.12	51
TOTAL						1585		5212

factors were derived from the ADAMAT scores. The values of EXP (1.04) and COEF (2.8) were chosen to conform to recent Ada project analysis [4] and to reflect current understanding of the developer. The actual development effort, in this case, included an excessive amount of effort to resolve problems related to the immaturity of the Ada development environment at the time the project was started (1984) and would not reflect the current capabilities of the developer.

The assembly code effort was calculated using Equation 2 with the Intermediate COCOMO coefficients for semi-detached mode [1, p.117], i.e., ENVIR equal to 0.6, COEF equal to 3.0, and EXP equal to 1.12. The values for QUAL and PORT are assumed to be similar for the assembly language code because of the consistency of the ADAMAT scores within the Ada code. These factors are most closely related to the application of software engineering principles, which are as applicable to the assembly code as to the Ada code.

Estimates for the Options

The required data is now available for Equation 1 to be applied for each of the options in Figure 1. The results are presented in Table 2. Equation 1 clearly indicates that the porting effort is linearly related to the Change Traffic (CT).

SUMMARY

The decisions regarding the porting of the CEP code to the CHS involved the consideration of numerous factors other than the level of effort for the porting options outlined here. Because of the limited RAM (relative to the CEP prototype equipment), it was almost certain that the CEP OS would have to be converted to a virtual disk-based system if it were to be ported to CHS. This option required approximately five times the effort of the other options. In contrast there is relatively little difference in effort to port the CEP GKS, LAN and modem software versus using the CHS utilities. No estimate was made of the impact on the actual performance of the ported code.

The techniques presented can be used in the analysis of any large porting effort. As more information becomes available from porting projects, the predictive equations can be refined to provide more accurate estimates. The results presented here have not been substantiated by completed porting projects.

ACKNOWLEDGMENT

This work was performed under Contract Number DAAB07-87-C-B015, for the U.S. Army Communications-Electronics Command Center for Software Engineering, in support of the Program Manager Office, Field Artillery Tactical Data Systems at Fort Monmouth, NJ.

REFERENCES

[1] Barry W. Boehm, Software Engineering Economics. Prentice-Hall, Inc. Englewood Cliffs, NJ, 1981

[2] "AFATDS Portability Summary Report," TELOS Report TFSFMO 89-0410, 1989

[3] Harry F. Joiner, "Metrics Analysis of Ada Programming Practices on AFATDS, a Large Ada Project." Technical Presentations to the 43rd International AFCEA Convention, 1989

[4] Barry W. Boehm, "Software Cost Estimation Using COCOMO and Ada COCOMO." Lecture notes, 1989.

Table 2. **Level-of-Effort Estimates for the Six Porting Options**

OPTION	ADA MODIFIED LOC	ASSEMBLY MODIFIED LOC	TOTAL MODIFIED LOC	ADA CT PERCENT	ASSEMBLY CT PERCENT	TOTAL CT PERCENT	ADA EFFORT M - M	ASSEMBLY EFFORT M - M	TOTAL EFFORT M - M	RATIO TO MIN	OPTION
1	59785	3082	62867	3.81	21.81	3.97	204.8	11.1	215.9	11.3	1
2	64583	9201	73784	4.11	65.11	4.66	240.3	33.2	273.5	14.3	2
3	6463	3082	9545	0.41	21.81	0.60	31.1	11.1	42.2	2.2	3
4	11261	9201	20462	0.72	65.11	1.29	66.6	33.2	99.8	5.2	4
5	5865	0	5865	0.37	0.00	0.37	19.1	0.0	19.1	1.0	5
6	10663	6119	16782	0.68	43.30	1.06	54.7	22.1	76.7	4.0	6

BIOGRAPHIES



C. Alan Burnham

C. Alan Burnham is a Senior Systems Engineer with Telos at Ft. Monmouth, NJ. He is currently the Technical Task Leader for Ada Engineering Support.

Mr. Burnham has worked for over 20 years in the development and support of software-intensive military systems. He has performed software development, software quality evaluation, systems analysis/ engineering, software acquisition management, and program management. Mr. Burnham's recent experience has been primarily in developing specifications, defining requirements, and testing software for mission-critical defense systems. His earlier experience was in the development, validation, and application of large-scale, high-resolution combat simulations.

Mr. Burnham received his BA in Mathematics from Augustana College.



Peter N. Ho

Peter N. Ho is a Senior Software Engineer in the Advanced Software Technology group of Telos Corporation. His main areas of interest are CASE tools, Software Reverse Engineering Technology, Ada Software Reuse Technology, and large-scale software metric analysis.

Mr. Ho received his BA in Mathematics from Lincoln University, MS in Mathematics from Lehigh University and MSE in Computer and Information Science from University of Pennsylvania. Mr. Ho has over 10 years of experience in the development of software for both Government and Commercial applications.



Harry F. Joiner

Harry F. Joiner joined Telos as a software engineer in August 1986, working on the Firefinder field artillery location radar. He assumed his current position as Manager of Software Metrics in March 1988. Telos is the largest software engineering support contractor to CECOM CSE.

Before joining Telos, Dr. Joiner served as a consultant on project management and digital signal processing to various oil companies. He has over 20 years of experience in mathematical modeling, digital signal processing, engineering, and project management.

Dr. Joiner has a BA in Mathematics and Chemistry and an MS and Ph.D. in Mathematics. He has served on the faculties of the University of Massachusetts and Texas Christian University.

Experience Using Automated Metric Frameworks in the Review of Ada Source for AFATDS

Stan Levine
Chief, Technical Management Division
OPM FATDS
Building 457
Fort Monmouth, NJ 07703

J. D. Anderson and J.A. Perkins
Dynamics Research Corporation
Systems Division
60 Frontage Road
Andover, MA 01810

ABSTRACT

This paper discusses the application of automated metrics analysis to the Ada software developed during the concept evaluation phase of the Advanced Field Artillery Tactical Data System (AFATDS). The purpose of the analysis was to provide the Army, as a part of their evaluation of the quality of the AFATDS software, visibility into the Ada source by quantifying specific Ada language features and programming practices applied in the development of the delivered software.

Analysis of AFATDS software consisted of applying two automated, hierarchical, Ada-specific software metric frameworks to 1.5 million text lines (.5 million Ada statements) of Army-supplied Ada source, where the first framework assesses the level of adherence to accepted software engineering principles, and the second assesses inherent complexity by measuring the number of occurrences of each feature of the Ada language.

The analysis included: 1) automated calculation of metric scores for the entire supplied source; 2) metric-based human analysis to identify a) characteristics of source that augment quality, b) characteristics of the source that attenuate quality, c) programming standards actually applied during development, and d) under-utilized features of the Ada language; 3) comparison of the programming practices observed for AFATDS to those found in other Ada software developed for the Army, Navy, or Air Force, and 4) reporting of the metric findings to the Project Manager, Field Artillery Tactical Data Systems (PM FATDS), and the developer.

Discussed in the paper are 1) decisions and operational aspects relative to analyzing such a large software system, 2) actual metrics scores for the AFATDS software, 3) the results of the metric-based human analysis, 4) how PM FATDS used the the metrics information, and 5) AFATDS developers' comments to the reported findings.

KEYWORDS

Software metrics, software quality, Ada, software principles, software tools

ACKNOWLEDGEMENTS

The analysis discussed in this paper was performed by Dynamics Research Corporation as a subcontractor to TELOS Federal Systems for the Ada Software Quality Evaluation Tool Task, contract number DAAB07-87-C-B015/TFS-1014.

Portions of the Ada Complexity Analyzer were developed by Dynamics Research Corporation under contract to Naval Underwater Systems Center, Newport, Rhode Island, contract number N66604-88-D-0027.

AdaMAT is a registered trademark of Dynamics Research Corporation.

1.0 INTRODUCTION

This analysis was part of the Army's evaluation of the quality of the Ada software developed during the concept evaluation phase of Advanced Field Artillery Tactical Data System (AFATDS). The Army's goal was to determine whether the concept evaluation Ada source could be effectively used as a basis for the next phase of software development. In addition to testing the operational capabilities of the software, PM FATDS decided to evaluate the software engineering principles applied to the development of the Ada source. The purpose of the metrics analysis was to provide the Army with source-level visibility, resulting from quantification of the specific Ada language features and programming practices actually present in the Ada source, as the means to identify the set of applied software engineering principles [AFATDS-T1-88], [AFATDS-T2-89], [AFATDS-T3-89]. See [Sintic89] for a discussion of the rationale of applying independent metrics analysis to the AFATDS source code.

The metrics analysis of the AFATDS software consisted of collecting level-of-adherence and inherent complexity measures on approximately 1.5 million text lines (.5 million Ada statements) of Ada source, through a combination of automated static analysis and metric-based human evaluation of the code. The AFATDS software analyzed consisted of five sub-directories:

Movement Control (MC),
Fire Support Execution (FSX),
Fire Support Plan (FSP),
Data Base Design (DBD),
Common Function (CF).

The metrics analysis for PM FATDS comprised four major tasks. First, a) level-of-adherence metrics were collected on the application code for AFATDS to provide a profile of the programming practices used and, b) inherent complexity metrics were collected on the application code to provide a profile of the Ada constructs used. Second, these metric scores were examined to identify:

- characteristics of the Ada source that augment quality,
- characteristics of the Ada source that attenuate quality,
- under-utilized features of Ada,

- and the developer's coding standards.

Third, the metrics scores for the AFATDS code were compared to typical ranges found from analysis of other comparable systems. Lastly, the findings and recommendations were reported to PM FATDS and the developer.

2.0 ANALYSIS APPROACH

One aspect of the assessment of code quality of the concept evaluation phase of AFATDS was to provide visibility into software by the use of automated analysis tools to produce metrics data, which, in turn, was used to guide the human review. The analysis approach was primarily based on the premise that important aspects of the quality of the Ada software may be determined from the static analysis of the source code, and consequently was metric driven.

The analysis effort consisted of the following:

- The production of composite metrics reports for each of the five sub-directories to allow the most prominent characteristics of the sub-directories to be quickly identified.

- Individual reports were produced for each of the Ada files to isolate those modules having contributed significantly to the prominence of a characteristic.

- Human analysis was performed to divide these characteristics into the following six categories:

1. Specific examples of non-adherence - the report provided a definition of the metric, discussion of its rationale for use, an example of non-adherence taken from the Ada source, and a sample modification of the code to illustrate the effect of adherence for each metric selected for investigation.

2. Characteristics that augment quality - for these characteristics, the report contained a summary-level description of each, derived from actual metrics scores, the observations made during human analysis, and comparisons to other code analysis scores. (For further discussion of these results, see Section 4.2.)

3. Characteristics that attenuate quality - the report provided the same level of summary as for the characteristics augmenting quality. (For further discussion of these results, see Section 4.3.)

4. Coding standards - the report supplied a list of coding standards obviously being employed in the code, as demonstrated during metric analysis. (For further discussion of these results, see Section 4.4.)

5. Under-utilized features - the report listed software engineering practices which the analysis of metrics scores indicated were under-utilized. (For further discussion of these results, see Section 4.5.)

6. Comparisons of levels of adherence - the report contained a discussion of comparisons between scores for the overall characteristics and typical scores found in comparable systems. (For further discussion of these results, see Sections 4.2, 4.3, 4.6.)

The analysis effort began by using the AFATDS Ada source as input to the two automated analysis tools. At this stage, the process produced tabular reports listing level-of-adherence scores and inherent complexity scores (See Figure A). These metric scores served as a starting point in reviewing the code to determine the extent to which the features of the Ada language designed to support software engineering principles were employed in the Ada source. Each of the automated analysis tools produced two types of reports - composite and individual (Figure B in Section 3.3 and C in Section 3.4 are examples of composite reports).

The composite reports, containing scores for the a group (sub-directory) of Ada files as a whole, were used to determine those characteristics of the the Ada source to undergo the next step of metric-driven human review. The individual reports were used to locate specific examples of these characteristics in the developer's Ada source. The examples were reviewed to determine whether non-adherence to the principle represented by the metric was justified. For example, in some cases, concerns for efficiency may override the rationale for the metric, or the developer may have addressed the issue satisfactorily in an alternative manner not reflected in the metrics scores (for instance - AFATDS rarely used named loops; but because the nesting level was intentionally low, this practice was not the problem that it might be in more complex code).

The metrics scores were compared to scores obtained from analysis of other Ada software systems developed for the Department of Defense. This was done in order to provide PM FATDS with a comparative evaluation of the software's overall quality relative to the level-of-adherence metric scores and the number of occurrences of particular characteristics found from previous analysis efforts (See Sections 4.1, 4.2, 4.3).

3.0 AUTOMATED COLLECTION AND CALCULATION OF METRICS SCORES

This section describes: 1) the two automated metric frameworks used in the analysis (Section 3.1), 2) the operational considerations resulting from the need to analyze such a large quantity of Ada source in a timely fashion (Section 3.2), and 3) the hard-copy reports generated for the level-of-adherence and the inherent complexity metrics (Sections 3.3 and 3.4).

3.1 Metrics Frameworks

Two automated, hierarchical, and Ada-specific metric frameworks were applied in assessing the quality of the AFATDS software.

The AdaMAT (TM) metric framework is designed to improve the quality of Ada software by assessing the code-level programming practices applied in the implementation of the source. The 153 low-level metrics in the framework measure the level of adherence to software engineering principles and are documented to provide a rationale and method of improvement ([Keller85], [Perkins85], [Perkins86], [Perkins87], [Anderson88], [Anderson89], [AdaMAT88]). These metrics measure adherence to actual practices associated with the principles, such as initializing variables as part of the declaration, avoiding the use of PRAGMA SUPPRESS, avoiding the declaration of variables in PACKAGE specifications, avoiding the use of GOTOs, and naming loops. The the low-level metrics correspond to

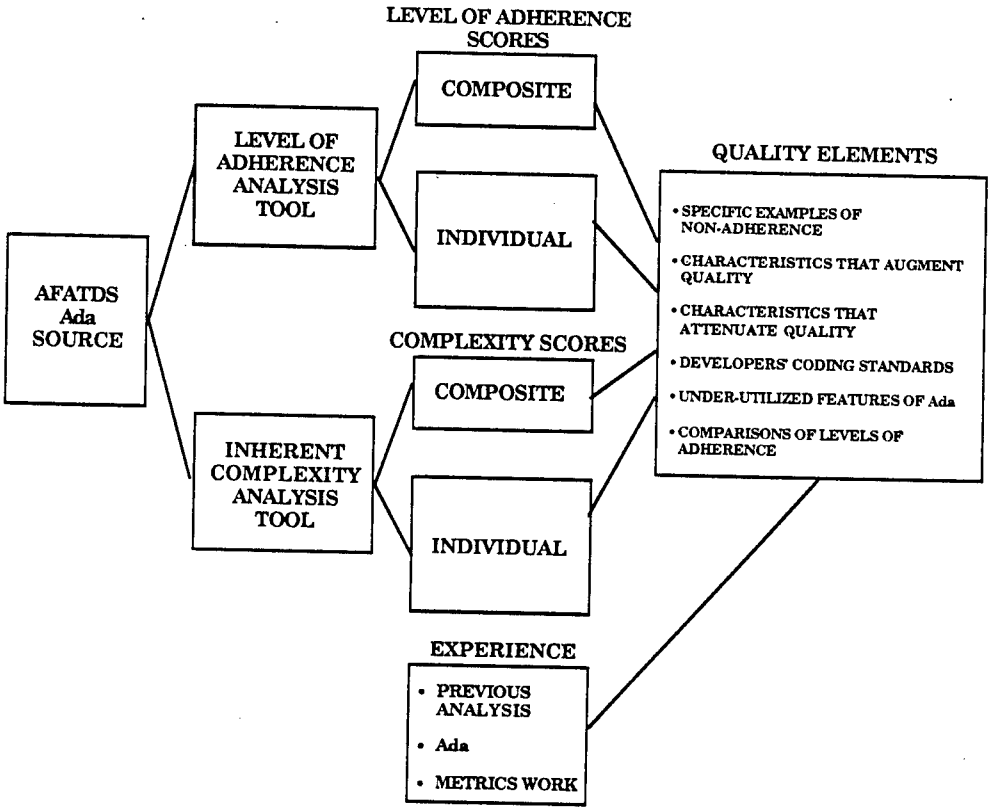


FIGURE A

specific features of the Ada language. The high-level metrics associate software engineering principles to software quality goals such as anomaly management, independence, modularity, self descriptiveness, simplicity, and system clarity.

The framework relates the language features to the software quality principles by:

- Identifying guidelines associated with achieving these goals, in terms of the Ada language.
- Defining coding practices that should be adhered to in order to follow the guidelines.
- Automating the collection of characteristics of the source code that reflect these coding practices (See the report in Figure B which shows the hierarchical metrics framework).

As a result, the framework serves as an action-oriented taxonomy which associates software engineering principles with actual coding practices.

The Ada Complexity Analyzer framework is designed to use analysis of the number of occurrences of each Ada construct to evaluate the inherent complexity of the software. The 600 Complexity Analyzer metrics are hierarchically organized to distinguish between 1) different constructs or 2) different forms of the same construct. The highest-level metrics categorize each Ada statement as either declarative, executable, library, or pragma. The lower-level metrics measure the number of occurrences of Ada constructs such as the number of multi-dimensional array types, package bodies, CASE statements with OTHERS, WITHs associated with SEPARATEs, or PRAGMA INTERFACEs. Hence, the framework provides a hierarchical, level-of-detail profile of the Ada statements constituting the source (See the report in Figure C which shows the hierarchical organization, [Complexity88]).

3.2 Operational Considerations Of Collection Process

In this section, the following operational aspects of the task are discussed:

- Automated data collection and metric score calculation for the level of adherence to software engineering principles
- Automated data collection of the number of Ada constructs to measure inherent complexity.

The analysis process started with selection of AFATDS sub-directories of the code for analysis. PM FATDS chose to have analysis performed on the application software for the system (FSX, FSP, CF, MC, DBD). The analysis was divided into three tasks. For each task approximately one-third of the code was analyzed. Task 1 consisted of a representative sample made up of one complete sub-directory, and portions of other sub-directories chosen to reflect both

various time periods in the development process and various applications. The results of analyzing this portion were later used as a baseline for comparison to the code analyzed in Task 2 and Task 3.

3.2.1 Level-Of-Adherence Analysis

The first step in the automated portion of the metric analysis was counting the occurrences of data items. "Data item" is the label given to the lowest elements in the metrics hierarchy, usually identified as specific features of the language, such as the number of record representation clauses or the number of references to user-defined types. Data item counts were collected on each of the Ada source files for AFATDS.

The next step in the automated portion of metric analysis was the calculation of metric scores. "Metric" is the label given to any element in the hierarchy above the data-item level, each of which is defined in terms of the data items or lower-level metrics. In the hierarchy, metrics are further aggregated upward into sub-criteria, and criteria.

The ANALYZE component calculated a single set of metric scores for each file. The scores are determined from the data-item counts and are reported as an ordered pair (good, total) of integer values indicating the number of "proper" occurrences and the number of "total" occurrences of specific features of the Ada language present in the source. For example, the score for the metric element applicative_declarations is the ordered pair (number of constant declarations, sum of constant and variable declarations), where constant declarations and variable declarations are data items [Perkins 87], [Anderson 88].

The ANALYZE component also processed sets of data-item counts for entire sub-directories as a whole in order to calculate composite scores. The third phase of the automated analysis was the production of hard-copy reports containing metric scores. For the AFATDS project, individual reports were produced for each Ada source file and composite reports for the set of files in each sub-directory. For the individual reports, only those metric scores which were less than 1.00 were reported since such scores indicate the presence of at least one occurrence of a non-adherence. This allows analysts to focus quickly on the "improper" occurrences. For the sub-directory/composite reports, all metric scores were reported. This enabled the analyst to consider the overall characteristics, including both those enhancing and those detracting from quality (See Figure B in Section 3.3).

3.2.2 Inherent Complexity Analysis

The automated collection of the number of constructs followed a similar procedure. However, the ANALYSIS process differs in that only counts (not scores) are produced. For the AFATDS study, the individual reports included only the counts greater than zero. In the composite reports, all counts were included, so that the analyst could evaluate the presence as well as the absence of various features. Figure C (in Section 3.4) is an example of the inherent complexity/composite report.

3.3 Level-Of-Adherence Metric Scores

A portion of the composite report for a sub-directory follows in Figure B. The metric hierarchy is reflected in the "level" column. At the lowest level in the hierarchy, the "total" represents the number of opportunities to adhere to a principle, the "good" represents the number of adherences to the principle. The values at the upper levels of the hierarchy (criteria, etc.) are computed by simply aggregating the scores below.

```
CF    0.85   365168  429997   |1---------- ALL_FACTORS
FSX   0.86   558558  646097   |1---------- ALL_FACTORS
FSP   0.84   752729  900678   |1---------- ALL_FACTORS

CF    0.63    25932   41111   |2---------- ANOMALY_MANAGEMENT
FSX   0.61    31367   51210   |2---------- ANOMALY_MANAGEMENT
FSP   0.62    69464  112765   |2---------- ANOMALY_MANAGEMENT

CF    0.33     4805   14589   | 3---------- PREVENTION
FSX   0.29     5791   20302   | 3---------- PREVENTION
FSP   0.38    17501   46053   | 3---------- PREVENTION

CF    0.28     2268    8049   |  4---------- APPLICATIVE_DECLARATIONS
FSX   0.19     2120   10938   |  4---------- APPLICATIVE_DECLARATIONS
FSP   0.35     9311   26633   |  4---------- APPLICATIVE_DECLARATIONS
...
CF    0.35     2046    5764   |  4---------- DEFAULT_INITIALIZATION
FSX   0.39     3439    8805   |  4---------- DEFAULT_INITIALIZATION
FSP   0.40     6905   17320   |  4---------- DEFAULT_INITIALIZATION
...
CF    1.00        8       8   |  4---------- CONSTRAINED_NUMERICS
FSX   0.71       35      49   |  4---------- CONSTRAINED_NUMERICS
FSP   0.48       16      33   |  4---------- CONSTRAINED_NUMERICS
...
CF    0.78    18680   24074   | 3---------- DETECTION
FSX   0.82    24726   30043   | 3---------- DETECTION
FSP   0.77    50649   65370   | 3---------- DETECTION

CF    0           0           |  4---------- SUPPRESS_PRAGMA
FSX   0.00        0       0   |  4---------- SUPPRESS_PRAGMA
FSP   0           0           |  4---------- SUPPRESS_PRAGMA
...
CF    0.78    18680   24074   |  4---------- USER_TYPES
FSX   0.82    24726   30043   |  4---------- USER_TYPES
FSP   0.77    50649   65370   |  4---------- USER_TYPES
...
CF    0.99     2447    2448   | 3---------- RECOVERY
FSX   0.98      850     865   | 3---------- RECOVERY
FSP   0.98     1314    1342   | 3---------- RECOVERY

CF    0.99     2447    2448   |  4---------- USER_EXCEPTIONS_RAISED
FSX   0.98      850     865   |  4---------- USER_EXCEPTIONS_RAISED
FSP   0.98     1314    1342   |  4---------- USER_EXCEPTIONS_RAISED

CF    0.99   232571  235072   |2---------- INDEPENDENCE
FSX   0.99   408182  412588   |2---------- INDEPENDENCE
FSP   0.99   458515  464867   |2---------- INDEPENDENCE
...
CF                      90    |  5---------- NO_INTEGER_DECLARATIONS
FSX   0.00        0     243   |  5---------- NO_INTEGER_DECLARATIONS
FSP                     1113  |  5---------- NO_INTEGER_DECLARATIONS
...
CF    0.00        0       6   |  5---------- FIXED_CLAUSE
FSX   0.00        0      33   |  5---------- FIXED_CLAUSE
FSP   0.00        0       9   |  5---------- FIXED_CLAUSE
...
CF    0.84    11134   13318   |2---------- MODULARITY
FSX   0.80    11705   14580   |2---------- MODULARITY
FSP   0.81    26656   32902   |2---------- MODULARITY
...
CF    0.99     5725    5781   |  5---------- VARIABLES_HIDDEN
FSX   0.98     8602    8818   |  5---------- VARIABLES_HIDDEN
FSP   0.97    16827   17322   |  5---------- VARIABLES_HIDDEN
...
CF    0.16       48     298   |  5---------- PRIVATE_TYPES
FSX   0.27      149     554   |  5---------- PRIVATE_TYPES
FSP   0.03       23     791   |  5---------- PRIVATE_TYPES
...
CF    0.92     1102    1200   | 3---------- COUPLING
FSX   0.68      338     496   | 3---------- COUPLING
FSP   0.83     1073    1298   | 3---------- COUPLING

CF    0.90      537     600   |  4---------- NO_MULTIPLE_TYPE_DECLARATIONS
FSX   0.55      137     248   |  4---------- NO_MULTIPLE_TYPE_DECLARATIONS
FSP   0.78      506     649   |  4---------- NO_MULTIPLE_TYPE_DECLARATIONS
...
CF    0.52    39715   76254   |2---------- SIMPLICITY
FSX   0.47    39967   84914   |2---------- SIMPLICITY
FSP   0.56    78738  139928   |2---------- SIMPLICITY
...
CF    0.28     8298   29522   |  4---------- DECLARATIONS_CONTAIN_LITERALS
FSX   0.15     3350   21959   |  4---------- DECLARATIONS_CONTAIN_LITERALS
FSP   0.27    10629   39712   |  4---------- DECLARATIONS_CONTAIN_LITERALS
...
CF    0.49     9701   19823   |  4---------- BRANCH_AND_NESTING
FSX   0.42    10892   25912   |  4---------- BRANCH_AND_NESTING
FSP   0.48    19388   40735   |  4---------- BRANCH_AND_NESTING

CF    0.87    55816   64242   |2---------- SYSTEM_CLARITY
FSX   0.84    56115   66844   |2---------- SYSTEM_CLARITY
FSP   0.79  119356150216   |2---------- SYSTEM_CLARITY
...
```

```
CF      1.00     5129     5129   |  5------ NO_DEFAULT_MODE_PARAMETERS
FSX     1.00     4587     4605   |  5------ NO_DEFAULT_MODE_PARAMETERS
FSP     1.00    10711    10756   |  5------ NO_DEFAULT_MODE_PARAMETERS
...
CF      0.46      319      699   |  6------ NAMED_LOOPS
FSX     0.37      472     1274   |  6------ NAMED_LOOPS
FSP     0.22      651     3004   |  6------ NAMED_LOOPS

CF      0.62      623      998   |  6------ NAMED_BLOCKS
FSX     0.63      379      598   |  6------ NAMED_BLOCKS
FSP     0.68      392      576   |  6------ NAMED_BLOCKS
...
CF      0.87    16130    18555   |  5------ QUALIFIED_SUBPROGRAM
FSX     0.75    10168    13486   |  5------ QUALIFIED_SUBPROGRAM
FSP     0.66    18794    28332   |  5------ QUALIFIED_SUBPROGRAM
```

FIGURE B

3.4 Inherent Complexity Metrics Scores

The following figure is a sample from the complexity metric scores for a sub-directory. The "total" is the count of the number of occurrences of a construct. The "level" shows the hierarchical relationship.

```
FSX     96640   |1---------- STATEMENTS
CF      82714   |1---------- STATEMENTS
FSP    175458   |1---------- STATEMENTS

FSX     35035   | 2--------- DECLARATIVE_STATEMENTS
CF      31410   | 2--------- DECLARATIVE_STATEMENTS
FSP     77899   | 2--------- DECLARATIVE_STATEMENTS

FSX      1146   |  3-------- TYPE_DECLARATIONS
CF       1561   |  3-------- TYPE_DECLARATIONS
FSP      2794   |  3-------- TYPE_DECLARATIONS

FSX       764   |   4------- NONSUBTYPE_DECLARATIONS
CF        872   |   4------- NONSUBTYPE_DECLARATIONS
FSP      1330   |   4------- NONSUBTYPE_DECLARATIONS

FSX       734   |    5------ FULL_TYPE_DECLARATIONS
CF        863   |    5------ FULL_TYPE_DECLARATIONS
FSP      1162   |    5------ FULL_TYPE_DECLARATIONS

FSX       697   |     6----- NONDERIVED_TYPE_DECLARATION
CF        811   |     6----- NONDERIVED_TYPE_DECLARATION
FSP      1130   |     6----- NONDERIVED_TYPE_DECLARATION

FSX       296   |      7---- SCALAR_TYPE_DECLARATION
CF        251   |      7---- SCALAR_TYPE_DECLARATION
FSP       318   |      7---- SCALAR_TYPE_DECLARATION

FSX       214   |       8--- DISCRETE_TYPE_DECLARATION
CF        237   |       8--- DISCRETE_TYPE_DECLARATION
FSP       276   |       8--- DISCRETE_TYPE_DECLARATION

FSX       159   |        9-- ENUMERATION_TYPE_DECLARATION
CF        177   |        9-- ENUMERATION_TYPE_DECLARATION
FSP       259   |        9-- ENUMERATION_TYPE_DECLARATION

FSX       159   |         10 ENUMERATION_NO_CHARACTER
CF        170   |         10 ENUMERATION_NO_CHARACTER
FSP       257   |         10 ENUMERATION_NO_CHARACTER

FSX         0   |         10 ENUMERATION_CHARACTER
CF          7   |         10 ENUMERATION_CHARACTER
FSP         2   |         10 ENUMERATION_CHARACTER

FSX        55   |        9-- INTEGER_TYPE_DECLARATION
CF         60   |        9-- INTEGER_TYPE_DECLARATION
FSP        17   |        9-- INTEGER_TYPE_DECLARATION

FSX        82   |       8--- REAL_TYPE_DECLARATION
CF         14   |       8--- REAL_TYPE_DECLARATION
FSP        42   |       8--- REAL_TYPE_DECLARATION

FSX        16   |        9-- FLOATING_POINT_TYPES
CF          2   |        9-- FLOATING_POINT_TYPES
FSP        24   |        9-- FLOATING_POINT_TYPES

FSX         2   |         10 FLOATING_RANGE_CONSTRAINT
CF          2   |         10 FLOATING_RANGE_CONSTRAINT
FSP         7   |         10 FLOATING_RANGE_CONSTRAINT
...
FSX     25525   | 3--------- OBJECT_DECLARATIONS
CF      22129   | 3--------- OBJECT_DECLARATIONS
FSP     59180   | 3--------- OBJECT_DECLARATIONS

FSX      2128   |  4-------- CONSTANTS
CF       2282   |  4-------- CONSTANTS
FSP      9311   |  4-------- CONSTANTS

FSX      1963   |   5------- NON_DEFERRED
CF       1849   |   5------- NON_DEFERRED
FSP      8995   |   5------- NON_DEFERRED

FSX      1961   |    6------ CONST_NONDEFR_NOT_ARRAY
CF       1836   |    6------ CONST_NONDEFR_NOT_ARRAY
FSP      8987   |    6------ CONST_NONDEFR_NOT_ARRAY

FSX         2   |    6------ CONST_NONDEFER_ARRAY
CF         13   |    6------ CONST_NONDEFER_ARRAY
FSP         8   |    6------ CONST_NONDEFER_ARRAY

FSX        12   |   5------- DEFERRED_CONSTANTS
CF         14   |   5------- DEFERRED_CONSTANTS
FSP         0   |   5------- DEFERRED_CONSTANTS

FSX        12   |    6------ CONST_DEFR_NOT_ARRAY
CF         14   |    6------ CONST_DEFR_NOT_ARRAY
FSP         0   |    6------ CONST_DEFR_NOT_ARRAY

FSX         0   |    6------ CONST_DEFR_ARRAY_TYPE
CF          0   |    6------ CONST_DEFR_ARRAY_TYPE
FSP         0   |    6------ CONST_DEFR_ARRAY_TYPE

FSX       153   |   5------- ANONYMOUS_TYPE
CF        419   |   5------- ANONYMOUS_TYPE
FSP       316   |   5------- ANONYMOUS_TYPE

FSX      8745   |  4-------- VARIABLES
CF       5780   |  4-------- VARIABLES
FSP     17197   |  4-------- VARIABLES
...
FSX      5656   |2---------- LIBRARY_STATEMENTS
CF       7668   |2---------- LIBRARY_STATEMENTS
FSP      8600   |2---------- LIBRARY_STATEMENTS

FSX      5652   | 3--------- WITH_CLAUSES
CF       7663   | 3--------- WITH_CLAUSES
FSP      8576   | 3--------- WITH_CLAUSES

FSX       945   |  4-------- WITH_SPECIFICATIONS
CF       1910   |  4-------- WITH_SPECIFICATIONS
FSP      1643   |  4-------- WITH_SPECIFICATIONS

FSX         0   |   5------- WITH_SUBPROGRAM_DECLARATIONS
CF          0   |   5------- WITH_SUBPROGRAM_DECLARATIONS
FSP         0   |   5------- WITH_SUBPROGRAM_DECLARATIONS
...
FSX       851   |   5------- WITH_PACKAGE_DECLARATIONS
CF       1756   |   5------- WITH_PACKAGE_DECLARATIONS
FSP      1473   |   5------- WITH_PACKAGE_DECLARATIONS
...
FSX     55667   |2---------- EXECUTABLE_STATEMENT
CF      43595   |2---------- EXECUTABLE_STATEMENT
FSP     88917   |2---------- EXECUTABLE_STATEMENT

FSX     55440   | 3--------- NON_TASKING_STATEMENT
CF      43399   | 3--------- NON_TASKING_STATEMENT
FSP     88862   | 3--------- NON_TASKING_STATEMENT
...
```

FIGURE C

4.0 METRIC-BASED HUMAN ANALYSIS

This section discusses human analysis of the generated metric data and scores as a means of outlining metric-by-metric potential non-adherence to accepted software quality principles and for identifying overall characteristics of the code that augment or attenuate quality.

The composite reports, output by the automated tools, suggest which metrics should be subject to further investigation. For example, the score for the metric no_multiple_type_declarations (reported in Figure B) indicated that FSX contained 248 package specifications, of which 137 had one or no type declarations. Comparing the values to results obtained from prior analysis, that declaring many types in a single package causes dependencies which result extensive recompilation when even a minor change to a single type is made, the analysts reviewed a portion of the 111 files containing multiple type declarations to determine whether this practice detracted from quality (See Figure D). The individual reports were used to determine those files to review for this purpose. A search of the individual reports was made to find which files contained a score for the metric no_multiple_type_declarations because only those files have non-adherences. Knowing what to look for and where to find it enables the analyst to locate actual examples of coding practices in the Ada source to illustrate adherences or non-adherences to software engineering principles. The results of

this detailed metric analysis were presented in a report containing the examples and recommendations for improvement.

Of the 153 metrics, 28 metric scores indicated a level of potential non-adherence to accepted software quality principles sufficient to warrant further analysis (the other metrics were used to identify overall characteristics that augment or detract from quality). Of the 28, two were chosen for discussion in this paper (See Figures D and E below). For each metric the report provided a definition, its rationale showing the relationship of the language feature to a recognized software engineering principle, and the conclusions drawn on the code as a result of the actual score for the metric. The same 153 metrics were used to determine those software quality principles for which there was a high level of adherence.

From the report in Figure B:
FSX 0.55 137 248
NO_MULTIPLE_TYPE_DECLARATIONS

metric name: NO_MULTIPLE_TYPE_DECLARATIONS of modularity

There are 137 package specifications with one or no type declaration from a total of 248 package specifications, resulting in a metric score of 0.55.

metric definition - The proportion of package specifications containing no more than one type declaration in the non-private part.

rationale - In some cases, the presence of multiple type declarations in a package specification may indicate definition of multiple objects in a single package. The need to reference any of these objects will result in the ability to access all of the objects even when such access is not desired.

In other cases, the presence of multiple types may indicate definition of a single object structurally too complex for representation by a single data type. The object represented may require operators not directly supported by any type and may not require all the operators supported by the types.

method of improvement - In some cases, multiple type declarations in a package specification should be replaced by multiple package specifications that each declare a single type.

In other cases, one of the multiple type declarations should be defined to be private and others should be moved to the private part of the package specification.

EXAMPLE: In FSX_MESSAGE_BASIC_TYPES_SPEC.ADA

```
with DB_Basic_Types;

package FSX_Message_Basic_Types is--$..
   type Observer_To_Target_Distance_Type is range 0..9_999;
      --| Distance in meters from the observer to the target
--$..
→type Gun_Target_Range_Type is
                     delta 0.01 range 0.00..99900.00;

   type Mission_Direction_Type is
                     (Msn_1_0000_0999_Mils,   --| Mission #
--$..
   type Purpose_Of_Message_Type is
                     (--| Why message was sent
                      Additional_Fires,
                      --$..
                      );

   type Type_Of_Message_Type is
                     (--| Types of messages
                       Additional_Fires,  --|Additional fire from
                                          --|TDAR process
                      --$..
                      );
    type Target_Position_Adjustment_Type is
                     (OKTGT,
                       --$..
                      REJDIS
                      );
   type Situation_Of_Rounds_Type is
                     (Good_Round,
                       Lost_Round
                       --$..
                       );

   type    Shell_Designator_Type is
                     (Adjusting_Shell,
                       Initial_Shell,
                       Subsequent_Shell,
                       All_Shells
                       );

   type    Powder_Charge_Type is range 1..9;

   type Record_As_Registration_Type is new Boolean;
   type Record_As_Time_Registration_Type is new Boolean;
--$..
```

SAMPLE MODIFICATION

```
PACKAGE DIRECTION_DISTANCE_PACKAGE
IS
--$ notice package still contains multiple types but far fewer.
--$ Each of these types are closely related and would usually be used together.
type Observer_To_Target_Distance_Type is range 0..9_999;
    --| Distance in meters from the observer to the target
--$..

  type Gun_Target_Range_Type is delta 0.01 range 0.00 .. 99900.00;

  type Mission_Direction_Type is (Msn_1_0000_0999_Mils,   --| Mission #
--$..
--$ DECLARE OPERATORS FOR THE THREE TYPES
END DIRECTION_DISTANCE_PACKAGE;
```

FIGURE D

From the report in Figure B:
CF 0.28 8298 29522
DECLARATIONS_CONTAIN_LITERALS

metric name - DECLARATIONS_CONTAIN_LITERALS

There are numeric literals referenced in non-declarative parts in the module.

There are 8298 numeric literals referenced in type or constant declarations from a total of 29522 numeric literals referenced, resulting in a metric score of 0.28.

metric definition - The proportion of referenced numeric literals referenced in constant declarations or type declarations.

rationale - Multiple uses of a literal in the non-declarative portion of the code reduces clarity and increases the likelihood that a change in a single use of the literal will not lead to the appropriate modification for other occurrences.

method of improvement - Examine each occurrence of a literal in the non-declarative proportion to see whether the literal is a candidate for replacement by a constant or attribute. If a constant (type) is appropriate, but no such constant (type) exists, then the constant (type) should be declared.

EXAMPLE: In CF_BUILD_DIV_FM_SUBS_BODY.ADA

```
 --$..
package body CF_Build_Div_FM_SUBS is
--$..
```

```
procedure Build_Div_FM_SUBS
  (
  AFATDS_Body       : in
    FSX_Target_Message.Target_Message_Type;
                          --| AFATDS message to be converted from
  Field_List        : in    CF_Fields_Available.Available_Field_List_Type;
                          --| List of fields which are available
  Non_AFATDS_Message : in out
    CF_MM_String_Package.Variable_String_Type
                          --| Non-AFATDS message string to build
  ) is
--$..
                          --| Convert Target_Number
    CF_MM_String_Package.Concatenate
     (This_String     => AFATDS_Body.Target_Number,
      To_This_String  => Non_AFATDS_Message,
-->   For_A_Length_Of => 6);
--$..
```

SAMPLE MODIFICATION:

```
--$..
package body CF_Build_Div_FM_SUBS is
--$..

procedure Build_Div_FM_SUBS
  (
  AFATDS_Body       : in
    FSX_Target_Message.Target_Message_Type;
                          --| AFATDS message to be converted from
  Field_List        : in    CF_Fields_Available.Available_Field_List_Type;
                          --| List of fields which are available
  Non_AFATDS_Message : in out
    CF_MM_String_Package.Variable_String_Type
                          --| Non-AFATDS message string to build
  ) is
--$..
                          --| Convert Target_Number
    CF_MM_String_Package.Concatenate
     (This_String     => AFATDS_Body.Target_Number,
      To_This_String  => Non_AFATDS_Message,
-->   For_A_Length_Of => AFATDS_BODY.Target_Number'Length);
--$..
```

FIGURE E: Replacing the numeric literal by a constant or attribute reduces the possibility that a change in a single use of a literal will not result in appropriate changes to other occurrences. Also, if there are multiple uses of the same literal not being used for the same purpose, adherence to the principle reduces the possibility that inappropriate modifications to these values will be made.

4.1 Metrics Experience Base

The scores for the metrics that augment and attenuate quality, discussed in Sections 4.2 and 4.3, were compared to scores obtained for similar Ada software systems to provide a relative sense (below normal, normal, above normal) of the level of adherence or occurrences of particular characteristics [Perkins89]. The other systems include applications such as local area networks, command center processing, and simulation/stimulation.

Prior to examining AFATDS, the analysts' experience included: 1) research into the proper use of Ada language features such as generics, private types, and exceptions [Gorzela87], [Keller89], [PerkinsE88], Perkins T88], 2) evaluation of the following kinds of Ada-implemented systems:

- Army-developed support tools for the Army Center for Software Engineering
- Navy-developed Compiler Benchmarks for the Naval Underwater Systems Center
- Trident Sonar Maintenance Trainer-FES for the Naval Underwater Systems Center
- Air Force Software for the Electronic Systems Division
- WWMCCS Information System for the Electronic Systems Division
- Command Control Processor Display System-Replacement for the Electronic Systems Division
- Submarine Satellite Information Exchange System for the Fleet Combat Direction Systems Support Activity, Naval Ocean System Center.

In these wide-ranging analysis efforts, more than 500,000 text lines of code were analyzed.

4.2 OVERALL CHARACTERISTICS THAT AUGMENT QUALITY

The human review of the software, the analysts' previous experience, and the composite metrics scores were employed in concert to identify overall characteristics of the software that augmented quality. Based on metrics scores for the composite reports, twenty-six such characteristics were identified in Task 1. The Task 1 characteristics were used again as a the basis for comparison of the segments of the software analyzed in Task 2 and Task 3. For these later tasks, only those characteristics deemed significantly different from Task 1 were subjected to detailed analysis.

A list of overall beneficial characteristics follows in Figure F. They are grouped by the criterion with which they are associated in the metric framework. For each characteristic, the report contains the benefits of adherence to the practice and the extent of its use relative to typical industry practices, reported as greater than normal, normal, less than normal (See Figure F).

Note: Comparisons between systems should be treated cautiously, since they may be associated with differences in application characteristics, system requirements, or development processes.

CHARACTERISTICS THAT AUGMENT QUALITY

ANOMALY MANAGEMENT

The software does not use PRAGMA SUPPRESS.

The practice of not using PRAGMA SUPPRESS, which is a normal practice, is beneficial since it permits use of the run-time error-detection mechanisms provided by Ada. This increases the possibility that Ada can detect errors which might otherwise produce incorrect information or exhibit a different error.

The software includes initialization as part of the variable declarations. Record components are being initialized as part of the record type declaration.

These practices of initializing during declaration which are being applied to the normal degree are beneficial since they reduce the possibility of referencing the value of an undefined variable.

Adherence was higher in package specifications than in package bodies.

The software has a high proportion of normal loops, where a normal loop implies the existence of a conditional exit or return.

The practice of employing normal loops, which is applied more often than normal, is beneficial since it implies that the loops may be expected to behave in a meaningful and normal manner.

The software has a high proportion of constant declarations in package specifications.

The practice of using constants instead of a variable when possible, which is being applied to the normal degree, is beneficial since it eliminates the possibility of unintended change of invariant values. Also, the declaration of a constant clarifies that the object is invariant.

The software uses the "EXCEPTION" mechanism of Ada. User-defined exceptions are declared, raised and handled. Blocks are being used to isolate the handling of exceptions. The executable portion of package bodies is in most instances empty.

These practices involving the exception mechanism, which are applied more often than normal, are beneficial for the detection and the recovery from unusual but anticipated situations.

The software contains very few unconstrained array types, and variant record types.

The absence of these two kinds of dynamic types, which is normal, is beneficial since the more static the data structures the more effective the compile time error checking.

Note: The software does have several access types. Access types are dynamic structures.

INDEPENDENCE

The software does not use system-dependent I/O.

This practice, which occurs to a greater degree than normal, is beneficial when porting to a new machine or operating system.

The software does not use machine code statements.

The absence of machine code statements, which is normal, is beneficial since it reduces target machine dependence.

Note: This conclusion is based upon a small portion of the code. It may be that some of the subprograms called by the analyzed software, but which are not part of the sample, are machine dependent.

The software does not use implementation dependent pragmas.

The absence of implementation-dependent pragmas, which is greater than normal, is beneficial since it decreases possibility of dependence on the host machine, or target environment.

The software does not use fixed-point or floating-point types.

This practice is normal. The absence of fixed-point and floating-point types is range or accuracy of these types can lead to non-portable code.

The software does not mix tasking and non-tasking statements.

This practice, which occurs more than normally, is beneficial since it makes porting to a new system less complex.

The software does not mix I/O and non-I/O processing.

This practice which is adhered to more than normally is beneficial since it increases the ability to modify the format of the input and output.

MODULARITY

The software uses the package mechanism of Ada. Packages are being withed only when required and the withing is being restricted to bodies when possible. The separate mechanism of Ada is being used. Each with clause contains a reference to a single library unit.

These practices, which are followed more often than normal, are beneficial since they prevent the "withers" from having unintended access to unneeded types, operators, and objects. The ability to perform separate compilation is enhanced.

In most cases, the software does not have variables declared in package specifications.

This practice, which is applied more often than normal, is beneficial since it prevents users from creating unintended dependencies on the underlying implementation.

The subprograms are parameterized.

The practice of parameterizing subprograms, which is applied more often than normal, is beneficial since parameterization permits the functionality of the subprograms to be performed for multiple sets of objects.

The software uses the GENERIC mechanism of Ada.

This practice, which is applied more often than normal, is beneficial since it eliminates the repetition of portions of the program in cases where the logic is independent of the types.

SELF-DESCRIPTIVENESS

The objects are given meaningful names. Pre-defined words are not used as names.

This practice, which is applied to a normal degree, is beneficial since it increases explanation of the code.

The software is extensively commented.

This practice occurs to a greater degree than normal. Algorithms are described in a comment block. Declarations and executable statements are commented sufficiently. This observation was made as a result of reading the code. Since the commenting standards for this software differ from

those used by AdaMAT, the scores do not accurately reflect the amount or distribution of comments.

SIMPLICITY

Array types are declared explicitly.

This practice, which is applied to a greater degree than normally, is beneficial since it provides the ability to declare parameters, variables, and constants of these types.

Subtypes are declared explicitly.

This practice, which is applied more than normally, is beneficial since it provides the ability to declare parameters, variables, and constants of these types.

The branching structure is simple. The software does not use goto's. Very few branches per modules. Most ifs contain no elsifs. For loops iterate forward. Procedures contain no returns.

These practices involving simplifying the branching structure which are being applied more often that normal are beneficial since it increases the ease of understanding the flow within a module.

The average level of nesting is not excessive.

This practice of limiting the nesting level, which is applied more often than normal, is beneficial since the complexity of flow relates to the difficulty of understanding and testing the functions of a module. In addition, simpler modules are easier to maintain.

SYSTEM CLARITY

The software references the names of aggregates associated with components.

This practice, which is applied more often than normal, is beneficial since, unlike positional notation, it results in an explicit association of the components and the aggregate.

The software uses qualification when referencing subprograms.

This practice, which is applied more often than normal, is beneficial since it makes explicit the association of the subprogram to the declarer of the subprogram. This eliminates potential ambiguity for the human as well as the compiler.

The mode of subprogram and generic parameters is explicitly specified.

The practice of explicitly specifying the mode of parameters which is being applied more often than normal, is beneficial since explicit specification of the mode reduces the likelihood that a parameter intended to be out or in out mode will default to in.

The software contains no use clauses.

The practice of not employing use clauses, which is applied more often than normal, is beneficial since it forces explicit qualification of all references to the contents of withed packages.

Declaration lists contain a single object declaration.

The practice of declaring a single object in a declaration list, which is applied more often than normal, is beneficial since it enhances the ability to follow other good practices concerning commenting, initialization, strong type checking, and constraint checking.

FIGURE F

4.3 Characteristics That Attenuate Quality

Similarly, the software was reviewed for the purpose of identifying overall characteristics that attenuate quality. Fifteen characteristics were found to be in this category. These characteristics are listed below in Figure G along with the benefits of adherence to the metric and the extent to which the developer's use of this practice compares to that found on other systems. These comparisons are reported as greater than normal, normal, less than normal. If a characteristic is reported as "normal (seldom)", this means that within the industry the practice is seldom used. The characteristics are grouped by the criterion with which they are associated in the metric framework.

CHARACTERISTICS THAT ATTENUATE QUALITY

ANOMALY MANAGEMENT

The software is heavily dependent on the Ada system-defined NATURAL, POSITIVE, and INTEGER types, rather than using the declaration of user-defined types.

The preferred practice of declaring of user-defined types increases the effectiveness of strong type-checking in the detection of the unintended transfer of values between conceptually different objects and the constraint checking in the detection of out of range values being placed into these objects.

This practice is being used to a normal degree (seldom).

In most cases, the presence of these system-defined types seems unjustified.

The software contains few derived types.

The preferred practice of using derived types is beneficial, since it increases the effectiveness of strong type-checking in the detection of the unintended transfer of values between conceptually different objects.

This is being applied to the normal degree (seldom).

The software contains many unconstrained subtypes.

The preferred practice of constraining subtypes is beneficial, since it increases the effectiveness of the constraint checking in the detection of out-of-range values being placed into objects.

The practice of constraining subtypes is being applied less often than normal.

The software contains many variables that are invariant, and therefore could be declared as constants.

The preferred practice of representing invariant objects by the use of a constant rather than a variable is beneficial, since it prevents the unintentional modification of an invariant object.

This practice is being applied to the normal degree (only for universal constants).

The software contains many procedure parameters of IN OUT mode that are invariant within the procedure. Invariant parameters can be declared to be IN mode.

The preferred practice of using of IN mode parameters rather than IN OUT mode when possible is beneficial since it prevents the unintentional modification of an invariant object.

The practice of specifying invariant parameters to be IN mode is being applied to the normal degree.

INDEPENDENCE

The software is heavily dependent on ACCESS types rather than statically allocated types.

The preferred practice of limiting the use of ACCESS types is beneficial since the efficiency and effectiveness of deallocation varies with each implementation.

The use of access types occurs to a normal degree.

The software is heavily dependent on predefined types INTEGER, NATURAL, and POSITIVE. Integer constants are heavily dependent on these predefined types.

The preferred practice of declaring user-defined integer types is beneficial since it prevents reliance on assumptions about the bounds which may lead to non-portable code.

This practice is being applied to the normal degree (seldom).

MODULARITY

The software is heavily dependent on a few packages which declare many types.

The preferred practice of declaring few types per package specification is beneficial for several reasons: it prevents coupling between conceptually different objects, if a change is made in a data structure in one of these packages extensive recompilation will not be required.

This practice of declaring many types in a few packages occurs to a normal degree.

In general, the declaration of many types in a single package seems unjustified. These packages should be decomposed into smaller packages.

Note: Although FSP.GUID has just 7 packages which declare multiple types, the WITHing structure indicates the dependency of many packages on two packages (not within this sub-directory) which declare many types.

The software contains few private types.

The preferred practice of declaring types in package specifications to be private types is beneficial since specifying the types to be private decreases the dependency of the wither of the type on the underlying data structure actually being used.

This practice is being applied to the normal degree (seldom).

Most subprograms are procedures that could be functions.

The preferred practice of specifying subprograms that have no internal side effects as functions when possible is beneficial since functions are able to be used as part of expressions.

This practice is being applied to the normal degree.

SIMPLICITY

There are array ranges not declared as explicit types.

The preferred practice of declaring explicit types for array ranges is beneficial since it permits the declaration of array index variables for expressing that array type.

This practice is applied to a lesser degree than normal.

The software is heavily dependent on the use of literals in the executable portions of the code.

The preferred practice, using constants or attributes to represent the literals, is beneficial since it increases clarity and reduces the likelihood that a change in a single use of the literal will not lead to the appropriate modification for other occurrences.

This practice is applied to a normal degree.

SYSTEM CLARITY

The software is heavily dependent upon formatting to indicate the BEGIN and END of structures.

The preferred practice of naming structures and using the name at the end of the structure is beneficial since it makes an explicit association of the BEGIN and END.

This practice is applied to a normal degree.

In general, delineation of the software is not difficult, however, more extensive naming of loops and blocks would be beneficial.

The software contains groups of nearly identical files.

The preferred practice of using generics when the logic of the subprogram is independent of the type is beneficial, since modifications will only be required in one place.

This practice is being applied to a greater than normal degree.

In some cases, the groups could be replaced by a generic subprograms, so that the modification would only be done once.

The software is heavily dependent on implicit subtypes for the ranges in FOR LOOPs.

The preferred practice of declaring explicit types for ranges of FOR LOOPs is beneficial, since the use of a type for the range of a for loop results in the explicit declaration of static ranges for this type of looping. In addition, it also permits association with an array range when the loop depends upon this array range.

This practice is applied to a lesser degree than normal.

CHARACTERISTICS OF SUB-DIRECTORY DBD

Comparison of DBD to the other four sub-directories is not appropriate since DBD is a wholly structural entity. All statements contained in the DBD directory were found to be declarative or library. The criteria used in the metrics framework are intended for functional entities.

- The software references user-defined types rather than pre-defined types. This practice enhances the strong type checking and constraint checking provided in Ada, and increases Ada's ability to detect the unintentional combinations of distinct but similar objects.

- The software contains packages that declare many types.

Although the practice of declaring many types in a few packages is not unusual the practice of declaring a few types and the associated operators per package specification is preferable. The preferred practice is beneficial, since it limits the visibility of the objects to those users who need visibility. The reduces the rippling effects of recompilation.

- DBD contains 27 files, each of which is a package specification. Of these, 15 contain indications of a lack of specificity.

See [Temte84] for a detailed discussion of the developer's application of the Ada package mechanism to a component of AFATDS.

FIGURE G

4.4 PROGRAMMING STANDARDS

The AFATDS metric scores at the composite level were also reviewed to determine the coding standards applied by the developer. The findings consist of only those practices that are obvious in AFATDS Ada source and reflect only the analysts' examination of the metrics collected by the analysis tools. No attempt was made to identify all standards employed with the code development effort. Thus, the following list in Figure H represents a subset of the coding standards actually used to govern the development of the software.

PROGRAMMING STANDARDS

- Limit on the number of branch statements per module.
- No goto statements.
- No return statements in procedures.
- No declarations of packages or subprograms in blocks.
- No use statements.
- Qualification of all external references.
- Rename of infix operators of withed types so that these operators can be used as infix operators inside the consumer of the withed unit.
- Type and subtype are declared explicitly.
- The mode of generic and subprogram parameters are specified explicitly.
- With clauses contain only a single name.
- Declaration lists contain only a single name.
- Subprogram calls and aggregates use named notation.
- The ends of modules are delineated by the name of the module.
- No use of pre-defined words.
- Commenting of specifications and bodies.

FIGURE H: This list is based upon metrics scores and human analysis. For example, the conclusion "qualification of all external references" may be drawn from the complexity report: for FSX there are 5656 library statements of which 5652 are WITH clauses, therefore there are only 4 USE clauses, thus almost all external references must be qualified.

4.5 Under-Utilized Ada Features

The analysis of the metric scores reflecting the use of accepted software engineering practices indicated that the following features, listed in Figure I, of the Ada language were under-utilized for one or more of the the six sub-directories of AFATDS Ada source. The list of under-utilized Ada features uses only the analysts' examination of the metrics collected by the analysis tools. No attempt was made to identify the external circumstances that caused these features not to be used more (See Section 5.2 for the developer's response as to why some of these features were not heavily utilized.). Thus, although modern accepted software engineering practices suggest that these features should be used more, there may be valid justifications for the infrequency of their use.

UNDER-UTILIZED FEATURES

- User-Declared Integer Types
- Derived Types
- Private and Limited Private Types
- Generic Packages
- Constant Declarations

FIGURE I: The metrics reports and human analysis were used to prepare this list. For example, the conclusion that user-declared integer types were under-utilized is based the complexity report in Figure C. Of the 2974 type declarations in FSP, only 17 are user-defined integer types.

4.6 Comparison To Other Programs

To assist in the determination of the effectiveness of the concept evaluation software and of its applicability to the next phase of development, PM FATDS requested that the quality assessment be compared to the results of previous studies for comparable systems.
This was done at two levels:

As discussed in Sections 4.2 and 4.3, part of the overall quality assessment for Task 1 included comparisons to metric scores found for other Ada software systems. These comparisons were made at the metric-element level (low level in the hierarchy reflecting actual coding practices).

For the final report, the criteria scores (top level in the hierarchy) for each sub-directory were compared to typical ranges determined from Ada systems analyzed previously. The following is a list of the criteria, definitions, and typical scores found for these criteria. In each case, the score is the ordered pair (sum of number of adherences to the principles, sum of number of opportunities to adhere).

anomaly management - Those attributes of the software providing for prevention, detection, and recovery from non-nominal conditions. The score for anomaly management is typically .55 to .65.

independence - Those attributes of the software that determine its non-dependency on the software environment (computing system, operating system, utilities, input/output routines, libraries). The score for independence is typically .85 to .99.

modularity - Those attributes of the software providing a structure of highly cohesive modules with optimum coupling. The score for modularity is typically .20 to .60.

simplicity - Those attributes of the software providing for the definition and implementation of the functions of a module in the most non-complex and understandable manner. Coding simplicity, design simplicity and flow simplicity are considered. The score for simplicity is typically .30 to .40.

Note: This range is considered acceptable, since the simplicity criterion includes decisions, nesting, branches. All occurrences of these lower the score for simplicity; however, they are unavoidable.

Note: The method employed for accumulating the scores for the criteria and other high levels of the metric framework was modified prior to the analysis of the AFATDS Ada source. Consequently, slight differences in scores should not be interpreted as significant in terms of level of adherence.

system clarity - Those attributes of programming style providing for a clear and understandable description of the program structure. The score for system clarity is typically .30 to .80.

Figure J compares the five AFATDS sub-directories by criteria. The bold boxes indicate the ranges found for criteria scores on the first AFATDS sample. The thin boxes show ranges for criteria scores found as a result of previous analysis of comparable systems.

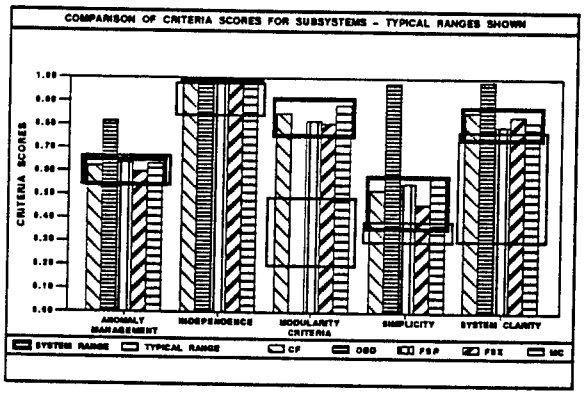


FIGURE J

This graph (Figure J) indicates that the AFATDS scores for anomaly management are within the range found in other systems. The AFATDS score for modularity is generally much higher than that found in other systems with the notable exception of the sub-directory DBD. This sub-directory, discussed in Section 4.3, utilizes very few of the features of Ada designed to enhance modularity. The simplicity scores show that sub-directory FSX has the lowest score. This suggests that more detailed reports should be reviewed to determine the reasons. (For example, the functions may require complex computation.) The scores for system clarity are above those typically found and their consistency indicates a highly disciplined programming style.

5.0 PRESENTATION OF FINDINGS

5.1 Form Of The Presentation

The findings were presented to PM FATDS in the form of reports and briefings.

Hard-copies of the tabular reports for each of the approximately 6000 files were delivered. For each of the three tasks, a summary level report was also prepared. Figures B - K were taken directly from one of these reports, and are examples of the types of information provided [AFATDS].

In the interests of efficiency, PM FATDS chose to have the scope of the summary reports for Tasks 2 and 3 limited to

analysis of those characteristics of the software that differed significantly from those found in the first sample. The metrics-driven reports were similar to those found in Figures D, E, F, G; however, they also included the identification of the sub-directories where such characteristics were found. This information assisted PM FATDS and the contractor in associating these characteristics with code produced at different times and for different applications.

Finally, following the entire analysis of the application software, overall characteristics of the code were identified in a manner that permitted comparison and evaluation across sub-directories as well as to other similar Ada systems. Figures K and L show statement distribution for two of the sub-directories determined from the inherent complexity analysis.

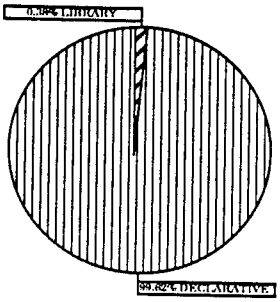


FIGURE K

Figure K illustrates the statement distribution for DBD (See Section 4.3 for more details). The relatively large percentage of declarative statements and small percentage of library statements, and the absence of executable statements indicates that the overall design, not unlike that of other large Ada systems reviewed by the analysts, depended upon relatively few packages to provide most of the TYPEs. (See Section 5.2 for the developer's response to why this characteristic may not be desirable).

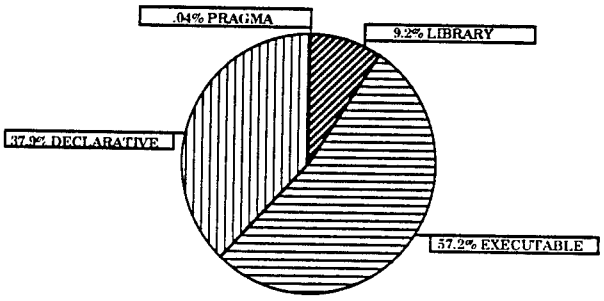


FIGURE L

Figure L shows the statement distribution for sub-directory CF. This is typical of the remainder of the application software. The high proportion of declarative statements is indicative of the developer's practices of declaring constants, operators, and user-defined types, which the packaging mechanism of Ada is intended to support. The large percentage of library statements further confirms the extensive use of this Ada mechanism for encapsulation.

Figure J (Section 4.6) was also part of this comparative analysis.

5.2 AFATDS Developer Response

The AFATDS developer was present at the briefing of the metrics analysis. A portion of the developer's responses to the analysis findings is presented below:

Analysis Finding: Many invariant objects declared as variables could have been declared as constants.

Developer Response: The developer intentionally chose to declare many invariant objects as variables rather than constants. During the early stages of development, the specific value for these invariant objects was not known. Leaving these objects as variables allowed the developer to patch object code and save recompilation time.

Analysis Finding: Explicit types were used in the declaration of numeric constants. This practice forces the compiler to represent the values of the constants as pre-defined types.

Developer Response: The developer stated that the decision to use typed constants rather than universal type constants had caused some problems during integration.

Analysis Finding: A few packages contained most of the type declarations.

Developer Response: The developer stated that these large global types packages caused a major problem. A change to any of these types packages resulted in the need to recompile most of the AFATDS source. In the early stages of the development, such recompilation could take as long as 6 days.

By the end of development, improvements in compiler performance reduced the recompilation time associated with such changes to 1 day.

The developer is considering reworking these packages in the next phase of development.

Analysis Finding: Structured programming techniques were employed in the development of the software.

Developer Response: The developer had metrics tools to control the number of branches per module, coding standards governing the development of structured code, and procedures for insuring adherence to such standards.

Analysis Finding: WITH clauses were frequently associated with subunits.

Developer Response: The developer stated that the SEPARATE mechanism was a useful way of reducing compilation order dependencies; thus decreasing the amount of recompilation required in those instances where the specification portion of a withed units is modified. See [Barnes 84], [Ada83] for an in-depth discussion of withing by subunits.

Analysis Finding: In a few instances, a module would declare, raise, and handle a given exception.

Developer Response: The developer stated that this

practice was associated with debugging and that such code was not intended to be in the delivered version.

Analysis Finding: Private types were under-utilized despite the extensive use of composite types.

Developer Response: Early versions of the compiler did not support private types; for this reason, only the code developed during the later stages of development contained private or limited private types.

Analysis Finding: Generics were under-utilized despite having a large number of similar packages and modules.

Developer Response: Early versions of the compiler did not support generics; therefore, generics were instantiated by hand before being submitted to the compiler.

5.3 Analysts' Response

The human analysis of the AFATDS Ada source indicated the need for additional metrics. After the AFATDS metrics analysis, several new metrics were added to the two frameworks. The level-of-adherence framework was updated to contain metrics such as the proportion of 1) record components initialized, 2) case statements without others, and 3) WITHs associated with separates. The inherent complexity framework was updated to contain a usage-breakdown by name for the pre-defined 1) pragmas, 2) attributes, 3) packages, 4) types, 5) exceptions, and 6) I/O operators.

The hard-copy report for the inherent complexity metric framework was modified to better show the distribution of statements among the modules making up the software. Reported are 1) what proportion of the modules contain no more than a specified threshold for a certain kind of statement and 2) what proportion of these kind of statements are contained in modules satisfying the threshold. Eleven different thresholds are allowed to be set for each of the different kinds of statements identified in the framework. Figure M is an example of the new form for inherent complexity reports, showing distributions for all statements and declarative statements over modules.

MODULES			STATEMENTS			THRESHOLD	
1.00	176	176	1.00	4609	4609	256 | 1	STATEMENTS
0.99	174	176	0.94	4310	4609	128 | 1	STATEMENTS
0.95	167	176	0.79	3642	4609	64 | 1	STATEMENTS
0.74	131	176	0.47	2155	4609	32 | 1	STATEMENTS
0.34	59	176	0.13	580	4609	16 | 1	STATEMENTS
0.09	15	176	0.01	56	4609	8 | 1	STATEMENTS
0.04	7	176	0.00	15	4609	4 | 1	STATEMENTS
0.01	2	176	0.00	2	4609	2 | 1	STATEMENTS
0.00	0	176	0.00	0	4609	1 | 1	STATEMENTS
1.00	176	176	1.00	1392	1392	256 | 2	DECLARATIVE
0.99	175	176	0.90	1259	1392	128 | 2	DECLARATIVE
0.98	172	176	0.70	969	1392	64 | 2	DECLARATIVE
0.95	168	176	0.57	787	1392	32 | 2	DECLARATIVE
0.90	159	176	0.40	553	1392	16 | 2	DECLARATIVE
0.80	140	176	0.26	357	1392	8 | 2	DECLARATIVE
0.63	110	176	0.15	204	1392	4 | 2	DECLARATIVE
0.26	45	176	0.03	45	1392	2 | 2	DECLARATIVE
0.00	0	176	0.00	0	1392	1 | 2	DECLARATIVE

FIGURE M: 7 out of 176 modules contained less than 4 statements each. 15 out of 4609 statements were in these 7 modules.

6.0 CONCLUSIONS

The metrics analysis indicates that the AFATDS software was developed in a highly disciplined fashion. A rich set of the language features was utilized and a large collection of programming practices was applied. Rigorous controls were used by the developer to insure the consistent use of and high adherence to these practices. The absence or the low use of a quality-enchancing language feature was often related to the lack of compiler support for the feature during the early stages of development. Although in some instances the level of adherence to important software engineering principles was lower than what might be desired, in many of these instances, the AFATDS software had a normal or above normal level of adherence to these principles when compared to other Ada software analyzed using the same two metrics frameworks.

As a result of the findings of this metrics analysis, other related quality analysis performed on the AFATDS software [Sintic89], and the success of the month-long operational software testing [Green89], the Army is confident that the overall code quality of the Ada source developed during the concept evaluation phase indicates the code's suitability for use in the next phase of the AFATDS development.

PM FATDS found automated metrics analysis to be effective means of identifying the software engineering principles present in the 1.5 millions text lines of AFATDS Ada source. These findings formed an objective basis for communication between PM FATDS and the developer concerning the programming practices applied during development of the code. As a result of these discussions, areas of improvement have been identified and will be implemented during the next phase of development.

The metrics scores for the Ada source developed during the the concept evaluation phase will provide valuable guidance in deciding what are acceptable scores for the next phase of AFATDS; however, PM FATDS understands that for a given software component the quality goals and the quality trade-offs caused by these goals must be the primary drivers in determining what is acceptable. For instance, frequently executed components have to emphasize aspects of efficiency; critical components have to emphasize aspects of reliability; and multi-targeted components have to emphasize aspects of portability. PM FATDS realizes that meaningful interpretation of metric scores requires knowledge of the quality goals. The expected level of adherence to the software principles may need adjustment for certain cases in order to meet overall system-specific requirements. PM FATDS accepts that some degree of non-adherence to desired practices is unavoidable while supporting the premise that such nonadherence is only acceptable in those instances where adherence is either not possible or not practical due to overriding circumstances.

The Army's ability to evaluate the quality of the AFATDS software was greatly enhanced by the source-level visibility provided by the metrics analysis. PM FATDS has determined that quality metrics are both useful and necessary during the development of large software programs. PM FATDS is committed to applying automated metrics analysis during the next phase of the AFATDS development.

REFERENCES

[Ada83]

"Reference Manual for the Ada Programming

Language", ANSI/MIL-STD-1815A, United States Department of Defense, Washington, D. C., January 1983

[AdaMAT88]

"AdaMAT Reference Manual" Dynamics Research Corporation, Andover, MA, 1988

[AFATDS-T1-88]

"Technical Report AFATDS Ada Code Analysis Volume 1-2 Task 1", Telos Federal Systems, Fort Monmouth Operations, Shrewsbury, NJ, November 88.

[AFATDS-T2-89]

"Technical Report AFATDS Ada Code Analysis Volume 1-2 Task 2", Telos Federal Systems, Fort Monmouth Operations, Shrewsbury, NJ, January 1989.

[AFATDS-T3-89]

"Technical Report AFATDS Ada Code Analysis Volume 1-2 Task 3", Telos Federal Systems, Fort Monmouth Operations, Shrewsbury, NJ, February 1989.

[Anderson88]

Anderson, J. D., Perkins, J. A., "Experience Using an Automated Metrics Framework in the Review of Ada Source for Wis", Six Annual National Conference on Ada Technology, March 1988, pp. 32-41.

[Anderson89]

Anderson, J. D., Perkins, J. A., "AdaMAT ANALYSIS of AFATDS", 1st Annual Software Quality Workshop, August 1989.

[Barnes 84]

Barnes, J. G. P., Programming in Ada, Addison-Wesley Publishing Company, 1984, pp. 114-115.

[Complexity88]

"Complexity Analyzer Reference Manual." Dynamics Research Corporation, E-14646U, Andover, MA, September 30, 1988.

[Gorzela87]

Perkins, J. A., Gorzela, R. S., "Programming Paradigms Involving Exceptions: A Software Quality Approach", Fifth Annual Conference on Ada Technology, March 1987.

[Green89]

Green, Robert, "Army Gives Its Blessing to Controversial AFATDS", Government Computer News, August 21, 1989, p. 56.

[Keller85]

Keller, S. E., Perkins J. A., "Ada Measurement Based on Software Quality Principles", Washington Ada Symposium, March 1985, pp. 195-203.

[Keller89]

Keller, S. E., Perkins J. A., O'Leary, K., "Layering and Multiple Views of Data Abstraction in Ada: Techniques and Experiences", TRI-Ada 89, October 1989, pp. 625-640.

[Perkins85]

Keller, S. E., Perkins, J. A., "An Ada Measurement and Analysis Tool", Third Annual National Conference on Ada Technology, March 1985, pp. 188-196.

[Perkins86]

Perkins, J. A., Lease, D. M., Keller, S. E., "Experience Collecting and Analyzing Automated Software Quality Metrics for Ada", Fourth Annual National Conference on Ada Technology, March 1986, pp. 67-74.

[Perkins87]

Perkins, J. A., Gorzela, R. S., "Experience Using an Automated Framework to Improve the Quality of Ada Software", Fifth Annual Conference on Ada Technology, March 1987, pp. 277-284.

[PerkinsE88]

Perkins, J. A., "How Programmers Code: Impact on Quality", Ada EXPO 88, October 1988, session 22.

[PerkinsT88]

Perkins, J. A., "The Myth: Anyone Can Code the Software If the Requirements and Design are ...", TRI-Ada 88, October 1988, pp. 258-273.

[Sintic89]

Sintic, John H., Joiner, Dr. Harry F., "Managing Software Quality", Journal of Electronics Defense, May 1989, pp. 37-42,55.

[Temte84]

Temte, Mark, "Object-Oriented Design and Ballistics Software", Ada Letters, ACM SIGAda, November 1984, pp. 25-36.

ABOUT THE AUTHORS

Stan Levine is the Chief of the Technical Management Division for the Project Manager, of Field Artillery Tactical Data Systems, U.S. Army. He has a Bachelor of Science in Electronic Engineering and Master of Science in Physics from Monmouth College. He has also taken extensive post-graduate coursework in Computer Science and Computer Engineering. Mr. Levine has held positions on a wide variety of computer software and systems engineering projects. He is responsible for all technical aspects on many Army Command and Control systems. He is responsible for ensuring that all Army Fire Support Command and Control systems are interoperable. Mr Levine is presently serving as chairman of a working group tasked with the responsibility of maximizing the reuse and sharing of Ada software among the Army Tactical Command and Control Systems.

Jane Anderson is a member of the Software Research

Development Group at Dynamics Research Corporation. She has a Bachelor of Arts degree in Mathematics from Brown University and a Master of Science in Mathematics from the University of Lowell. She has written the functional requirements document for the Ada Complexity Analyzer, and tested portions of AdaMAT. She has used AdaMAT and the Complexity Analyzer to analyze over 2 million lines of Ada source for the Air Force, Army, and Navy. She has developed and presented training materials covering the operational aspects and capabilities of both tools for metric analysis.

John Perkins is a member of the Software Research Development Group at Dynamics Research Corporation. He has a Bachelor of Science in Mathematics from Purdue University and a Master of Science in Mathematics from the University of Illinois. He has been involved in the development of a math library and communication software (Digital Message Device), translators for multi-processor scientific computers (Burroughs Scientific Processor and Flow Model Processor), an attribute grammar-based translator-writing system (SSAGS), and static analyzers for assessing the quality of Ada source (AdaMAT and Ada Complexity Analyzer). He is currently involved in defining trust metrics for Ada and in specifying requirements for a rule-based consultant for aiding in the development of trusted Ada software.

A DEVELOPMENT METHODOLOGY FOR DISTRIBUTED REAL-TIME SYSTEMS

A.J. Clough*, R.F. Vidale** and T.J. Yuhas**

* C. S. Draper Laboratory, Cambridge, MA **Boston University, Boston, MA

ABSTRACT

This paper describes a project to create and evaluate a development methodology for distributed systems with hard deadlines for both periodic and aperiodic processing. The methodology covers the developmental phases of system specification, system architectural design, and system implementation. Tool support for the evaluation of performance has been specified and prototyped for the development and maintenance of distributed systems.

INTRODUCTION

While the use of Ada's tasking feature allows the software developer an elegant way to define actions that are logically executed in parallel, the actual execution of concurrent tasks can lead to non-deterministic behavior and poor system performance. The non-determinism of Ada tasking and concern about performance make it particularly important for the developer to model performance early in the development process. [BUHR88] Embedded applications with stringent performance requirements may otherwise not meet their requirements or require much fine tuning in order to do so. [LEHR89] When Ada is used in a *distributed* system, even more difficulties arise. [FIRT87] In addition to the issues of performance and the non-determinism of the Ada tasking model, there are requirements for system flexibility, maintainability, and reusability. Memory space is often severely limited. Allocation of software to processors also affects how well the system performs. [CORN84] Despite these obstacles, there is little guidance available for the development and maintenance of such systems.

In an internal research and development effort at C. S. Draper Laboratories, a development methodology for distributed systems with hard deadlines has been formulated. This methodology addresses all phases of the lifecycle of a system: specification, design, implementation in Ada, and maintenance. Though software development is emphasized, it should be noted that the methodology is really a *system* methodology that addresses both hardware and software issues. In the specification phase, the interaction of system software and hardware is precisely specified. Performance evaluation tool support for the development and maintenance of distributed systems has been developed in conjunction with methodology development.

In this paper, we present the engineering principles which guided our research efforts and outline our approach to significant development phases. Particular emphasis will be placed on the specification stage for which unique and descriptive icons have been defined and on performance modeling using the prototype simulation tool developed in the project. How our methods relate to the use of Ada in distributed systems will be stressed. The paper is organized into five major sections:

SYSTEM CHARACTERISTICS: Outlines the characteristics of distributed systems of particular interest and describes the distributed system being used to test the methodology.

METHODOLOGY DEVELOPMENT PRINCIPLES: Presents the principles guiding methodology formulation.

METHODOLOGY OVERVIEW: Describes the overall scope of methodology development.

SYSTEM SPECIFICATION PHASE: Describes the specification of a distributed system. Icons which support Ada as the implementation language are introduced at this phase.

PERFORMANCE MODELING: Introduces the use of PREDICT, a simulation tool, at the architectural design phase. PREDICT is used to determine the performance of a distributed system design before it is implemented.

SYSTEM CHARACTERISTICS

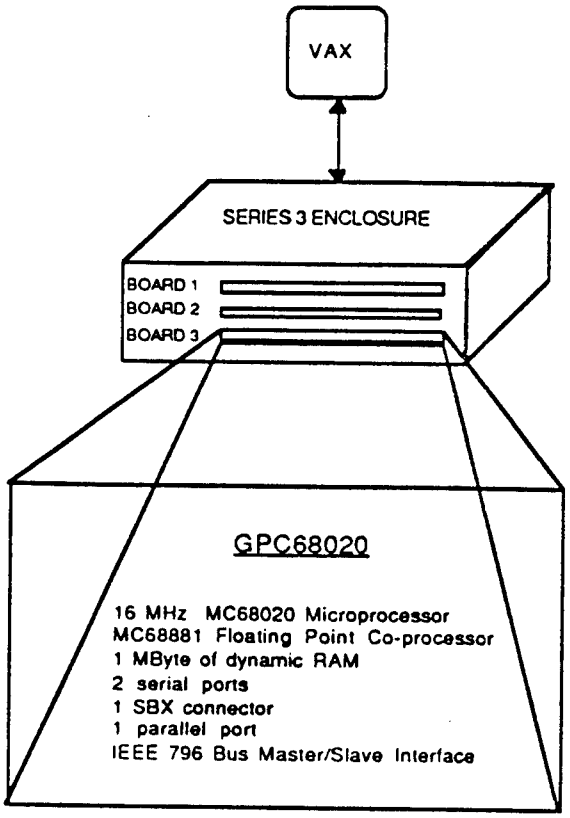
Distributed systems of specific interest in this project have the following characteristics: Overall memory capacity is limited. Hard deadline timing requirements exist for both cyclic and acyclic system activities. Tightly coupled processors which communicate via shared memory constitute a site within a larger distributed system. Sites are loosely coupled, communicating using message passing with other sites in the system. Each microprocessor in the system runs a single Ada program under control of a kernel supporting the execution of Ada tasks. Although communication between tasks which reside on different processors within the same site cannot be achieved using high level language constructs such as the Ada rendezvous in this kind of system, such tasks can communicate using shared memory.

The system is characterized by cooperative computing: programs running on different processors work together as a system. Systems of interest are *embedded* distributed systems which interact with larger systems that may include hardware. Software is developed using a host-target approach. A program that will execute on an embedded target computer is developed on a host computer which offers extensive support facilities.

The system is not dynamically reconfigurable; that is, tasks and data structures do not move from processor to processor when the system is running. The runtime software schedules periodic and aperiodic processes and it may manage some shared resources. Alternatively, a real-time operating system can be used to schedule processes and manage shared resources.

Though multiple sites will be fabricated in the future, the current testbed configuration used at Draper Laboratories for the IR&D project simulates a single site in a distributed system. This configuration incorporates three tightly-coupled Microbar GPC68020 boards housed in one enclosure, as shown below.

OVERALL SYSTEM HARDWARE



Each board communicates with other boards in the system using shared memory. In this configuration, each processor in the site is able to read from and write to the shared memory of any other processor in the site. The Verdix Ada Development System (VADS) is being used as a cross-development system. The ARTEWG (Ada Run-Time Executive Working Group) scheduler has been added to the Verdix runtime software to provide cyclic and acyclic scheduling of processes. The scheduling of processes uses a pre-emptive priority-based algorithm.

METHODOLOGY DEVELOPMENT PRINCIPLES

In creating the development methodology, the research and development team was guided by the following principles:

1. The development process should employ techniques which are formal and comprehensive, thus allowing the developer the ability to specify, design and implement the system being developed in the appropriate detail at each stage of development. [DAVI88]

2. Graphical representations of distributed systems are superior to textual forms in their ability to portray concurrency and in their ability to be understood by others. [SCAN89], [BUHR88], [LEHR89]

3. Synchronization issues and shared resource contention must be understood in the specification and system architectural design phases to avoid the high cost of correcting related design errors in the implementation and testing phases. [BUHR88] More effort in the early stages of distributed system development will have a direct and positive effect on the costs and maintainability of the resulting system. [BOEH87]

4. Executable models should be provided to assess the performance of a system architectural design during design and implementation. [BLOM89]

5. Techniques developed should be testable for completeness and consistency, preferably by CASE tools. In general, CASE tools should support all phases of system development and maintenance and should provide a smooth transition from one phase to another. [BLOM89]

6. Guidelines should be provided that will guide the developer to maximize the maintainability of a design while satisfying system performance requirements.

METHODOLOGY OVERVIEW

The major developmental phases in the building of a distributed system are system specification, system architectural design and system implementation.

System Specification Phase

In the system specification phase, both functional requirements and constraints must be specified. In our methodology, an activity is defined as an abstract unit which represents a system operation. System functionality is represented graphically in a form called activity interaction diagrams, which show communication,synchronization and the flow of information. In addition to specifying the necessary system functionality, the diagrams define the system boundary and allow some types of constraint requirements to be expressed graphically. The graphical specification is formal and comprehensive. In particular, it supports the specification of the types of interactions that Ada provides, such as timeout, rendezvous and the conditional rendezvous. As part of the system specification, a system specification dictionary is created which defines data items and data stores used in activity interaction diagrams. The internal system behavior, and hence the logical constraints imposed upon individual system activities, are specified using activity behavior diagrams. Space constraints, such as memory size, are usually identified as implementation constraints in the system definition phase. Timing constraints on the system and environment are specified in the specification phase in a timing requirements document. More detailed information about the specification phase can be found in the section of the paper entitled SYSTEM SPECIFICATION PHASE.

System Architectural Design Phase

The system architectural design phase begins with preliminary decisions about which system activities will be implemented in hardware and which in software. The physical attributes of the hardware must first be determined. These attributes include the number of processors in the system, how the processors are to be grouped into sites, CPU speeds, layout and size of shared memory, size of local memory, and performance of communication links within and between sites. The design of the software architecture can then proceed. Software architecture design will

include the determination of necessary runtime support software characteristics, the mapping of application activities into processes (tasks), the creation of software allocation units (packages), and the mapping of the software allocation units onto the identified hardware architecture. Prototyping system communication at this stage is appropriate in order to avoid common problems, such as starvation or deadlock.

In the system architectural design phase, the issue of object-oriented design versus performance-driven design must be addressed. Ada systems should be designed with performance and maintainability as key goals. Our approach is to employ object-oriented techniques to promote maintainability of the system using, as much as possible, objects as allocation units. If system behavior and performance are evaluated during the architectural design phase using performance evaluation tools, then the designer can determine where the design must be adjusted to meet performance objectives. Behavior and performance of the proposed architectural design are evaluated using a prototype simulation tool, PREDICT, to simulate the timing of the scheduling of processes, the processor contention for the bus, process interactions, and the process contention for shared resources.

System Implementation Phase

In the system implementation phase, separate Ada programs are written for each microprocessor in the distributed system. Each program consists of the set of software allocation units that have been allocated to a particular processor. Processes, routines invoked within a process, and allocation units will map into tasks, subprogram units, and packages respectively. When objects are used as allocation units, the encapsulation of the object data type and the operations on that data type create a software allocation unit that has significant potential for reuse. [BOOC87] If the reusable component is itself a particular instance of a more general reusable component, it may be appropriate to create a generic unit to increase its potential for reuse. Since there is one Ada program per processor, re-allocation or re-configurations due to system maintenance will necessitate changing the Ada source code for all processors that are re-configured. However, the encapsulation of data stores and their associated processes into packages will facilitate system changes and upgrades during the maintenance phase.

SYSTEM SPECIFICATION PHASE

One goal in the system specification phase is to provide a graphical representation of a distributed system that will clearly and adequately portray the concurrency inherent in the system, thereby enabling the developer to examine issues of system interaction and shared resource contention. Another goal of this phase is to capture implementation constraints when applicable. Existing notations, as found in structured analysis/real time [WARD85] [HATL88], do not adequately portray the requirements of system/environment interactions, timing, synchronization, and shared resource management inherent in the problem description. Therefore new graphical representations reflecting required system functionality and behavior, and fully capturing the interactions between the system and its environment, have been introduced. The activity interaction diagram captures system functionality. Structured descriptions, called activity behavior diagrams, capture behavior. In addition, the graphical representation of the specification clearly shows concurrency and provides ways to represent a variety of activity interactions.

The approach adopted differs from the data-flow approach in its emphasis on the detailed specification of the characteristics of the interactions between the system and its environment, rather than on the data interchanged. The activity interaction diagram defines the system boundary, identifies all important system activities, describes interactions between environmental entities and system activities, and shows the information that crosses the system boundary. Interactions between system activities are also specified without specifying who calls whom, since this is a software design issue. Information that flows between system activities is also shown.

A context diagram provides the first view of the system under development by

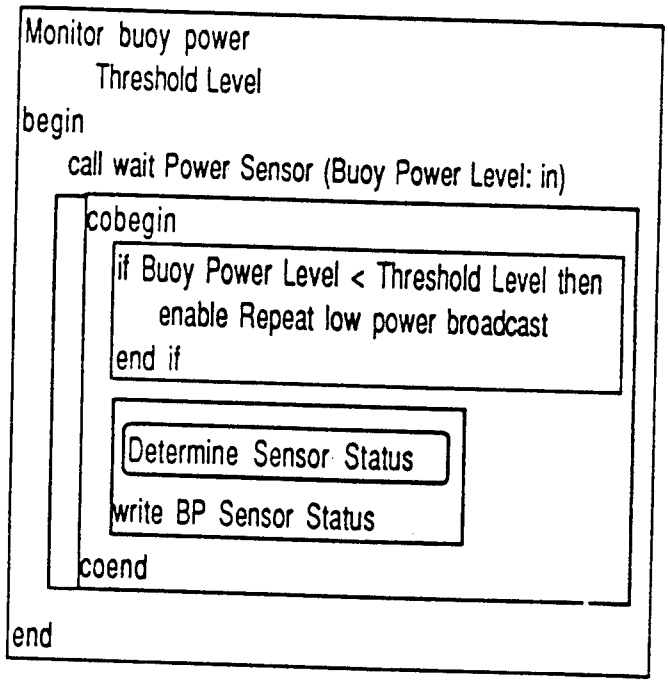
specifying the system boundary, thus separating the system being developed from its environment. A single object represents the system being developed. The environment is a set of real-world objects that interact with this system. In Figure 1, a context diagram for a sample application of an instrumented buoy is shown. The parallelograms represent environmental objects, while the single box in the center is the system to be developed. Solid arrows represent directed interactions where data is passed, and dotted arrows represent directed interactions without data being passed. Arrows indicate one entity "calling" another entity to initiate the interaction; if the direction of the call is undetermined, lines are used. The dotted arrows indicate the direction of a synchronization call; lines represent a synchronization "handshake" in which the calling relationship is not specified. Solid arrows and lines are labelled with the data item(s) that is being passed; the small arrows associated with the data item indicate the direction of the data flow.

A level 0 or top level system specification diagram is shown in Figure 2. In this diagram, boxes represent system activities and two encapsulating horizontal parallel lines indicate a data store. Numbered circles are used to reduce the number of crossing lines on the diagram. Where resource management is necessary, new graphical icons have been developed to represent resource managers that provide access to shared resources. The box "Send message" has a shared resource manager icon preceding it, in this case indicating a priority based processing of calls to access the resource. The solid black boxes on the edges of the diagram represent the connections to the environmental objects that were shown in Figure 1. Thick lines cross the system boundary. Thin lines are used within the system being developed. Thinnest lines are used for reads/writes to data stores.

Note that the basic unit of decomposition in the specification phase is a system activity, not an object. We feel that the specification stage of distributed system development is not the appropriate phase to introduce object-oriented techniques. The diagram technique adopted provides a more convenient way to specify concurrent threads inherent in the system. It makes effective use of hierarchy and is therefore better for specifying large systems. To introduce object-oriented techniques at this phase is encroaching upon design decisions that should more properly be made at the design stage. The full benefits of the object-oriented approach can be realized when these techniques are applied during the architectural design phase.

Each of the activities in the level 0 diagram can be further decomposed if they are still too complex to be easily understood. Such repeated decomposition of activities in lower level diagrams (level 1, level 2, ---) provides hierarchy and information hiding which promotes understanding of large complex systems. The lowest level diagrams of the graphical structure contain activities that perform system functions that can be understood without the need for further decomposition.

The internal behavior of the identified system activities is then specified using structured descriptions, called activity behavior diagrams. Behavior diagrams, created using action diagram techniques [MART85], supplement the activity interaction diagrams of the system by indicating the execution order of activities and the possible parallelism in the system. An activity behavior diagram from the buoy application is shown below.



Activity interaction diagrams and activity behavior diagrams can be drawn by hand or using CASE tools. CASE tools can also be used to automate the checking of diagrams for completeness and consistency. DESIGN 2.0, developed by Meta Software Corporation, was used by our project because its icon building capability allowed us to incorporate our new icons into the data flow diagram format already supported by the tool. Unfortunately, use of DESIGN 2.0 does not permit the automatic checking for completeness and consistency.

Timing requirements on the system and environment are specified using stimulus/response statements:

> SOS Request / Write broadcast message (SOS signal) within 60 milliseconds

and scheduling requirements statements:

> Repeat SOS broadcast every 60 milliseconds

All such requirements are compiled in a timing requirements document. A system information dictionary, including data and control information, completes the specification. A more complete description of the techniques and methodology of system specification can be found in "Development Methodology for Hard Real-Time Distributed Systems." [YUHA89]

PERFORMANCE MODELING

Altering system specification or design to achieve performance goals is both easier and more cost-effective than changing the system implementation. It has therefore been a goal of our methodology to assess performance of the distributed system under development as early in the development process as possible. To this end, we looked carefully at an executable model of the system specification. Though such an executable model might yield some information about system functionality, it became obvious that information about the behavior and performance of a distributed system can only be determined after the system software has been partitioned and mapped onto a given system hardware architecture. Process synchronization and communication across processors cannot be modeled in any pre-allocation view.

Before attempting to model a distributed system, the system must be partitioned into hardware and software activities. The software activities must then be mapped into processes (tasks). Allocation units can then be defined using object oriented techniques to synthesize the software allocation units. After this has been done, the software allocation units are mapped onto the identified hardware architecture. After a system architectural design has thus been chosen, the behavior and performance of the system design can then be evaluated by simulation.

The purpose of evaluating the system architectural design at this stage of system development is twofold: 1) to insure that the proposed system design will behave as required by the specified behavioral requirements, and 2) to provide an early indication of whether or not the proposed system design can satisfy its timing requirements. Because all of the implementation details are not known during design, testing the system architectural design cannot guarantee that the proposed design will satisfy all of its timing requirements after it is implemented. Rather, this testing can be used to verify system behavior, determine whether or not a proposed design is *likely* to satisfy its timing requirements, and compare the performance of alternative system architectural designs.

Currently there is a lack of analytical methods and tool support that will enable a developer to determine the performance and behavior of hard real-time distributed systems before such systems are completely implemented. Because of this, systems are often implemented and then tested to verify their timing requirements. If timing requirements are not met, the system is revised and re-tested. This approach is time consuming and prone to errors and may never provide satisfactory results. [BUHR88]

A system model must include at least the following details if it is to provide a realistic assessment of the proposed system design. It should model the interactions between all processes whether the interactions take place on the

same processor or between processors. It should model the scheduling algorithms, services provided by the runtime system, and the services provided by the system hardware to the application software. The more accurately these characteristics are modelled, the more accurate the assessment of system performance will be.

A system simulation tool could virtually eliminate behavioral errors and the need for performance tuning. It is essential, however, that the simulation be a cost-effective option for the system developer. In addition to the cost of developing tool support, effort must be expended to gather necessary modeling information through such means as benchmarking of runtime system software and system hardware. If the simulation cost is greater than the expected value of the information it provides, or if tools used for the simulation are not properly designed to provide useful information without excessive tailoring, performance evaluation at this stage may not be cost-effective. On the other hand, if the system will require extensive maintenance, then even a high cost for performance modelling may be cost-effective due to the valuable assistance of modelling in the maintenance phase.

We believe that a system simulation tool can provide a desirable and cost-effective alternative for the development and maintenance of real-time distributed systems. A prototype of such a simulation tool, which allows simulation of hard real-time uniprocessor systems scheduled by a cyclic executive, has already been built. The user can specify CPU characteristics, the type of process for each process in the system and the scheduling algorithm(s). The tool allows the designer to verify that all deadlines are met, under whatever conditions the designer specifies, and reports at the end of a simulation the percent of CPU idle time. The behavior of the system can be reported in either a textual or timeline format.

An expanded tool, PREDICT (Performance Requirements Evaluation of Distributed Computer Targets), which will model distributed systems, is currently being specified and built. PREDICT will provide simulation of a site in a distributed system. It will simulate the scheduling of cyclic and acyclic processes, asynchronous interrupts, the synchronization between any processes within a site, and the competition among processes for shared resources. Its ability to simulate the interactions between processes that are on different processors within a site is essential for accurately determining the performance of the site. These modeling capabilities will assist the developer in the mapping of software allocation units to processors within a site. PREDICT shows the system behavior of a given system architectural design so that feasible designs with different performance capabilities can be measured and compared.

The use of reusable components in distributed systems will be supported by PREDICT. For software components to be usable in systems with stringent performance requirements, information about their performance should be available. Using this performance information, a simulation tool can show how the use of the component will affect system performance.

A similar benefit will accrue during the maintenance phase. When new requirements are added to a distributed system or when existing system requirements change, it is advantageous to know the impact of the changes before they are fully implemented in the system. PREDICT can be used to gauge the impact on the existing system design and thereby help the developer to assess the effect of the proposed changes on system performance.

CONCLUSION

In this research and development project, we have faced some of the difficult challenges that face any real-time distributed system developer who wishes to use Ada in applications with hard deadlines. Neither adequate system development methodologies nor adequate tool support currently exist.

To overcome this deficiency, we are developing a methodology which will support all phases of the system life cycle, accompanied by a review of current CASE technology and the specification and design of tools that are needed

to supplement those CASE tools currently available.

The graphical specification of system/environment interactions requires specification notation which can express the functionality that is later provided in the Ada rendezvous model. Consequently, new specification techniques must be formulated. Similarly, tool support must provide the ability to model capabilities available to the developer in Ada. New developments in both of these areas have been described in this paper.

Development techniques are now being applied to a representative problem, implemented on a tightly-coupled site. Future research efforts will expand the scope to encompass a study of issues that arise when loosely-coupled sites are added.

The use of Ada in distributed systems presents challenges and difficulties but, at the same time, new power and capabilities. With the proper methodology and tool support, developing and maintaining high-performance distributed systems using Ada in a cost-effective manner should be a realizable goal.

BIBLIOGRAPHY

[BLOM89] R. Blomseth, "RISC and Real-Time", *Embedded Systems Programming,* August 1989.

[BOEH87] B. Boehm, "Improving Software Productivity", *IEEE Computer,* September 1987.

[BOOC87] G. Booch, " Software Engineering with Ada", Second Edition, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[BUHR88] R. Buhr, "Visual Prototyping in System Design (preliminary report)", Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, CANADA, Sept. 14, 1988.

[CORN84] D. Cornhill, "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets," *IEEE Computer Society 1984 Conference on Ada Applications and Environments,* 1984.

[DAVI88] A. Davis, "A Comparison of Techniques for the Specification of External System Behavior," *Communications of the ACM,* vol. 31, no. 9, Sept. 1988.

[FIRT87] R. Firth, "A Pragmatic Approach to Ada Insertion", International Workshop on Real-Time Ada Issues, *Ada Letters,* vol.7, no. 6, Fall 1987.

[HATL88] D. Hatley and I. Pirbhai, Strategies for Real-Time System Specification, New York, NY: Dorset House Publishing Company, 1988.

[LEHR89] T. Lehr et. al., "Visualizing Performance Debugging", *IEEE Computer,* vol. 22, no. 10, October, 1989.

[MART85] J. Martin and C. McClure, *Diagramming Techniques for Analysts and Programmers,* Englewood Cliffs, NJ: Prentice-Hall, 1985, pp 183-209.

[SCAN89] D. Scanlan, "Structured Flowcharts Outperform Pseudocode: An Experimental Comparison", *IEEE Software,* vol. 6, no. 5, Sept. 1989.

[WARD85] P.T. Ward, S.J. Mellor, Structured Development for Real-Time Systems, Volume 1-3, Englewood Cliffs, NJ: Prentice-Hall, 1985.

[YUHA89] T. Yuhas, A. Clough, R. Vidale, "Development Methodology for Hard Real-Time Distributed Systems", CSDL Report, November, 1989.

AUTHORS

Anne Clough, Section Chief in the Engineering Services Directorate at C.S. Draper Laboratory, has been actively involved in introducing Ada technology to CSDL. Toward this end, she has developed and taught courses in Ada and software engineering, participated in Ada projects, provided support for Ada compilers and tools, and been involved in the benchmarking of Ada products. She is presently principal

investigator of an internal research and development effort directed at developing a methodology for hard real-time distributed systems. In an earlier IR&D effort, she participated in the development of a graphical analysis tool, the Embedded Ada Execution Analyzer (EAEA), which generates timing diagrams after the execution of an Ada tasking system on an embedded target.

Dr. Vidale has been a member of the faculty at Boston University since 1964, serving as Chairman of the Department of Electrical, Computer, and Systems Engineering from 1971 to 1981. He has been involved in applications of structured programming and software engineering since 1977. Dr. Vidale first began teaching software engineering and Ada in 1982. Since then, his research has focused on Ada design methodologies for embedded computer applications. Dr. Vidale has consulted in the areas of software engineering and Ada at C.T. Main, Inc., GTE, US Navy, MITRE Corporation, Data General Corporation, C. S. Draper Laboratory, Kollsman, and The Analytic Sciences Corporation (TASC).

Mr. Yuhas is currently a Research Assistant at Boston University in the Electrical, Computer, and Systems Engineering Department. His research includes the evaluation of Computer Aided Software Engineering (CASE) tools for use in developing a methodology for specifying and designing real-time, distributed, embedded systems. In addition, he is working toward a MS in Software Systems Engineering; his advanced graduate work includes embedded computer software design and applications of formal methods.

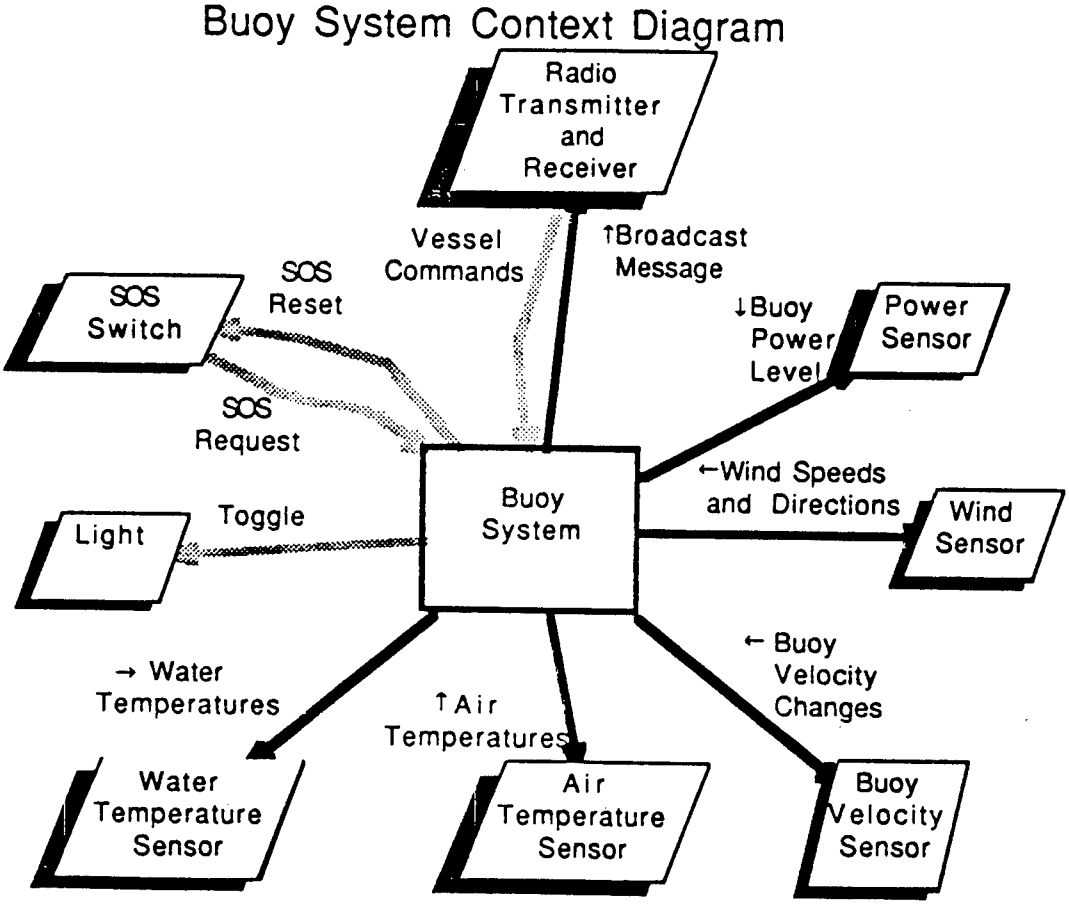


FIGURE 1

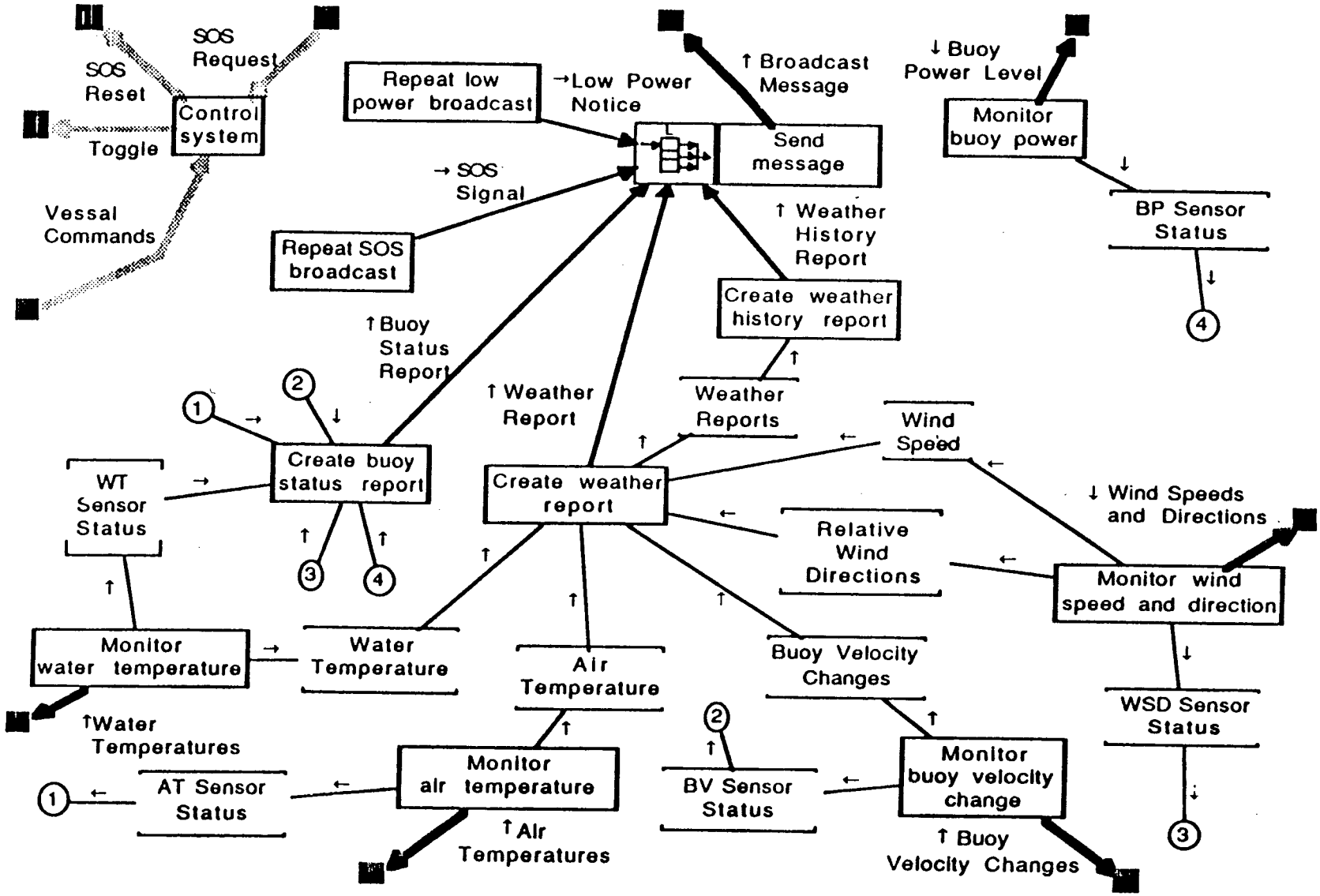

SOS Request
SOS Reset
Toggle
Vessal Commands
Control system
Repeat low power broadcast
→Low Power Notice
↑ Broadcast Message
Send message
→ SOS Signal
Repeat SOS broadcast
↑ Weather History Report
Create weather history report
↓ Buoy Power Level
Monitor buoy power
BP Sensor Status
↑Buoy Status Report
↑ Weather Report
Weather Reports
Wind Speed
Create buoy status report
WT Sensor Status
Create weather report
↓ Wind Speeds and Directions
Relative Wind Directions
Monitor wind speed and direction
Monitor water temperature
Water Temperature
Air Temperature
Buoy Velocity Changes
WSD Sensor Status
↑Water Temperatures
AT Sensor Status
Monitor air temperature
BV Sensor Status
Monitor buoy velocity change
↑Air Temperatures
↑ Buoy Velocity Changes

FIGURE 2

A Development Interface for an Expert System Shell

Jill E. Courte Verlynda S. Dobbs

Wright State University
Dayton, Ohio

Abstract

This paper describes a continuing effort to produce a frame-based expert system shell, written in Ada, and incorporating AI and object-oriented techniques. The first stage of producing the *Ada eXpert System*, or *AXS*, involved implementing the inference engine, allowing backward and forward chaining through a knowledge base consisting of frames and associated rules.

The focal point of this paper is the second phase of the implementation of AXS, the design of the development interface, the means by which an expert system designer constructs a knowledge base to be run by the inference engine.

The AXS system and inference engine is briefly outlined, followed by discussion of interface development techniques, expert system shell development interfaces and description of the implementation of AXS.

Introduction

The use of expert systems in defense and related software has been limited, in part due to problems with reliability and maintainability of large scale projects implemented in traditional AI languages. Little commercial effort has gone into creating expert system shells in Ada, and many AI researchers resist Ada as a viable option for AI projects. One notable advantage of traditional AI languages over Ada is their suitability for rapid prototyping. However, Ada offers improvements in reliability and maintainability of large scale systems. There is potential for the use of expert systems in defense related systems, and since Ada is the required DOD language, the use of Ada in expert systems deserves attention.

AXS is by no means the first or only attempt to blend AI and Ada. Previous recommendations for implementing AI applications in Ada include the use of variant pointers to simulate variant list structures, and the use of task types as dynamic objects [6]. Variant pointers provide the means to overcome Ada's strict typing rules, allowing for simulation of typeless data objects, an important concept for smooth operation of an expert system inference engine. Utilizing task types as dynamic objects was explored in the implementation of the development interface, and will be discussed later.

The AXS Inference Engine

AXS is a tool to develop an expert system which executes rules from a knowledge base developed in frames. Frame attributes are stored in slots, which may have rules and associated demons, and frames may inherit attributes from a parent frame. Communication between frames is achieved either through message passing, or through the parent-provided linear inheritance structure

The inference engine is implemented using an object oriented style of programming. Ada is not an object oriented language, but does provide the means for representing data abstractions as objects. True object oriented programming languages use frame-based knowledge representation methods, and the inference engine components of frames, slots, rules, demons, messages, etc., lent themselves nicely to an object-based, rather than procedural, implementation [6].

Conclusions drawn from the design and implementation of the inference engine correspond with the opinions of others, that is, that Ada is suitable to be the base language of AI applications with some limitations. For instance, Ada is reasonable for an application such as an expert system shell which does not directly depend on language intrinsics for symbolic processing. As previously noted, Ada does not provide a means for rapid prototyping, but may offer advantages in terms of reliability, reusability, and maintainability [5] [6].

Following the successful completion of the inference engine, the second phase of implementing the expert system shell involved the design of the development interface. The state of interface design in general will be briefly reviewed, followed by discussion of the AXS development interface.

The Human-Computer Interface

Human-computer interface researchers as a body agree on only one thing: defining, implementing and evaluating a human-computer interface is difficult. Taken to a ridiculous simplicity, human-computer interaction requires a human, a computer, and some kind of task. However, humans and computers are both complex ar d variable entities, making a common interface tightly dependent on the computer task, the user, the computer hardware and associated interface software. Despite the complexities, there seems to be a general assumption that there are some universal "goods" to

include in an interface, and some universal "bads" to avoid. Much research in the past decade has been directed toward determining what comprises a good interface, both from the human factors and computer science standpoints.

The Human

Human factors researchers and cognitive psychologists attempt to discover if there are innate human characteristics which facilitate human-computer interaction, and consequently define an optimal interface. Typically, human factors oriented researchers assume that metrics such as time or error comparisons indicate the utility of an interface. It is often not uncommon to see interface evaluation tied in with accepted cognitive theories, such as G. A. Miller's 1956 hypothesis that human short-term memory is limited to approximately seven "chunks".

As an example of human-factors research, much effort has gone into trying to discover if either mice, text keys, or arrow keys are superior from the user's point of view for data input [1] [4] [15] [21]. Not surprisingly, there is no consistent clear-cut superiority for one over the others between experiments. Much of the research in the human factors field has produced similar conflicting results, and suffers somewhat from lack of replication and unifying theories [21], in addition to small numbers of experimental subjects and questionable experimental assumptions. Landauer [16] also points out that human factors theories are not often applied to the creative phase of engineering an interface, but are used after the fact as measures of usability or trouble shooting guides, and in addition, that the field lacks an applied science.

This criticism is not to discount human factors research, merely to illustrate the difficulty of trying to draw conclusions from a few factors drawn from the vast number at work in a human-computer interaction. And in fact, there is one element which most people seem to agree is important—consistency, although this has recently been questioned [11].

The Computer

Research and advances oriented to the mechanical partner in the human-computer interaction center around computer hardware and technology, operating systems, graphics capabilities and communications protocols. In addition, there have been conceptual changes in the way interface construction is viewed.

Where just a decade ago the standard interactive interface to a computer was character and line oriented, most often restricted to keyboard input, today's most sophisticated interfaces are graphical full screen windowing systems, utilizing bitmapped displays and varied mechanical input devices. Yesterday's interface code was generally firmly embedded in the applications code, non-portable and non-reusable [7] [13]. Today the concept is layers. At the minimum, an application layer, a communications layer and an interface layer, as separate as possible, communicate through well defined software interfaces. This separation is generally know as *dialogue independence*, promoting reuse and ease of modification.

Implementation of state of the art interface designs is complex and unwieldy, and especially difficult is implementation of the currently popular direct-manipulation interfaces, often referred to as *WIMP* interfaces, for windows, icons, menus, and pointers, or windows, icons, mice, and pop-up/pull-down menus, depending on the source. As Myers [18] puts it,

> As interfaces are easier to use, they become harder to create.

As illustration, one survey of expert systems estimates that 30 to 50 percent of application code is dedicated to the interface [2]. By informal evaluation, sophisticated graphic interfaces are closer to the high end of this estimate, depending on the system used to create the interface. Looking at one application containing 100,000 lines of code, 60,000 lines, or 60 percent of the code was dedicated to the interface [22]. Large numbers of lines of code are not necessarily indicative of complexity, and may at best provide strong argument for the need for reusable interface components. However, consider that this interface software controls multiple devices, with asynchronous event handling and a high need for efficiency between i/o and application events [18], and it is not hard to imagine the inherent difficulty of the problem.

To reduce the difficulty of creating a sophisticated user interface, toolkits and development systems have been created. Toolkits are libraries of functions and objects which provide the necessary elements for an interface, for instance, i/o functions for input devices, or code to produce menus, icons, scrollbars, and so forth. Development systems combine these tools with control and analysis components, and a programming framework to provide an environment for interface development and management. In theory, development environments will allow intimate involvement in the interface creation process by non-programming design and engineering professionals, in "al a carte" fashion i.e., a menu from here, scrollbars from there, which can be combined and tested at a very high level.

In spite of their conceptual attractiveness and promise, toolkits and interface development systems are currently not a panacea, and generally suffer from being complex languages and/or systems in their infancy. Hopefully, the shortcomings will be worked out to produce flexible and unrestricted environments, useful to a wide range of disciplines.

Interface Design

It is not at all difficult to find opinions about what constitutes a good interface. Isolating the elements that cause an interface to be perceived as good by users is much more difficult. In addition, the rapid evolution of interface technology in the past few years which has produced complex

interactive and direct-manipulation interfaces has dramatically changed our perception of what is even an *acceptable* interface, let alone a good one. There really aren't any hard and fast rules about interface design, although there are elements which seem to contribute to an overall perception of desirability and utility across a variety of users. These are sometimes distinguished as *principles* and *guidelines*. Also important are the assumptions that interface researchers and designers make about the design process.

Assumptions

There seem to be a few assumptions that provide a background for interface design and evaluation. These generally aren't explicit, and are certainly not claimed to be universal.

- *For novices, the most comfortable interface is the closest to reality.* The desktop metaphor illustrates this assumption that people will learn more easily if computer objects and acts, for instance, files and file deletion, are related to familiar objects, like folders and waste cans.
- *Expert users desire more efficiency and fewer keystrokes.*
- *The targeted user population is very important in design decisions.* This seeming truism actually presents some inherently flawed reasoning, that is, that *most* of any given group of users of a computer program are homogeneous. There doesn't seem to be any evidence that this is true, or even reasonable. Obviously, there are some aspects of the user population that are important, for instance an advanced linear algebra program would not seek to teach basic algebra, but it would be silly to assume that all mathematicians had good visual-spatial skills. It can be important to be aware if the targeted group will be expert or novice computer software users, but on the whole, designers should avoid making generalizations about the group of intended users.
- *Human factors analysis will make a difference in the design process and the success of the product.* Rubin [21] cites an experiment which showed that, among other things, that users made 0.7 fewer mistakes when using hardcopy help (printed manuals) versus with online help. While statistically significant, this finding seems to lack applied significance. At the very least, as Rubin points out, manuals get lost and stolen. Care needs to be taken when generalizing the results of contrived laboratory experiments to a full-scale, real situation. This is not a dismissal of human factors research, only a caution about overgeneralization.

Principles

Principles are rules which cover overall interface design, while guidelines are the results of testing and/or implementing certain interface conditions. It is emphasized that these are by no means universal edicts, and should not be followed blindly without thought and research. As Norman [19] puts it,

> Statements that proclaim "Consider the user" are valid, but worthless. We need more precise principles.

Most designers have their favorite principles, to mention a few: know the user; minimize memorization; optimize operation; provide good error handling; focus on users and tasks early in the design process; prototype and experiment; use iterative design; separate the interface from the application; and provide help to the user [1] [10] [19].

Four additional principles deserve elaboration:

- *Provide the greatest number of options possible.* Humans are very adaptable and will generally adjust to working with an interface which they may not favor. However, especially for commercial success, it is important to have satisfied customers. Since humans are a varied lot, users of an interface should have choices where possible, such as choice between keyboard, mouse, or arrow key input.
- *Be consistent.* It is difficult to read anything about interface design without the admonishment to be consistent. Prevailing opinion indicates that it doesn't matter so much what style you choose, so long as you're consistent, and users know what to expect. For instance, truncation and vowel deletion are two popular methods of abbreviating commands. Neither is inherently superior to the other, but both produce equally good interfaces when used consistently throughout.
- *Learning and use are different.* Grudin points out that much of the empirical work on consistency has focused on learning or transfer of learning. This is true throughout user interface research. An interface which holds the user's hand by minute guidance throughout the interface may be easy to learn but frustrating once mastered. A concise command line system may be difficult to learn, but efficient to use. Obviously, some balance between learnability and usability is necessary, and here is a place where "Know the user" is a helpful edict. For instance, an interface designed entirely for use by computer professionals will require less hand-holding during the learning process than one targeted toward a general population. Good documentation and manuals can contribute to a balance here, as well as online tutorials.
- *Make sure the user thinks the system does what the designer thinks the system does.* It is very important that the user's mental images of the system or software be accurate. This is another argument for good documentation and manuals, as well as for careful phrasing of commands and displayed information.

Guidelines

As mentioned, guidelines are methods which are thought to produce good interfaces. There are hundreds of guidelines available throughout the user-interface literature, for the most part unproven and inconsistent. Smith and Mosier [23] provide over 900 guidelines which they compiled from existing literature and research. Many of these seem to fall into the common sense category, such as advising designers to avoid uncommon abbreviations or to separate paragraphs by blank lines. Importance is also placed on consistency. These and other guidelines mentioned in the interface literature will be mentioned in more detail later as they pertain to the AXS development interface.

Dialogues

A dialogue is the actual exchange between the human and computer that is supported by the interface software [13]. Dialogues fall roughly into two categories, sequential and asynchronous. Sequential dialogues are conversational, or involve turn-taking. Asynchronous dialogues are event driven, and involve multiple threads or tasks. Within these groups, there are many types of user interaction techniques to support dialogues, and a single interface may use one or more of the following:

- Command line dialogues, as in most operating systems
- Programming language dialogues
- Natural language dialogues
- Menu systems
- Form filling, or template systems
- Iconic interfaces
- Window systems
- Direct manipulation, or using a mouse to choose options or to push or pull objects about the screen
- Graphical interaction, in multiple dimensions, such as a CAD system

Interfaces are formed using these techniques singly or in combination. Obviously, there are varying ways to implement these, and most designers and users of interfaces have their favorites. Guidelines exist for all of these, as well as opinions about their usefulness. Again, an interface which allows the most options is probably best, but there is nothing inherently wrong with a well-designed and well-implemented interface using only one style if that is suited to the interfaced task.

Design Issues

During the design phase of an interface the following issues need to be dealt with:

response time The importance of response time is obvious; a system requiring long user waits will almost certainly be less than successful.

user help Studies indicate that any help at all reduces errors [21].

error handling Many issues need to be addressed regarding interface error handling. Not so straightforward is defining what is an error. For instance, is it an error if a system allows a file to be overwritten? This question, often referred to as "idiot-proofing", is often an emotional one. At one extreme are interfaces which rest most of the responsibility on the shoulders of the user, with terse command line input. At the other are interfaces which require confirmation of nearly everything. Generally it is felt that an interface should at the most only confirm actions which could result in irreversible change. The issue of learning versus use is important here, as a user may require more "user-friendliness" while learning, but desire less when comfortable with the interface.

command style As previously stated, consistency is probably an important aspect of commands, for example, using the same verbs for actions, and having consistency between different forms of the same command. Consistency between different forms refers to interfaces which offer input options. For instance, an interface may have pointer input as well as command key input, and in this instance the actual command words should be closely related. The command name should also suggest the function of the command, as in *Delete*. Problems can arise with this, for instance, some systems use the word "file" as a noun while others use it as a verb. Abbreviating command names requires care to make them as clear as possible. Ambiguous commands are potentially frustrating for users who try to guess unknown commands. If the command abbreviation method is consistent, for example, all vowel deletion, or all truncation, a user stands a greater chance of successful guessing. [21].

Interface Design Overview

At this point, it may seem that the field of interface design is a chaotic anarchy, with non-binding principles and inexact guidelines. And, in fact, that is almost the case. Efforts are currently underway by several commercial venders to set interface standards for interface development systems and graphical interface tools [14], while standardized interface design environments already exist for some machines.

It is unclear whether an overall industry standardization can be achieved, or even if it is completely desirable.

There are those who believe all interfaces should be virtually identical for ease of learning, use, and design. However, humans, computers and their associated tasks all vary widely, and present endless variation and difficulty when combined. In addition, there is the risk of attributing too much importance to the superficial aspects of the interface, rather than to the task or tasks underneath the interface. It is also important to confirm via prototyping and experimentation that a system is as appealing as it looks.

Expert System Development

Expert System Development Systems

Expert system development systems, or shells, provide generic, standardized structures for construction of expert systems. There are dozens of expert system shells commercially available, with wide variation among them with respect to sophistication, the class of problems they are targeted to solve, power, underlying data representation, and transparency of underlying concepts to the developer. For instance, shells may offer choices between frame or rule based systems, or between forward and backward chaining, or blackboard systems.

The developer may be aware of the data structures and implementation details involved, for instance, actively manipulating the structures used by the inference engine, as is generally the case with shells targeted toward more expert programmers. Alternatively, the targeted developer is unfamiliar with AI and programming concepts and deals only with abstractions. In common to each other, expert systems tools provide, or should provide, an environment to develop, test and modify an expert system in much less time than programming from scratch [8] [9] [12] [20].

The Development Interface

Expert system shells are generally comprised of two basic parts, the inference engine and the development interface. As stated, the focus of this project is on the development interface. As shown in Figure 1, modified from Gevarter [9], the development interface can potentially be composed of many parts. A very comprehensive shell would result from the inclusion of all of these items, but many acceptable shells include only a portion of these features.

The *knowledge base* contains the information needed by the inference engine to create the expert system. For instance, in AXS, a frame based shell, the knowledge base defines the frames, along with their types, inheritance information, associated slots and their confidence factors, initial values and demons, in addition to rules. Some expert system shells consider this knowledge base to simply be an instance of a computer program, and the knowledge base is built with a word processor or standard editor. Others provide an extensive development environment, with graphics and choices between keyboard or pointing device input. Of these, the majority provide a menu-driven interface, and online help is almost universal. Also incorporated

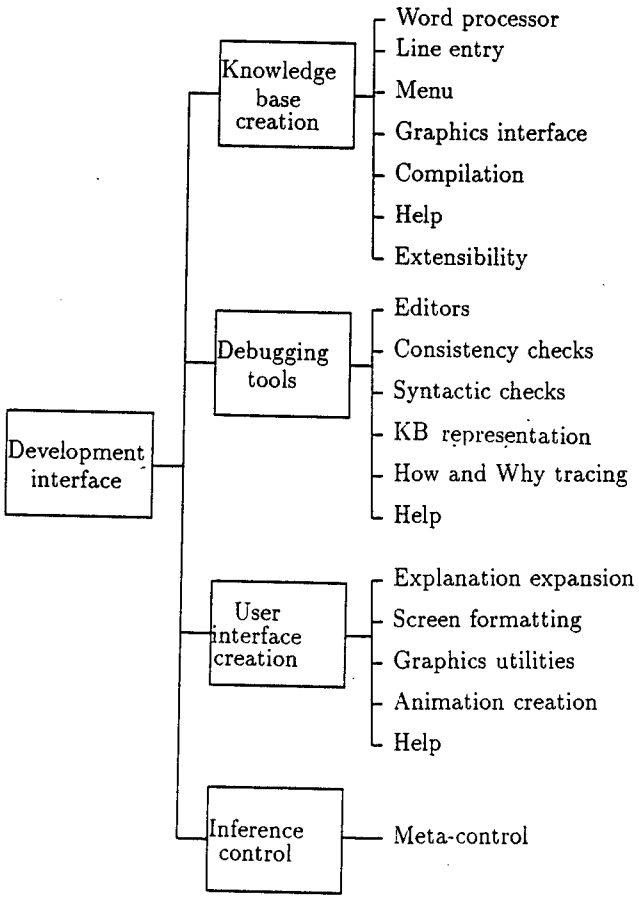


Figure 1: The development interface (adapted from Gevarter)

into many development interfaces are form-filling dialogues, or templates which allow the developer to fill in the parts of the knowledge base structures. Rules are popular template interfaced structures, where the interface presents to the developer a form with if and then sections which can be accessed through cursor or pointer movement and easily edited.

Debugging Tools allow for easy modification and testing of the expert system. The most complete environments provide the means to run and test a system and easily return to the development interface for modification. Varied trace facilities may be provided to follow the expert system as it runs, and in addition, graphics capabilities may provide on-screen representations of the underlying structures. An important debugging tool is internal consistency checking, for instance, not allowing the developer to mention an undefined variable in a rule, or making sure that all structures or variables mentioned are declared. This becomes particularly important in large knowledge bases where it is difficult, if not impossible, to determine manually if conflicts or redundant definitions exist [17].

Creation of an *end user interface* is also a part of the development process. This can be as simple as screen prompts and line entry of user input, or as elaborate as graphics interfaces. Also included here are explanation facilities which provide help or allow tracing of the logic of the system for the end user of the expert system.

Development System Evaluation

In one study which compared four PC-based expert system shells [24], issues which were noted by the researchers when evaluating the development interfaces were the presence or absence of online help or tutorials, documentation, time to learn the system, the method of building rules, targeted developer expertise, limits of data structures and ease of debugging. Specific criticisms were inadequate or poorly written help and documentation, or documentation which was too slanted toward novice users. Systems which did not allow direct manipulation of structures were criticized for a loss of control over the knowledge base. Also criticized was a system which placed numerical limits on data structures. Consistent with the guidelines mentioned earlier, systems which offered choices for development methods and also those which offered graphic interfaces were viewed most favorably.

AXS Development Interface

When designing an interface, the intended platforms are very important. Dialogue independence, or the separation of the application code from the interface code, makes it easier to port applications to varying platforms, but knowledge of the targeted platforms is necessary in the pre-design and design phases in order to select supporting software. Commercial expert system shells have the manpower and facilities to target their product to many computer systems. Research efforts, such as AXS, generally do not. Thus, it was felt that first priority would be given to constructing as generic an interface as possible which could be supported by any Ada compiler. One advantage of using Ada as the implementation language is Ada's standardization. The AXS system should work, with no or minimal change, on a variety of compilers, although the use of tasks may be restrictive.

Giving first priority to a portable system eliminated some of the power and "gee whiz" potential of a sophisticated graphics interface, but insured that the system would be available to as many users as possible. In addition, since AXS was to be written entirely in Ada, the set of graphics tools available was greatly reduced. In keeping with common wisdom, the interface code is as separate as possible from the application code which controls the knowledge base structures, which will allow the transition to a graphical interface to proceed more smoothly. This section details the design and implementation of the generic interface.

Interface Style

The knowledge base representation was established during design of the inference engine and Figures 2 and 3 show examples of the form of frame and rule structures. At the very least, the entire knowledge base could be prepared in this syntax using a word processor or editor, and then compiled or interpreted to check for internal consistency and proper syntax. This was thought unsatisfactory for several reasons. First, there are several long and unwieldy reserved words, such as "intermediate", "propagator", "local_slot", etc., which would make editor entry very tedious and error prone. As seen in figure 3, constants must be identified as such, accompanied by their data type. Typing in this information could be very frustrating for a large body of rules. Second, although the rules logically are if-then structures, they do not resemble familiar if-then form, and in addition, contain clues as to the location of the referenced slots, as in "dist_slot" and "local_slot", items which could be generated by the application.

In addition, the end-user interface at present is very simple, consisting of report and prompt strings sent to the standard output stream as rules containing them are fired. Thus, to the inference engine, all messages directed to the end-user are contained in rules, and for most applications these are rules with no antecedent parts. However, to the developer, the end-user interface is logically very different from the rule base, and for consistency, should be treated as a separate object. Also, if further developments to AXS include a more sophisticated end-user interface, then encapsulating

```
#F Plant INTERMEDIATE
    #S Family UNK 0.0 #D IF_NEEDED PROPOGATOR #N 1 #N 4
    #S Class UNK 0.0  #D IF_NEEDED PROPOGATOR #N 5 #N 6
    #S Type UNK 0.0   #D IF_NEEDED PROPOGATOR #N 8 #N 9
                      #D IF_ADDED FILTER #N 30
#F Leaf INTERMEDIATE #P Plant
    #S Shape UNK 0.0                  #D IF_NEEDED PROPOGATOR #N 23
    #S Pattern UNK 0.0                #D IF_NEEDED PROPOGATOR #N 24
    #S Has_Silver_Band UNK 0.0        #D IF_NEEDED PROPOGATOR #N 25
#F New_Leaf GOAL #P Leaf
    #S Plant_Family UNK 0.0           #D IF_NEEDED FILTER #N 14
```

Figure 2: Knowledge base representation of frames

```
#R 4
#A LOCAL_SLOT Class EQ CONSTNT STR "Gymnosperm"
#A DIST_SLOT Leaf Shape EQ CONSTNT STR "Needle_Like"
#A DIST_SLOT Leaf Pattern EQ CONSTNT STR "Even"
#A DIST_SLOT Leaf Has_Silver_Band NE CONSTNT BOOL TRUE
#C LOCAL_SLOT Family CONSTNT STR "Bald_Cypress"
#R 14
#C REPORT "Your leaf is family " LOCAL_SLOT Plant_Family
#C LOCAL_SLOT Plant_Family LOCAL_SLOT Family
#R 24
#C LOCAL_SLOT Pattern PROMPT "leaf pattern Random or Even? " STR
```

Figure 3: Knowledge base representation of rules

the messages and rules as logically different structures allows the development interface to be modified without total disruption.

In general, a more structured form of input removes much of the burden from the developer. In keeping with other shells, a menu system was chosen as appropriate, with keyboard data entry given the constraints of a non-graphical interface. As Baecker and Buxton [1] write, menus are occasionally viewed as cumbersome by expert users. However, they cite successful large systems with menu interfaces. In addition, whenever structured input is desired, with clear and fairly limited paths through the system, menus seem to be the method of choice at this time. One element which does need to be considered with menus is that they do limit screen space [19]. In the case of AXS, this does not seem to be a problem, as the choices are relatively few, and no cases have been found which present significantly reduced screen space.

Interface Menus

The menu hierarchy is shown in Figure 4. It is rather shallow, in keeping with guidelines and research which suggest that shallow menus with a reasonably large number of items are better than very deep menus with fewer items. [21]. The frames related menus are the deepest, reflecting the increased complication of the frame structure. Three of five areas available from the top level menu—frames, rules and the user interface, reflect the logical objects of the knowledge base. There are smaller sub-menus which pop up while in some of the object areas, such as demon or type choice menus, although these are not reflected in the hierarchy diagram. The *Knowledge base* menu is available for meta-control, limited at present to loading a knowledge base from a file.

Common features of all menus include consistent naming of commands, i.e., *Print*, *Delete*, *Quit*, and so forth. Commands common to all menus are *Help* and *Quit*, and consistent with user interface design guidelines, are positioned in the same place in each menu. Also consistent, with one exception, is the action taken at the *Quit* command, which returns the developer to the menu directly above in the hierarchy. The exception is the slots menu at the fourth level, which returns to the main frames menu, due to the limited nature of the third level menu. Command input is identical

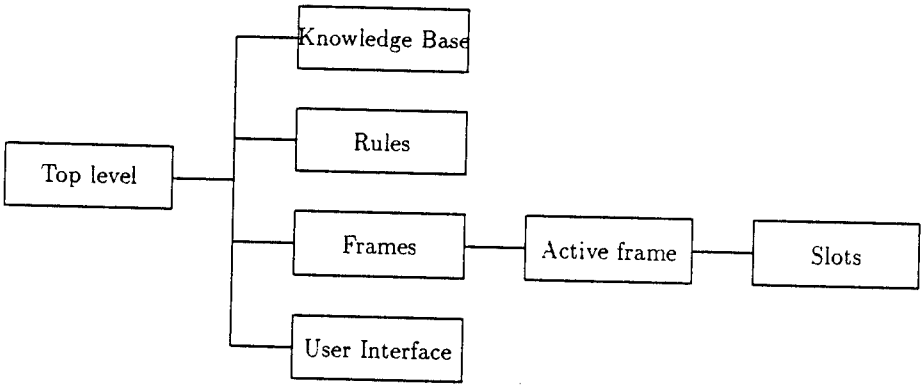


Figure 4: Menu hierarchy

for each menu, accepting the first letter of each option, followed by the [return] key. Using an ascii escape sequence, the screen is also cleared before the display of each menu, allowing a "clean" screen for each step.

Development Process

The development interface discussed here has some unique features, due in part to the desire for a portable implementation. The developer can choose to partially automate the formation of the knowledge base, or can develop it by hand through the interface. Since no graphical interface tools were used, editing of rules is restricted while in the interface. To alleviate this problem, the developer has the option of pre-entering rules into a separate file which are read in by the development interface. The input rule syntax resembles familiar if-then format, and from these rules, all of the necessary frames and slots are created, with default confidence values. The developer is queried during this process for information about frame and demon types, and confirmation of the suspected slot of attachment. Thus, knowledge base formation can be as simple as typing in a rule base with one's favorite editor, and adding desired embellishments to the frames. The alternative is to build the knowledge base completely using the menu structure of the interface.

Development Interface Implementation

As noted, AXS is written entirely in Ada, and object oriented techniques were used for implementation of the inference engine, with packages representing the objects in the system. For the most part, the development interface continued this style, in addition to experimenting with Ada task types as object types. A lexical analyzer and parser are a necessary part of the development interface, and these are written in a traditional functional style, rather than utilizing objects, as these modules were modeled from an earlier project, with the intent to reuse existing code. Also used were generic tool packages, such as trees and maps, modeled after those described by Booch [3].

The application code is as independent as possible from the interface code, although error messages from the application are sent directly to the standard output stream. This could be improved upon by implementing an error interface module for internal error message output, which would be better for a graphical interface.

Ada and Objects

Ada is not an object oriented language, although it has features which allow object programming, namely packages and task types. Non-generic packages allow for the representation of specific, single objects, while generic packages model reusable objects instantiable for multiple data types, for instance, trees. Task types allow for the dynamic creation of objects, an important part of object-oriented languages, although there are problems associated with the designation of task types as objects in this case.

The inference engine utilized non-generic packages for frame, slot, demon, and rule objects, which resulted in what could be called a large granularity of objects. For instance, all frames were represented by record types and included in one package with associated operations, instead of each individual frame being an object in itself. This smaller granularity, with multiple instances of one class and their associated operations being instantiated dynamically as needed, is one of the properties of object-oriented languages which is difficult to implement with Ada.

In the development interface, objects were represented in three ways: by non-generic packages, by generic packages and by task types. For instance, rules were represented by an instantiation of a generic map package, mapping a rule number to a rule record object, and all associated rule operations were collected with this structure in a single non-generic package.

Task Objects

It was decided to implement the frames as instances of a single task type declared through an access type, allowing for frames to be dynamically created as needed. Within this task type, a generic tree package is used to represent the frame slots. Entry points to the task designate the operations possible for a frame. All frames resulting from activations of this task type are managed by a single frame management package. Conceptually, objects as task types are closer to the representation of objects in an object oriented language than are packages. However, the way in which tasks are terminated causes problems for the development interface in terms of allowing work on more than one knowledge base at the same time.

For instance, suppose that a knowledge base is loaded, causing tasks representing each frame to be created. In order to allow for multiple knowledge bases to be worked on during the same development session, the frame objects and manager for each knowledge base need to be able to be terminated, allowing for a new knowledge base and its associated objects to be loaded. Given the conditions under which tasks and task objects terminate, this is not a trivial matter. At present, the frames manager and frames objects are not defined in such a way to allow for easy termination, except at the end of a development session. Thus, only one knowledge base can be worked on at any given time.

Restarting the AXS development interface for each knowledge base is not much of a problem with the generic interface as currently defined. However, a graphical interface would need a significant amount of time to be loaded, an inconvenience for a developer who wished to work on multiple knowledge bases if the interface had to be reloaded for each. It is common for development systems to have small example knowledge bases available, for instance. Thus, some redesign of the frame objects is needed for an optimum graphical interface, and it is not clear at this time if task type objects are viable in this circumstance.

Conclusion

The conclusion drawn from the implementation of the AXS inference engine was that AI and Ada could be blended successfully to produce an expert system shell, despite the fact that Ada is not an optimal language for AI applications.

This part of the project looked at using Ada to incorporate objects into a human-computer interface. While an interface was successfully completed which met the initial requirements, some questions remain as to the use of Ada for interface objects. One question involves the use of generic packages for reusable data structures, typically those which require the use of multiple function and procedure calls to provide higher level functions. An important aspect of an interface is timely response, and multiple layers may be unsatisfactory here. Further testing is needed in this area before any conclusions can be drawn.

Task types provide objects, but can be difficult to manage and terminate. In addition, using task types as objects may present problems in terms of space requirements. An expert system development interface requires a great number of objects, dependent on the size of a knowledge base, and can probably not be totally implemented using task types alone.

Overall, Ada continues to prove to be an appropriate base language for a frame-based expert system shell. Continued development involves the inclusion of certainty factors, how and why tracing, and a graphical X Window System interface.

References

[1] Ronald M. Baecker and William A. S. Buxton, editors. *Readings in Human-Computer Interaction: A Multidisciplinary Approach.* Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[2] D. G. Bobrow et al. Expert systems: Perils and promise. *CACM*, 29(9):880–894, September 1986.

[3] Grady Booch. *Software Components with Ada.* Benjamin/Cummings, Menlo Park, CA, 1987.

[4] S. K. Card et al. Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a CRT. In Baecker and Buxton [1], pages 81–89.

[5] James E. Cardow. Toward an expert system shell for a common ada programming support environment. Master's thesis, Wright State University, Department of Computer Science, 1989.

[6] James E. Cardow and Verlynda S. Dobbs. Toward an expert system shell for a common ada programming support environment. *NAECON*, 1989.

[7] Gerhard Fischer. Human-computer interaction software: Lessons learned, challenges ahead. *IEEE Software*, pages 44–52, January 1989.

[8] Richard Forsyth. Software review: Leonardo. *Expert Systems*, 5(2):160–164, May 1988.

[9] William B. Gevarter. The nature and evaluation of commercial expert system building tools. *Computer*, pages 24–41, May 1987.

[10] John D. Gould and Clayton Lewis. Designing for usability: Key principles and what designers think. In Baecker and Buxton [1], pages 528–539.

[11] Jonathon Grudin. The case against user interface consistency. *CACM*, 32(10):1164–1173, October 1989.

[12] Paul Harmon et al. *Expert System Tools and Applications.* Wiley, New York, NY, 1988.

[13] Rex Hartson. User-interface management control and communication. *IEEE Software*, pages 62–70, January 1989.

[14] Jon Kannegaard. Open look: Outlook/overview. *Sun Technology*, 1(4):58–62, 1988.

[15] J. Karat et al. A comparison of menu selection techniques: touch panel, mouse and keyboard. *International Journal of Man-Machine Studies*, 25:73–88, 1986.

[16] Thomas K. Landauer. Relations between cognitive psychology and computer system design. In John M. Carroll, editor, *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, pages 1–25. The MIT Press, Cambridge, Massachusetts, 1987.

[17] William Mettrey. An assessment of tools for building large kb systems. *AI Magazine*, 8(4):81–89, 1987.

[18] Brad A. Meyers. User-interface tools: Introduction and survey. *IEEE Software*, pages 15–23, January 1989.

[19] Donald A. Norman. Design principles for human-computer interfaces. In Baecker and Buxton [1], pages 492–501.

[20] Mark H. Richer. An evaluation of expert system development tools. *Expert Systems*, 3(3):166–181, July 1986.

[21] Tony Rubin, editor. *User Interface Design for Computer Systems.* Ellis Horwood Limited, West Sussex, England, 1988.

[22] Personal communication with John Sloan, October, 1989.

[23] Sidney L. Smith and Jane N. Mosier. Guidelines for designing user interface software. Technical report, The MITRE Corporation, 1986. NTIS No. ADA 177198.

[24] Richard G. Vedder. PC-based expert system shells: Some desirable and less desirable characteristics. *Expert Systems*, 6(1):28–42, February 1989.



Verlynda Dobbs received her Ph.D. in computer science from the Ohio State University in 1985. Her research interests are in the areas of software engineering, artificial intelligence, and Ada for artificial intelligence. Dr. Dobbs is currently on the faculty of the Department of Computer Science and Engineering at Wright State University, Dayton, Ohio 45435.

Jill E. Courte has a B.S. in Psychology and a M.S. in Computer Science from Wright State University. She is currently employed by BCD Technology Inc., Dayton, Ohio.

THE AUTOMATIC TRANSLATION OF LISP APPLICATIONS INTO ADA

Henry G. Baker, Ph.D.

Nimble Computer Corporation
Encino, California

This work was supported in part by the U.S. Air Force
Contract No. F33615-89-C-1083

Many years of effort and over a billion dollars have been invested in the development of artificial intelligence and expert systems applications written in Lisp. DoD is facing costs of tens of millions of dollars to convert some of these applications into the Ada programming language for deployment. The *Nimble Project* is developing an *applications compiler* to compile Common Lisp applications into the Ada language. Success in this project will reduce the cost of reprogramming these applications by at least one order of magnitude for both Government and commercial users, and speed the deployment of these systems into the field.

Introduction

Artificial Intelligence research and the Lisp programming language have developed hand-in-hand for the past 25 years. It is not surprising, then, that Lisp is an excellent host for AI and expert systems. But because Lisp was developed for use within the research community, its objectives were considerably different from those of a language such as Ada, which is tuned for applications deployment. In particular, the original objectives of Lisp were:

- ease of algorithm prototyping
- concise programs (no declarations)
- powerful data and control structures
- very general constructs
- rich programming environments
- incremental development
- fast compilation
- ease of debugging

These objectives have been largely realized in "AI workstations" such as Lisp Machines. As a result, Lisp has been utilized as a "rapid prototyping" environment for investigating the algorithms and user interfaces for many "expert systems" being developed for DoD applications. These investigations have shown the feasibility of expert systems in improving resource allocation, threat assessment, robot weapon control, and other "brilliant" applications. However, the time has come to move these expert systems from development into deployment, and a whole new set of requirements for deployed software must be addressed. While Lisp maximizes the productivity of the prototyping phase of development, it trades off this productivity for increases in cost, weight and size, and decreases in speed, safety and maintainability. Some means must be developed to aid in the conversion of this prototype software into efficient, reliable and maintainable production software.

The hardware environment for deployed systems will also be significantly different then that for development. While special-purpose chips have been developed specifically for Lisp (e.g., TI Micro-Explorer, Symbolics Ivory), the number, quality and speed of RISC alternatives offers so much more in performance and flexibility (e.g., SUN SPARC, MIPS, Motorola 88000, TI 320C30, Intel 80960, AMD 29000, etc.) that the majority of delivery systems will utilize stock instruction-set architectures. These architectures require different compiler techniques and hence different compilers than those for Lisp architectures in order to achieve an acceptable level of efficiency.

Thus, Lisp development software systems are not appropriate for deployment because:

- they are too slow, especially on arithmetic
- they require extremely large run-time systems—50 megabytes or more for Lisp Machines
- they do not offer the sort of real-time response mechanisms required for embedded systems
- they do not do global consistency checks on variable or function usage
- they require a choice between "safety" and speed
- they are not easily ported to a wide variety of instruction set architectures

An expert systems developer thus has two choices for achieving these applications goals: reprogram his system in Ada, or find a Lisp system which is more appropriate for applications delivery. The cost of reprogramming in a non-Lisp language can easily exceed the original cost of developing the application in Lisp. This high reprogramming cost stems from several sources. Lisp itself is such a highly productive programming language, that moving the application to Ada will mean moving to a far less productive programming environment. In addition, many features of Lisp, such as run-time typing, garbage-collected heap storage, generic arithmetic, and sophisticated object-oriented programming, must either be dispensed with or duplicated during the reprogramming process.

One is thus led to consider the alternative—a Lisp system which is tuned for the *production* of applications rather than the *development* of algorithms. If such a system could take an existing Lisp application program, and recompile it for efficiency, safety, and real-time performance, then it would solve most of the problems facing developers of expert application systems. If this Lisp system generated compiled code which is as fast as that generated by more traditional languages, then a major incentive for reprogramming into a non-Lisp language would be eliminated. If this Lisp system could also generate code in Ada, then portability and compliance with the DoD mandate would be assured.

Therefore, a whole new set of objectives for a Lisp implementation is required to move applications from the laboratory to the field:

- very high *execution speed*, especially on arithmetic
- *small run-time system* incorporating only the features used
- *real-time response* capabilities
- high level of *compile-time checking*
- high level of *safety*
- high level of *portability* through the generation of Ada instead of machine language
- *stock hardware target*—especially modern DSP and RISC architectures

Three major issues were addressed in the development of the Nimble translator from Lisp into Ada:

- Is Ada an appropriate target language for such a translation?
- Can Lisp applications translated in this way achieve predictable real-time operation?
- Can Lisp applications translated in this way achieve the parallelism and embedded operation required of avionics systems?

Is Ada an appropriate target language for translation from Lisp?

Whether an application is to be translated from Lisp to Ada by hand or automatically by a computer, many of the same issues must be addressed:

- Translate from high-level specification language or from Lisp?
- Program to be maintained subsequently as Ada?

Many of the objectives of the requirement of programming embedded systems in Ada involve the whole programming process, not just the resulting Ada code. The objectives of programming in Ada include reducing the cost of initial programming, increasing the reliability of the produced code, and decreasing the cost of maintenance of the code. A one-time translation of the Lisp code into Ada code would reduce the cost of the initial programming, and might retain or possibly increase the reliability of the produced code. It would not, however, necessarily decrease the cost of maintenance, because the maintainer would not be working with the original source language, but a derivative program. As a result, it could be much more difficult to understand the interactions of the portions of the program, and such a maintenance effort would very likely introduce more bugs than it might fix.

One might realize the objective of lowered maintenance costs if the source code were kept and maintained in Lisp, but translated when necessary into Ada. Unfortunately, unless this translation cost were very low, the overall cost of maintaining this code would be dominated by the continual translation cost. In addition, this method for maintaining the application would violate the spirit of the "program in Ada" rule, because the program would in reality be in Lisp, with Ada but a step on the path toward machine code.

The only translation of Lisp to Ada which meets the spirit of the Ada requirement would translate not only the Lisp code itself, but also the specifications and documentation of the code. This complete translation would allow the maintenance programmer to access a specification and source code that would be consistent, without any hint of the original Lisp. As a result, he would find no difference between maintaining this code and any other code which was initially developed in Ada.

While an automatic translation of Lisp source, specifications and documentation might eventually be feasible, we consider such a possibility unlikely within 3 to 5 years. We must therefore look for alternatives to a complete translation of specifications, code and documentation from Lisp into Ada.

Another model for producing Ada code exists already in the form of "specification languages" and "problem description languages". After an algorithm is specified in one of these languages, the appropriate Ada code to implement these specifications is produced. All of the arguments about maintaining Ada code produced by translation from one of these description languages are the same as those about maintaining Ada code produced by translation from Lisp. Yet the Ada and DOD community are starting to accept these "preprocessed" Ada programs as legitimate mechanisms to produce and maintain Ada code.

One way to view this translation process is to consider Lisp a high level "specification language" (or "fourth generation language" in the sense of Brooks[13]), with the Ada code produced by translation from this Lisp specification—whether automatically produced or not—a legitimate product of a valid programming technique. Of course, the Ada code produced by such a translation process must then meet certain standards of readability and comprehensibility, assuming that this is possible.

At least one defense contractor has taken the idea of "Lisp as a specification language" to a certain conclusion, and produced a system called "InnovAda"[38]. InnovAda is the direct interpretation of Zetalisp's *Flavors* object-oriented programming constructs as a specification language for Ada, except that the executable code within the bodies of the Flavor definitions must be valid Ada code instead of Lisp code. This specification is then processed by the InnovAda translator to produce legal Ada source code by expanding the Flavor constructs. (A system called "Classic-Ada"[39] has similarly grafted the object-oriented ideas of Smalltalk onto Ada.)

While InnovAda and similar systems may be valuable tools to help *build* an AI or expert system from scratch in Ada, they are less help in *translating* an existing system from Lisp into Ada. This is because these techniques still require a programmer to have a very deep understanding of the overall program in order to translate it correctly, and the resulting translation must still be tested and debugged again.

Our approach to solving the Ada problem is to develop an automatic translator from Common Lisp with the CLOS ("Common Lisp Object System") extensions into Ada. This translator can then be used either once to translate a Lisp program into Ada and henceforth have the program maintained in Ada, or used many times during the course of program maintenance to continuously regenerate a new Ada program from a modified Lisp source.

We feel that the most important objective of this translation should be the generation of correct, runnable Ada code from the Lisp source, even if this Ada code is not very readable. If we

can later "tune" this translation to better translate "high level" constructs such as Lisp loops into equivalent Ada loops, this would be a welcome improvement. If later stages of this project can similarly take advantage of most high-level Ada constructs, then the project will be an overwhelming success.

What are the costs of translating Lisp into Ada? There are several costs associated with any translation of Lisp into Ada. First, there is the cost of the original translation, which can be quite substantial if performed manually. Second, there is the cost of debugging the translation, which debugging can take as long or longer than the original debugging. This is because the debugging environment for Ada code can be more hostile than the Lisp environment, and because the person most likely debugging the Ada version may not be the person who debugged the Lisp version, and will have less insight into the nature and behavior of the program. Finally, there are the costs of maintaining multiple sources in both Lisp and Ada, in the case that significant algorithms may have to be replaced and debugged; this will likely take less time in Lisp than Ada.

The cost of translating a Lisp application into Ada is dependent upon the Lisp facilities used, and whether they have similar counterparts within Ada. Many simple constructs are available in both languages, such as lexical variables, assignment, arithmetic expressions, loops, functions, parameters and arguments. However, there are constructs in Lisp with no built-in Ada counterparts, e.g., infinite-precision integers, rational numbers, generic arithmetic, "CONS" lists, etc. While Ada can solve many of the problems these constructs are meant to solve in a similar way, the translation is not straight-forward.

The complexity of a translator goes up significantly if high level Lisp constructs are required to be translated into "equivalent" high level Ada constructs instead of into lower level constructs. This is because a faithful translation of Lisp into lower level Ada is no harder than a translation of Lisp into C or a simple assembly language; it may even be somewhat easier. However, the low-level Ada code produced by such a translation may be quite difficult to relate back to the original specifications.

What about garbage collection? A major problem arises with Lisp's *garbage collection*, which is required in all Lisp implementations, but optional (and hence almost never present) in Ada implementations. One of Lisp's major advantages in programming productivity over Ada is its assumption that the system is responsible for the allocation and deallocation of all storage. As a result of this assumption, many programs become more elegant, easier to understand, and easier to prove correct than their counterparts with explicit storage management. While the Ada language allows for garbage collection to solve the problem of storage management, this option is almost universally ignored. Perhaps this is because Ada is difficult enough to implement efficiently without the added headache of expensive and hard-to-implement additional options, or because most methods of garbage collection do not allow for "real-time" operation, or seem hopelessly complex and inefficient on stock hardware.

Since AI and expert systems programmed in Lisp assume garbage collection, the translator responsible for producing an Ada equivalent program is faced with a quandary: how to allocate/deallocate storage without requiring garbage collection, or alternatively, how to simulate garbage collection. Replacing garbage-collected heap allocation/deallocation by other mechanisms by a compiler/translator is a problem which has been studied[35,7,34,28], but these techniques can only succeed in a fraction of the cases. One is still left with the problem of deallocating the residual garbage. Simulating garbage collection in a higher level language like Ada is also possible[46], but difficult, and may involve incomprehensible transformations of the original program.

Can Lisp applications translated automatically into Ada achieve predictable real-time operation?

The question of how to produce software for predictable real-time control has been hotly debated in regard to Ada [15,30]. However, we assume that most of these issues will find a solution within the Ada framework because much more is at stake here than just AI applications. We therefore address only the Lisp component in the translation sequence.

Common Lisp[17], even extended with CLOS[9], does not address the issues of real-time operation and parallel execution. The only two Common Lisp primitive operations which deal with time at all are those which read one of several clocks, and one (`"sleep"`) which waits a specified number of seconds.

Steele, in his overview of Common Lisp[42], mentions that Lisp is used for "symbolic processing, numerical processing, text manipulation and *systems programming*" and "nearly all the I/O drivers for the Lisp Machine, for devices such as disks, networks, graphic displays, and dot-matrix printers, are written completely in Lisp". Nevertheless, the Common Lisp standards committee felt that it was premature to standardize on multiprogramming/multiprocessing constructs. According to Steele,

> Common Lisp does not specify whether or how multiprocessing and multiprogramming is to be accomplished, nor how concurrent processes may interact or communicate. Some planned implementations of Common Lisp will provide extensions for multiprocessing; experience with these implementations may lead to extension of the Common Lisp definition.[42]

Due to system and user requirements, virtually every Lisp implementation in use offers some form of multiprocessing. The Lisp Machine[23] and its derivatives offer "stack groups" as a model of multiprogramming; Lisp "special variables" (Lisp's global variables) become the major means of communication between these "lightweight processes". (Stack groups are lightweight processes because they share the same address space; a heavyweight process, such as those found in UNIX, has its own address space.) The Scheme standard offers "continuations" as a means for implementing non-stacking control structures, but stops short of offering a true multi-threaded multiprogramming environment; this lack is easily rectified[45].

As a result of the decision by the Common Lisp committee to

delay action on a multiprocessing standard, Common Lisp has not failed as a language for real-time control; it was never even considered for this task. As a result, Lisp cannot currently be viewed as an appropriate vehicle for expressing real-time constraints or concurrent control in an embedded system.

Yet AI systems are being proposed to solve many real-time problems, from the control of robots in the factory, to the control of autonomous land vehicles, to the aiding of a pilot in a single-seat aircraft. The Lisp programs in these applications are not required to handle the closed loop control of low-level flight operations, for example, but they are required to control their own processing requirements, and to keep their processing in synchrony with the external environment, as would be required to follow the aircraft progress on an internal map. As a result, these programs must have the means to start various processes, check on their progress, and redirect or abort them as necessary.

We have already shown in Baker[5] that Lisp's garbage collection is compatible with real-time operation. This algorithm overlaps and distributes the work of garbage collection across all allocation operations, so that a Lisp system never has to stop at an unpredictable moment for an indefinite amount of time in order to rearrange its storage. An approximation to this algorithm is utilized in both commercially available "Lisp Machines". However, as a result of tuning for an *interactive*, rather than a *real-time*, environment, both of these implementations have trouble responding reliably to requests in less than a few seconds.

Can Lisp applications automatically translated into Ada achieve parallelism?

While explicit multiprogramming constructs were not specified for Common Lisp/CLOS, these standards are silent on the issue of parallel evaluation of Lisp programs. Curiously, the Common Lisp standard[17] does *not* force the sequential left-to-right evaluation of the arguments to a function; it only requires that they all be evaluated to a value:

> Any and all remaining elements of the list are forms to be evaluated; one value is obtained from each form, and these values become the *arguments* to the function.[17]

However, many examples given in Steele[17], and nearly every other Common Lisp program, fail using other than a left-to-right argument evaluation order. Most other obvious candidates for parallel evaluation, such as "`let`" expressions and mapping functions are required to be sequentially evaluated by the Common Lisp standard.

Even though Common Lisp is envisioned as a sequential programming language, an implementation is free to execute a program in a parallel manner, so long as it returns the same answer it would have, had it been executed in a sequential manner. Performing this type of optimization is very difficult, even in simpler languages like Fortran, because it requires knowledge of where the parallel threads might interact or conflict[19]. There are several projects to determine these conflicts in Lisp at compile-time[32,26], or deal with them at run-time[21,24,44]. Interestingly, the type of compile-time analysis required for detecting conflicts is essentially the same as that required to detect *aliasing*; as a result, any optimizing compiler which can perform alias analysis is in a good position to be converted to parcel out evaluation to multiple functional units.

Implicit parallelism of the type considered above will probably not solve the problem of real-time response in Lisp, even though it is capable of substantial parallelism[4]. This is because the Lisp programmer has no control over resource allocation—particularly the allocation of processing time. While processing time may grow to be a relatively cheap resource in the future due to systems with thousands of processors, it is not that cheap now, nor is it likely to be in the 3-5 year time frame envisioned for the translation of Lisp programs into Ada. As a result, giving up control of processor allocation to the system, as in Baker[4], may be elegant, but not suitable for near-term embedded systems requirements.

What about the requirements of embedded systems? Many existing Lisp systems attempt to provide both software development and applications delivery with the same software. Their "solution" to the problem of efficient code for application delivery is to perform debugging with the `SAFETY` parameter set to the highest level; when sufficient confidence has been reached with the program, more efficiency is gained (at the expense of safety) by then telling the compiler to eliminate most run-time checking by setting `SAFETY` to the lowest level.

This method of achieving speed by increasing the risk of field failure is unacceptable for most applications. While the benefits of additional speed are greatly appreciated by the customer for a while, the risk of sudden and catastrophic failure which can result from the elimination of the run-time checks is not worth the increased speed. As expert systems find their way into embedded applications such as graceful nuclear powerplant shutdown and a "pilot's associate", the lowering of safety standards in the software is not to be tolerated.

The following are the sources of speed/safety tradeoffs and inefficiencies on stock architectures:

- run-time type checking
- run-time argument/parameter checking
- run-time storage management
- incompatibility of Lisp datatypes with hardware datatypes
- preference of heap allocation over static and stack allocation

The Nimble Common Lisp to Ada Compiler

The Nimble compiler achieves efficiency without compromising safety through several means:

- use of programmer-supplied *declarations*, when available
- sophisticated compile-time *type inferencing*
- extensive compile-time *storage management*
- more extensive set of data representations including *native datatypes*
- recognition and special handling for global values, arrays and argument lists to expunge systematic inefficiencies
- traditional compiler optimization techniques which move safety checks out of inner loops
- output to Ada, which provides efficient code generation for pipelined and RISC architectures

One way to minimize run-time type checking and the incompatibility of Lisp and hardware datatypes is through *declarations*. Common Lisp offers the programmer the option of giving declarations for data which are quite similar to the declarations found in Ada. In theory, these declarations should allow the compiler to produce code similar to that found in Ada. However, because compiling code for native datatypes is not typical behavior for development Lisp compilers, and because native datatypes require different sorts of calling sequences—with ramifications regarding incremental compilation and debugging—a completely different compiler is required in order to achieve the maximum run-time efficiency allowed by these declarations.

Type inferencing can often supply additional type information, even in the absence of declarations. Lisp is usually described as a language with only run-time data typing, for which little compile-time typing can be done. Yet researchers have found that in many real programs, the datatypes of many variables and temporaries can be *inferred* through a process called *type inference*. For example, type inference has been successful in APL[14], Smalltalk[10,43], and Lisp[8].

In order to perform type inference, the Nimble compiler performs a very deep and thorough *flow analysis* of the application, which due to the structure of Lisp must incorporate both *control flow* and *data flow* analysis at the same time. This analysis determines bounds on the *datatype* of variables, which allows for the more traditional methods of compiling *strongly typed* languages like Ada. In order to get the best possible information, this flow analysis is performed on the complete application, not just on a single module. While called "type" inferencing, this same process also develops some *range* information, which allows for the elimination of some array-bounds checking.

The Nimble flow analysis enables many run-time type checks to be eliminated, and allows for the code generator to often utilize the datatypes which are *native* to the hardware. While this flow analysis requires far more time than that usually required to compile a Lisp program, this analysis is very important, because it can turn up problems such as argument/parameter incompatibilities that would prove embarrassing in the field. Furthermore, the analysis can sometimes find conflicts with programmer-supplied declarations, which usually indicate some sort of an error.

Another major source of inefficiency in Lisp is its preference for storage allocation on a dynamic *heap*, which is very elegant and general, but which costs more during run-time. A phase of the Nimble compiler builds on the flow analysis information to compute the *lifetimes* (i.e., *extents*) of the data objects. The goal of this analysis is to determine which objects can be *statically* allocated in the manner of Fortran variables; which objects can be *stack* allocated in the manner of Ada local variables; and which objects must continue to be heap allocated.

Once flow and lifetime analysis has compiled sufficient information on the data objects within a program, Nimble can perform a better job of choosing an appropriate data *representation* which maximizes the efficiency of the resulting code by better matching the target hardware.

Even after the type inferencing and lifetime analysis optimizations have been performed, there will remain many checks that could not be eliminated by the compiler, e.g., array bounds checks or variable type checks. In these cases, traditional compiler optimizations can be utilized to eliminate redundant checks—as in *common subexpression elimination*—or to move them to places where they will not be executed nearly as often—as in *strength reduction*. Thus, even though the type-checking is still performed in the program to enhance safety, its run-time cost becomes negligible.

Current State

The Nimble Compiler is currently under development, with the type inferencing phase working well enough run small examples such as the *Gabriel benchmarks*, which are standard Common Lisp programs used to test the efficiency of Common Lisp implementations. The results from type inferencing are surprisingly good, fully supporting our contention that most Common Lisp datatypes can be pinned down from their context, and hence compiled into very efficient code. The type inferencing algorithm has also borne out our expectation that it would be *slow*, but we are less concerned with this, since it is certain that this type inferencer is still faster than a human, and the cost of running this very important phase amortized over the execution time of the deployed program is still negligible.

Comparisons with Previous Work

Burns[15] is an excellent discussion of the Ada tasking model and its pitfalls; Kontak[30] is a case study of the use of Ada tasking for a real-time system.

Schwartz[36] was one of the first to discuss the use of Ada for AI; Ausnit[3] is one of the more recent discussions of the use of Ada for AI applications; Brauer[11], Simonian[38], SPS[39], and Ausnit[3] describe methods of implementing AI applications in Ada; Fortier[20] and Marmelstein[33] discuss Ada development environments; Gabriel[22] gives requirements for future algorithm development workstations which provide a smooth path from a Lisp-like language to Ada.

Steele[17] and Bobrow[9] are standards for Common Lisp and CLOS, respectively; Keene[29] is an excellent introductory programming book for CLOS.

Halstead[24] discusses the implementation of Lisp on a MIMD parallel processor; ParaTran[44] is the use of "time warp" ideas to reduce the synchronization required in the parallel evaluation of sequential Scheme; Harrison[26] and Larus[32] discuss the compile-time analysis of Lisp to detect parallelism and conflict.

Baker[5] gives the first real-time algorithm for garbage collection; Appel[2] shows how to utilize paging hardware to efficiently implement this algorithm on stock hardware. Schwartz[35], Barth[7], Steele[40,41], Brooks[12], Chase[16], Ruggieri[34], Hederman[27], and Inoue[28] discuss methods and mechanisms for doing a portion of the garbage collection effort during compile time.

Budd[14] has shown the efficacy of type inferencing in APL, Borning[10] and Suzuki[43] have shown the efficacy of type inferencing in Smalltalk, and Beer[8] has shown the efficacy of type inferencing in Common Lisp. Steele has shown the efficacy of lifetime analysis for Maclisp[40] and for Scheme[41], Brooks has shown the efficacy of lifetime analysis for Common Lisp[12]; Chase[16], Ruggieri[34], and Hederman[27] have recently shown algorithms for lifetime analysis in C-like and Ada-like languages. Inoue[28] has recently demonstrated compile-time reclamation of lists in "pure" Lisp. Cooper[18] shows the efficacy of static analysis of whole programs. Aho[1] is a standard reference for the optimization of Ada-like languages.

The Nimble Project for compiling Common Lisp into Ada has a method similar to that of Harrison[25], numeric performance goals similar to S-1 Common Lisp[12], and utilizes a higher-level language for code generation similar to Kyoto Common Lisp[46]. However, it performs a level of flow analysis deeper and more comprehensive than Beer[8], Steele[41], Shivers[37], or ORBIT[31].

References

1. Aho, Alfred V.; Sethi, Ravi; and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. Appel, Andrew W.; Ellis, John R.; and Li, Kai. "Real-time concurrent garbage collection on stock multiprocessors". *ACM Prog. Lang. Des. and Impl.*, June 1988, p.11-20.
3. Ausnit, Christine, et al. "U.S. Air Force Program Office Guide to Ada, Edition 3". DTIC AD-A189651, Dec. 1987.
4. Baker, Henry G., Jr.; and Hewitt, Carl. "The Incremental Garbage Collection of Processes". *Proc. AI*PL Conf.*, also *SIGPLAN Notices 12*, 8 (Aug. 1977).
5. Baker, Henry G., Jr. "List Processing in Real Time on a Serial Computer". *CACM 21,4* (April 1978), pp.280-294.
6. Baker, Henry G. "The NIMBLE Project—Real-Time Common Lisp for Embedded Expert Systems Applications". 1989 AIAA Computers in Aerospace Conference, Monterey, CA, Oct. 1989.
7. Barth, J. "Shifting garbage collection overhead to compile time". *CACM* 20,7 (July 1977), pp.513-518.
8. Beer, Randall D. "The compile-time type inference and type checking of Common Lisp programs: a technical summary". TR 88-116, Ctr. for Automation and Intell. Sys. Res., Case Western Reserve Univ., May 1988.
9. Bobrow, et al. "Common Lisp Object System Specification X3J13", *ACM SIGPLAN Notices*, v.23, Sept. 1988; also X3J13 Document 88-002R, June 1988.
10. Borning, Alan H., and Ingalls, Daniel H. H. "A Type Declaration and Inference Systems for Smalltalk". *ACM POPL 9*, 1982, 133-141.
11. Brauer, D.C., et al. "Knowledge-based Systems: A Prototype Combining the Best of Both Worlds". McDonnell Douglas Astro. Co., Huntington Bch, CA, Oct. 1986.
12. Brooks, Rodney A.; Gabriel, Richard P.; and Steele, Guy L., Jr. "An Optimizing Compiler for Lexically Scoped LISP". *ACM Lisp and Functional Programming 1982*, pp.261-275.
13. Brooks, Frederick P., Jr. *Report of the Defense Science Board Task Force on Military Software.* Office of the Under Secretary of Defense for Acquisition, Wash. DC, Sept. 1987.
14. Budd, Timothy. *An APL Compiler.* Springer-Verlag, NY, 1988.
15. Burns, Alan; Lister, Andrew M.; and Wellings, Andrew J. *A Review of Ada Tasking.* Springer-Verlag, New York, 1987.
16. Chase, David. "Garbage Collection and Other Optimizations". PhD Thesis, Rice University Comp. Sci. Dept., Nov. 1987.
17. Steele, Guy L., Jr. *Common Lisp: The Language.* Digital Press, 1984.
18. Cooper, Keith D.; Kennedy, Ken; and Torczon, Linda. "Editing and Compiling Whole Programs". *SIGPLAN Notices* 22,1, Jan. 1987, pp. 92-101.
19. Ellis, John R. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1986.
20. Fortier, P.J., Bergandy, P.A., et al. "Software Architect's Workstation: A Tool for Ada Program Development". NUSC Tech. Doc. 6630, DTIC AD-A183 505, July 1987.
21. Gabriel, R.P., and McCarthy, J. "Queue-based multiprocessing Lisp". *ACM Symp. on Lisp and Funct. Prog.*, Austin, TX, 1984, 25-43.
22. Gabriel, Richard P., ed. "Draft Report on Requirements for a Common Prototyping System". Available from Robert Balzer, Chairman of Common Prototyping Working Group, Information Sciences Institute, Marina del Rey, CA, Nov. 11, 1988.
23. Greenblatt, R., et al. "The LISP Machine". Working Paper No. 79, MIT AI Lab., Camb., MA, Nov. 1974.
24. Halstead, R. "MultiLisp: A language for concurrent symbolic computation". *ACM TOPLAS 7*, 4 (Oct. 1985),501-538.
25. Harrison, William. "A New Strategy for Code Generation: The General Purpose Optimizing Compiler". *ACM 4'th POPL '77*, pp.29-37.
26. Harrison, W.L., III; and Padua, D.A. "PARCEL: Project for the Automatic Restructuring and Concurrent Evaluation of Lisp". *Proc. 1988 ACM Supercomputing Conf.*, St. Malo, France (July 1988),527-538.
27. Hederman, Lucy. "Compile Time Garbage Collection". MS Thesis, Rice University Computer Science Dept., Sept. 1988.
28. Inoue, Katsuro; Seki, Hiroyuki; and Yagi, Hikaru. "Analysis of Functional Programs to Detect Run-Time Garbage Cells". *ACM TOPLAS 10*, 4 (Oct. 1988),555-578.
29. Keene, Sonya E. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, Reading, MA, 1989.
30. Kontak, Roger E. "Applicability of Ada Tasking for Avionics Executives". M.S. Thesis, AFIT, TR AFIT/GE/MA/87D-4, DTIC AD-A188827, Nov. 1987.
31. Kranz, David, *et al.* "Orbit: An Optimizing Compiler for Scheme". *Proc. SIGPLAN '86 Symp. on Compiler Construction* (June 1986), pp.219-233.
32. Larus, J.R.; and Hilfinger, P.N. "Restructuring Lisp Programs for Concurrent Execution". *ACM PPEALS*, also *SIGPLAN Notices 23*, 9 (Sept. 1988),100-110.

33. Marmelstein, Lt. Robert E. "The Interactive Ada Workstation: A Prototype for Next Generation Software Environments". *Proc. Joint Ada Conf.* 5'th Natl. Conf. on Ada Tech. and Wash. Ada Symp., March 1987,54-61.
34. Ruggieri, Cristina; and Murtagh, Thomas P. "Lifetime analysis of dynamically allocated objects". *ACM POPL '88*, pp.285-293.
35. Schwartz, J.T. "Optimization of very high level languages—I. Value transmission and its corollaries". *J. Computer Lang. 1*, 1975,161-194.
36. Schwartz, R.L.; and Melliar-Smith, P.M. "The Suitability of Ada for Artificial Intelligence Applications". SRI International, Menlo Park, CA, May 1980.
37. Shivers, O. "Control Flow Analysis in Scheme". *ACM SIGPLAN Conf. '88*, pp.164-174.
38. Simonian, Richard; and Crone, Michael. "InnovAda: True Object-Oriented Programming in Ada". Available from Richard Simonian, Harris Corp., PO Box 98000, MS W1/7736, Melbourne, FL, 32902, 1988.
39. Software Productivity Solutions, Inc. (SPS). "Classic-Ada Technical Specification". Available from Software Productivity Solutions, Inc., PO Box 361697, Melbourne, FL 32936, 1988.
40. Steele, Guy L., Jr. "Fast Arithmetic in Maclisp". *Proc. 1977 Macsyma User's Conference*, NASA Sci. and Tech. Info. Off. (Wash., DC, July 1977), pp.215-224. Also AI Memo 421, MIT AI Lab., Camb. Mass.
41. Steele, Guy L., Jr. *Rabbit: A Compiler for SCHEME (A Study in Compiler Optimization).* AI-TR-474, Artificial Intelligence Laboratory, MIT, May 1978.
42. Steele, Guy L., Jr. "An Overview of Common Lisp". *ACM Conf. on Lisp and Funct. Prog.*, Pittsburgh, 1982,98-107.
43. Suzuki, Norihisa. "Inferring Types in Smalltalk". *ACM 8'th POPL '81*, pp.187-199.
44. Tinker, Pete; and Katz, Morry. "Parallel Execution of Sequential Scheme with ParaTran". *1988 ACM Lisp and Funct. Prog. Conf.*, Snowbird, UT, 28-39.
45. Wand, Mitchell. "Continuation-based Multiprocessing". *1980 ACM Lisp Conf.*, Stanford, CA, Aug. 1980,19-28.
46. Yuasa, Taiichi; and Hagiya, Masami. *Kyoto Common Lisp Report.* IBUKI Corp., Los Altos, CA, 1985.

PORTABILITY:
A KEY ELEMENT IN LIFE CYCLE PRODUCTIVITY

Tom Archer *Andrew Easson*

TELOS Systems
Lawton, Oklahoma

ABSTRACT

This paper presents TELOS Systems' experience in applying Ada style guidelines, portability aids, guides, and standards to improve life cycle productivity during Ada software development efforts on Mission Critical Defense Systems (MCDS). Also presented in this paper is a systems standards approach to improving portability. TELOS provides technical and engineering support to our Government customer for U.S. Army Battlefield Automated Systems comprising the Fire Support Segment of the Army Tactical Command and Control Systems.

PORTABILITY ASPECTS

Portability is a key element in improving and maintaining life cycle productivity. A positive relationship exists between the qualities of portability and software reusability, software/system maintenance, software/system performance, software testability and software/system quality. Increased portability means that a system can use software that has already been coded and debugged. Portability increases reusability, improves testability and quality, and reduces system maintenance and cost activities by reducing the design and coding effort, reducing the testing effort, and decreasing the performance risks through the use of software that has already proven its performance reliability. Increases in developer productivity, with corresponding decreases in development costs, can be realized by reuse of previously developed software.

A critical aspect of software reusability is portability. Portability refers to the ease (or difficulty) of moving the software to the current development effort. In other words, reusability and portability are tied together, in that once software has been identified for reuse, the effort required to "port" the software to the new system is evaluated.

An additional aspect of portability, not related to reuse of software, is the migration of system software from the current target computer to a new and improved target computer (or at least different). Again portability refers to the ease with which such a migration can occur.

A final aspect of portability is the ease with which software developed in one environment can be ported to a different target environment. This "porting" typically occurs when the target computer does not have or has weak program development tools. The developer uses a different environment for generating the software and then relies on cross-compilers and linkers to create system software for the target.

All three of these portability aspects are affected by many factors. Availability of compilers, cross-compilers, linkers, editors, and operating systems has tremendous impact on the ease of moving software. Personnel capabilities and experience also have a large impact on the effort and cost. Yet, given complete toolkits and fully qualified personnel, it is still possible for a "porting" job to be difficult. This difficulty arises because the software being ported is unusable on the "new" system, unreadable by engineers working on the "new" system, or has critical dependencies on the "old" system.

PORTABILITY AIDS

Portability aids are those automated tools provided by compiler and environment developers, such as IBM, DEC, HP, etc., that allow definition and technical enforcement of Ada style guidelines, as well as standard aids such as Pretty Printer, Language Sensitive Editors, syntax checks, etc. The quality of compiler-provided portability aids and guides selected for use directly affects life cycle productivity. Improvements to CASE tools, portability aids and guides, and programming support environments such as editors that catch and identify naming conventions and ambiguities during compiles, and compilers that deal with the generation of the most efficient code will increase the importance of portability as an element of life cycle productivity.

TELOS makes the maximum use possible of vendor tools that aid in producing portable code. Pretty Printers and language sensitive editors each help developers to produce consistently formatted and structured software.

Language sensitive editors are used to provide an environment that assists the programmers by providing language construct templates. Careful design of the templates provides consistent style and construct usage across development efforts. These editors also provide help facilities that explain the use of the language features. The use of language sensitive editors results in source code being developed faster and with fewer, first time, compiler errors.

Printer utilities such as Pretty Printer can also be used to increase portability of software by enforcing a consistent Ada programming style. Text files processed by a printer utility will produce source files with consistent appearance across all programmers. While these utilities do not normally flag non-portable language features, their improvement in readability simplifies the identification of these non-portable language features during analysis and search processes.

Source code analyzers and syntax checkers enable both producers and users alike to grade software according to its portability and to aid in identifying areas that may require

work to increase the portability. One such tool is ADAMAT which is a source code analysis tool whose purpose is to rate (or score) Ada source code. The tool scans the source code and, based on the statements found, rates the software for various aspects of good software engineering, high reusability, and maximum portability.

ADAMAT rates the software for six major attributes of good software engineering. The first attribute, Anomaly Management, relates to those qualities that provide for prevention, detection, and recovery from non-nominal conditions. The second attribute, Independence, includes those qualities that determine the dependency of the software on the environment. The third attribute, Modularity, refers to the software qualities providing a structure of cohesive modules with minimal coupling. The fourth, Self Descriptiveness, includes those qualities which provide explanations of the implementation within the software. The fifth, Simplicity, is those qualities that provide for definition and implementation of a module in the least complex and understandable manner. The sixth, System Clarity, is the set of qualities that determine the understandability of the software. These are primarily related to the style of programming used.

For each of these six sets of attributes, ADAMAT computes scores and ratings. These ratings are then used to identify modules and packages that may require more effort, i.e., those that have a poor rating as compared to similar modules.

TELOS uses ADAMAT in three ways. First, ADAMAT's rules and ratings are evaluated to determine what is considered good software engineering. These rules are then incorporated into existing standards and guidelines. Second, modules that receive low ADAMAT scores will be revisited with more development effort applied, thus improving the quality of those modules. Third, reported violations of particular ADAMAT criteria may identify a training deficiency. If such deficiencies are identified, then corrective measures will be taken for the training or retraining of project personnel.

APPROACH TO IMPROVING PORTABILITY

A practical solution to this problem is to establish and use aids, guidelines, and standards that steer the developers into writing software that will be portable to later systems.

Application of the portability guidelines and aids provides varying results. TELOS' approach to the incorporation of Ada style guides and portability aids and guides into Ada Programming Standards, Ada Software Engineering Standards, and Ada Project Management Standards for the Forward Entry Device development effort has provided the greatest benefit during the design, coding, and testing phases. TELOS is currently using internally developed policies and procedures to minimize the effort required for porting software across a wide range of compilers. The portability guides, aids, and rules addressed in TELOS' Ada Programming Standards, Ada Software Engineering Standards, and Ada Project Management Standards affect and benefit reusability, maintenance, performance, testability, and quality of the software. Examples of such guidelines that TELOS has established and uses are:

- The isolation of machine dependent code. Later users of the software will know exactly where to find and alter software so the code will operate on their machine.
- Minimize the use of Language Reference Manual Chapter #13 features.
- Standardizing of company-wide style guides and module format structures that increase understandability of the software by those other that the original developer.
- Requiring implementation of standards and style guidelines that rely heavily on Ada packaging techniques. These increase the software portability.
- Requiring the use of definitions type packages to describe types that are dependent on the underlying architecture.

SPECIFIC STANDARDS APPROACH

The specific standard approach to improve portability is based on our recommendations to tailor the DOD-STD-2167A Interface Requirements Specification Data Item Description (DID) DI-MCCR-80026A and the Interface Design Document DID DI-MCCR-80027A to facilitate and consolidate portability information into a standard location for systems developed in accordance with DOD-STD-2167A. This would be accomplished by adding an appendix A to each document. This appendix A would address factors affecting portability from the concept formulation phase through the enhancement and sustainment phase of a MCDS software life cycle. Locating this information in a common appendix will facilitate the use of this information and improve the portability factor of the software especially during the enhancement and sustainment phase of the systems life cycle. A proposed format for the appendix for both documents would be as shown in the following example.

EXAMPLE

APPENDIX A. PORTABILITY INFORMATION FOR
SYSTEM NAME
Date of Appendix and Revision or Change Number

10. SYSTEM NAME Portability Factors.

10.1.1 List and explain the equation and factors used to measure the software's portability (include tools used before and after porting i.e., static program analysis, test case generators, and dynamic program analyzer and software monitor.

10.1.2 Describe the software to be ported along with references to supporting documentation.

10.1.3 Describe the current development environment host and target equipment.

10.1.4 Describe the porting environment tools to be used.

10.1.5 Describe the portability/adaptability approach to be used.

10.1.5.1 Describe the designers approach.

10.1.5.2 Describe the installers approach.

10.2 Describe the host or development environment hardware and software.

10.2.1 Describe the language structure: length of words; method of representing values, addressing data, negative numbers and binary configuration special meaning; method of addressing word limits/constraints; problems connected in presenting the data methodology; the use of registers to include the use of stack

mechanisms; the organization of addressable memory; the addressing approach used to access structured data components; and the storage of useful data at procedure cells.

10.2.2 Describe the operating system features and constraints: command language; resources used to include limitations and constraints applied plus naming convention used in resource identification; various services that exist; and method of errors handling.

10.3 Describe the target or deployment hardware and software.

10.3.1 Describe the language structure: length of words; method of representing values, addressing data, negative numbers and binary configuration special meaning; method of addressing word limits/constraints; problems connected in presenting the data methodology; use of registers to include the use of stack mechanisms; organization of addressable memory; addressing approach used to access structured data components; and storage of useful data at procedure cells.

10.3.2 Describe the operating system features and constraints: command language; resources used to include limitations and constraints applied plus naming convention used in resource identification; various services that exist; and method of errors handling.

10.3.3 Describe the devices and files: printing format constraints - line lengths, line or page feed, configuration of programmed type, and number of lines on a page; character-set used and accepted; structure of files and their access; maximum length and characteristics of magnetic media used; use and control of interactive terminals; and standard input and output files to include any known peculiarities.

10.4 Describe the programming language for host systems.

10.4.1 Describe the host programming language and any language extensions or improvements incorporated: explicit or implicit programming assumptions made; method of interpretation and treatment of errors; character SET and transport media to be used; software maintenance plans supporting documentation, testing, and updates to programs; changes to storage medias; and any change to operating system during porting.

10.4.1.1 Describe the data base: architecture to include levels; internal schema; conceptual schema; length; structure; and access methodology.

10.5 Describe the programming language for target systems.

10.5.1 Describe the target programming language and any language extensions or improvements incorporated: explicit or implicit programming assumptions made; method of interpretation and treatment of errors; character SET and transport media to be used; software maintenance plans supporting documentation, testing, and updates to programs; changes to storage medias; and any change to operating system during porting.

10.5.1.1 Describe the data base: architecture to include levels; internal schema; conceptual schema; length; structure; and access methodology.

CONCLUSION

The value of applying the Ada style guidelines, portability aids, guides and standards has been proven internally to TELOS personnel. This internal value has not been equated to resources or dollars saved, but can be observed in the improved quality of the Mission Critical Defense Systems developed. The Specific Standards Approach example will be further developed. This development will be accomplished and enhanced by soliciting your comments and suggestions for example improvement when the paper is presented. The resulting example will be completed and provided to our Government customer for forwarding to the appropriate Government representative for review and evaluation of the example and the approach. Upon approval, the approach and example will be presented by the Government representative to the appropriate DOD-STD-2167A working group for evaluation and future inclusion in the DIDs.



TOM ARCHER

TELOS Systems
P. O. Box 33099
Ft. Sill, Oklahoma 73503-0099

TOM ARCHER has a bachelor's degree in Mathematics with minors in Computer Science and Applied Physics. He has over 27 years experience in designing, developing, and implementing Government applications of various Electronic Data Processing (EDP) machines and languages. He has worked on numerous weapon system projects such as the E-3A Airborne Warning and Control System (AWACS), the M-X Missile Development, and currently is assisting in various Fire Direction Ada language system developments. Tom is a senior systems engineer, currently involved in technical analysis and management support for the Fire Direction System Branch of TELOS Systems, Region 1.



ANDREW EASSON

TELOS Systems
P. O. Box 33099
Ft. Sill, Oklahoma 73503-0099

ANDREW EASSON has both a Bachelor and a Master of Science degree in Nuclear Engineering, as well as a Master of Science degree in Computer Science. He has 13 years of experience in scientific computing and five years experience in PDSS. Andrew is a software engineer, currently involved in designing and developing of Ada language software and the new Army Common Hardware/Software equipment for the Fire Direction System Branch of TELOS Systems, Region 1.

An Ada Runtime Supervisor Simulator

R. T. Lewis, D. A. Workman, S. D. Lang

Department of Computer Science
University of Central Florida, Orlando, Florida 32816

Abstract

In this paper, we describe the design and implementation of a simulator for the Ada runtime supervisor. The input to the simulator is a tuple description of an Ada program specifying the tasks involved, the entry calls used, and the task calling relationships. The tuples describing a task body simulate Cpu operation, I/O operation (request I/O and wait I/O), task activation, termination and abort, real time delay, simple looping, and task select/accept statements. All of Ada's entry calling and accepting semantics are implemented (including guards), with the exception of entry and task attributes and interrupt entry calls.

The simulator accommodates the "single-stepping" of the simulation of the program execution allowing the user to inspect the states of all tasks, the contents of all entry queues, and the contents of the supervisor's ready, delay, and I/O wait queues. Upon termination of the simulated program, a detailed accounting of the times spent by each task in their various states is produced along with the average queue lengths, the average queue wait times, the peak queue sizes, and the number of calls made and missed for all entries in the simulated program. The user also has read access to all of the supervisor's data structures at any time.

1. Introduction

In general, modeling the behavior of the constituent modules in a concurrent system in a way that facilitates the study of their dynamic behavior is a difficult task. The execution of such a system of modules depends upon many factors that include: program design, compiler design, and operating system/compiler interface design. In the programming language ADA the overhead of the supervisory code inserted into the compiled program to implement the ADA tasking semantics along with the inherent queueing delays introduced by the entry queues can become significant. Such delays can become intolerable in real-time ADA applications [RICH87].

The supervisor runtime overhead in ADA includes: task activation, task termination, task scheduling and dispatching, context switching, exception propagation, open entry selection within selective wait statements, queueing management of entry calls, and allocation of task control blocks [NIEL88]. Burger and Nielson [BURG87] have provided a time assessment of the actual overhead associated with these and other tasking idioms used in ADA. Their statistics suggest that certain program design decisions should simply be avoided - such as, recurrent dynamic task allocation via procedure calls, to minimize supervisor overhead.

Figure 1 (in the appendix) illustrates how the ADA program is typically interfaced to the operating system in a multi-programming system [RICH87]. The Ada runtime supervisor manages the tasks within the Ada program. This task scheduling is a second level of scheduling on top of that already done by the operating system. Rich demonstrated the performance degradation due to the underlying operating system support. These problem areas include processor scheduling, I/O blocking, interrupt servicing, and exception handling. One can circumvent many of these problems by directly implementing many of these operating system support functions in ADA - a possibility since ADA has interrupt handling facility built-in.

The Embedded Software Design Simulator (ESDS) [LEFK85] uses an event driven scheme to analyze proposed ADA program designs specified in an event-encoded table format. ESDS provides input/output-pair response measures and operational efficiencies for all tasks and events (such as external interrupts). These numbers indicate an overall performance measure for a proposed program design without investing the resources to fully implement the design only to find that it has poor performance characteristics.

This paper presents an ADA runtime supervisor simulator (or, just ADA simulator) which is used together with an operating system simulator to study the behavior of tasks within an ADA program. The main goal of these simulators is similar to that of ESDS, i.e., to prototype ADA program designs. However, the ADA simulator meets the following operational requirements:

(1) It interfaces with an operating system simulator.

(2) It uses a tuple description of ADA programs.

(3) It allows concurrent execution of multiple ADA programs.

(4) It allows inspection of individual task state information and supervisor concurrent structures at each step of simulation.

The ADA simulator provides:

(1) Complete task state information for each task.

(2) Complete entry state and entry queue information, such as average queue length and average time spent by tasks in the queue.

(3) Contents of all queues: the ready queue, the delta queue, and all entry queues.

(4) A complete trace of system execution.

The remainder of this paper provides a description of the tuple specification used to capture the semantics of these concurrent facilities, the design of the necessary data structures and the program modules used to perform the supervisor simulation, and a discussion of the limitations, extensions, and overall usefulness of this simulator. All figures and an ADA program with its tuple description used in simulation are provided in the appendix.

2. ADA Runtime Supervisor Simulator Design

The XINU Operating System Design [COME84] was used as a model for the runtime supervisor. In particular, many of the queue support routines were used with little modification (however, none of the higher level XINU routines were used). As in XINU, a task is linked to a queue implicitly by its position in the task control block. This linkage works well since a task cannot be in more than one queue at a time except for the one situation where it is in a timed entry call. In this case, a separate delta queue structure is used. Once all ADA subtasks have completed, the main ADA task terminates.

Figure 2 illustrates the top-level design of the Ada simulator. The operating system simulator and the supervisor modules initialize their data structures from a sequence of program scripts at the start of the simulation. A script refers to a description of an ADA program and it includes header information that specifies the main module name, task names, priorities, code locations, entry names associated with the tasks,

and variable names with their initial values. Following this header is a memory image consisting of tuples representing the ADA program.

The operating system simulator then interprets each tuple calling the supervisor module when supervisor intervention is necessary. This intervention occurs at:

(1) simulation start,

(2) task initiation and termination,

(3) select statements,

(4) rendezvous completion,

(5) delay statements,

(6) I/O wait requests,

(7) timer interrupts,

(8) and global variable modification.

The supervisor considers all ADA programs to be simultaneously executing. Consequently, upon entry to the supervisor module a switch to the variables describing the currently executing, operating system process is done. The supervisor performs all of the necessary bookkeeping, and then schedules a new task to run. This new ADA program state is saved before returning to the operating system simulator.

The user can set various tracing parameters as well as inspect all supervisor data structures and main memory contents. This interaction occurs prior to each return from the supervisor module to the operating system module. The user can stop the simulation after a fixed number of supervisor calls or single-step the simulation (i.e., stop after each supervisor call), or allow the simulation to run with no interruption.

The statistics produced at the end of simulation (that is, at a normal program end, or at a forced end because of deadlock) include the times spent by each task in each supervisor state, the total queueing time for each task, the lifetime of each task, and the various entry queue statistics for each task. All of these statistics are described more fully in section 4.

2.1. The Tuple Simulation Language

The simulation is performed by interpreting the program script associated with the ADA task currently in the running state. The script itself is a sequence of "tuples" that describe the dynamic behavior of an executing program in terms of the events it generates. An "event" is any program action requiring the intervention of the operating system, or the ADA runtime supervisor; e.g., entry call, accept statement, rendezvous, select statement, abort, terminate, start I/O, wait for I/O, etc.

The tuples define a simulation language rich enough to capture most of the semantics of ADA's tasking facilities as well as the semantics of I/O processing. The tuple language is structured much like an assembly language in its syntactic form and is an extension of the tuple language developed for the OSSIM (Operating System SIMulator) project [WORK88]. The remainder of this section gives a brief overview of the tuple language along with examples illustrating the most relevant features.

2.2. Tuple Descriptions

Tuples have the following general form where square brackets [] denote optional syntax:

OPCODE [XXX [/M]] [YYY [/M]]

The OPCODE describes the function of the tuple. Operands XXX and YYY are optional. The optional modifier "/M" applies only to operands representing numeric values. When M = "F" the corresponding operand is FIXED at the specified value. When M = "R" the value of the operand is to be selected at RANDOM, but not to exceed the value specified. In general, the operands can denote a variety of values including Cpu cycles, byte counts, tuple address, and names of program entities.

In the discussion to follow, tuples will be described in functional groups corresponding roughly to ADA language statements they simulate. Text in upper case denotes required syntax while text in lower case denotes information that must be specified by the script writer.

2.2.1. Initiate I/O

The following pair of tuples defines a burst of Cpu activity that ends with a request to start an I/O operation on some device.

```
SIO     nnn/m1
device  bbb/m2
```

SEMANTICS

"nnn" - denotes an instruction count; the number of Cpu cycles executed before the Start I/O is given. "M1" is its modifier.

"device" - denotes a device identifier. An SIO queues an IO request for the designated device. If the device is not busy, the operation can be initiated immediately.

"bbb" - denotes a byte count implicitly specifying the duration of the I/O operation. Each device defined to the simulator has a data transfer rate. This rate is used in conjunction with the byte count to determine the completion time of the operation. "M2" is its modifier.

2.2.2. Wait for I/O

The following pair of tuples defines a burst of Cpu activity that ends with a request to wait for an I/O operation to complete. If the designated operation is not complete at the time the request is made, the task enters a blocked state. Otherwise, the task is allowed to continue.

```
WIO  nnn/m1
REQ  addr
```

SEMANTICS

"nnn" - denotes an instruction count; the number of Cpu cycles executed before the Wait I/O is given. "M1" is its modifier.

"addr" - the REQ tuple specifies the relative script address, denoted by "addr", of a device tuple associated with an SIO operation. It is this operation for which the executing task requests to wait.

2.2.3. Task Completion

The following tuple defines a burst of Cpu activity that ends with a request to terminate. The task will in fact terminate if and only if all of its subtasks have terminated and there are no I/O operations pending or in-progress. If this condition is not met, the task is said to be "complete" and remains in this state until all subtasks and I/O have terminated.

```
END  nnn/m1
```

SEMANTICS

Similar to that of the SIO and WIO tuples.

2.2.4. Create Subtask

The following tuples define an immediate request to dynamically create the identified tasks. "Kkk" denotes the number of TASK-tuples to follow. Each TASK-tuple identifies the name of a subtask to be created. The task issuing a FORK becomes the master of each subtask.

```
FORK  kkk
TASK  name1
TASK  name2
...   ...
TASK  namekkk
```

2.2.5. Task Abort

The following tuples define an immediate request to abort the identified tasks. "Kkk" denotes the number of TASK-tuples to follow. Each TASK tuple identifies the name of a task to be aborted.

```
ABORT kkk
TASK  name1
TASK  name2
...   ...
TASK  namekkk
```

2.2.6. Task Delay

The following tuple defines an immediate request to delay execution for "nnn/m1" Cpu cycles.

```
DELAY  nnn/m1
```

When a task delays, an entry is made in the Delta queue of the ADA main task and the task is suspended.

2.2.7. Conditional Control Flow

The SKIP and JUMP tuples are used to build simple loops and conditional control transfers.

```
SKIP  nnn/m1
JUMP  addr
```

SEMANTICS

"Nnn" together with the modifier "m1" determine a non-negative count when the tuple is first executed. Every time the tuple is executed, the count is tested. If the count is zero, the next tuple is execute; otherwise, the count is decremented and the next tuple is ignored (skipped). The JUMP tuple causes an unconditional transfer the to tuple at relative address "addr".

Example: A simple loop structure (10 iterations).

ADDRESS	TUPLE		COMMENT
100	SKIP	10/F	Top of loop
101	JUMP	151	
102	...		Body of loop
	...	...	Body of loop
150	JUMP	100	End of loop
151	....		Script continuation

In the example above, the first 10 times tuple 100 is executed its count field will be positive and the JUMP at 101 will be skipped. The eleventh time tuple 100 is executed its count will be zero and control will take the JUMP to tuple 151 terminating the loop.

2.2.8. Generalized Select Group for Entry Calls

Leathrum [LEAT84] describes a generalized ADA Select construct that encompasses the semantics of timed and conditional entry calls as well as timed and conditional asynchronous waits. It is his model that forms the basis for the Select groups described in this section and the next. The generalized form of the ADA Select for timed and conditional entry calls is given below.

	TUPLE			COMMENT
	SELBEG	ENT	addr2	
	OPTION	x	y	
	task	entry	addr1	
	[DELAY	nnn/m1]		Omitted if y = 0.
	....			Code executed if call
	....			not accepted.
	JUMP		addr2	
addr1	....			Code executed when
	....			rendezvous complete.
addr2	SELEND			End of Select group.

SEMANTICS

The SELBEG tuple specifies the beginning of a Select structure. The modifier, ENT, indicates the ENTRY-CALL form of the Select. "Addr2" specifies the tuple address of the corresponding SELEND tuple. The OPTION tuple has two operands, "x" and "y". For an ENTRY-CALL, x must be "1" and designates the number of entry call tuples to follow (exactly one). Y must be "0" or "1" and specifies the number of DELAY tuples following the entry call tuple. The entry call tuple identifies name of the "task" and "entry" being called. For conditional entry calls (y = 0), if the call is not immediately accepted, the tuples following the entry call are executed and the Select group terminates with a JUMP to the SELEND. For timed entry calls (y = 1), the DELAY tuple is evaluated to determine the number of Cpu cycles the calling task must wait for the callee to accept. If the call is not accepted within the specified delay, the calling task continues by executing the tuples following the DELAY. For either timed or conditional calls, if the call is accepted, the calling task continues after the rendezvous is complete by executing the tuples beginning at "addr1".

Example. Conditional Entry Call.

ADA FRAGMENT	TUPLE REPRESENTATION			
LOOP	100	SELBEG	ENT	108
SELECT	101	OPTION	1	0
Counter.Increment;	102	Counter	Increment	106
Put("Rendezvous!");	103	SIO	50/F	
ELSE	104	STDOUT	14/F	
Put("No Rendezvous!");	105	JUMP	108	
END SELECT;	106	SIO	50/F	
END LOOP;	107	STDOUT	11/F	
	108	SELEND		
	109	JUMP	100	

Example. Timed Entry Call.

ADA FRAGMENT	TUPLE REPRESENTATION			
LOOP	100	SELBEG	ENT	109
SELECT	101	OPTION	1	1
Counter.Increment;	102	Counter	Increment	107
Put("Rendezvous!");	103	DELAY	50000/F	
OR	104	SIO	50/F	
DELAY 1.0;	105	STDOUT	14/F	
Put("No Rendezvous!");	106	JUMP	109	
END SELECT;	107	SIO	50/F	
END LOOP;	108	STDOUT	11/F	
	109	SELEND		
	110	JUMP	100	

2.2.9. Generalized Select Group for Asynchronous Waits

The generalized form of the ADA Select for timed and conditional asynchronous waits is given by the tuple group below.

	TUPLE			COMMENT
	SELBEG	ACC	addrSEL	
	OPTION	x	y	x > 0 y ≥ -1
	Aguard1	accept1	addr1	Accept alternative.
	Aguard2	accept2	Aaddr2	Accept alternative.
	.....	.....	.....	.....
	Aguardx	acceptx	Aaddrx	Accept alternative.
	Dguard1	nnn1/m1	Daddr1	Delay alternative.
	Dguard2	nnn2/m2	Daddr2	Delay alternative.
	.....	.....	.....	.....
	Dguardy	[nnny/my	Daddry]	Operands omitted
				if y = -1.
Aaddr1	....			Rendezvous code
				for accept1.
	ACCEND			Rendezvous end
				for accept1.
	....			Additional code
				for accept1.
	JUMP	addrSEL		
Aaddr2	....			Rendezvous code
				for accept2.
	JUMP	addrACC		

	....		etc. for other accepts.
addrACC	ACCEND		End of Rendezvous.
	JUMP	addrSEL	
Daddr1	....		Code for Delay1.
	JUMP	addrSEL	
Daddr2	....		Code for Delay2.
	JUMP	addrSEL	
	....		etc. for other Delays.
addrSEL	SELEND		End of Select group.

SEMANTICS

The the second field of the SELBEG tuple is "ACC" indicating the ACCEPT form of the Select group. The OPTION tuple specifies the number "x" (> 0) of Accept alternatives, and the number "y" (≥ 0) of Delay alternatives in the Select group. For y = 0, the Select becomes an unconditional asynchronous wait for x > 0 Accept alternatives. For y = -1, the Select group ends with a terminate alternative. For y > 0, one or more Delay alternatives are indicated. A conditional wait is simulated with a single Delay alternative (y = 1) with a delay time of zero.

Following the OPTION tuple is one tuple for each Accept alternative, and one tuple for each Delay (or Terminate) alternative, if $y \neq 0$. Each of these tuples identifies the name of a guard function that will compute the value of the guard when the Select group is initialized. For Accept tuples, the second operand identifies the name of the task entry, while for Delay (non-Terminate) tuples, it specifies the value of the delay in Cpu cycles. The third operand specifies the address of the rendezvous code for Accept alternatives, or the default code for Delay alternatives. Since Delay, Terminate and Conditional alternatives are mutually exclusive, the last Delay tuple is specified only for $y \neq 0$. If y = -1, the one and only Delay tuple specifies the guard function for the Terminate alternative. If y = 0, no Delay tuples are defined.

Accept alternatives begin with the tuples simulating a rendezvous. Once the rendezvous is complete, the Accept alternative may complete, or it may contain an additional computation. Both possibilities are accommodated by providing a special ACCEND tuple that signals the end of an Accept rendezvous. More than one Accept alternative may transfer to the same ACCEND, but only one ACCEND can be executed within a single activation of the Select group. The ACCEND tuple generates a call to the ADA Supervisor to update task states and make the calling task ready to run.

Example. Simple Unconditional Accept.

ADA FRAGMENT	TUPLE REPRESENTATION			
ACCEPT Signal;	100	SELBEG	ACC	104
	101	OPTION	1	0
	102	Ftrue	Signal	103
	103	ACCEND		
	104	SELEND		

NOTE: In tuple 102, the guard "Ftrue" is a special guard function that always returns TRUE. Consequently, this structure represents an unconditional ACCEPT statement.

Example. Conditional Wait with Rendezvous.

ADA FRAGMENT	TUPLE REPRESENTATION			
SELECT	100	SELBEG	ACC	114
ACCEPT Signal(X:STRING) DO	101	OPTION	1	1
PUT(X);	102	Ftrue	Signal	104
END Signal;	103	Ftrue	0/F	110
ELSE	104	SIO	25/F	
PUT(" No Calls waiting!");	105	STDOUT	80/R	
END SELECT;	106	WIO	0/F	
	107	REQ	105	
	108	ACCEND		
	109	JUMP	114	
	110	SIO	25/F	
	111	STDOUT	18/F	
	112	WIO	0/F	
	113	REQ	111	
	114	SELEND		

NOTE: The ACCEPT alternative consists of tuples: 104-109. The ELSE alternative is coded as a Delay alternative with time delay of zero. Its tuples begin at 110. The guards on both alternatives are open (TRUE).

Example. Selective Wait with Guards and Terminate.

ADA FRAGMENT	TUPLE REPRESENTATION			
SELECT	100	SELBEG	ACC	109
WHEN (IP < OP + 10) =>	101	OPTION	1	-1
ACCEPT Signal(X:STRING) DO	102	Guard1	Signal	104
PUT(X);	103	Guard2		
END Signal;	104	SIO	25/F	
OR	105	STDOUT	80/R	
WHEN (OP < IP) =>	106	WIO	0/F	
TERMINATE;	107	REQ	105	
END SELECT;	108	ACCEND		
	109	SELEND		

NOTE: In the OPTION tuple, Y = -1 indicates a Terminate alternative. Guards in tuples 102 and 103 simulate the outcome of the conditions for the Accept and Terminate alternatives.

2.2.10. Global Variable Set and Increment.

To provide for realistic simulation of guard conditions on alternatives of selective waits, global variables can be initialized and modified and later tested by the pre-defined guard functions referenced in the tuples of a Select group. The following tuples are designed to define and update the values of global variables.

	TUPLE	
SET	variable	constant
INC	variable	constant

SEMANTICS

The SET tuple initializes the variable identified by the first operand to the value in the second operand. The INC tuple increments the value of the variable identified in the first operand by the value of the constant specified in the second operand. The value of the constant operand may be positive or negative. All variables referenced in SET and INC tuples are treated as global shared variables. It is the responsibility of the user (designer) to use these variables in accordance with the ADA LRM [DOD83] Section 9.11.

2.3. Data Structure Design

Figure 3 illustrates the interdependencies of the major data structures used in the supervisor part of the simulator. These data structures are necessary to perform the bookkeeping required by the ADA runtime semantics and to interface the operating system simulator to the ADA runtime supervisor simulator.

Each ADA program has its own task control block (*ptcb), queue structures (*pq, *pdq), variables (*globals), and its own I/O device-to-task map table (*io_map). *Ptcb contains a record for each task in the ADA program. Each of these records points to a block of entry records (*entry) and to the current select structure (*tselptr) in use by the associated task. *Pq is used for the ready queue and entry queue management. *Pdq is used for the task delay management and is organized as a delta queue; only the time difference between one delay quantum and the next is actually recorded in the queue. All of these structures collectively define the state of the ADA program at any given simulation time.

The structure 'program' functions as the ADA program control block for the supervisor and allows the supervisor to distinguish between multiple ADA programs. Its records contain pointers to the ADA programs' data structures. Each ADA program record is distinguished by its operating system process identifier that is passed to the supervisor module in the generalized program counter '*pc'. Details of all of these structures can be found in Lewis [LEWI89].

3. Executable Modules

The controlling routine for the ADA runtime supervisor simulator is 'SUPV'. It in turn is called from and therefore controlled by the main operating system simulator routine (see the system overview in Figure 2). The operating system simulator used to test the SUPV module uses a simple round robin scheduling. Preemption occurs on a timer interrupt, an I/O completion interrupt, or expiration of the time slice. The operating system simulator also performs the following functions required by the supervisor module:

(1) It initially loads the program scripts into the memory array, calling SUPV to read the header information for each script. SUPV needs this header information to initialize its data structures.

(2) It repeats steps 3 - 5 continuously until no more processes can execute.

(3) It reads the current tuple specified by its local program counter and advances the clock and local program counter accordingly. A structure is used to record the duration of each tuple in cpu time units. In this way the supervisor overhead, such as that discussed by Burger [BURG87], can be simulated.

(4) It calls SUPV when supervisor intervention is necessary.

(5) It checks for I/O completion and timer events calling SUPV accordingly.

Apart from reading the header information in the script files on system initialization, the SUPV module needs to know the current simulation time, current tuple location, whether a task must wait for the completion of an I/O operation, and when an I/O operation has completed (so that it can make ready the task responsible for the I/O initiation). A closer look at the structure '*pc' shows that there is also a device id. This device identifier and the address of the I/O device specification are sufficient to uniquely identify the task. This mapping is possible since the present system disallows re-entrant code, i.e., no dynamic allocation of tasks and no instantiation of task types. (The complexity of the supervisor is greatly reduced with these restrictions.) Multiple instances of a task type is modeled by replicating the task type program script for each task instance. Additional details of the design and implementation of the ADA simulator can be found in Lewis [LEWI89].

4. Simulation

Included in the appendix is an ADA program used in simulation. The ADA program modeled by the tuple description is listed first, followed by the equivalent script file contents as read upon initialization. The ADA program consists of a procedure that contains a buffer manager task, two producers (buffer writers), and two consumers (buffer readers). The buffer is a fixed length buffer of five slots with variables I and O to keep track of the input and the output positions respectively.

The simulator provides an inspection menu for tracing the system's status. Listed below are the possible choices in the inspection menu.

program counter	show the contents of the program counter
io wait list	show which tasks are waiting for I/O to complete
ready queue	show the contents of the task ready queue
delay queue	show the contents of the delta queue
all queues	show the contents of all queues
task's select	show the contents of a task's select structure
task's stats	show a task's present statistics
entry record	show the contents of an entry descriptor record
entry stats	show an entry's current statistics
task record	show an entire task record
entire program	show all data for a particular program
single step	cause the simulator to stop each time the SUPV is ready to return to the operating system simulator (this is a toggle)
trace a task	traces the action of a particular task (toggle)
task trace off	turn the trace off for all tasks
look at a tuple	allows inspection of a memory location
toggle tuples	causes the statements to be printed before the operating system simulator executes them
delay inspect	causes a breakpoint to occur after a user-specified number of calls to SUPV (a value of -1 will allow the simulation to run interrupted if single step is off)
status	show what parameters are being traced
trace variable	toggle the tracing of a variable

At the end of simulation overall statistics are generated showing where the tasks spend their time. Life time represents the task's total active time. Active time is the interval from the task's first entrance into the ready queue to its completion time (which occurs when it enters into the complete, terminated, or abnormal state). Queueing time represents the time spent in all queues - entry, ready, and delta queues. In a sense, this time is a measure of the task's idle time. The individual state times yield a time profile of the task's life and can be used to discover where the task spends its time. Listed below are the task states maintained by the simulator.

TINAC	Inactive state for the task. A task becomes active only as the result of a FORK tuple.
TREADY	The task is in the ready queue waiting its turn on the cpu.
TCURR	The task is the currently executing task for a particular ADA program. In general the time in this state is not the task's execution time.
TCOMP	The task is waiting to complete. It has reached its END tuple.
TTERM	The task has terminated.
TSUSP/WAITIO	The task is suspended waiting for an I/O operation to complete.
TSUSP/WAITACCALT	The task is suspended waiting for some entry call to be made in an unconditional selective wait.
TSUSP/WAITENTRYQ	The task is waiting in some entry queue.
TSUSP/WAITRZ	The task is waiting for a rendezvous to complete.
TABNL	Tasking error was raised for this task or this task felt the effect of the ABORT tuple.
TDELD/DLYSTMT	A DELAY tuple is being executed by the task.
TDELD/ACCDELALT	The task is executing a selective wait statement which has positive delay expression in the delay alternatives and has at least one open accept alternative with empty entry queues for the corresponding alternatives.
TDELD/TENTCALL	The task is waiting in a timed entry call.

5. Conclusions

This simulator is useful for determining how tasks behave in an ADA program by showing the tasks' life cycles. One can also get some understanding of the entry usage via the reported queue statistics. This information can indicate performance bottle-necks in the program design.

The scheduler routines in both the operating system simulator and in the ADA supervisor simulator can be reprogrammed to provide different scheduling policies. Much flexibility is possible in the process scheduling. But, in the task scheduling one must ensure that the highest priority task is always running(the only ADA language requirement [DOD83]). One can therefore investigate the effects on an ADA program design by the supervisor using a round-robin policy among tasks of equal priority (e.g.,to ensure fairness of task execution).

Present system limitations include: no task dynamic instantiations; no handling of non-task compilation units such as procedures, block statements, etc., for task dependencies; no direct support for interrupt entry calls is provided in the ADA supervisor, but is delt with indirectly through the operating system simulator; no support for non-ADA processes is provided in the operating system simulator.

Apart from removing the above limitations, this system could conceivably be integrated with the GRIP system [WORK85], currently a C-based visual programming environment that allows users to describe program and data structures graphically. The capabilities of GRIP coupled with the ADA simulator would make a more dynamic and friendly design environment in which the user could modify a program design and 'test' it via simulation before committing to detailed design and implementation. The graphical interface of GRIP would enhance the designers ability to visualize the dynamics of program execution and to observe the effects of a change. As it stands, the simulating system presented in this paper is a step toward such a prototyping environment for ADA program designs.

BIBLIOGRAPHY

[BURG87]Burger, T. M., K. W. Nielson, "An Assessment of the Overhead Associated With Tasking Facilities and Tasking Paradigms in Ada," Ada Letters, Vol VII, no. 1, JAN/FEB 1987.

[COME84]Comer, D., *Operating System Design - the XINU Approach*, PRENTICE HALL, 1984.

[CONW63]Conway, M.E., "Design of a separable Transition Diagram and Compiler," Comm. ACM, 6,7, Jul(63) pp 396 - 408.

[DOD79] "Rationale for the design of the Ada Programming Language," SIGPLAN Notices, v14, no6, part B, June 1979.

[DOD83] U.S. Department of Defense, "Ref. Manual for the Ada Programming Language," Feb. 1983, ANSI/MIL-STD-1815A-1983.

[LEAT84] Leathum, J. F., "Design of an Ada Run-Time System," IEEE 1984 ADA Applications and Environments, pp. 4 - 13.

[LEFK84] Lefkovitz, D., R. Lee, P. R. Nelson, "An Embedded Software Design Simulator for Ada Multitasking," Proc. of the 1984 Intl. Conf. on Parallel Processing, Aug. 1985.

[LEWI89] Lewis, R. D., "An ADA Program Design Analyzer," Master's Project Report, Department of Computer Science, University of Central Florida, Orlando, FL, Aug. 1989.

[NIEL88] K. Nielsen, K. Shumate, *Designing Large Real-Time Systems with ADA*, McGraw-Hill 1988.

[RICH87] Rich, Vincent F., "A Multiprocessor Bare Machine Ada System for Flight Simulators," Proceedings of the 9th Interservice Industry Training Systems Conf., Nov-Dec 1987, pp. 1-7.

[WORK85]Workman, D., "GRIP : A Formal Framework For Developing A Support Environment For Graphical Interactive Programming," Conf. on Comp. Software Tools, Apr. 1985

[WORK88]Workman, D., "An Operating System Simulator," Technical Report CS-TR-88-07, Department of Computer Science, University of Central Florida, Orlando, FL, Apr. 1988.


```
EXAMPLE 1.  Buffer Manager.

-- Two Producers
-- Two Consumers

PROCEDURE Prod_Cons IS
  PRAGMA Priority(10);

  TASK Buff_Mgr IS
   PRAGMA Priority(1);
   ENTRY Append;
   ENTRY Remove;
  END Buff_Mgr;

  TASK Prodcr1;
   PRAGMA Priority(5);
  TASK Prodcr2;
   PRAGMA Priority(5);
  TASK Consmr1;
   PRAGMA Priority(2);
  TASK Consmr2;
   PRAGMA Priority(2);


  TASK BODY Buff_Mgr IS
   I : NATURAL := 0;
   -- I = Buffer Input index
   O : NATURAL := 0;
   -- O = Buffer Output index
   Buffer : ARRAY(1..5) of INFO;
  BEGIN
   LOOP
    SELECT
      WHEN ( I < O+5 ) =>
      ACCEPT Append DO
         -- Write to buffer
         END Append ;
      I := I + 1;
    OR
      WHEN ( O < I ) =>
      ACCEPT Remove DO
         -- Read from buffer
         END Remove;
      O := O + 1;
    OR
      TERMINATE;
    END SELECT;
   END LOOP;
  END Buff_Mgr;

  TASK BODY Prodcr1 IS
  BEGIN
    FOR i IN 1..10 LOOP
      Buff_Mgr.Append;
    END LOOP;
  END Prodcr1;

  -- Prodcr2 is identical
  -- to Prodcr1.

TASK BODY Consmr1 IS
BEGIN
  FOR i IN 1..10 LOOP
    Buff_Mgr.Remove;
  END LOOP;
END Consmr1;

-- Consmr2 is identical
-- to Consmr1.

BEGIN -- Main task
 NULL;
END Prod_Cons;
```

```
HEADER
5     2     2
TASK              PRI   LOC   #ENT
Prod_Cons          10     0     0
buff_mgr            1    10     2
ENT     append
ENT     remove
Prodcr1             5    30     0
Prodcr2             5    40     0
Consmr1             2    50     0
Consmr2             2    70     0
I     0
O     0
Buff_Mgr
Prodcr1
Prodcr2
Consmr1
Consmr2
_end_of_callers
_end_header
0     FORK        5
1     buff_mgr
2     Prodcr1
3     Prodcr2
4     Consmr1
5     Consmr2
6     DELAY       10/F
7     END         10/R

10    SELBEG      ACC        22
11    OPTION        2        -1
12    FIGRD     append       15
13    FOGRD     remove       19
14    Ftrue
15    DELAY         5/R
16    ACCEND
17    INC           I         1
18    JUMP         22
19    DELAY        10/R
20    ACCEND
21    INC           0         1
22    SELEND
23    JUMP         10
24    END          10/F

30    SKIP         10/F
31    JUMP         37
32    SELBEG      ENT        35
33    OPTION        1         0
34    buff_mgr append         35
35    SELEND
36    JUMP         30
37    END          20/F

40    SKIP         10/F
41    JUMP         47
42    SELBEG      ENT        45
43    OPTION        1         0
44    buff_mgr append         45
45    SELEND
46    JUMP         40
47    END          10/R

50    SKIP        10/F
51    JUMP        61
52    SELBEG     ENT         55
53    OPTION       1          0
54    buff_mgr remove        55
55    SELEND
56    SIO          5/F
57    DISK        10/F
58    WIO         10/F
59    REQ         57
60    JUMP        50
61    END         10/F

70    SKIP        10/F
71    JUMP        81
72    SELBEG     ENT         75
73    OPTION       1          0
74    buff_mgr remove        75
75    SELEND
76    SIO          6/F
77    STDOUT      10/F
78    WIO         10/F
79    REQ         77
80    JUMP        70
81    END         10/R
```

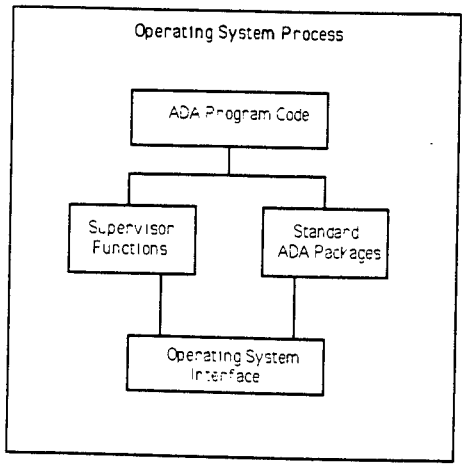


Figure 1: An ADA Process
[RICH87]

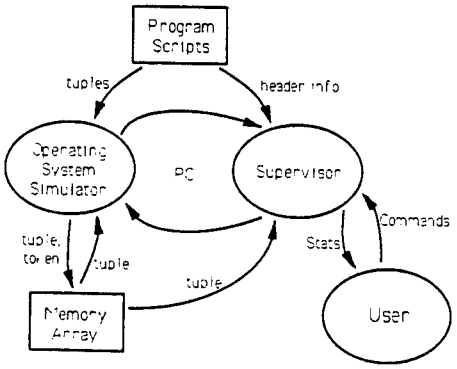


Figure 2: System Overview

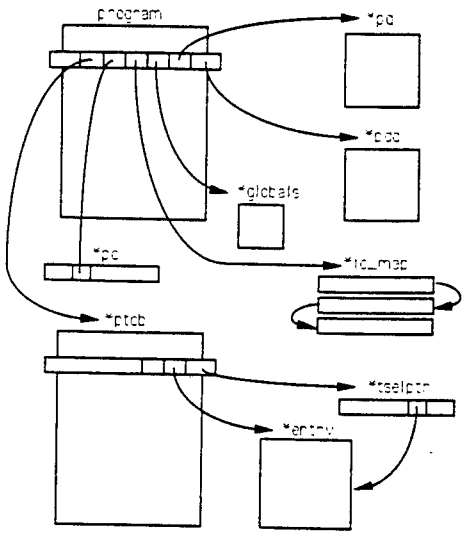


Figure 3: Data Structures



RUSSELL T. LEWIS received the B.S. degree in Applied Mathematics from the Montana College of Mineral Science and Technology in 1981, and the M.S. degree in Computer Science from the University of Central Florida in 1989. He worked as a systems analyst for the Montana Nuclear Power School (separating as a Lieutenant from the USN after 6 years), and is presently a mathematician at the Yuma Proving Ground. His research interests include: software engineering, operating systems, and formal (mathematical) system theory.



DAVID A. WORKMAN is currently Associate Professor of Computer Science at the University of Central Florida. Dr. Workman's research interests lie in visual programming tools and environments, software metrics, and Ada technology. He is currently heading the development of GRIP (GRaphical Interactive Programming), a C-based visual programming environment. The author's present address is: Department of Computer Science, University of Central Florida, P.O. Box 25000, Orlando, Florida, 32816.



SHEAU-DONG LANG is an Assistant Professor in the Computer Science Department of the University of Central Florida. He received the B.S. degree in Mathematics from the National Taiwan University, and the M.S. degree in Computer Science and the Ph.D. degree in Mathematics both from the Pennsylvania State University. His research interests include: software engineering, real-time simulation, and database file systems. His present address is: Department of Computer Science, University of Central Florida, P.O. Box 25000, Orlando, Florida, 32816.

RESOLUTION OF PERFORMANCE PROBLEMS IN AN ADA COMMUNICATIONS SYSTEM

Thomas L.C. Chen and Ralph P. Cooley
E-Systems, ECI Division and Sequent Computer Systems
St. Petersburg, Florida and Tampa, Florida

Abstract

Operating systems, high level languages, and software development methodologies are introduced to make software development easier; however, good intention does not always lead to a gratifying result. This is particularly true of software developed for high performance systems. This paper discusses the resolution of the performance problems of a specific Ada data communication software system. This software system was built on fault-tolerant distributed processors, loosely coupled together by a UNIX compatible operating system. Upon initial integration, the principal problem was that the performance was 58 times slower than required by contract.

Introduction

The intention to make software development easy by the introduction of operating systems, high level languages, and various software development methodologies does not always lead to a gratifying result.

This paper discusses the performance problems of an Ada data communication software system, the methodology used to resolve the problem, the key element to make the methodology practical, and a few examples on how the method applies to a real situation.

A common sense cyclic method was adopted to enhance the performance of the software. The performance problem was resolved by the modification of the software design in accordance with conventional system programming practices, the selection of efficient programming constructs, and the conversion of selected program modules to C.

Developed according to MIL-STD-2167, the software system was built on fault-tolerant distributed processors, loosely coupled together by a UNIX compatible operating system.

Between proposal and critical design review, three separate timing and sizing studies were made. Two of the studies were conducted by the contractor, and one by the separate monitoring agency. The project won an award for its design efforts.

The software design, conceived during the proposal phase of the contract, was based on performance claims of the hardware vendor and supported by in-house benchmark data collected for several fault-tolerant systems.

The design produced following the Preliminary Design Review (PDR) was substantially different from the original proposal. Due to the results of a timing study conducted during the preliminary design phase of the project, the following decisions were made:

- To use UNIX processes instead of Ada tasking.
- To use operating system provided fail-over instead of application fail-over.
- To use contractor developed software instead of vendor provided application software.

The standard UNIX performance measurement tools were improved and augmented to identify the bottlenecks of the application. Upon initial integration of the system, the performance was 58 times slower than required by the contract.

The methodology to resolve this performance problem, alternative approaches, implementation of the methodology, and validity of the proposed methodology are discussed in the following paragraphs. The software terminology and abbreviations can be found in any comprehensive UNIX manual.

Problem Resolution

The problem is how to optimize the software application for a selected hardware platform.

Thus, the Resolution is to develop the most effective software in terms of what combination of hardware, concurrent processing, and System time may be available for a specific application. This usually means that the program has to:

a. Use a minimum amount of hardware resources to achieve the functional requirement of the contract.

b. Arrange each piece of the software into concurrent processes to make full use of each part of the dedicated hardware platform.

c. Reduce the overhead in creating or rearranging processes to less than the idle time in the alternative architectures. The overhead includes the creation and maintenance of each process and the interaction between processes.

Since the software was already built, the following cyclic approach was adopted to reach the goals. This approach was born of experience and common sense.

a. Use the existing process architecture and optimize the code within each process. A process level instrumentation is used to identify the wasteful code within a process.

b. A system level instrumentation is used to identify the idle resources in the dedicated hardware platform while the application is subjected to load.

c. Repeat steps a and b until significant idle resources in the hardware platform are identified.

d. Measure the total CPU consumption of all the processes under load and the wall clock time to process the load.

e. Rearrange software functions and processes stepwise to reduce the wall clock time and idle resources of the dedicated hardware platform without undue increase in the total CPU consumption of all the processes under load.

f. Repeat the above steps until the desired performance is reached. Different levels of solutions are introduced in each repeat.

At each performance enhancement cycle a set of performance enhancement solutions is introduced. Candidates were selected on perceived cost and benefit. Only Candidates that produce results are retained. The following are some Candidate solutions:

a. Introduce equivalent program constructs which are more efficient for the selected platform.

b. Trim down the application code that exceeds the contract requirements.

c. Introduce more efficient alternate software solutions.

d. Rearrange function to process and hardware allocation.

e. Update compiler.

f. Fine tune the operating system.

g. Optimize I/O access paths.

h. Compile with suppress checking.

i. Introduce inline.

j. Convert to more efficient language for the selected hardware platform.

k. Adjust the priority of each UNIX process.

Solutions introduced in each performance enhancement cycle were selected according to their perceived cost and benefits.

The Alternative Solutions

The mainstream of formal solutions dealing with performance problems in the software system is the Waterfall method dictated by MIL-STD-2167, its processors, contemporaries, and followups.

In essence, this methodology dictates that no code should be written until all the design details are resolved. The cause of failure of any project that follows this methodology is generally attributed to not enough design work up front, not following the 'real' method, or an inexperienced (incompetent) labor force. The viability of this method has been debated and no doubt will continue to be debated. This project has been conducted according to the required methodology and won an award for the design phase of the project. The design is sanctioned by three separate timing/sizing studies. Two of the studies were done by the project; one was done by the independent monitoring contractor. The code has also been verified for compliance to design. When the software was operated, the performance had no resemblance to any of the studies.

Most published work on performance problems for Ada software is concentrated on tasking and the interactions between tasks. These references are not applicable to this project because Ada tasking was precluded from this project. The principle findings by Bender and Griest [1] agree with the experience on this project. There are errors in validated Ada compiler; source code of Ada runtime is needed for producing the performance code. The speed improvement in the Ada compiler is not necessarily proportional for various custom applications and the improvement schedule is not dependable.

How Does the Proposed Methodology Work

The application provides receive and send time stamps for each message. The number of messages processed in a specific time, together with the delay introduced by the processing, makes up the acceptance criteria.

While the time stamps on messages provide a pass and fail indication of the performance test, they are not sufficient to analyze the cause of slow performance.

To support the proposed methodology, it is essential to have instrumentation to identify the utilization of all parts of the selected hardware platform, and show how the resources are utilized by the software application. The vendor of the UNIX based Ada development system does not provide instrumentations for performance. Each project must provide its own instrumentation. Once instrumented, the Candidates are selected intuitively, as well as experimentally. The instrumentation assembled to support the approach and two of the Candidates are discussed below.

Customized Performance

To support the resolution of the performance problems, two sets of tools are assembled out of utilities provided by UNIX like systems. The first set is a profiling tool to break down the resource consumption of each Ada program in the system, which is mapped into a UNIX process. The second set is a tool to detail the interactions of all processes and the system resources usage.

Process Level Instrumentation. Each Ada program in the application is mapped into a UNIX process. An instrument is required to analyze how each part of the Ada program uses the system resource. A customized software collection, mostly from the UNIX native utilities, is produced to serve this purpose.

Profiling of an Ada Program

Most UNIX based systems provide process profiling capability through the use of the monitor library function (see UNIX Programmer's Manual), the profile system call (see UNIX Programmer's Manual), and the gprof utility (see UNIX User's Manual).

The usual method of creating a profileable UNIX executable is to compile the source code with the proper command line flags (e.g., the C compiler command line might be cc -gp src_file.c). This forces the compiler to modify its behavior in three ways:

a. by inserting calls to a routine called mcount at the start of each subroutine,

b. by invoking the object file link editor with an argument to force linkage of a profileable process startup routine, and

c. by invoking the object file link editor with arguments to force linkage of other profileable run-time libraries.

The purpose of mcount is to increment a counter each time this subroutine is entered. There is one count accumulator data space set aside for each subroutine linked into the executable.

The process startup routine which gets linked in is different from the normal (unprofiled) routine. This routine, named _start in this system, is the text entry point for each UNIX process.

The _start routine allocates memory for each mcount accumulator and notifies the operating system that the current program counter (pc) should be range checked each time a system clock interrupt occurs (in this case, 100 times a second). At each such interrupt the system should record where in the text the process is executing. At the completion of profiling, the clock interrupt counts can be used to produce a time histograph of CPU usage per subroutine. That is, the CPU counters can be analyzed to determine in which areas of the process CPU time is being consumed, on a subroutine level basis. Note that this is an approximation of the activities of the process. It is dependent upon a process life span which is long enough to provide count totals which are indicative of the true run-time behavior of the process.

The end result of compiling profileable code is an executable file which will include instructions to maintain a count of the number of times a function is called, as well as provide the date, to produce a histograph of processing time approximations for each function.

The Ada compiler vendor in this case did not provide profiling capabilities as part of the compiler. Therefore, it was necessary to modify the behavior of the Ada startup routine (found in the standard object file startup.o) and the linkage editor commands to produce any profiling data. Note, however, that insertion of calls to mcount could not be accomplished without source code to the compiler. Since we did not have the source code, we were not able to provide subroutine call counting except for the system-provided profileable library calls.

The features of our eventual profiling implementation include the following:

a. No modification of Ada source code by developers. The significance of this feature is obvious.

b. Conditional profiling, based upon a shell environment flag. With this flag we could use the same executable for non-profiled runs without invoking any profiling overhead.

c. Usage of a standard UNIX utility (gprof) to provide a portion of the profile analysis. We did not desire to create new tools.

d. Accumulation of C language (but not Ada) routine entry counts (via mcount), which were useful when determining the number of individual system calls made by a process.

e. Control of profiling by (external) UNIX signals: SIGQUIT halts profiling, SIGSYS writes profile data to files, SIGIOT continues profiling, and SIGEMT restarts profiling (e.g., like SIGIOT but resets mcount and CPU time counters to zero before continuing). UNIX profiling normally starts at process initiation and completes at process termination. With this control, we were able to turn on and then turn off profiling to focus on activities of interest during the life of the process.

f. Profile data is written out to unique file names upon receipt of a predefined signal (SIGSYS) to the process. Normally, UNIX profiling data is placed in a file named gmon.out. However, since our intention was to run a large number of concurrent profiles over a long period of time under many different conditions, the unique data file naming was necessary.

g. Additional process information was provided as part of the profiling data. This data was provided by the UNIX getrusage system call and included CPU usage, context switch counts, and various memory usage statistics (see Table 1).

Of course, most features are not free. In particular, here are the cost elements of the new profiling implementation:

a. Modification of the Ada startup routine. This was accomplished by disassembling the startup routine, modifying it, and then reassembling.

b. Modification of the standard UNIX monitor functions. This was also accomplished via a disassembly/ reassembly.

c. Modification of the standard UNIX exit function, accomplished by disassembling and reassembling.

d. Creation of an intermediate pre-linker that was invoked by the compiler and which then called the standard UNIX linker. The pre-linker ensured that the proper profiling libraries were linked in.

Table 1. Snap-Shot Summary Provided by the Custom Ada Program Profiler

	Current	Previous	Difference
CPU usage (%)	3.68	0.00	3.68
Total CPU time (CPU-SEC)	20.30	0.00	20.30
User CPU time (CPU-SEC)	12.80	0.00	12.80
System CPU time (CPU-SEC)	7.50	0.00	7.50
Total memory (KB) Shared memory (KB) Unshared data (KB) Unshared stack (KB)	1368 0524 0844 0	0 0 0 0	0 0 0 0
Max resident memory size (KB)	1657	0	1657
Page reclaims (no I/O)	460	0	460
Page faults (req'd I/O)	161	0	161
Swaps of this process	0	0	0
Block input operations	188	0	188
Block output operations	395	0	395
Messages sent	0	0	0
Messages received	0	0	0
Signals received	1	0	1
Voluntary context switches	415	0	415
Involuntary context switches	1262	0	1262
Total context switches	1677	0	1677

It is our feeling that the overhead incurred in our profiling process is insignificant. The mcount routine consists of a few instructions, while the CPU time data overhead was incurred at the expense of normal operating system kernel activities. There is some initial overhead incurred during process initialization to set up various data count areas. However, since our intent was to study the performance of the system under steady state conditions, we usually started profiling activities via a UNIX signal after initialization activities had been completed.

Analysis of Profile Data

A program was produced to use with 'gprof' to analyze the profiling data. An example of the output is in Table 2. Some of the problems introduced by this profiling approach are:

a. The '__write' system call values reflect the calls to write by the profiling software itself. Be wary of examining performance problems based on large calls to write.

b. In some cases, one function will show an inordinate number of calls due to some memory rounding error in gprof. If such a condition exists and is not easily explainable, you should consider it a measurement error.

Table 2. Profile of Ada Program

Sorted by Usage Non-zero Cumulative CPU Hits per Subroutine		
23.1%	(94)	__A__bcopy.13S13.mm__bcopy
22.6%	(92)	__A__lmult.5S13.1a__mul
9.3%	(38)	__writerandom
8.6%	(35)	__open
4.4%	(18)	__A__init__duplicate__elim__queue.76S14.duplicate__elimination
2.7%	(11)	__A__init__work__order__queue.419B14.cp__retention__queue
1.7%	(7)	__select
1.7%	(7)	__A__la__mul..BODY
1.5%	(6)	__close
1.5%	(6)	__A__init__status__intf__pkg.268S14.sts__intf__common
1.2%	(5)	__shmat
1.2%	(5)	__A__enum__value.14S12.at__evalue
1.0%	(4)	__A__init__rtn__cntl.417B14.cp__retention__queue
1.0%	(4)	__A__bzero.13S13.mm__bzero
1.0%	(4)	__A__init__msg__store.415B14.cp__retention__queue
0.7%	(3)	__A__receive__from__server.202S14.sts__server__intf
0.7%	(3)	__A__init__server__intf.147S14.sts__server__intf
0.7%	(3)	__detzcode

System Level Instrumentation. System level instrumentation provides information on the interactions between UNIX process and system resource usage.

Native Commands and their Derivatives

UNIX provides a set of useful instrumentation. The following are those used frequently:

vmstat- report statistics on system activity prints iterative statistics over time for: processes, memory usage, traps, disk activity, and CPU usage.

For example, "vmstat 5 100" prints system statistics every 5 seconds, for 100 times.

ps- process status is printed for the user at the time it is requested. Process status includes such information as cumulative CPU, memory, and disk usages for all processes in the system.

time- The given command is executed; after it is complete, 'time' prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds. An improved version of this function is provided. The output of this improved version is shown in Tables 3 and 4. Elapsed time is accurate to the second, while the CPU times are measured to the 60th second. Thus the sum of the CPU times can be up to a second larger than elapsed time.

pstat print system facts interprets the contents of certain system tables, such as the print process table for active processes and a summary of swap space usage.

It also prints the number of used and free slots in the several system tables and is useful for checking to see how full system tables have become if the system is under heavy load.

Table 3. Improved Time Function Output and Large Write No Copy Results

Test Sequence	1	2	3
Wall time (SEC)	9	17	13
Total CPU time (CPU-SEC)	1.5	1.317	1.383
Percentage of time CPU used	7	8	11
User time (CPU-SEC)	0.117	0.100	0.100
System time (CPU-SEC)	1.382	1.217	1.283
Max resident memory size (KB)	192	190	192
Average shared memory size	4074	3937	3838
Average unshared data size (KB)	5845	5640	5447
Average unshared stack size (KB)	0	0	0
Page reclaims (no I/O)	47	47	47
Page faults (req'd I/O)	3	3	3
Swaps of this process	0	0	0
Block input operations total (per second)	0(0)	0(0)	0(0)
Block output operations total (per second)	1172 (781.37)	1172 890.162	1172 847.26
Messages sent total (per second)	0(0)	0(0)	0(0)
Messages received	0(0)	0(0)	0(0)
Signals received total (per second)	0	0	0
Voluntary context switches	43	51	48
Involuntary context switches	0	0	0
Total context switches total (per second)	40 (68.57)	51 (38.74)	48 (34.700)

Customized System Profile Tool

A simple command file consisting of the following three UNIX commands is used as the primary system level instrumentation:

```
ps -aug
vmstat 1 300
ps -aug
```

This command file is activated to record the activities in the system every second for 300 seconds during which a specific load (of message traffic) is applied to the system. Figure 1 is derived from data collected by this command file. The figure shows that the CPU usage is hindered by disk access and suggests that the CPU intensive function should be redistributed so that the CPU will not be left idling while each process is waiting for data from disk. Potentially 40% more messages can be processed by the CPU.

The coordination for application load and the initiation of the command file is an awkward operation. Time stamps provided by the applications are used to coordinate and verify the results of this instrument.

Performance Enhancement

The performance problem was resolved by the modification of the software design in accordance with conventional system programming practices, the selection of efficient programming constructs, and the conversion of selected program modules to C. The majority of the improvement came from the modification of the software design. The original design consisted of data abstraction packages, loosely fitted in a data flow design. Services within each data abstraction were invoked by passing data. The modified design utilizes global data packages to eliminate excessive data transfers between data abstractions, accounting for the majority of required improvements. The following listings are examples of performance improvements carried out during the performance enhancement effort.

An Inefficient Program Construct Problem. The Ada compiler available in the selected hardware platform has a well known problem [2]. This problem is confirmed by the Customized Ada program profiling tool.

Table 4. Improved Time Function Output and Different Loop Construct Performance

Test Sequence	1	2	3	4	5	6	7
Wall time (SEC)	1	1	1	1	1	1	1
Total CPU time (CPU-SEC)	0.350	0.267	0.283	0.250	0.265	0.517	0.400
Percentage of time CPU used	35%	27%	28%	25%	27%	52%	40%
User time (CPU-SEC)	0.150	0.100	0.117	0.100	0.133	0.317	0.267
System time (CPU-SEC)	0.200	0.167	0.167	0.160	0.133	0.200	0.133
Max resident memory size (KB)	52	52	58	58	58	100	66
Average shared memory size (KB)	600	592	1030	1016	952	1188	1152
Average unshared data size (KB)	1180	1117	1129	1064	1132	2132	2025
Average unshared stack size (KB)	0	0	0	0	0	0	0
Page reclaims (no I/O)	10	10	8	8	8	8	8
Page faults (req'd I/O)	1	1	3	3	3	3	3
Swaps of this process	0	0	0	0	0	0	0
Block input operations Total (per second)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
Block output operations Total (per second)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
Messages sent Total (per second)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
Messages received Total (per second)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)	0(0)
Signals received	0	0	0	0	0	0	0
Voluntary context switches	29	28	24	24	22	23	23
Involuntary context switches	0	0	0	0	0	0	0
Total context switches Total (per second)	29 (82.86)	28 (105.0)	24 (84.71)	24 (96.00)	22 (82.50)	23 (44.52)	23 (57.50)

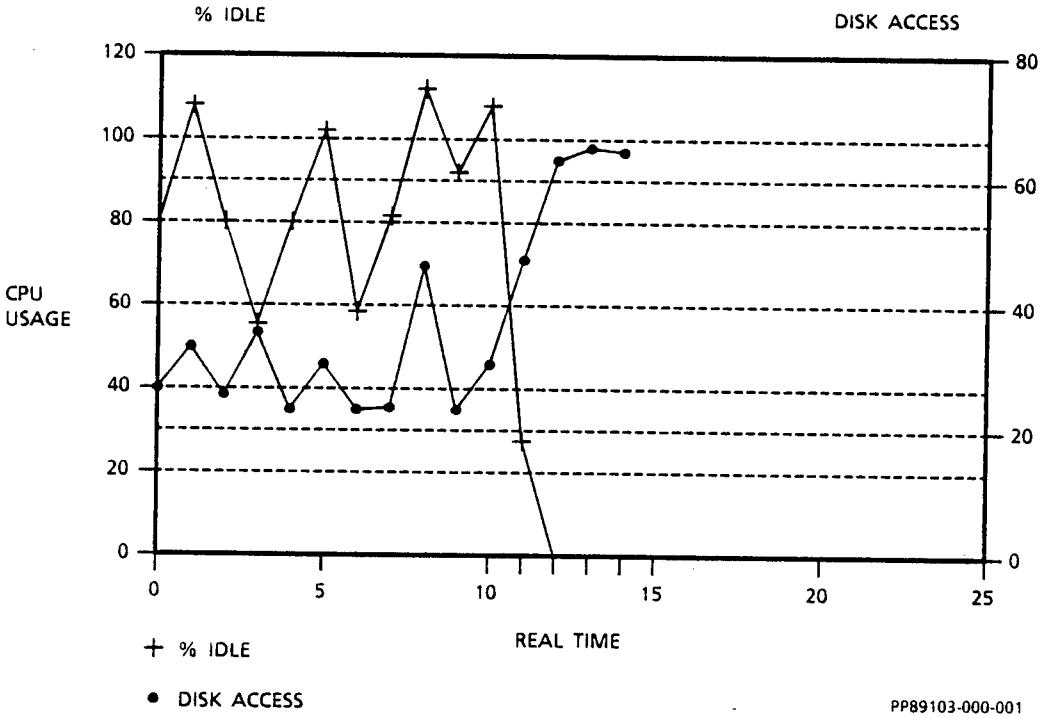


Figure 1. Data Collected by the Customized System Profiling Tool

Table 2 is an example of the profile of an Ada program before the first performance enhancement cycle. About 46% of the CPU is used in copy from one memory location to another and in long integer multiply (call to __A__lmult). The long integer multiply is related to the known problem in the compiler. The CPU consuming multiply is used in array index. By liberal use of 'RENAME', the call to '__A__lmult' is curtailed. Table 5 is a profile of the same Ada program after 'RENAME' is liberally introduced. Because of the other performance enhancement solutions introduced in the same performance enhancement cycle, the total CPU consumed by the program is greatly reduced. After this enhancement, the call to '__A__lmult' still consumes 20% of the CPU even though the application has no requirement for long integer multiply.

The reasons for the remaining use of the long integer multiply are found in loop programming constructs that process large arrays. The 'RENAME' solution cannot be applied to the loop programming constructs.

Listing 1 shows seven test programs used to identify the most efficient loop constructs to process large array. Table 3 shows their performance.

Table 4 is the performance of these seven alternatives:

test1: Simple 'for' loop with loop variable N for individually indexing and assigning each item in a record.

test2: Simple 'for' loop with loop variable N for indexing but assigning all items in a record in one statement.

test3: Simple 'while' loop with a short integer N for index.

test4: Uses a pointer to record in an array in the heap and increment the pointer in a 'while' loop.

test5: Same as 4 but uses a 'for' loop.

test6: Uses a link list in a 'for' loop instead of array.

test7: Uses an array but adds a link in each record in the array so each record can be accessed as a link list.

This reflects the current implementation of a particular compiler.

If an updated compiler can be made available within the time frame of the project, then this procedure is not necessary.

Table 5. Profile for an Ada Program Optimized for Large Copy and Static Indexing

Time	Cumsecs	Seconds	Calls Name
22.4	2.60	2.60	BZERO
20.8	5.02	2.42	BCOPY
19.9	7.33	2.32	LMULT
12.6	8.80	1.47	__A__mainline
3.4	9.20	0.40	__A__comm__transmit__msg
2.4	9.48	0.28	49 __open
1.9	9.70	0.22	mcount
1.3	9.85	0.15	__A__number__io
1.3	10.00	0.15	__A__process__registration__to__mmi
1.1	10.13	0.13	17 __writerandom
1.0	10.25	0.12	INTR
0.7	10.33	0.08	6 __select
0.6	10.40	0.07	210 __malloc
0.6	10.47	0.07	ENUM__VALUE
0.6	10.53	0.07	VAL__TO__POS
0.6	10.60	0.07	__A__host__message__data__definition__package
0.6	10.67	0.07	__A__host__retention__intf__pkg
0.4	10.72	0.05	50 __sbrk
0.4	10.77	0.05	18 __close
0.4	10.82	0.05	__A__close__all
0.4	10.87	0.05	__A__tx__calls
0.3	10.90	0.03	278 __asctime
0.3	10.93	0.03	20 __read
0.3	10.97	0.03	8 __write
0.3	11.00	0.03	AA__GLOBAL__NEW
0.3	11.03	0.03	ENUM__IMAGE
0.3	11.07	0.03	RAISE
0.3	11.10	0.03	__A__byte__ops
0.3	11.13	0.03	__A__cp__queue

Listing 1. Inter-process Transfer Prototype Programs

```
with SSH_IO_SPRT;

procedure LG_WRITE is

   package IO        renames SSH_IO_SPRT;

   type INT_ARRAY_TYPE is array (1 ..
1500) of TINY_INTEGER;
   type STR_ARRAY_TYPE is array (1 .. 5)
of INT_ARRAY_TYPE;

   package MSG_IO  is new IO.TXIPC_IO
(STR_ARRAY_TYPE);

   STR_FILE
      : IO.FILE_TYPE;
   ORIG_STRING     : INT_ARRAY_TYPE :=
(others => 55);
   MSG_STRING      : STR_ARRAY_TYPE;

   TIMER_ITER      : constant := 10;
   NUMBER_OF_BLKS : constant := 4;

begin

   MSG_IO.OPEN (FILE => STR_FILE,
                MODE => IO.OUT_FILE,
                NAME => "BENCHMARK_2");

   for TIMER in 1 .. TIMER_ITER loop
      for BLKS in 1 .. NUMBER_OF_BLKS
loop

         for I in 1 .. 5 loop
            MSG_STRING (I) :=
ORIG_STRING;
         end loop;

         -- Write message block to
 IPC
         MSG_IO.WRITE (FILE => STR_FILE,
                       ITEM =>
MSG_STRING);

      end loop;
   end loop;

end LG_WRITE;
with SSH_IO_SPRT;

procedure LG_WRITE_NO_COPY is

   package IO        renames SSH_IO_SPRT;

   type INT_ARRAY_TYPE is array (1 ..
1500) of TINY_INTEGER;
   type STR_ARRAY_TYPE is array (1 .. 5)
of INT_ARRAY_TYPE;

   package MSG_IO  is new IO.TXIPC_IO
(STR_ARRAY_TYPE);

   STR_FILE     : IO.FILE_TYPE;
   MSG_STRING   : STR_ARRAY_TYPE :=
(others => (others => 55));

   TIMER_ITER      : constant := 10;
   NUMBER_OF_BLKS : constant := 4;

begin

   MSG_IO.OPEN (FILE => STR_FILE,
                MODE => IO.OUT_FILE,
                NAME => "BENCHMARK_2");

   for TIMER in 1 .. TIMER_ITER loop
      for BLKS in 1 .. NUMBER_OF_BLKS
loop

         -- Write message block to IPC
         MSG_IO.WRITE (FILE => STR_FILE,
                       ITEM =>
MSG_STRING);

      end loop;
   end loop;

end LG_WRITE_NO_COPY;
with SSH_IO_SPRT;

procedure LG_WRITE_NO_COPY_OPEN is

   package IO
       renames SSH_IO_SPRT;

   type INT_ARRAY_TYPE is array (1 ..
1500) of TINY_INTEGER;
   type STR_ARRAY_TYPE is array (1 .. 5)
of INT_ARRAY_TYPE;
```

```
   package MSG_IO  is new IO.TXIPC_IO
(STR_ARRAY_TYPE);

   STR_FILE        : IO.FILE_TYPE;
   TIMER_ITER      : constant := 10;
   NUMBER_OF_BLKS : constant := 4;

   MSG_STRING      : STR_ARRAY_TYPE;

begin

   MSG_IO.OPEN (FILE => STR_FILE,
                MODE => IO.IN_FILE,
                NAME => "BENCHMARK_2");

   for TIMER in 1 .. TIMER_ITER loop

for BLKS in 1 .. NUMBER_OF_BLKS loop

         -- Write message block to IPC

         MSG_IO.READ (FILE => STR_FILE,
                      ITEM =>
MSG_STRING);

      end loop;
   end loop;

end LG_WRITE_NO_COPY_OPEN;
with SSH_IO_SPRT;

procedure LG_WRITE_OPEN is

   package IO        renames SSH_IO_SPRT;

   type INT_ARRAY_TYPE is array (1 ..
1500) of TINY_INTEGER;
   type STR_ARRAY_TYPE is array (1 .. 5)
of INT_ARRAY_TYPE;

   package MSG_IO  is new IO.TXIPC_IO
(STR_ARRAY_TYPE);

   STR_FILE        : IO.FILE_TYPE;
   TIMER_ITER      : constant := 10;
   NUMBER_OF_BLKS : constant := 4;

   MSG_STRING      : STR_ARRAY_TYPE;

begin

   MSG_IO.OPEN (FILE => STR_FILE,
                MODE => IO.IN_FILE,
                NAME => "BENCHMARK_2");

   for TIMER in 1 .. TIMER_ITER loop
      for BLKS in 1 .. NUMBER_OF_BLKS
loop

         -- Write message block to IPC
         MSG_IO.READ (FILE => STR_FILE,
                      ITEM =>
MSG_STRING);

      end loop;
   end loop;

end LG_WRITE_OPEN;
with SSH_IO_SPRT;

procedure SM_WRITE is

   package IO        renames SSH_IO_SPRT;

   type INT_ARRAY_TYPE is array (1 ..
1500) of TINY_INTEGER;

   package MSG_IO  is new IO.TXIPC_IO
(INT_ARRAY_TYPE);

   STR_FILE        : IO.FILE_TYPE;

   MSG_STRING      : INT_ARRAY_TYPE :=
(others => 55);
   TIMER_ITER      : constant := 10;
   NUMBER_OF_BLKS : constant := 20;

begin

   MSG_IO.OPEN (FILE => STR_FILE,
                MODE => IO.OUT_FILE,
                NAME => "BENCHMARK_1");

   for TIMER in 1 .. TIMER_ITER loop
      for BLKS in 1 .. NUMBER_OF_BLKS
loop

         -- Write message block to IPC
         MSG_IO.WRITE (FILE => STR_FILE,
                       ITEM =>
```

```
MSG_STRING);

      end loop;
   end loop;

end SM_WRITE;
with SSH_IO_SPRT;

procedure SM_WRITE_OPEN is

   package IO        renames SSH_IO_SPRT;

   type INT_ARRAY_TYPE is array (1 ..
1500) of TINY_INTEGER;

   package MSG_IO  is new IO.TXIPC_IO
(INT_ARRAY_TYPE);

   STR_FILE        : IO.FILE_TYPE;
   TIMER_ITER      : constant := 10;
   NUMBER_OF_BLKS : constant := 20;


   MSG_STRING      : INT_ARRAY_TYPE;

begin

   MSG_IO.OPEN (FILE => STR_FILE,
                MODE => IO.IN_FILE,
                NAME => "BENCHMARK_1");

   for TIMER in 1 .. TIMER_ITER loop
      for BLKS in 1 .. NUMBER_OF_BLKS
loop

         -- Read message string from IPC
         MSG_IO.READ (FILE => STR_FILE,

                      ITEM =>
MSG_STRING);

      end loop;
   end loop;

end SM_WRITE_OPEN;
```

An Interprocess Interaction Problem. An earlier study in the project already identified that passing messages between UNIX processes in the selected platform is a high CPU demand function. The initial architecture calls for one process in the CPU to buffer up all inputs from the front end processors then pass along to other processes. This interprocess interaction is reevaluated as part of the performance enhancement effort.

Listing 2 collects the performance data of six Ada programs for the three different design approaches for the selected platform. Each Ada program is mapped into a separate UNIX process. The six programs are arranged into three pairs. Each pair simulates a different design.

a. LG_WRITE — Copy and Send large data blocks

b. LG_WRITE_OPEN — Read large block

c. LG_WRITE_NO_COPY — Send large block

d. LG_WRITE_NO_COPY_OPEN — Read

e. SM_WRITE — Send small block

f. SM_WRITE_OPEN — Read small block

a and b together simulate Buffer data in one CPU process. c and d simulate buffering up in front end processors. e and f simulate no additional buffering. Table 6 is the data for each process. Table 7 is the estimated performance of each different design. The original study overview indicates that at least one memory-to-memory copy is necessary for data buffering in the selected platform. Memory copy is very expansive in the selected platform.

Listing 2. Prototype Programs to Measure the Performance of Different Program Constructs for a Unique Ada Implementation

```
-- -- This iteration of the for loop
overhead test, uses a
      standard for loop
-- -- on a typical array of typical
records.  This loop
      initializes the array.
--
-- with SYSTEM;
--
-- procedure TESTMAIN_1 is
--
--    MAX_LENGTH : constant
SHORT_INTEGER := 1000;
--
--    type RECORD_TYPE is
--       record
--
       FIELD_1 : INTEGER;
--          FIELD_2 : STRING(1..4);
--          FIELD_3 : SYSTEM.ADDRESS;
--       end record;
--       for RECORD_TYPE'size use 96;
--
--    type ARRAY_TYPE is array (1 ..
MAX_LENGTH) of
      RECORD_TYPE;
--
--    SEARCH_ARRAY : ARRAY_TYPE;
-- begin
--    for N in 1 .. MAX_LENGTH loop
--       SEARCH_ARRAY(N).FIELD_1 := 1;
--       SEARCH_ARRAY(N).FIELD_2 :=
"1234";
--       SEARCH_ARRAY(N).FIELD_3 :=
MAX_LENGTH'ADDRESS;
--    end loop;
-- end TESTMAIN_1;

-- -- This iteration of the for loop
overhead test, uses a
      standard for loop
-- -- on a typical array of typical
records.  This loop
      initializes the array.
-- -- 1) This time initialize the array
element as a record and
        not fields.
--    MAX_LENGTH : constant
SHORT_INTEGER := 1000;
--
--    type RECORD_TYPE is
--       record
--          FIELD_1 : INTEGER;
--          FIELD_2 : STRING(1..4);
--          FIELD_3 : SYSTEM.ADDRESS;
--       end record;
--       for RECORD_TYPE'size use 96;
--
--    type ARRAY_TYPE is array (1 ..
MAX_LENGTH) of
      RECORD_TYPE;
--
--    SEARCH_ARRAY : ARRAY_TYPE;
-- begin
--    for N in 1 .. MAX_LENGTH loop
--       SEARCH_ARRAY(N) := (FIELD_1 =>
1,
--                           FIELD_2 =>
"1234",
--                           FIELD_3 =>
MAX_LENGTH'ADDRESS);
--    end loop;
-- end TESTMAIN_2;

-- This iteration of the for loop
overhead test, uses a
   standard for loop
-- on a typical array of typical
records.  This loop
   initializes the array.
-
- 1) This time initialize the array
element as a record and not
     fields.
-- 2) Change the for loop to a while
loop.

with SYSTEM;

procedure TESTMAIN_3 is

   MAX_LENGTH : constant SHORT_INTEGER
:= 1000;

--
-- with SYSTEM;
--
-- procedure TESTMAIN_2 is
--
```

```
   type RECORD_TYPE is
      record
         FIELD_1 : INTEGER;
         FIELD_2 : STRING(1..4);
         FIELD_3 : SYSTEM.ADDRESS;
      end record;
      for RECORD_TYPE'size use 96;

   type ARRAY_TYPE is array (1 ..
MAX_LENGTH) of RECORD_TYPE;

   SEARCH_ARRAY : ARRAY_TYPE;
   N : SHORT_INTEGER := 1;

begin
   while N <= MAX_LENGTH loop
      SEARCH_ARRAY(N) := (FIELD_1 => 1,
                          FIELD_2 =>
"1234",
                          FIELD_3 =>
MAX_LENGTH'ADDRESS);
      N := N + 1;
   end loop;
end TESTMAIN_3;

-- This iteration of the for loop
overhead test, uses a
   standard for loop
-- on a typical array of typical
records.  This loop
   initializes the array.
-- 1) This time initialize the array
element as a record and
      not fields.
-- 2) Change the for loop to a while
loop.
-- 3) Use an access type instead of an
index.

with SYSTEM;
with UNCHECKED_CONVERSION;

procedure TESTMAIN_4 is

   MAX_LENGTH : constant SHORT_INTEGER
:= 1000;

   type RECORD_TYPE is
      record
         FIELD_1 : INTEGER;
         FIELD_2 : STRING(1..4);
         FIELD_3 : SYSTEM.ADDRESS;
      end record;
      for RECORD_TYPE'size use 96;

   type ARRAY_TYPE is array (1 ..
MAX_LENGTH) of RECORD_TYPE;

   type RECORD_PTR_TYPE is access
RECORD_TYPE;

   function PTR_TO_SYS is new
UNCHECKED_CONVERSION
      (SOURCE => RECORD_PTR_TYPE, TARGET
=> SYSTEM.ADDRESS);

   function SYS_TO_PTR is new
UNCHECKED_CONVERSION
      (SOURCE => SYSTEM.ADDRESS, TARGET
=> RECORD_PTR_TYPE);

   SEARCH_ARRAY : ARRAY_TYPE;
   REC_PTR : RECORD_PTR_TYPE :=
      SYS_TO_PTR(SEARCH_ARRAY'ADDRESS);
   N : SHORT_INTEGER := 1;
begin
   while N <= MAX_LENGTH loop
      REC_PTR.all := (FIELD_1 => 1,
                      FIELD_2 => "1234",
                      FIELD_3 =>
MAX_LENGTH'ADDRESS);
      REC_PTR :=
SYS_TO_PTR(SYSTEM.INCR_ADDR

       (PTR_TO_SYS(REC_PTR),
RECORD_TYPE'SIZE/8));
      N := N + 1;
   end loop;
end TESTMAIN_4;

-- This iteration of the for loop
overhead test, uses a
   standard for loop
-- on a typical array of typical
records.  This loop
   initializes the array.
-- 1) This time initialize the array
element as a record and
      not fields.
-- 2) Change the for loop to a while
loop.
-- 3) Use an access type instead of an
index.
-- 4) Change while loop to for loop.
```

```
with SYSTEM;
with UNCHECKED_CONVERSION;

procedure TESTMAIN_5
 is

   MAX_LENGTH : constant SHORT_INTEGER
:= 1000;

   type RECORD_TYPE is
      record
         FIELD_1 : INTEGER;
         FIELD_2 : STRING(1..4);
         FIELD_3 : SYSTEM.ADDRESS;
      end record;
      for RECORD_TYPE'size use 96;

   type ARRAY_TYPE is array (1 ..
MAX_LENGTH) of RECORD_TYPE;

   type RECORD_PTR_TYPE is access
RECORD_TYPE;

   function PTR_TO_SYS is new
UNCHECKED_CONVERSION
      (SOURCE => RECORD_PTR_TYPE, TARGET
=> SYSTEM.ADDRESS);

   function SYS_TO_PTR is new
UNCHECKED_CONVERSION
      (SOURCE => SYSTEM.ADDRESS, TARGET
=> RECORD_PTR_TYPE);

   SEARCH_ARRAY : ARRAY_TYPE;
   REC_PTR : RECORD_PTR_TYPE :=
      SYS_TO_PTR(SEARCH_ARRAY'ADDRESS);
begin
   for N in 1 .. MAX_LENGTH loop
      REC_PTR.all := (FIELD_1 => 1,
                      FIELD_2 => "1234",
                      FIELD_3 =>
MAX_LENGTH'ADDRESS);
      REC_PTR :=
SYS_TO_PTR(SYSTEM.INCR_ADDR
                  (PTR_TO_SYS(REC_PTR),
RECORD_TYPE'SIZE/8));
   end loop;
end TESTMAIN_5;

-- This iteration of the for loop
overhead test, uses a
   standard for loop
-- on a typical array of typical
records.  This loop
   initializes the array.
-- 1) This time initialize the array
element as a record and
      not fields.
-- 2) Change the for loop to a while
loop.
-- 3) Use an access type instead of an
index.
-- 4) Change while loop to for loop.
-- 5) Change array structure to a linked
list.

with SYSTEM;
with UNCHECKED_CONVERSION;

procedure TESTMAIN_6 is

   MAX_LENGTH : constant SHORT_INTEGER
:= 1000;

   type RECORD_TYPE;
   type RECORD_PTR_TYPE is access
RECORD_TYPE;
   type RECORD_TYPE is
      record
         FIELD_1 : INTEGER;
         FIELD_2 : STRING(1..4);
         FIELD_3 : SYSTEM.ADDRESS;
         NEXT    : RECORD_PTR_TYPE;
      end record;
      for RECORD_TYPE'size use 128;

   function PTR_TO_SYS is new
UNCHECKED_CONVERSION
      (SOURCE => RECORD_PTR_TYPE, TARGET
=> SYSTEM.ADDRESS);

   function SYS_TO_PTR is new
UNCHECKED_CONVERSION
      (SOURCE => SYSTEM.ADDRESS, TARGET
=> RECORD_PTR_TYPE);

   SEARCH_ARRAY : RECORD_PTR_TYPE := new
RECORD_TYPE;
```

```
   REC_PTR      : RECORD_PTR_TYPE :=
SEARCH_ARRAY;
begin
   for N in 1 .. MAX_LENGTH loop
      REC_PTR.all := (FIELD_1 => 1,
                      FIELD_2 => "1234",
                      FIELD_3 =>
MAX_LENGTH'ADDRESS,
                      NEXT    => new
RECORD_TYPE);
      REC_PTR := REC_PTR.NEXT;
   end loop;
end TESTMAIN_6;

-- This iteration of the for loop
overhead test, uses a
   standard for loop
-- on a typical array of typical
records.  This loop
   initializes the array.
-- 1) This time initialize the array
element as a record and
      not fields.
-- 2) Change the for loop to a while
loop.
-- 3) Use an access type instead of an
index.
-- 4) Change while loop to for loop.
-- 5) Change array structure to a linked
list.
-- 6) Preallocate the space for the
linked list.  (Don't use
      'new').

with SYSTEM;
with UNCHECKED_CONVERSION;

procedure TESTMAIN_7 is

   MAX_LENGTH : constant SHORT_INTEGER
:= 1000;

   type RECORD_TYPE;
   type RECORD_PTR_TYPE is access
RECORD_TYPE;
   type RECORD_TYPE is
      record
         FIELD_1 : INTEGER;
         FIELD_2 : STRING(1..4);
         FIELD_3 : SYSTEM.ADDRESS;
         NEXT    : RECORD_PTR_TYPE;
      end record;
      for RECORD_TYPE'size use 128;

   type ARRAY_TYPE is array (1 ..
MAX_LENGTH) of RECORD_TYPE;

   function PTR_TO_SYS is new
UNCHECKED_CONVERSION
      (SOURCE => RECORD_PTR_TYPE, TARGET
=> SYSTEM.ADDRESS);

   function SYS_TO_PTR is new
UNCHECKED_CONVERSION
      (SOURCE => SYSTEM.ADDRESS, TARGET
=> RECORD_PTR_TYPE);


   SEARCH_
ARRAY : ARRAY_TYPE;
   REC_PTR : RECORD_PTR_TYPE :=
       SYS_TO_PTR(SEARCH_ARRAY'ADDRESS);
begin
   for N in 1 .. MAX_LENGTH loop
      REC_PTR.all := (FIELD_1 => 1,
                      FIELD_2 => "1234",
                      FIELD_3 =>
MAX_LENGTH'ADDRESS,
                      NEXT =>
SYS_TO_PTR(SYSTEM.INCR_ADDR

(PTR_TO_SYS(REC_PTR),RECORD_TYPE
                      'SIZE/8)));
      REC_PTR := REC_PTR.NEXT;
   end loop;
end TESTMAIN_7;
```

Table 6. CPU Usage for Interprocess Transfer

	Small Write	Small Write Open	Large Write	Large Write Open	Large Write No Copy	Large Write No Copy Open
User time	0.2166	3.760	2.04	3.7366	0.09	3.683
System time	1.3668	1.5666	1.2566	1.5798	1.25	1.57

Table 7. Comparative Advantage of Different Blocking Design

Design	User Time	System Time
Small block design : small write + small read =	3.98	2.93
Buffer in CPU: Large write + large read =	5.78	2.84
Buffer in front-end : Large write no copy + large read =	3.78	2.83

Findings and Conclusions

This methodology improves the performance of the application without introducing unnecessary changes. The following are some observations from this impromptu experiment.

1. The common sense spiral approach to enhance the performance of the software is in conflict with the rigid enforcement for MIL-STD-2167. Expensive delays result from this conflict with no tangible proof of quality improvements.
2. There is no disagreement on the interpretation of results of each performance enhancement cycle as in timing/sizing study using analytical method or simulation method.
3. This method only deals with a real problem instead of a perceived problem. One example is that the elimination of a particular disk write for recording produces no measurable improvements in the software application.
4. Availability of source code is essential for software performance design.

References

1. Mary E. Bender, Thomas E. Griest: "Real Time Ada Demonstration Project" pp. 159, Proceedings of the 7th Annual National Conference on Ada Technology.
2. Bruce A. Bergman: "What Lurks Behind RENAMEing" May 1989, Embedded Systems Programming.

Biography



Thomas L. C. Chen is a Member of the Technical Staff in the Software Systems Department, E-Systems, ECI Division. He has over 20 years experience in the development of communications software. He holds an M.E. from Taipei Institute of Technology.



Ralph P. Cooley is a Sales Systems Analyst with Sequent Computer Systems, Inc. Over the last 12 years he has been involved in all aspects of UNIX systems operations, programming, and instruction. Before joining Sequent, he served as an in-house UNIX consultant for Tolerant Systems at E-Systems. He received his B.S. Ocean Engineering in 1976 and an MBA in 1983.

AUTHORS' INDEX

Name	Page
Adamus, S.	69
Anderson, J.D.	597
Anderson, P.B.	358
Archer, T.	63, 640
Ardoin, C.D.	118
Auty, D.	
Bagley, K.S.	546
Bailey, J.	477
Baker, H.G.	633
Basili, V.	477
Belgrave, C.C.	19
Bender, A.	383
Bollinger, T.B.	436
Brown, I.L.	546
Buck, P.D.	195
Burnham, C.A.	592
Burns, A.	367
Butler, D.E.	546
Byrnes, C.M.	321
Carver, R.H.	419
Chappel, B.	525
Chen, T.L.C.	651
Chester, R.O.	424
Clough, A.J.	613
Conger, S.A.	311, 408
Cooley, R.	651
Couevas, Y.D.	255
Courte, J.E.	623
Crossland, K.	69
Daubenspeck, G.	63
Day, S.L.	195
Devi, M.U.,	251
Divine, D.	214
Dobbs, V.S.	187, 623
Easson, S.A.	640
Elrad, T.	236
Ernst, G.W.	98
Every, T.	241
Fee, S.J.	283
Fenton, T.D.	56
Ferrer, A.	36
Fitzgibbon, J.P.	84
Fowles, M.	214
Fox, J.A.	187
Fraser, M.D.	311,408
Fulbright, R.	299
Fussichen, K.	132
Gagliano, R.A.	311, 408
Goel, A.	147, 383
Gonzalez, D.	195
Guerrieri, E.	460

Name	Page
Guilfoyle, R.	537
Hankley, W.	392
Harous, S.	269
Heidma, J.H.	21
Henry, S.	525
Ho, P.	592
Hooper, J.W.	424
Hornung, C.J.	289
Hsieh, C.Y.	275
Jensen, L.	399
Joiner, J.K.	25
Joiner, H.F., II	448, 592
Kitaoka, B.J.	471
Korson, T.	12
Kumar, K.	311, 408
Lahti, L.	241
Lang, S.D.	643
Latour, L.	106
LeBaron, J.	147
Leach, R.J.	338, 546
Levin, D.	236
Levine, S.	597
Lewis, R.T.	643
Lin, F.	98, 269
Linn, J.L.	118
Lipscomb, A.	241
Longshore, W.	241
Madsen, R.	84
Marable, C.D., Jr.	19
Mayo, K.	525
McKay, C.W.	367
McLean, E.R.	311, 408
Meadow, C.	106
Mraz, R.Y.	307
Mulraz, D.E.	157
Muralidharan, S.	515
Nohl, D.	236
Owen, G.S.	311, 408
Palanisaml, S.	205
Papanicolaou, B.	533
Pappas, T.L.	500
Park, E.K.	358
Perkins, J.A.	597
Peters, J.	392
Peters, J.F.	226
Pfleeger, S.L.	436
Pitts, W.H.	129
Pope, L.	255
Radi, T.S.	399
Ramanna, S.	226
Randall, C.	367

Name	Page
Richardson, J.	36
Rogers, P.	367
Rothrock, J.	452
Schmidt, D.	76
Shoenecker, J.L., III	49
Smith, M.C.	157
Smith, T.Z.	157
Sodhi, S.	63
Spann, P.W.	546
Stautz, D.J.	349
Steinberg, R.	142
Sutherlan, S.	281
Tai, K.C.	419
Taylor, W.N.	419
Vaishnavi, V.K.	311
Vidale, R.F.	32, 613
Vitaletti, W.	460
Vogelsong, T.	452
Weide, B.W.	515
Weyrich, O.R., Jr.	262
Wheeler, T.	36
Wilder, W.L.	165
Woodward, H.P.	343
Workman, D.A.	643
Wu, Y.Y.	13
Yuhas, T.J.	613