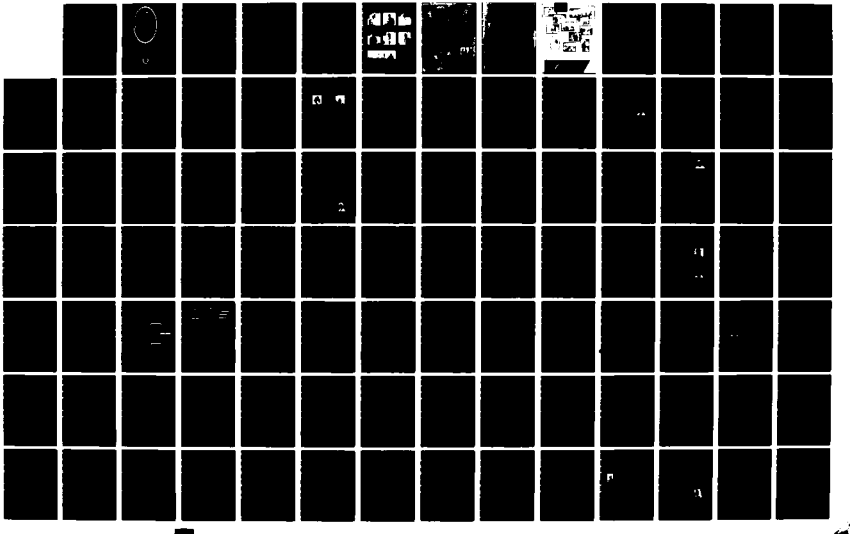
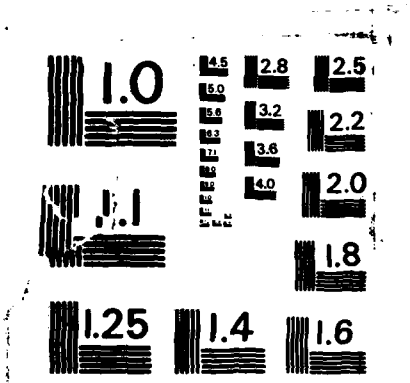
AD-A167 802 PROCEEDINGS OF THE ANNUAL NATIONAL CONFERENCE ON ADA (TRADE NAME) TECHNOL (U) ARMY COMMUNICATIONS-ELECTRONICS COMMAND FORT MONMOUTH NJ 1/2

UNCLASSIFIED 1986 F/G 9/2 NL





Proceedings of the 4th Annual National Conference on ®Ada Technology

March 19, 20, 1986

SPONSORED BY U.S. ARMY COMMUNICATIONS—ELECTRONICS COMMAND
FORT MONMOUTH, NEW JERSEY

Host College—Atlanta University, Atlanta, Georgia

PROCEEDINGS OF
4th ANNUAL NATIONAL CONFERENCE ON
Ada® TECHNOLOGY

Sponsored By
U.S. Army Communications-Electronics Command
Fort Monmouth, New Jersey

Host College
Atlanta University
Atlanta, Georgia

Hyatt Regency Atlanta
Atlanta, Georgia
March 19-20, 1986

Approved for Public Release: Distribution Unlimited

4th ANNUAL NATIONAL CONFERENCE ON Ada TECHNOLOGY

CONFERENCE COMMITTEE

Elmer F. Godwin, Director, GEF Associates (201) 741-8864
Melissa Herrera, Assistant, U.S. Army CECOM (201) 544-2112
Michael Danko, RCA, Morristown, NJ
Mary R. Ellis, Hampton Institute, Hampton, VA
Edward J. Gallagher, U.S. Army CECOM
Judy Giles, Intermetrics, Inc., Huntington Beach, CA
Charlene Hayden, GTE Communication Systems Div., Needham, MA
Arthur M. Jones, Morehouse College, Atlanta, GA
Benjamin Martin, Atlanta University, Atlanta, GA
Isabel Muennichow, TRW, Redondo Beach, CA
William M. Murray, General Dynamics, DSD, St. Louis, MO
Susan Richman, The Pennsylvania State University, Middletown, PA
John Roberts, The BDM Corp., Norfolk, VA
Susan Rosenberg, SOFTECH, Inc., Waltham, MA
Ruth Rudolph, Computer Science Corp., Moorestown, NJ
Jeff Simon, M/A COM Linkabit Division, San Diego, CA
Ken Taormina, Teledyne Brown, Tinton Falls, NJ
James Walker, Prairie View A&M University, Prairie View, TX
Paul Wolfgang, Boeing Vertol, Philadelphia, PA

TECHNICAL SESSIONS

Wednesday, March 19, 1986

9:00 AM	Session I	Panel Discussion—Commitment to Ada
2:30 PM	Session II	Ada Applications I
2:30 PM	Session III	Ada Research/Methodology

Thursday, March 20, 1986

9:00 AM	Session IV	Ada Impact on Secure Operating Systems Panel
9:00 AM	Session V	Ada Education and Training
2:00 PM	Session VI	Ada Applications II
2:00 PM	Session VII	Ada Program Support Environment and Development Tools

PAPERS

Proceedings

Bound—Available at Fort Monmouth

4th Annual National Conference on Ada Technology

1st-3rd copy—$20.00 each; 4th-10th copy—$15.00 each; 11 copy and above—$10.00 each

Make check or bank draft payable in U.S. dollars to the Annual National Conference on Ada Technology and forward request to:

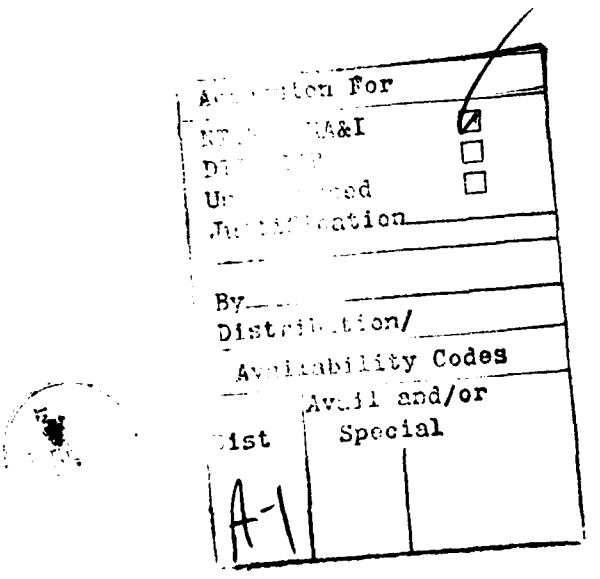
Annual National Conference on Ada Technology
U.S. Army Communications-Electronics Command
ATTN: AMSEL-COM-IE (Melissa Herrera)
Ft. Monmouth, New Jersey 07703

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

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

Include title, year, and AD Number:

2nd Annual Conference on Ada Technology 1984 -AD A142403.
3rd Annual Conference on Ada Technology 1985 -AD A164338.



Highlights of the
3rd Annual National Conference on Ada Technology
March 20-21, 1985
Hyatt Regency Houston
Houston, Texas

Greetings

Mr. James E. Schell, US Army, CECOM

Honorable Ernest McGowan, Councilman, Houston, Texas

Dr. Percy A. Pierre, President, Prairie View A&M University, Prairie View, Texas

Banquet Guest Speaker

Dr. Hugh Gloster, President, Morehouse College, Atlanta, GA

Honorable Pat Hillier, Assistant Secretary of the Army (Financial Management) Department of the Army, Washington, D.C.

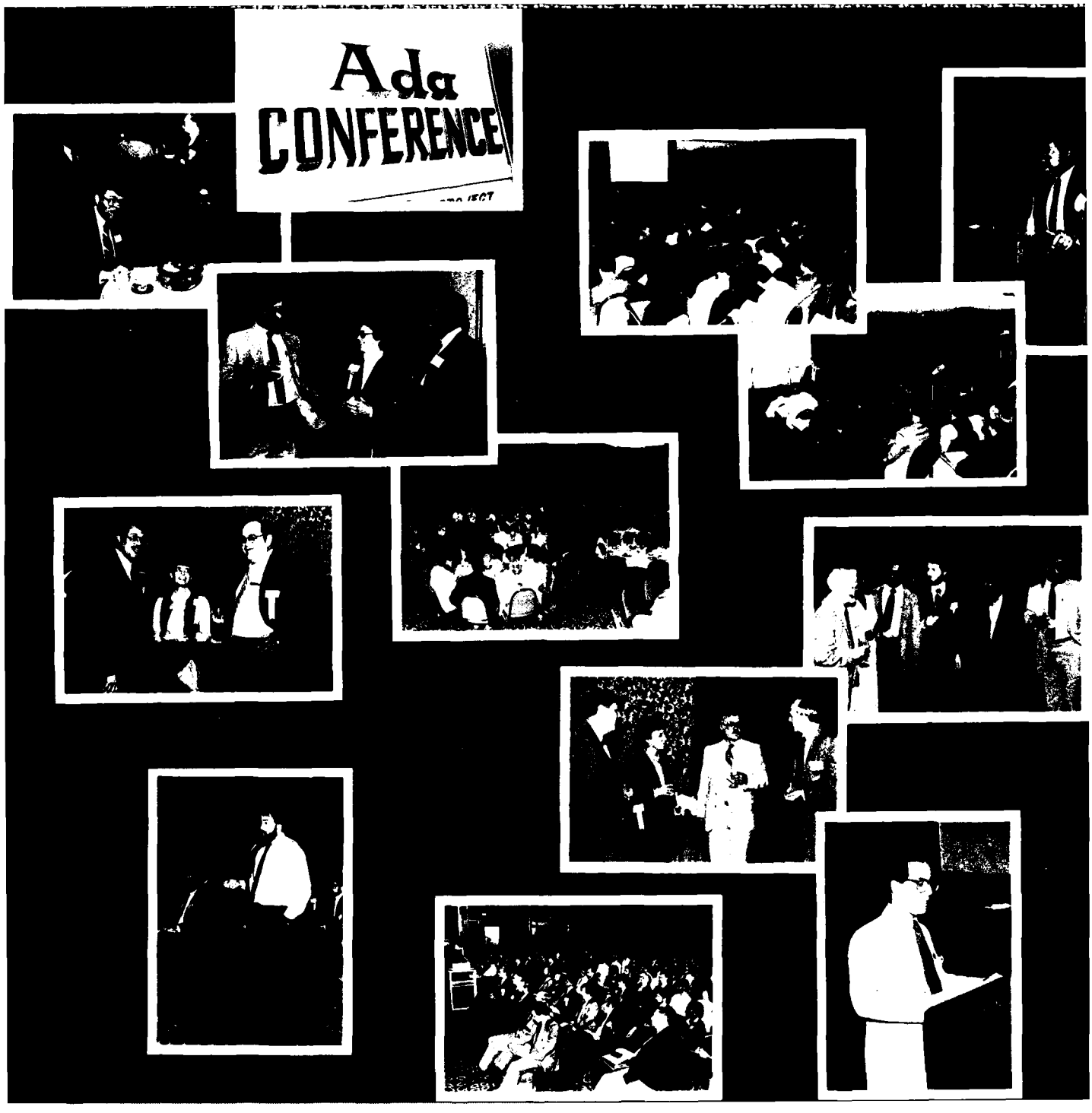
Luncheon Guest Speaker

Mr. Martin B. Zimmerman, Technical Advisor to the Assistant Chief of Staff for Information Management, Department of the Army, Washington, D.C.

Opening Session Panel Members

(Left to Right) Dr. Harland Mills, IBM, Bethesda, Maryland, Dr. Barry Boehm, TRW, Redondo Beach, CA, Dr. Nico Haberman (STARS, Software Engineering Institute) and Dr. Johannes Grande, Microelectric & Computer Technology Corp.


3rd Annual
1985 Ada
Conference


Ada
CONFERENCE

CONTRIBUTORS

AAI CORPORATION
Baltimore, Maryland

HONEYWELL, INCORPORATED
Minneapolis, Minnesota

INFORMATION SYSTEMS & NETWORK CORPORATION
Chevy Chase, Maryland

MAGNAVOX ELECTRONIC SYSTEMS COMPANY
Fort Wayne, Indiana

MERDAN GROUP, INC.
San Diego, California

MODCOMP
Ft. Lauderdale, Florida

RCA
Cherry Hill, New Jersey

ROCKWELL INTERNATIONAL CORPORATION
Dallas, Texas

SOFTECH-WALTHAM
Waltham, Massachusetts

TELOS
Santa Monica, California

VITRO CORPORATION
Silver Spring, Maryland

TABLE OF CONTENTS

WEDNESDAY, MARCH 19, 1986—9:00 AM-12:00 N

Falcon Room—Hyatt Regency

Greetings:
Mr. James E. Schell, Deputy Program Manager, ACCS US Army Communications-Electronics Command, Fort Monmouth, NJ
Dr. Luther Williams, President Atlanta University, Atlanta, GA

SESSION I: COMMITMENT TO ADA

Chairperson: Mr. James E. Schell, US Army Communications-Electronics Command, Fort Monmouth, NJ

Panel Members:
MG Alan B. Salisbury, USA, Commander, US Army Information Systems Engineering Command, Fort Belvoir, VA
BG Michael H. Alexander, USAF (Invited), Joint Program Manager, WWMCCS Information Systems, McLean, VA
Rear Admiral Harry S. Quast, USN, Director, Information Systems Division, Ch of Naval Operations, Washington, DC
MG David J. Ramsbotham, CBE, UK (Invited), General Officer, Commanding, 8th HQ, 3rd Armoured Division, Germany

WEDNESDAY, MARCH 19, 1986—2:30-5:30 PM

Falcon Room

SESSION II: ADA APPLICATIONS I

Chairperson: Paul Wolfgang, Computer Science Corp., Moorestown, NJ

An Ada* Tracker—Experiences and Lessons Learned—*T. Rodriguez* and *L. Griffin*, Ford Aerospace & Communications Corp., Newport Beach, CA 1

An Experimental Utilization of Ada in a Real-Time Interactive Avionics Communication Application—*W. S. Pepper IV*, Boeing Military Airplane Co., Wichita, KS 8

A Communications Project in Ada*—*P. J. Dousette*, The Singer Co., Librascope Div., Glendale, CA 13

The Army's MAFIS Command and Control—*M. T. Perkins* and *J. E. Bolger*, The BDM Corp., Austin, TX 22

An Ada* Tasking Application in an Air Defense System—*C. Ausnit*, SofTech, Inc. 28

Phoenix Room

SESSION III: ADA RESEARCH/METHODOLOGY

Chairperson: Susan Rosenberg, Softech, Inc., Waltham, MA

Floating Point Computation Using Ada's Model Numbers—*J. J. Buoni* and *R. L. Burden*, Youngstown State University, Youngstown, OH .. 38

Development of an Ada*Package Library—*B. Burton* and *M. Broido*, Intermetrics, Inc., Huntington Beach, CA 42

Experience with the Integration of Ada* Design Methods—*P. L. Baker*, Computer Technology Associates, Inc., McLean, VA 51

Applying the Spiral Model: Observations on Developing System Software in Ada—*F. C. Belz*, TRW, Redondo Beach, CA 57

Experience Collecting and Analyzing Automatable Software Quality Metrics for Ada*—*J. A. Perkins, D. M. Lease*, and *S. E. Keller*, Dynamics Research Corp., Wilmington, MA 67

The Technology Life Cycle and Ada—*M. A. Carrio, Jr.*, Teledyne Brown Engineering 75

THURSDAY, MARCH 20, 1986—9:00 AM-12:00 N

Lancaster Rooms ABC

SESSION IV: ADA IMPACT ON SECURE OPERATING SYSTEMS

Chairperson: M. Margaret Zuk, Mitre Corp., Bedford, MA 83

Panelists:
R. Platek, Odyssey Research Associates, Ithaca, NY 84
E. Anderson, TRW DSG, Redondo Beach, CA 85
W. E. Boebert, Honeywell, Inc., St. Anthony, MN .. 86
S. Hart, M/A-COM Linkabit, Inc., San Diego, CA ... 87

Lancaster Rooms D & E

SESSION V: ADA EDUCATION AND TRAINING

Chairperson: Art Jones, Atlanta University, Atlanta, GA

Development of a Corporate Ada Training Curriculum—*L. F. Blackmon*, General Dynamics, Ft. Worth, TX 88

Using Structured Techniques to Teach Real-Time Embedded Computer Applications—*R. S. Rudolph*, Computer Sciences Corp., Moorestown, NJ 96

The Implementation of a Graphics Package in Ada—*B. J. Martin*, Atlanta Univ., Atlanta, GA, *B. Setzer*, Kennesaw Col., Marietta, GA, and *R. Walker*, Atlanta Univ., Atlanta, GA 100

Experiences of Pascal Trained Students in an Introductory Ada Course—*R. C. Mers*, North Carolina Agricultural and Technical State University 104

The Development and Implementation of an Ada* Tutorial System—*J. E. Walker*, Prairie View A&M University, Prairie View, TX 109

The Use of Computer-Assisted Instructions in the Areas of Reinforcement and Testing for the L202 Module (Basic Ada Programming) of the US Army's Ada Training Curriculum—*P. Caverly, R. Canavan, P. Goldstein*, and *K. Pastuzyn*, Ada Technology Center, Jersey City State College, NJ 112

THURSDAY, MARCH 20, 1986—2:00-5:00 PM

Lancaster Rooms ABC

SESSION VI: ADA APPLICATIONS II

Chairperson: John Roberts, The BDM Corporation, Norfolk, VA

Implementation of an Ada* Real-Time Executive—A Case Study—*J. D. Laird*, *B. A. Burton*, and *M. R. Koppes*, Intermetrics, Inc., Huntington Beach, CA 114

Practical Experiences of the Ada Language for Real-Time Embedded Systems Development for the Defence-Related Market—*M. Selwood*, Plessey-UK Limited, England 125

Tactical Database Management System—An Ada Technology Project for the US Army—*J. Bamberger*, *P. Ritter*, and *J. Wilson*, TRW Defense Systems Group, Redondo Beach, CA 132

A Practical Approach for Translating FORTRAN Programs to Ada—*V. Santhanam*, Wichita State University, Wichita, KS 142

Lancaster Rooms D & E

SESSION VII: ADA PROGRAM SUPPORT ENVIRONMENT AND DEVELOPMENT TOOLS

Chairperson: Jeff Simon, M/A COM Linkabit Division, San Diego, CA

Verification of Diana Producers and Diana Consumers—*C. F. Schaefer*, Intermetrics, Inc. 149

The Back-End of a Multi-Target Compiler—*G. De Bartolo* and *R. Richards*, Intermetrics, Inc., Cambridge, MA 158

Automated Drawing of Data Structure Diagrams—*P. Mateti* and *G. M. Radack*, Case Western Reserve University, Cleveland, OH 165

Ada* and the PC, Its Time Has Come—*F. L. Moore*, Texas Instruments, Inc., McKinney, TX ... 173

AN ADA* TRACKER - EXPERIENCES AND LESSONS LEARNED

Terri Rodriguez and Lorraine Griffin

Ford Aerospace & Communications Corporation
Newport Beach, CA

Abstract

Although the Ada language was designed for use in embedded computer systems (ECS), a relatively small amount of work has been done with Ada in embedded, real-time environments. The goal of this project was to determine the amount of work and the types of problems that would be encountered using Ada for ECS. The redesign and coding in Ada of a small subset of a target tracker program that exists in 68000 assembly language and runs on a custom built, 68000 system was used as the medium for obtaining this information. From this project it was concluded that it is possible to use Ada for embedded computer systems, although the current lack of maturity in Ada tools and compilers for real-time ECS work discourages it for immediate use in large-scale ECS projects.

Background

Project Overview.

This project was a twelve man-month effort executed over eight calendar months' time in 1985. The project was divided into several overlapping stages:

- Learning/Training - 3 man-months
- Architectural/High Level Design - 2 man-months
- Detail Design and Ada Code - 3 man-months
- Integration and Debug - 4 man-months.

Tracker Description.

A target tracker can be compared to the combined working of the eyes and brain of a human as a moving object is followed from one point in the field of view to another (Figure 1). The eyes send image data to the brain which first recognizes and 'locks on' to the object. As the eyes continue to send image data to the brain, the brain performs 'calculations' and determines that the object is moving in a specific direction at a specific rate. For each set of image data received from the eyes, the brain sends information resulting from its 'calculations' back to the eyes to adjust the eye position so that the object can be kept within the field of view.

In a similar fashion a camera or other input source acts as the eyes of the tracker (Figure 2). The input source sends image data to the tracker hardware and software that together act as the brain. The hardware and software perform calculations on the image data to determine the target movement. Based upon the calculation results, position information is obtained and used to keep track of the target as it moves within the input source field of view.

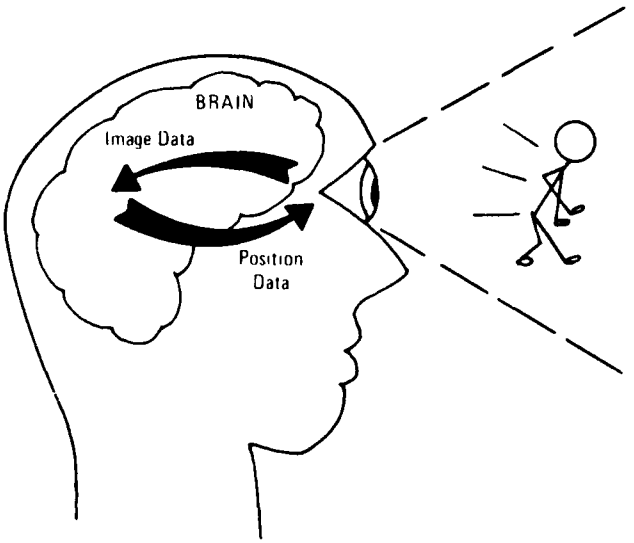


Figure 1. Human Tracker

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

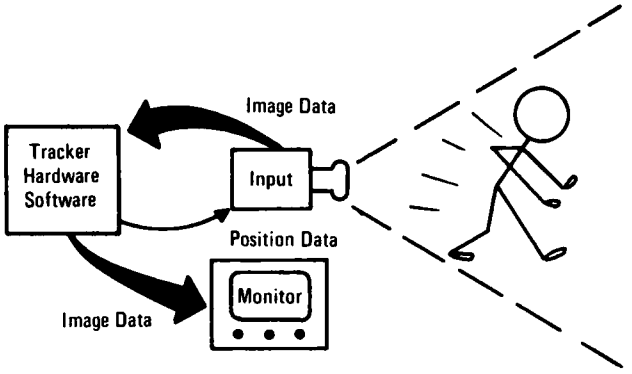


Figure 2. Machine Tracker

Hardware and Software Resources.

The following resources were available for this project:

- An Intellimac IN7000M (Motorola 68000 based) with the Telesoft subset Ada compiler (version 1.3d) and the Intellimac Embedded Systems Kit (ESK),
- A Data General MV10000 with the Ada Development Environment and validated Ada compiler,
- Two VAX 11-780s,
- The target 68000-based tracker hardware.

The tracker hardware for this project consisted of the following major components:

- Video camera
- Analog/Digital processor
- Motorola 68000 main processor with subordinate graphics processor and acquisition processor
- Video monitor

The video camera sends an analog stream to the Analog/Digital processor which converts the data to a digital format. This digital stream is sent to the acquisition and graphics processors of the tracker system. The acquisition processor receives the video data from the camera and extracts portions of the video data based on positional information from previous tracker calculations. These portions of data are stored in an area of the 68000 RAM called 'target memory'. The 68000 main processor directs the execution of the subordinate processors and performs the tracker calculations on image data in the target memory. It also sends position information to the acquisition and graphic processors. From the position information received from the 68000, the graphics processor generates a tracking gate and sends the graphics gate along with the image data received from the camera to the video monitor. The video monitor displays the image data and the graphics sent from the graphics processor. Figure 3 illustrates the tracker hardware system.

The Intellimac IN7000M was selected as the development computer since it was capable of generating s-record code for an embedded Motorola 68000 processor. The Data General MV10000 was selected for high level test and debug of algorithms because of its Ada tool set which included a source level debugger. The VAXs (VAX A, VAX B) were necessary resources for the transfer of generated 68000 s-records since the tracker hardware had no link to the Intellimac IN7000M.

Thus for the greater part of this project the development cycle began on the Intellimac, where Ada code was compiled and Motorola 68000 s-records were produced through the use of the ESK. Next, a direct link to VAX A was utilized to upload the s-records where a tape of the s-records was produced for transfer to VAX B. From VAX B the s-records were then downloaded to the tracker hardware for test and debug (Figure 4). Portions of the code that could be tested on the Data General were done so before being subjected to the development cycle described above.

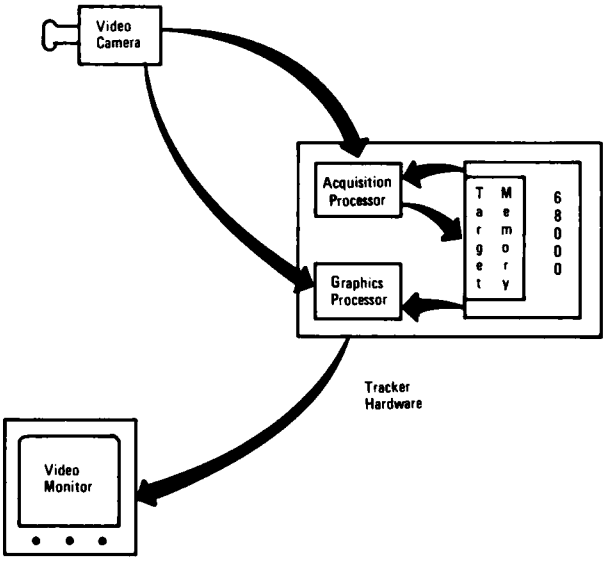


Figure 3. Tracker System

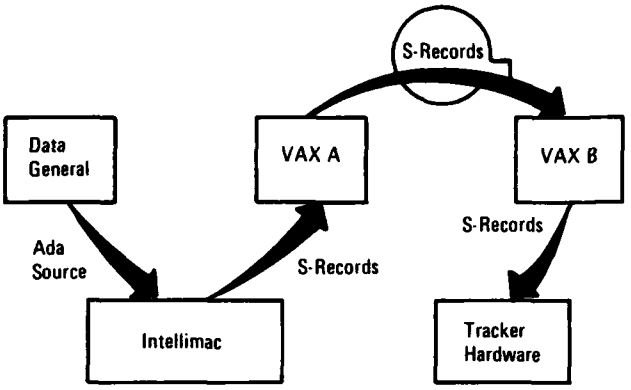


Figure 4. Transfer Process

This process was repeated for each test program or modification. However, during the integration and final debug portion of this project, direct access to VAX A from the tracker hardware was provided. This link eliminated the inefficient and time-consuming tape transfer of s-records between VAXs.

Human Resources.

At the start of the project two persons were assigned for part-time work. Both had a background in Ada and some familiarity with the ESK process. One had a little tracker experience and some 68000 familiarity. Neither had any knowledge of the target tracker hardware or software.

For the final portion of the project three more people were assigned for part- and full-time work. None had any previous Ada, 68000, or tracker experience. To become educated as quickly as possible about the project, these people took part in an 'Instant Ada Training' program. The program consisted of two one-hour sessions on Ada, and approximately three one-hour sessions on the Intellimac, ESK tools, tracker hardware, tracker design, and the debug process. They were also enrolled in a twelve-week, 36-hour Ada course, and assigned small ECS tasks to reinforce the material they learned. This unorthodox method of training was used for the following reason: the new personnel joined the project only six weeks before project completion and needed to be brought up to a contributory position as fast as possible. At this stage all the high level design was complete, as well as most of the detailed design. Since Ada was used as the design language for the detailed design most of the code was already complete.

For this project there were three 'consultants' available on a casual basis who provided initial instruction about the tracker hardware and software, provided information on the 68000, and provided hardware support and maintenance.

Goals

The main goal of this project was to create a real-time embedded computer system application that would demonstrate the capability of Ada for this type of work. The application selected for the Ada project had to satisfy the following requirements:

- Be applicable to the work of image processing;
- Demonstrate the feasibility of Ada for pre-existing specialized embedded system hardware;
- Be able to execute sufficiently well;
- Be completed in the 12 man-month time frame.

Thus the application of redesigning and recoding in Ada a small subset of an existing tracker was selected.

In reaching the primary goal, a secondary goal required answers for the following questions concerning Ada for embedded computer systems in general, and this project in particular:

- What problems were encountered using Ada?
- Can Ada be used for 100 percent of the application code?
- What are the performance degradations and their causes?
- What tools and environment are necessary for this type of ECS project?
- What types of personnel are required for this type of ECS project?

In addition, the project required documentation of all the project results and recommendations for an embedded systems development environment.

The Experiences

Experience 1 - A Prototype Mode.

Once it was determined to use a small-scale tracker as the application for this project, a prototype model of the tracker was quickly put together. The prototype model used all the mathematical algorithms that would be part of the final tracker, but since the prototype ran solely on a Data General MV10000, special modules for image input, target display, graphics, acquisition and de-acquisition were used. Also, the prototype did not simulate the interrupts the final tracker software would be receiving from the actual tracker hardware.

The purpose in building a prototype model was three-fold.

(1) First it acquainted project members with the tracker algorithms that would be used in the final tracker code. Implementing them for the prototype required an understanding of the tracker algorithms, how they interacted, and what the expected inputs and outputs were. It also revealed the scope of the tracker problem and what subset of the tracker would be feasible to implement for this project.

(2) Second it resulted in a rough design for the tracker and provided the opportunity to evaluate design decisions. Implementing the prototype required decisions on overall structure, interfaces, package contents, and subprogram breakdowns. The prototype demonstrated the pros and cons of these design decisions without the commitment of a final design.

(3) Third, it produced something tangible that worked. Although the prototype tracker was not the final product and did not run at real-time speed, the fact that Ada was tracking stirred interest in management levels, giving the project, and Ada, greater visibility.

Experience 2 - The Specifications Document.

After the successful completion of the prototype model came the task of detailing the tracker requirements. Since the goal of this project was to demonstrate the capability of Ada for embedded computer systems, and not to build a full-blown tracker, the specifications detailed only those algorithms necessary to perform centroid tracking.

The specifications document established requirements for selection of a target to track using the control panel, defined the algorithms that would be used for tracking and the portions of the tracker hardware that would be used, specified that tracking would be performed for a single target, defined the hardware interrupts that would be serviced, and defined the format of the graphics display.

Experience 3 - The Design And Pretests.

From the specifications document and basic information of the tracker hardware, the high level design was developed. Several methodologies were considered for the design - Process Abstraction Method, Object Oriented Design, and Functional Decomposition. Functional Decomposition was selected since the algorithms used for the tracker would execute in a serial manner, and the only tasking required was to handle the two hardware interrupts. After the initial design was completed, it was subjected to the walk-through and redesign cycle. The results of the high level design are illustrated in Figure 5.

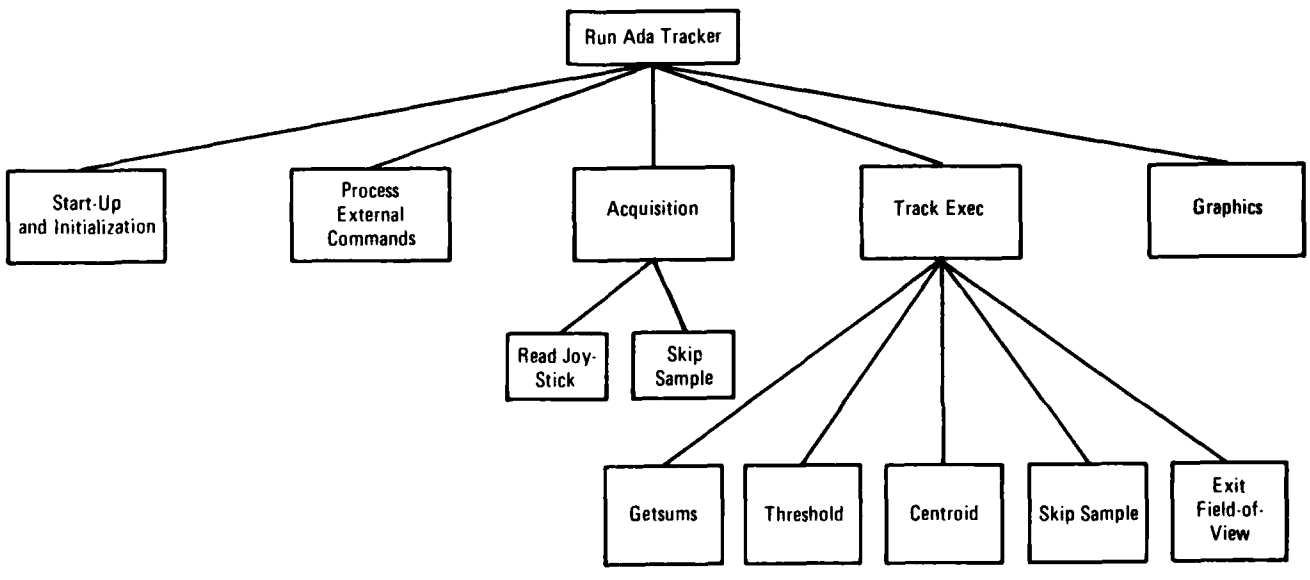


Figure 5. High Level Design

Upon approval of the high level design, work began on the detailed design. Ada was selected as the design language for this stage with liberal use of English text both to document the design and to describe intermediate implementation ideas and algorithms. At this stage it was realized that more information about the tracker hardware and the way in which the compiler would represent data structures on the hardware was needed. This information was extremely important to the detail design because there would be no hardware-software tradeoffs in design. Designers had to know the exact workings of the hardware and conform the design to it if necessary. To obtain this information a series of 'pretests' were devised. Each pretest was based on a specific design question or hardware question that needed to be solved in order for the Ada tracker to work. For instance, one pretest was designed to find the way in which arrays were mapped into target memory - row major or column major order. The acquisition processor would fill the target memory with image data in a row major fashion. This information was necessary to access the data in the same manner. Other information needed to complete the design concerned the method by which data structures were represented in portions of the memory space that were not fully addressable (i.e., the upper eight bits of the word were not accessible); how to handle the hardware interrupts using the Ada tasking mechanism; and what needed to be included in the run-time system that would be suitable for executing the Ada code. All problems were investigated individually until proper information and solutions were found. The pretests proved it was possible to implement all critical portions of the design in Ada and brought out hardware peculiarities previously undocumented. They also provided a way of testing ideas and concepts before the design was completed thereby eliminating the need for redesign during the integration and test phases. Additionally, the pretests unveiled the metamorphosis required to produce s-record format code from Ada source code. The metamorphosis required use of the Ada compiler to generate executable code followed by the use of the ESK to bind the code to specific locations in memory, to provide the necessary links into the target run-time kernel, and to generate the s-records.

Finally, the pretests allowed an early opportunity to tailor the run-time kernel to suit the tracker needs. The pretests determined which features would be needed in the run-time kernel to support the tracker execution and how to use the ESK to build the kernel. The Intellimac run-time kernel turned out to be a subset of the Intellimac ROS operating system. From that basic operating system floating point arithmetic and TEXT_IO were eliminated. Support packages for interrupt handlers were added.

The detailed design was completed based on results from the pretests.

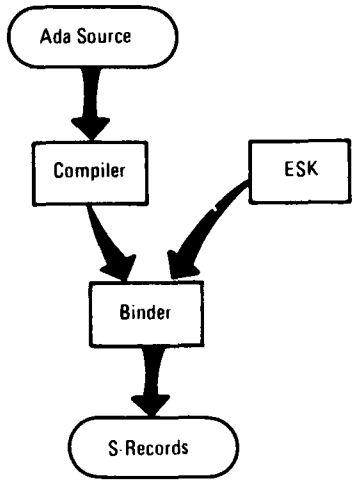


Figure 6. The Metamorphosis

Experience 4 - The Implementation Final Debug.

Upon completion of the detail design, the Ada design language was finalized as Ada code. Next a test plan for the tracker was developed. It was decided to start with small, easily tested portions of the code. New portions of code would be logically added to the old as it successfully passed all tests until the entire tracker was debugged. The actual debug method used in this phase was the same as that used for the pretests. For the debug process no interactive tools were available except for a monitor resident on the 68000 tracker. The monitor allowed the display and modification of memory, the display and modification of registers, and the setting of break-points.

The following process was used to debug the pretests and final tracker code:

(1) Perform as much source level testing as possible on the Data General. When testing is complete, move source code to the Intellimac.

(2) Place variables in the Ada source code to monitor the execution progress of the code. Tie these variables, as well as any other variables that will contain useful debug information, to hardware locations so that their contents can be examined using the monitor on the tracker. For the execution progress variables use values such as hexadecimal value 7777 or 3333 so that the location of these instructions can be easily found in the disassembled listing.

(3) Compile the code, generate the s-records and transfer the s-records to the tracker hardware.

(4) From the executable code produced by the compiler, generate a disassembled listing of the Ada source code. Get hardcopies of all source code, disassembled listings, and hardware maps.

(5) Run the program on the tracker hardware and hope it works.

(6) When the program does not work, use the Ada source map and the Ada source disassembled listing to determine appropriate places to halt execution to help pinpoint the problem area. Use an editor to search for opcodes in the s-record file to determine the approximate load address for the instruction. (The disassembled listing did not contain load address since it was generated from the compiler generated code file and not the final s-records.)

(7) When the problem is identified, correct the Ada source and repeat the above process. Call upon the consultants for ideas and help when unable to locate the cause of the problem.

Due to the extensive use of pretests, the debug of the entire tracker code was minimal. After integration of the final code, 90 percent of the tracker code worked correctly the first time it was executed. The areas in which problems occurred were areas which had not been subjected to a pretest checkout.

The Lessons Learned

It Can Be Done.

This project demonstrated that it is possible to use Ada for small embedded system type projects. With meager resources it was possible to complete a practical application using Ada for 98 percent of the code. Only the lack of maturity in the tools and compilers, and the relatively small amount of experience in using Ada for ECS restrict its immediate use on large, time-critical projects.

Qualified Human Resources Needed.

It is possible to complete a small scale ECS project with little knowledge of Ada and the target hardware. However, for a large ECS project it is imperative that there are team members with the following qualifications:

(1) Detailed knowledge of the target system and the ability to express the way a hardware configuration will impact the software design;

(2) Detailed knowledge of Ada and the code that will be generated by the Ada compiler. This knowledge may impact the detailed design and implementation phase if the code generated by the compiler for various Ada constructs is space inefficient or time-consuming;

(3) Detailed knowledge of the tool set used for the ECS project. This knowledge is necessary for creating a good design of the ECS program, for implementation of the design, and for efficient test and debug of the final code.

Good Tools Needed.

For a successful embedded systems project a good tool set and a good ECS development environment are needed. Several characteristics of an ECS environment and tool set follow:

(1) The environment should be easy to work in and provide ready access to all ECS tools. It should also provide easy access to the target machine for quick download of code.

(2) The minimum tools included in the ECS environment should be the Ada compiler, a source level debugger, a target level debugger, and a disassembler.

(3) The compiler should generate efficient executable code and should support many features from Chapter 13 of the Ada language reference manual.

(4) There should be a complete set of hardware and software documentation for the target system and a complete set of documentation on the ECS tools including information on the size and characteristics of the run-time kernel, the amount of code generated for each Ada construct, and the execution speed of the Ada constructs.

Finally, the target hardware should be available for use during the design phase so that early testing of design concepts can take place.

Tying It All Together - Words Of Wisdom.

- Large systems are an outgrowth of experiences from small systems.
- When in doubt try it out as a small test program on the hardware.
- All compilers are not created equal. Before purchasing a compiler for ECS work, thoroughly evaluate the quality of the code it generates, its ease of use, its execution speed and required resources.
- Know the quality and characteristics of the run-time kernel to be used for the target system.
- Before embarking on contract work, obtain experience via a small ECS project such as this and learn the risks and problems in a non-critical environment.

Lorraine Griffin, Engineering Specialist, has been at Ford Aerospace since 1977. She is the project manager of the Ada Technology Special Project and a Corporate Ada project. She has taught several semesters of in-plant, University of California at Irvine Ada extension courses and is the chairperson of the AdaJUG Education committee. She has a BS degree in Mathematics from Youngstown State University.

Terri Rodriguez, Engineer, joined Ford Aerospace in 1983. In addition to being Responsible Engineer for the Ada Technology Special Project, Ms. Rodriguez has written several reusable Ada packages including a fixed-point math library, and guidelines for using Ada in embedded systems applications. Ms. Rodriguez holds a BS degree in Information and Computer Science from the University of California, Irvine.

Authors' Address

Ford Aerospace & Communications Corporation
Aeronutronic Division
Newport Beach, California 92658-9983

AN EXPERIMENTAL UTILIZATION OF ADA IN A REAL-TIME INTERACTIVE AVIONICS COMMUNICATION APPLICATION

WILLIAM S. PEPPER IV

BOEING MILITARY AIRPLANE COMPANY
WICHITA, KANSAS

ABSTRACT

The application of high-order language (HOL) to real-time avionics applications has been fraught with assembly-language subroutines and sundry workarounds to increase throughput. In order to ensure that Ada would in fact alleviate these concerns, work was undertaken to develop an interface between a BMAC advanced technology avionics processor and a touch-sensitive Integrated Control Display Unit via ARINC-429 and MIL-Standard 1553B protocol buses utilizing one of the available Ada compilers. These two protocols are the most widely utilized communication protocols for general aviation and military avionics applications.

INTRODUCTION

The avionics systems of existing aircraft continue to be upgraded for various reasons. The replacement of systems for purely logistic (maintainability) reasons and the addition of new equipment for new mission requirements has caused undesirable complexity as well as an additional work load for aircrew members. Additionally, the customization of cockpit layouts at the squadron level in an attempt to alleviate the problem has caused additional divergence and training complexity. As a result, the Boeing Military Airplane Company has established an on-going Aircraft Cockpit Technology IR&D project. In order to meet the technological challenges, BMAC developed a Bus Interface Computer Unit (BICU) to upgrade aircraft by interfacing older existing systems to newer, more maintainable systems. Part of the effort was to program the BICU in Ada and use it in a laboratory cockpit scenario.

OBJECTIVES

The primary objective of the software portion of the research was to develop Ada applications which would utilize RS-232C, ARINC-429 and Mil-Standard 1553B protocols to establish a communication link between the central avionics processor of the BICU and a touch-sensitive Integrated Control Display Unit (ICDU) and other test devices. The goals established for the work were:

1. To determine if all device drivers could be written in Ada,
2. To test the efficacy of the Ada task type and,
3. To determine if code execution times were adequate to support a real-time avionics environment.

HARDWARE ENVIRONMENT

In order to conduct the research, BMAC developed an Ada Integ Laboratory representative of an aircraft cockpit avionics installation. The Bus Interface Computer Unit (BICU) prototype with ARINC 429, MIL-Standard 1553B and RS232C interface cards was the core of the hardware configuration. A serial bus analyzer was used to provide 1553B inputs and a test aid was connected to the RS232C channel. A pre-production touch-sensitive Interactive Control Display unit was connected to the ARINC 429 bus to provide a simulated aircrew interface. The Ada source code was developed on a host computer and then compiled and downloaded to a commercially-available integration unit. This unit was used as an emulator to run the object code and also served as a test and debugging tool.

GENERAL SOFTWARE DESCRIPTION

In order to provide a realistic scenario, demonstration programs were developed which would simulate flight activities in a tanker/airlift scenario. Programs were developed to insert waypoint data, fly-to the waypoints and "delivery" cargo via the airborne extraction method. An engine monitoring demonstration was also developed which included emergency decision making and crew alerting functions. The software exercised the ARINC-429 bus when generating displays and when controlling the touch screen on the ICDU. The BICU acted as the 1553B bus controller when it statused the Engine Monitor System simulated by the serial bus analyzer.

Six basic packages were used to implement the demonstration program. A brief description of each package follows:

- The ARINCIO package served as the ARINC-429 I/O driver. It provided two user interface programs which translated data on the ARINC-429 bus. To transmit SEND DATA formatted command strings into ARINC-429 protocol. To receive GET DATA interpreted the ARINC-429 protocol and returned a buffer with data sent over the bus.

- The ICDUIO package contained nine user interface programs to control the ICDU. Each of these programs called one of two ARINCIO interface programs to communicate to the ICDU via the ARINC-429 bus.

- The B155310 package drove the 1553B I/O card. It had two user interface programs which translated data on the 1553B bus. CHAIN BUILD assembled I/O chains for application programs. CHAIN EXECUTE issued I/O chains when the bus controller was available.

- EMSIO contained two user interface programs to control the simulated Engine Monitor System (EMS) via the 1553B bus. STATUS-EMS was used to regularly status engine data. EMS-WARN notified the application programs of engine warnings and directives received from the EMS. Both EMSIO interface programs employed the B155310 package to communicate with the serial bus analyzer.

- The MSNCTRL package was a non-executable data base accessed only by the Demonstration program. It contained data records for simulated missions and destination records for those missions.

- The DISPLAYDB package was another non-executable data base accessed only by the Demonstration program. It contained all the preformatted display data used in the demonstration. Most display data took the form of a record containing an array of 21 characters. Other records had arrays of only six characters which made manipulation of six or less characters more efficient. To conserve processing time, some of the larger records were grouped into arrays of records and accessed by enumeration types. DISPLAYDB also contained a decoding array used in determining what to do when the screen was touched.

The demonstration was controlled by a program of the same name (DEMONSTRATION). This program displayed the master menu and processed control requests as a result of menu selection, waypoint sequencing and engine warnings and directives. Menu selection transferred control to a secondary group of subprograms which control subordinate displays. The control hierarchy was designed as a tree structure of controlling subprograms in order to provide reusability of code.

Data was passed between subprograms as global data which resided at the top of DEMONSTRATION. To conserve memory and processing time, local data was not allocated to any subprogram. This data was also defined globally. DEMONSTRATION used overloaded operators with three functions ("+", "-", and "*"). These new functions allowed easy arithmetic on arguments of the form "minutes: seconds" for use in timer and navigation-oriented applications. Several utility subprograms provided routine manipulation of display data. Other utility subprograms were created from groups statements that repeated elsewhere only to conserve memory.

Each menu control subprogram cyclically called COMMONCALL. This subprogram was used to update dynamic mission data and request engine status. It also displayed engine warnings and directives on the title line. Regular checks for touch screen inputs were done in each menu control subprogram by calling GET-INPUT which was resident in ICDUIO. Regular updates of mission and engine data as well as regular sampling of input controls created the illusion of an ongoing mission.

Three types of displays were generated which included menus, profiles, and performance information. Displays could be selected at any time and performance information appeared automatically. The demonstration operator used menus to control missions and to make subordinate menu selections. Mission profile selection, mission start-up/abort, flight status and engine monitoring were menu-controlled. Profiles displayed mission starting location, waypoint locations and waypoint types. Performance information information displays showed mission progress and engine information. Flight status and cargo drop information displays were updated with fresh data as the mission progressed. Engine data could be interactively requested and problems or potential emergency situations were reported automatically.

GENERAL IMPLEMENTATION NOTES

The implementation problems inherent in the use of unproven hardware and an unproven Ada compiler proved to be a challenging and sometimes frustrating exercise. Several design problems were encountered, all of which were solved. Some were solved by software workarounds while some were solved by changes to the hardware. Although many of the solutions to the problems that were encountered are of a proprietary nature, the following design issues, problems, results and conclusions may be readily described.

Certain aspects of the Ada language made development on the executive and bus driver level more difficult than it might have been in another language. Although workarounds for the problems that were encountered were possible, they sometimes

proved costly in time and space utilization. As a result, it is of the utmost importance to note the potential effects of language limitations prior to coding because of their impact on program design.

The strong typing that Ada imposes proved to be a help in program correctness primarily by cutting debug time. In some cases, however, the inflexibility of this feature resulted in clumsy or wasteful constructs. For instance, when passing data to the bus driver for transmission to another device, the data types specified by the caller and the receiver must agree. In order to transmit data of several types, the caller must buffer data after performing unchecked type conversions or the bus driver must have several different entry points for different data types and perform the type conversion itself. The result is a choice between an inelegant coding practice or a seemingly unnecessary use of additional memory.

A very important capability at the executive level for efficient program design is the use of pointers to users' data structures. Ada provides a pointer with each access type which can only point to objects of the type that they are declared to point to. Ada further limits the access type by restricting it to "only designate an object created by an allocator", as described in the Ada Language Reference Manual (LRM), MIL-STD-1815A. The LRM continues "in particular, it cannot designate an object declared by an object declaration". In other words, access values cannot point to data structures that are created at compile time, but only to those created at run time (even though both could conceivably be of the same type). In an embedded avionics system, the creation of a large number of objects at run time is not desirable. The inability to point to a user's objects by an executive utility makes the design of that utility less efficient than it could be if that ability were provided.

When slices of equal lenghts are taken of arrays which have the same component type but are of a different length, a compiler error is raised. For example,

```
ARRAY1: array(1..10) of INTEGER:

ARRAY2: array(1..20) of INTEGER;
```

The following is an illegal statement:

```
ARRAY1(1..3) : = ARRAY2(1..3);
```

This restriction forces the use of separate assignment loop to accomplish the intent.

The priority pragma was not implemented in the Ada compiler used for this research. In a system consisting of two tasks and a main procedure, the last task activated gains control of the CPU first, followed by the first task, followed by the main procedure. The only way the latter two aren't starved is if the task that has control of the CPU relinquishes it in some predetermined way. For example, the execution of a "PUT" statement will force the task to release CPU control. In this particular compiler implementation this was sufficient to swap a waiting task into execution. Empirical analysis determined that a task context switch (resulting from a rendezvous for instance) required approximately 3.2 milliseconds.

Timing tests were performed to determine the most efficient method in terms of time to pass parameters using this particular compiler. Four parameters were passed. These consisted of two integers, one enumeration type object, and a record consisting of an array of 21 integers and another integer. The following three methods were considered:

1. All four were passed as paramters in the subroutine call. The time was 105 micro-seconds.

2. All but the record were passed as parameters, and the record was in the global database. The timing included the assignment statement to set up the record. The time was 160 micro-seconds.

3. All four parameters were in the global database. The timing included the assignment statements to set up the parameters. The time was 290 micro-seconds.

The obvious lesson learned as a result of these tests was that it did no good to try to circumvent the compiler in this instance. The case in which the compiler controlled the parameter passing was significantly more efficient than the other two cases.

SCHEDULING (EXECUTIVE)

It is generally accepted theory that when a significant level of control over the executive is possible, a tasking environment usually provides a greater level of flexibility in a complex system than other approaches. Flexibility is lost, however, if the executive cannot be molded into something that conforms to the needs of the applications it serves. In an embedded real time avionics system, reliable and quick response to events is paramount. Therefore, when designing an executive for avionics applications, executive overhead time is usually the most significant issue. With these points in mind, an attempt was made to design a viable executive for use on the BICU prototype using only the Ada language.

Several approaches to the design of the executive were considered in the preliminary design phase. The use of Ada tasking for job control was the ultimate goal, but the 3.2 millisecond task context switch time inherent with this compiler made such an approach unreasonable. If task context switch occurred only two times in the equivalent of a minor frame in a 64 Hz system, 41 per cent of the CPU time would have been consumed in executive overhead for tasking alone. Obviously, the use of tasking had to be avoided.

A second possible approach was the use of a tasking system that ran "on top of" the Ada tasking. In this approach there was one main Ada task and the tasking system was run within its context. This required two things that were either undesirable or impossible at this point:

1. Tasking primitives had to be written in assembly language because Ada doesn't allow the level of machine control necessary to perform this function and

2. A good working knowledge of the compiler's run-time support inner workings and the parameter passign protocol was necessary. This information was proprietary to the compiler vendor. (This version of the compiler didn't provide assembly code listings for the target computer).

The approach pursued was to use a single Ada task for all applications programs with a 16 Hz timer providing the synchronization for cyclic processing. All interrupts were handled using the compiler vendor's "fast interrupt" support to avoid task context switch delays. Studies on time consumed by the I/O drivers were used to determine the optimum scheduling and I/O support that could be provided by this system. This system was overflow tolerant to the extent that a particular minor frame's processing didn't overflow past the following minor frame's time allocation. A limit was also set on the number of consecutive minor frame overflows since a large number of overflows is indicative of an unhealthy system.

ARINC 429 COMMUNICATIONS

In the Ada Integration Lab, the only inputs via the ARINC-429 bus were from the ICDU. These inputs represented the status of the ICDU touch screen and consisted of six 32-bit words. The ICDU sent the screen status to the host every 12-20 milliseconds. The first implementation of the ARINC driver failed since it could not handle all the interrupts that it was receiving. This failure was due to two facts:

1. The interrupt handler was written to require a full Ada rendezvous on each interrupt which required a minimum of 3.2 milliseconds, and

2. The ARINC-429 driver was bombarded with unsolicited screen status interrupts. These interrupts were not event triggered (i.e. the screen wasn't being touched.)

Two steps were taken to alleviate this problem.

1. A latch was added in the hardware so that interrupts from the ARINC-429 could be masked on and off. Then when it was desired to receive interrupts (inputs) from the ICDU the interrupt latch was set appropriately.

2. The ARINC-429 interrupt handler was changed to use the compiler vendor's "fast interrupt" capability. Using fast interrupts meant that a procedure call took place when an interrupt occurred instead of a rendezvous.

These changes resulted in an interrupt handler that could accomodate the workload imposed by the ARINC-429 ICDU. A read of the ARINC-429 channel was performed as follows:

- When a read request was received, the ICDU interrupt latch was set to permit the processor to be interrupted by the ICDU.

- When an interrupt occurred, the data from the pertinent register was read.

- After all six words of the status transmission were received, the ICDU interrupt latch was set to prohibit the ICDU from interrupting the processor.

The results of the read were then analyzed by the program in order to determine which area of the screen had been touched.

MIL-STD 1553B COMMUNICATIONS

The original design of the 1553B bus driver incorporated features of Ada tasking to support user interface and physical driver control. The 1553B driver package was comprised of three separate tasks and one procedure. The tasks handled physical driver control, enqueuing of user requests for the physical driver task (a monitor task), and general housekeeping which included the notification of the user after receipt of the I/O complete interrupt. The procedure was available for users needing I/O chains to be built for them. A request for I/O execution was made via an entry call from either a user with a chain to be executed or from the chain building procedure after it had constructed an I/O chain to the user's specifications. In this way, requests could be made when the physical driver task wasn't in a position to handle them but its monitor task was. The monitor task would enqueue them (actually link them into a pending chain) for execution after the presently executing chain had completed.

When it was determined that the use of tasking as implemented in this particular compiler should be minimized to control execution in a cyclic environment, the 1553B driver package was revised to use only procedure calls (with the exception of the interrupt handling). The monitor task was turned into a procedure serving the same function as before. The execution task (physical driver) was also changed to a procedure, requiring that some cyclic scheduling of this procedure occur to accommodate chained up requests.

One of the major challenges of the 1553B driver design was linking together consecutive requests to the monitor task. In order to vector I/O controller execution through chains from two separate requests without CPU intervention the chains had to be linked together. A user of the driver had to allocate a link field at the end of the chain so that the request handler could insert either a link instruction (to the next user's chain) or a halt instruction. A difficulty arose in trying to retain reference to the end of a previous requestor's chain when handling the next request. Since Ada pointer (access type) can only designate objects created by an allocator (run time), either all chains had to end in such an object or an alternative approach to pointing to the end of the chain had to be devised.

The original solution to this dilemma was the creation of an array of "chain link cells". When a chain request was made to the monitor, a chain link instruction pointing to one of the chain link cells was inserted at the end of the user's chain. The index of the chain link cell pointed to by the end of the last chain was retained in the package data base so that when a subsequent request was made a chain link instruction pointing to the beginning of the next chain could be inserted into that link cell. This required a small amount of memory management to track available link cells and was generally a cumbersome design.

The method for linking consecutive requests that was subsequently employed used the Ada *unchecked* conversion function. Since objects of the access type and the address type (from package SYSTEM) were of the same length and both were actual memory addresses on the processor, an unchecked conversion between the two types was acceptable. This made it possible to designate an object declared by an object declaration (at compile time). Although this method was successful, the use of unchecked conversion in general is discouraged because it can potentially limit the portability of an Ada program. Its use in this circumstance, however, was acceptable because it was limited to the hardware implementation dependent portion of the driver code.

GENERAL RESULTS

Several other implementation problems were investigated and solved over the course of the experiment. Techniques were learned and methodologies developed to minimize the use of memory and increase throughput. Several of the problems were due to the immaturity and incompleteness of the Ada compiler that was used. In the final analysis, however, the experiment was successful in that all coding was performed in Ada and valuable implementation lessons were learned. In regard to the original goals for the work, the primary conclusions to date are:

1. The subject device drivers may be written entirely in Ada.
2. The code execution times warrant utilization of Ada in an embedded real-time application in the case of the aircrew/aircraft interface.
3. Conclusive proprietary data regarding the use of the Ada task type is now available.

This research and development project has advanced Ada technology in the real-time avionics arena in that concrete benchmarks have been established for the development of future applications. The project has proven invaluable in the training of software personnel and in the development of hardware to support design, testing, and implementation of Ada-based avionics applications. System definition practices in an Ada environment have been developed which will be utilized to great advantage in future programs.

ACKNOWLEDGEMENTS

The author wishes to thank Donald W. Higgins, Edwin D. Jones and James E. Kroening, all of BMAC, without whose contributions this work would not have been successful.

AUTHOR

Mr. Pepper is employed at Boeing Military Airplane Company as an Avionics Engineering Systems Analyst. He has held positions in real-time software development on tactical and strategic weapons systems. He performed undergraduate work at SUNY-Cortland, the University of Arizona and Wichita State University and graduate work at Kansas State University and Wichita State University.

Mr. William S. Pepper IV
Boeing Military Airplane Company
P.O. Box 7730, M/S K31-26
Wichita, KS 67277-7730

A COMMUNICATIONS PROJECT IN ADA *

Author: Patricia J. Dousette

The Singer Company, Librascope Division • 833 Sonora Ave., Glendale, CA 91201-0279

Abstract: The Communications Control System (CCS) is a front-end communications processor. It was one of the first mission critical systems to be implemented using the DOD developed language Ada. The CCS software development proved Ada technology to be practical for the following reasons:

- A large scale embedded software project, in excess of 45,000 lines of Ada is viable.
- A significant increase in programmer productivity was observed using Ada.
- The project pinpointed problems with existing Ada support tools and methodologies and emphasized the benefits to be gained by development of these tools.
- The use of Ada based Program Design Language as a design tool is both feasible and desirable.

In addition, information concerning the following areas of interest was obtained:

- Ada programmer training issues were analyzed.
- Ada Run-Time System problems i.e. speed of execution and code generator efficiency vs. Ada implementation were highlighted.

The following paragraphs will present the CCS project history i.e. how and why Ada was chosen and the problems and successes that were realized because of its use.

1.0 INTRODUCTION

In May of 1982 Singer-Librascope was awarded an Army contract (from Program Manager, Field Artillery Tactical Data Systems (FATDS)) for the Communications Control System (CCS). In the short term the CCS was to be the front-end communications processor for the Field Artillery's TACFIRE system and in the long term it was to be the communications front-end for the forthcoming Advanced Field Artillery Tactical Data System (AFATDS) Program and a candidate system for the Army Command and Control System (ACCS). The CCS was required to be able to interoperate with all Army Communications equipment both existing and future — up to the year 2010. The Contract was for an advanced development model of the system, both hardware and software. The software had been proposed to be written in PASCAL for the Motorola MC68000 processor, with any time critical software (the CCS is a real-time embedded system) written in Assembly language. Even though Ada has been specified in the CCS Preliminary Specification and contract, it had not been proposed because of the lack of an Ada compiler. However, shortly after the contract award information about the Ada language and its first implementation became available. A first generation compiler, specifically the Telesoft-Ada compiler had been released and was being hosted on the Intellimac 7000 series of computers. The decision was made, having been approved by the Program Office, to write the CCS software in Telesoft-Ada—except for time constrained software which would still be written in Assembly language.

2.0 THE CHOICE OF ADA

The CCS software development was approached with a great deal of confidence. The project functionally was very similar to previously implemented communications projects. However, had we been able to predict some of the problems we would encounter with a brand new host software development system we might have opted for a more standard approach. Admittedly, only a portion of the project problems can be attributed to the use of Ada and looking back now we would certainly not change our decision to use Ada.

The programming department was used to writing software for embedded systems in Assembly language. However, predicated on the magnitude of the CCS project we knew that a High Order Language (HOL) would be necessary. We had considerable experience in FORTRAN, more in CMS-2, and a little in PASCAL. For the MC68000, the CCS target processor, our choice was limited to PASCAL, Assembly language and the just released Telesoft-Ada. Our experience with PASCAL had not been particularly good. We found out subsequently that this was completely due to a poor quality compiler. Pure Assembly language was considered but was thought to be an extreme schedule risk—so the infant Ada was agreed upon. It was chosen after some feasibility studies, a trip to Telesoft (Singer and Government) and consultation with the customer. It appeared that Telesoft-Ada, even though it was a subset of Mil-Std-1815A Ada, supported the language features necessary for the CCS target hardware architecture. The MC68000 target Run-Time System was promised for delivery within a few months. The Intellimac 7000 series of computers was available and already hosting the new compiler. At this time however, only simple programs had actually been implemented in Telesoft-Ada.

2.1 ADA TRAINING

We began the training of programmers in the new language with a set of video tapes (18 hours of tutorial tapes by Jean Ichbiah, Robert Firth and John Barnes). We also relied heavily on Ada Language Reference manuals, John Barnes book "Programming in Ada" and a "learn by doing" rationale. This informal training in Ada went amazingly well and the reputed difficulty of Ada training touted in some Ada literature was never seen. We trained approximately 20 programmers and 5 lead programmers and found that personnel attitude had a great deal to do with the programmer facility in the language. Those programmers that wanted to learn the language and were enthusiastic about it did so with noteworthy ease. In addition, most of those programmers

*Ada is a trademark of the U.S. DOD Ada Joint Program Office

who originally had reservations about the language became enthusiastic as the project progressed.

2.2 PROJECT HISTORY

At the same time we were learning the language we were writing Software/Hardware Requirement Specifications and the CCS hardware was being developed. By the end of this specification process it turned out that both hardware and software were quite different from that proposed. A Computer Program Development Plan and appropriate software standards document were prepared and accepted. The design methodology, based on previous company experience and policy, was chosen to be structured top-down, using Ada-based Program Design Language (PDL) as the design recording methodology.

As in all software developments the schedule crunch began to be felt. We were still learning about the major system interface to the TACFIRE system. The CCS hardware, utilizing a distributed processing concept, became increasingly complex with the final 68000 processor count an impressive 53 (see Figure 1). Communications Modems within the system with identical software accounted for 40 of the processors but there was still distinct software to be written for 13 processors with interfaces being numerous and complex.

The software architecture comprised of four Computer Program Configuration Items (CPCIs) is shown in Table 1 and Figure 2. The interface between CCS processors was accomplished via Dual-ported Random Access Memory (DPR). The DPR was partitioned into status, control, and message buffers. Bi-directional queues were used to control the passage of data/messages between processors.

Messages entered/exited the system either via the TACFIRE/Fire Direction Center (FDC) interface or externally via the Communications Modems (denoted on Figure 1 as Modem Processors). Message flow through the CCS was then controlled by processing located in the Communications Processor (CP). Code in the CP—comprised of the Control Processing (COP) and Communications Processing (CMP) CPCIs—was almost entirely in Ada, while that in the surrounding peripheral processors was in Assembly language due to timing/memory constraints.

Messages were subject to the following processing:

- Reception/Transmission
- Forward Error Correction
- Off-line Encryption/Decryption
- Compaction/Decompaction
- Authentication/Serialization
- Routing/Relay
- Transaction Accounting

2.2.1 EARLY SUCCESSES

Some early successes kept us going in spite of mounting problems and missed software milestones. These were:

- An early prototype of the CCS operator interface software demonstrated Ada feasibility.
- Telesoft delivered their Run-Time System i.e. Embedded Systems Kit (ESK) as promised. The day we first ran preliminary software on the target, a sparsely populated prototype CCS, was a great one.
- The Intellimac proved reliable and several improvements were made to speed up its operation. However, more host development processing time was necessary to support the growing staff of programmers. Since the Telesoft compiler had by then been ported to the WICAT 150S desk top workstation, four of these were obtained for the project.
- The fact that the host computers—both Intellimac and WICATS—were MC68000 based and virtually the same as the target computer, allowed much of the code checkout to be accomplished on the host systems before the target CCS hardware was available. Even later when CCS hardware was available, checkout was still done on the host first since tools were more readily available and a time consuming load module "bind" was not necessary. Also if code ran on the host it was probable (not 100 percent) that it would also run on the target.
- Ada and tasking were ideally suited to the CCS architecture leading to a theoretically elegant though complex software design.
- Ada had been selected for another small Army project, Sigma-Heros Interoperability, which was very successfully implemented.

2.2.2 SHORTCOMINGS

The early disappointments and shortcomings faced were:

- Lack of software development support tools. Only an editor was provided with the compiler. While the company had a low level debugger for MC68000 Assembly language it was unusable for Ada due to the fact that a listing of Ada code with embedded Assembly language was not available. Considerable effort was expended in writing our own special purpose trace program.
- The size of the initial code greatly exceeded our estimates and memory was increased from the original 1/2 Megabyte to 2 Megabytes. The average Telesoft-Ada compiler expansion ratio was 22 bytes per line of Ada. The compiler overhead was about 2.5 when compared to handwritten Assembly language. This unforeseen increase in memory had a significant effect on the CCS hardware, but this was deemed less costly than heroic efforts to squeeze the code into insufficient memory.
- It became clear that the Telesoft compiler and Host Operating Systems were never meant to support a project the size of CCS. We constantly exceeded limits in file size, number of files, number of packages, stack size, etc. Most of them required fixes by Telesoft whose support—via a maintenance contract—was nearly always forthcoming and timely.
- The Ada-Assembly language interface which was a part of all the peripheral processor Dual-ported RAM interfaces took considerable time to design and debug. The interface required the use of the "for use at..." construct which worked fine, but without a debugger on both sides of the processor interface checkout was very cumbersome and difficult.
- Ada proved not to be well suited for a top-down design methodology since the implementation is bottoms-up. This was partially due to the lack of the "separate" construct in Telesoft-Ada, but even with this feature the special code needed to use it precludes some of its usefulness. However, Ada-PDL worked remarkably well even though it was not consistently used. The design, and code standards and guidelines had to be tightened up considerably and compliance to them carefully monitored.
- The separate compilation feature of the Ada language was not fully implemented by Telesoft and compilations were more frequent than planned.

2.2.3 INTEGRATION AND TEST

As the detailed design and implementation of the software proceeded it was recognized that the integration and test phase was going to take longer than planned. This phase

of the program was replanned in painstaking detail into 12 "stages" of CCS development and testing. With this new plan in place we proceeded to the end of the project slowly but steadily. We have concluded that without this type of phased integration and test, planned and successively refined into test plans and procedures, that large software projects would be impossible. The CCS integration and test planning and development was a "team" effort in the best sense of the word. Software, systems and hardware engineers worked and planned together toward the common goal of completing the 12 stages.

The following anomalies were experienced at various times during the development:

- The compiler was updated twice during the software development and both times was installed immediately. This naturally caused some delay, but the benefits of the improvements far outweighed the incurred delay of about one week each time.
- The source code to the Run-time Operating System was purchased under license agreement with Telesoft and several minor but important changes were made to it.
- The internal stack size for each Ada task was defaulted by the compiler to 2000 bytes. Since several of the CCS tasks needed much more than 2000 bytes, a patch to the compiler was obtained from Telesoft to vary the stack size. Tracking this problem down literally shut down the software development for about six weeks.
- Occasionally problems showed up at execution time that were only solved by a complete recompilation. Once the symptoms of this problem were recognized it wasn't so serious, only frustrating and time consuming. A total recompilation took approximately four hours.
- The process of "binding" the program packages into a load module for the target system was slow. Subsequent releases of the ESK speeded this up from an hour to about 15 minutes.

These problems were directly related to the relative "newness" of the first generation Ada compilers. Other major problems not directly related to the use of Ada were:

- The target hardware sustained some major modifications during the middle of integration and test. Although this took time it was beneficial to the software development in the long run. With the hardware running more reliably it was much easier to track software performance and differentiate between software/hardware problems. It should be noted here that the CCS hardware was also pushing the "state of the art".
- We were continuously plagued with timing problems in the TACFIRE interface. Even though the interface was used heavily it was never reliable and performance varied significantly between TACFIRE systems used for checkout.

3.0 CONCLUSIONS

The bottom line is that in about 2 1/2 years 45,000 lines of Ada were designed, coded, checked out, integrated and tested for the CCS. An equal number of assembly language instructions (equivalent to another 11,000 lines of Ada) were also developed for the project. In addition, a considerable amount of test software and system support software was also developed and integrated. The software staff peaked at 20 people with the average being about 12. The distribution over the length of the project is depicted in Figure 3. The peak represents that part of the program just before the CCS equipment was shipped from the Singer facility to the government test bed. Development and testing continued for some months after that until the test bed was concluded in April of 1985. The software was not of "production" quality, but it was sufficient for the concept evaluation test of CCS with TACFIRE.

One of the results gleaned from the CCS test bed was that the excessive compiler overhead was causing slow execution which in turn degraded the performance of the system. A more optimized design has certainly taken care of some of the problems, but it is obvious that major improvements are still necessary in code generation optimization.

A comparison with other software projects within the company showed an increase in programmer productivity (measured in lines of normalized code per day) between two and three times greater than for other projects using other languages. We attribute this to the use of Ada since a variety of projects and languages were compared. The increase in productivity was especially apparent when compared to two other similar communications projects, one in Assembly language and one in PASCAL. Improvements over Assembly language can be attributed to the use of a High Order Language (HOL), but even normalizing for that there was significant improvement, and the improvement over PASCAL and CMS-2 can only be attributed to Ada.

It is impossible to quantitatively measure at this time benefits gained by the use of Ada in readability, maintainability, reliability etc., but our subjective judgement is that they will be significant. The key characteristics of Ada, i.e. strong typing, tasks, packages, exceptions, generics (unavailable for CCS), and mechanisms for data and control abstraction cannot help but improve the readability of the code and reduce maintenance costs.

We look forward with great interest and anticipation to the second generation of Ada compilers and improved tool sets, some of which are already on the market.

TABLE 1. CCS CPCIS, CPCS

Control Processing	COP
Communication Processing	CMP
Network Protocol Processing	NPP
Modem Processing	MMP
COP CPCS	
System Initialization	COPINT
Diagnostics	COPDIAGS
FDC Interface	COPFDC
COMSEC Interface	COPCSC
Keyboard/Display Interface	COPKDP
Bubble Memory Interface	COPQIB
Modem Interface	COPMP
NPP Interface	COPNPP
System Executive	SYSEXC
Interrupt Handler	SYSINTR
CMP CPCS	
Communications Initialization	CMPINT
Network Management	CMPNET
Communications Data Base	CMPDATA
Voice Communications	CMPVCM
Authentication	CMPATH
NPP CPCS	
Network/Executive	NPPEXC
Abridgement/Compaction	NPPCPTN
Network Management Aids	NPPMGMT
MMP CPC	
Executive	MMPEXC
Initialization	MMPINT
Protocol	MMPPRT
Media/Device	MMPMDV
Error Processing	MMPFEC
Fiber-Optic Interface	MMPFOI

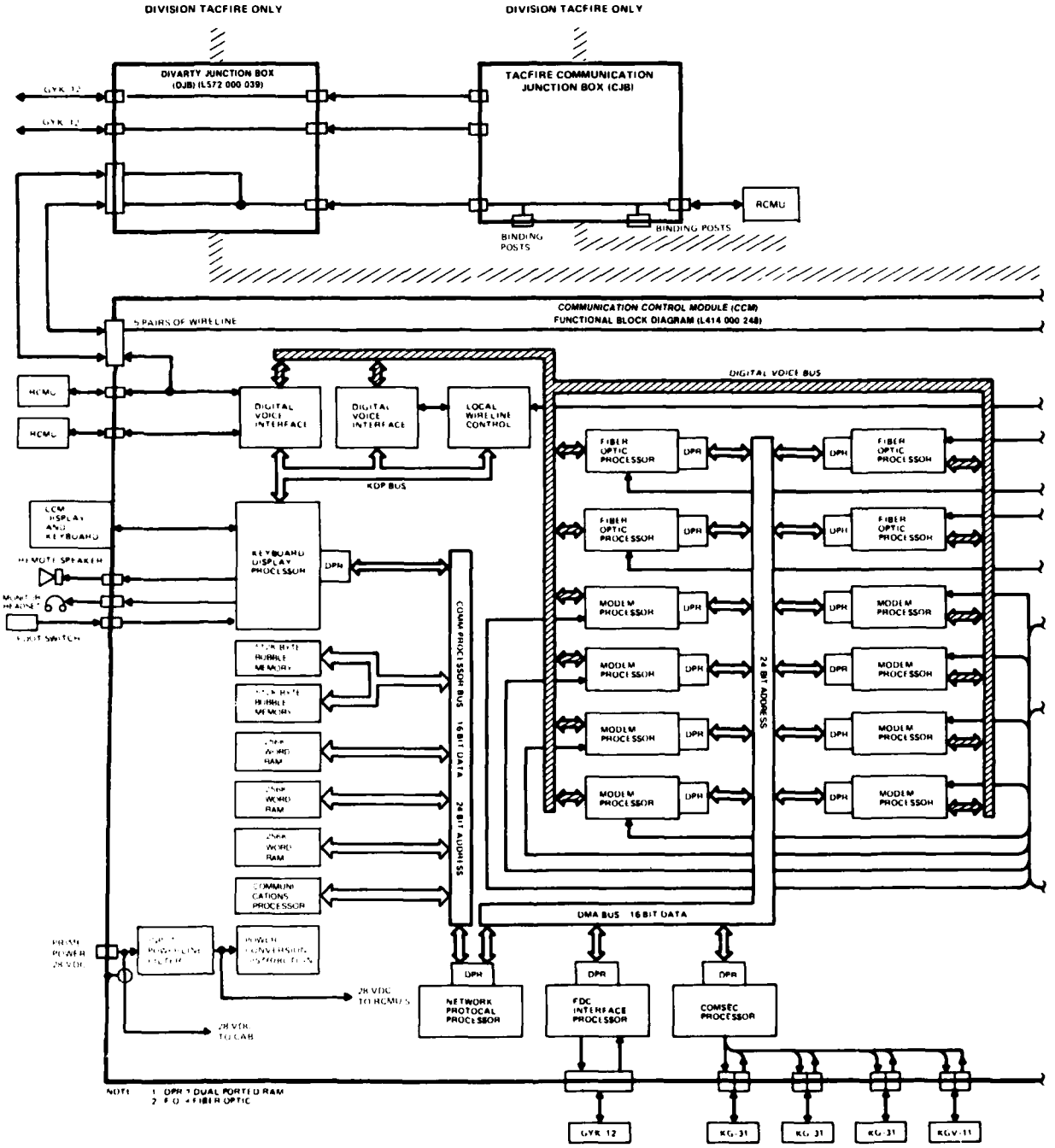


Figure 1. Communications Control System (CCS)
Functional Block Diagram
Final Design Model Rev. 3

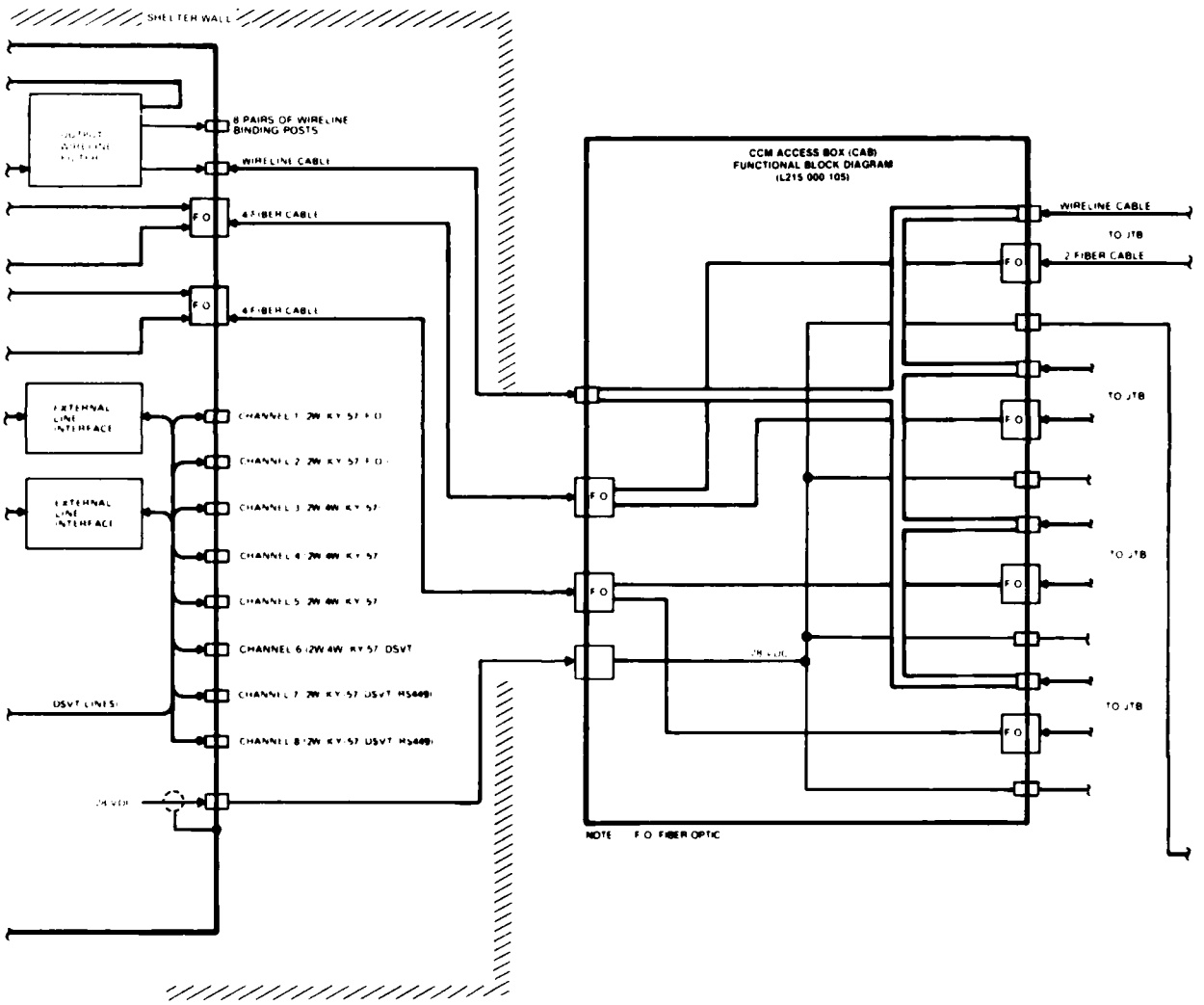


Figure 1. Communications Control System (CCS)
Functional Block Diagram
Final Design Model Rev. 3, Continued

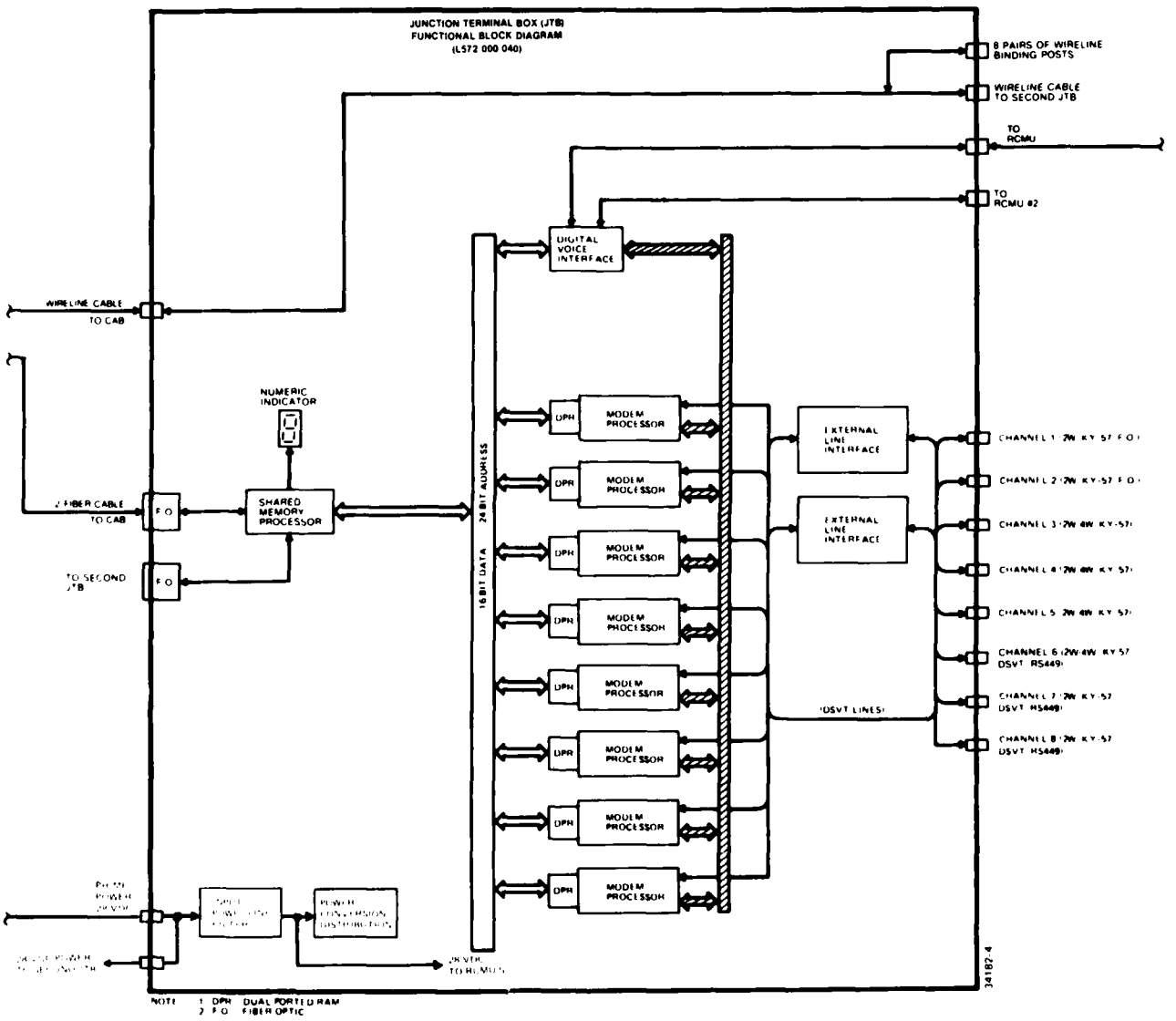


Figure 1. Communications Control System (CCS) Functional Block Diagram Final Design Model Rev. 3, Continued

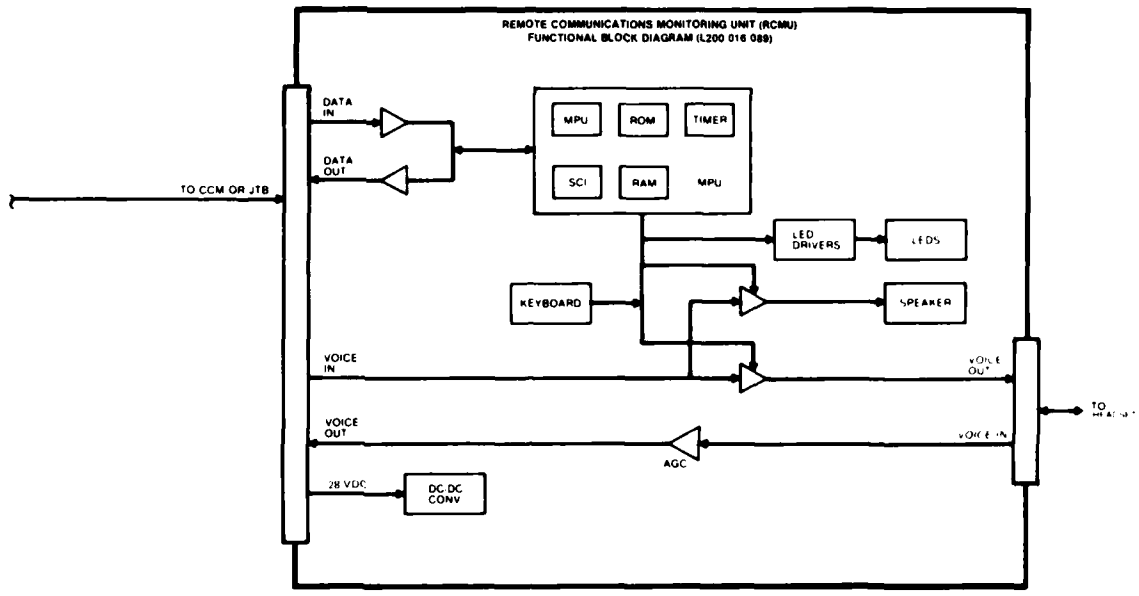


Figure 1. Communications Control System (CCS)
Functional Block Diagram
Final Design Model Rev. 3, Continued

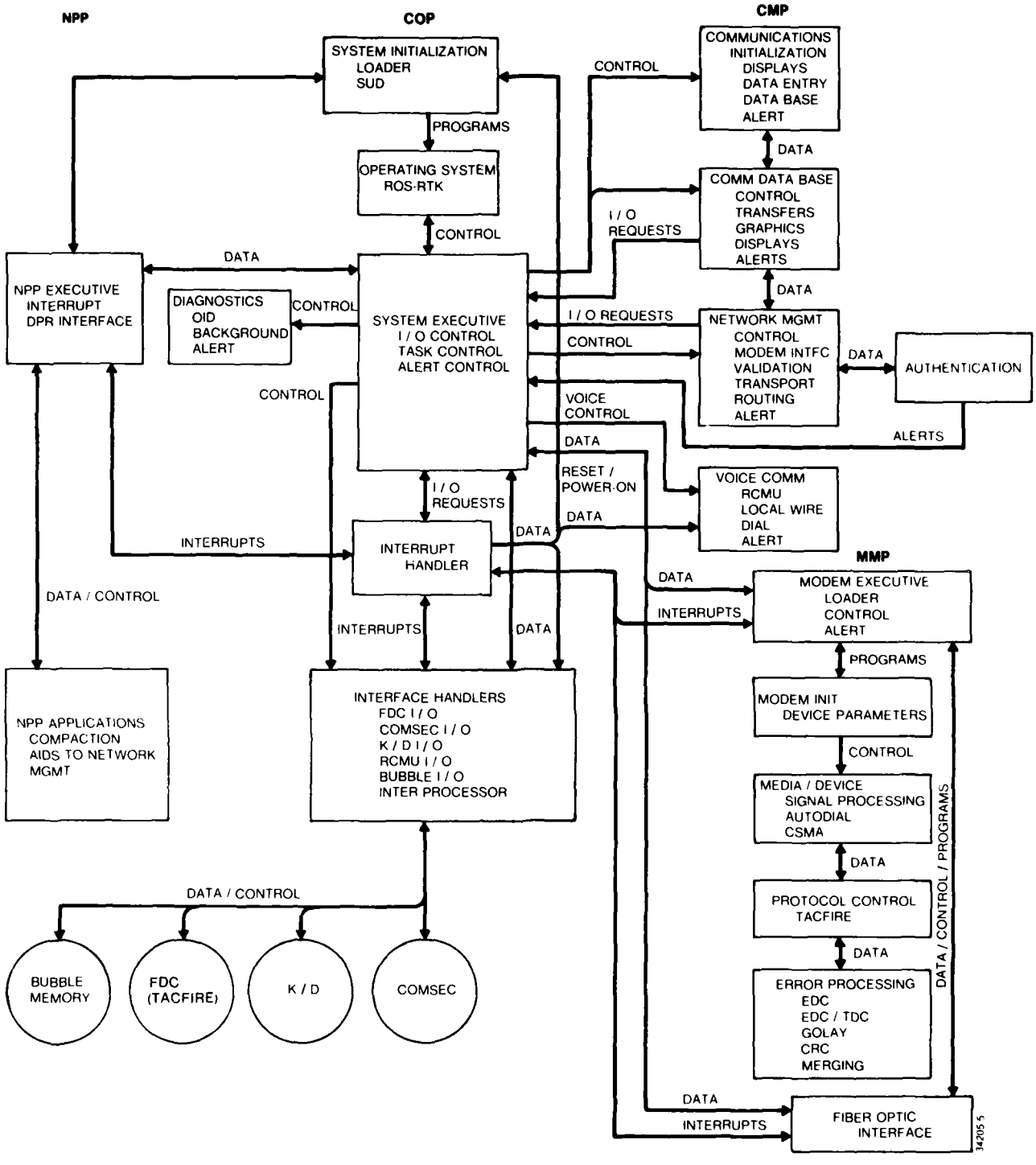


Figure 2. CPCI Interaction

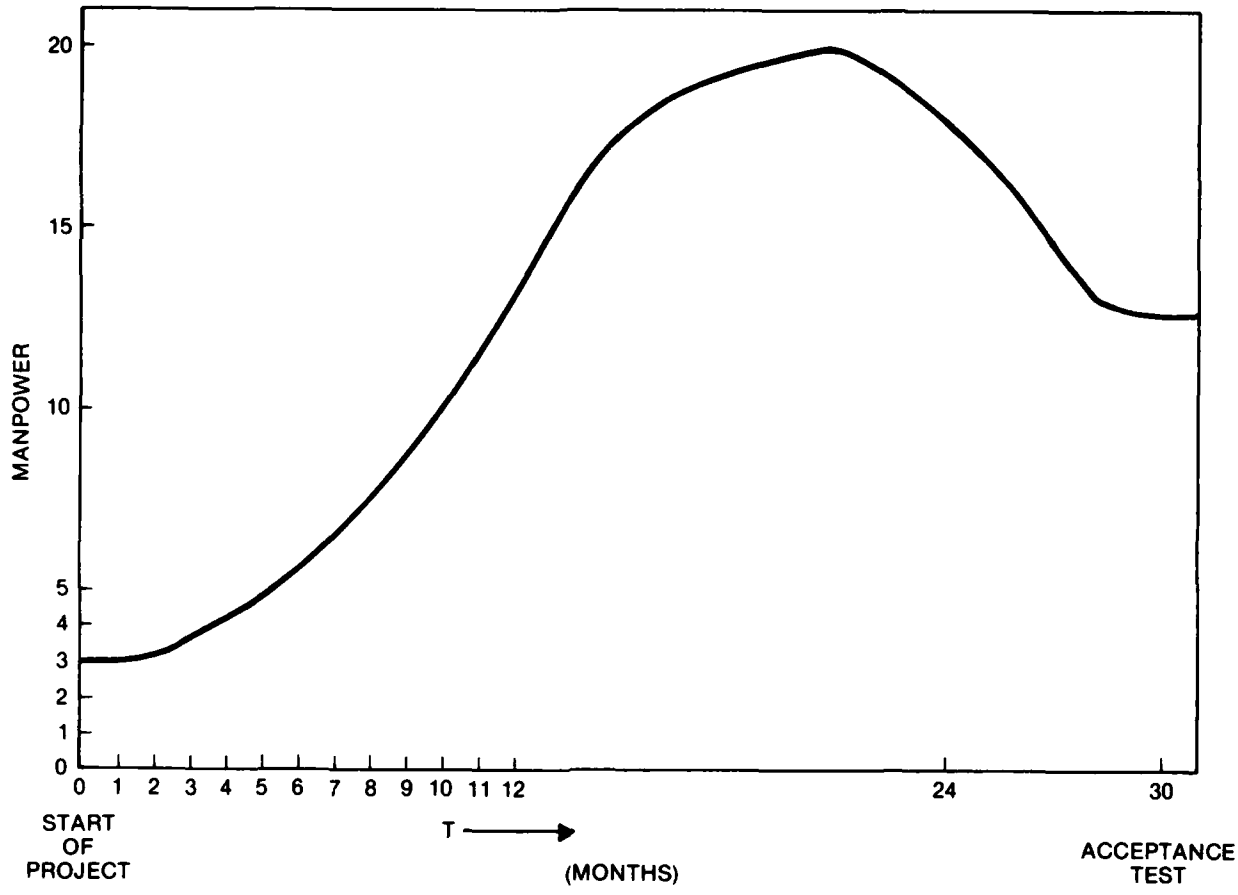


Figure 3. Project Manpower Distribution

Author: Patricia J. Dousette

Address: The Singer Company, Librascope Division
833 Sonora Ave.
Glendale, California 91201-0279

Biographical Sketch:

B.S. Mathematics, 1968 from California State Polytechnic University, San Luis Obispo, Ca.

M.S. Mathematics, 1977 from California State University at Los Angeles.

Employed at Singer-Librascope since July 1968 in the following capacities:

- Associate Mathematician
- Mathematician
- Technical Programmer
- Engineer
- Senior Engineer
- Systems Programming Specialist
- Research Programming Specialist
- Supervisor, Army Applications Programming

Extensive background in Systems Engineering and Analysis with emphasis on Software Engineering. Most recently, Manager of Software Development for the Communications Control System (CCS).

THE ARMY'S MAFIS COMMAND AND CONTROL

MICHAEL T. PERKINS AND JOE E. BOLGER

THE BDM CORPORATION
AUSTIN, TEXAS

Abstract

The Command & Control Subsystem of MAFIS is designed to monitor and control military doctrine and equipment testing exercises. The Command & Control is a distributed processing system that controls a communications network for the collection of data, processes applications (such as simulations and database management), and supports a user interface and graphics display. The Command & Control Subsystem is being designed and implemented in Ada*. A pilot portion of the system was tested at the end of FY85. This paper describes the requirements, development methodology, and implementation.

Introduction and Background

The Mobile Automated Field Instrumentation System (MAFIS), being built for the Army Training and Doctrine Command by BDM, allows for real-time monitoring of two-sided mechanized combat where weapon engagements are simulated. The MAFIS is composed of three subsystems: 1) the Field Instrumentation Subsystem, 2) the Command and Control (C&C) Subsystem, and 3) the Operations and Support Subsystem. The Field Instrumentation Subsystem equips combat participants with electronics to perform digital communication, position location, logic processing, and engagement simulation. The engagement simulation equipment consists of both low power, bore-sighted lasers that flash a coded message with each weapon trigger pull and an array of laser light detectors to score weapon hit information. The main processor is a Motorola 68010 with up to 2 Mbytes of memory. Position Location is to be provided by a module that receives transmissions from the Global Positioning System (GPS) space segment. The Position Location Module performs satellite multilateration. Player status and activity information is relayed to the C&C through a communications repeater network.

* Ada is a registered trademark of the Department of Defense (Ada Joint Program Office).

C&C processing is distributed among a Network Communications Processor, an Applications Processor, and Display Processors. The Network Communications Processor collects the data, filters it, and distributes it to other processors. The Applications Processor maintains a player database, records a history log, and executes simulations and other applications. The Display Processors serve as the user interface for monitoring and control of field activity. The processors communicate over a high speed Local Area Network (LAN). The primary flux of data across this LAN is field exercise activity broadcast to all other processors by the Network Communications Processor.

Requirements and Capabilities

The broad goals of the MAFIS are to instrument and monitor 200 (expandable to 2000) players interacting over a 50 by 50 kilometer area. A real combat environment is to be simulated as accurately as possible. The field portion of the MAFIS provides a practically invisible (greatly reduced in size) on-board equipment configuration. Both mechanized and (eventually) infantry platforms are to be supported. Most mechanized platforms will have multiple weapons systems that will need to be instrumented. Real-time casualty assessment (the ability to perform weapon engagement computer decision-simulations within one second), stimulated as a result of laser hit detection, will be done at the player level in the on-board logic processing unit. Players will report position and status information to C&C along with firing data and casualty assessments; configuration data will be sent to C&C to confirm player identity. Should the player's communications equipment lose contact with the network, the logic processor will be able to store several hours of data (enough for one trial) for transmission when network communication is re-established. C&C will be able to send messages to players to perform simulated ammunition reloads, to disable and enable laser firing, to perform an administrative kill simulation, to resurrect (re-enable) a platform, to report configuration status, and to report special simulations.

A major function of C&C is to control the communications network. Communications are implemented through a hybrid time-division multiple access and demand reservation network that uses line-of-sight, digital, spread-spectrum, 370 Mhz radios. Because of the line-of--sight communications constraints, a repeater routing network must be established that assigns players to repeaters according to signal strength. To ensure uninterrupted coverage as players move away from one repeater, they must be able to be reassigned to the succeeding network node. The network is configurable according to exercise and terrain requirements.

The C&C functionality also supports the processing of applications such as database management, history log recording, and simulation execution. A database is maintained that records player type and configuration, player location and status, player scoring data, and the status of all active simulations. An event history log is recorded in time order of event to support off-line replay and review of exercises. During the monitoring of an exercise, there is no guarantee that C&C will receive events in time order because of the logging capability that exists at the player level. Specific simulations are defined and programmed according to their specific requirements; however, software utilities are provided with the system to support the detection of a player's presence inside defined areas.

A third major function of C&C is to provide a human interface. The operator is presented a high-resolution color graphics map display generated from the Defense Mapping Agency's terrain data. The map display allows selectable feature visibility and supports zoom and virtual image panning from a disk data file. A player symbol display is superimposed over the map image. The text, line strokes, etc., that define the player are drawn in certain colors to designate attackers, defenders, or referees; the symbol background, drawn as a rectangle, is color-coded according to player status. Groups of players can be displayed as one symbol according to relative position in the military chain of command. Simulated weapon engagements are displayed as they occur. The operator can designate military planning control measures to appear on the map. In addition to the color display, an alphanumeric terminal is provided for notifications and reports. Operator commands are given by touching buttons drawn on a graphic tablet.

Development Methodology

The software development process follows a classical top-down design [1]. The project phases are requirements analysis, functional design, general design, detailed design, code, test, integration, and final test. Early in the design phase, Ada was chosen as the implementation language. The detailed design phase was tailored specifically to an implementation in Ada.

Solutions to many design issues documented by various authors have been incorporated into the development effort. The software design process is based on the Structured Analysis and Design Method (SADM) and uses Ada as a formal notation for expressing the (pictorial) results of SADM. The structured methodologies are well documented in Structured Analysis and System Specification, T. DeMarco [2] and Structured Design, Yourdon & Constantine [3]. The design method implementation draws heavily on concepts described in "A Software Design Method for Real-Time Systems", H. Goma [4] and Ada Design Language for the Structured Design Methodology, J.P. Privitera [5]. The translation from the design results of SADM into the Ada design notation is accomplished by a technique borrowed from the Process Abstraction Method [6]. The layering and hiding of various design decisions is detailed in "On the Criteria To Be Used in Decomposing Systems into Modules", D. Parnas [7].

Software design began with text descriptions of the primary functionality. A hardware architecture was developed based upon estimated processing capabilities required for that functionality. Sets of hierarchical data flow diagrams were created to formalize software module and interface definition. Data flow diagrams depicted software modules as circles, hardware interfaces as rectangles, and data flows as connecting arrows; a text label is associated with each object. Depending upon complexity, software modules were sub-categorized in a hierarchy of other data flow diagrams. At the conclusion of general design, data communicated over each processor's external interface was defined in the greatest possible detail, and updates were issued as necessary.

The creation of Ada code at the highest level began during detailed design. An Ada master procedure was created for each processor. The data flow diagrams were used to identify software modules and data structures. Because of the complexity of the software, the first level data flow diagrams were used to define Ada packages within the main procedure. To define internal software interfaces, package specifications were written within the main procedure according to the functionality defined by the highest-level data flow diagrams. To limit complexity, each package body was a separate compilation unit. Data structures required for communication between packages followed the data flow diagrams. Global data structures were encapsulated in a package specification called Global_Types. Concurrent tasks were identified on the data flow diagrams by enclosing software modules in rectangles or polygons. The Ada task specifications were then hidden inside the package bodies. The task bodies were again separate compilation units. For detailed design, the description of processing required inside the task bodies was no longer in compilable Ada, but was written in Ada psuedo-code. Auxillary procedures required for data set

preparation or wrap-up processing did not require multi-tasking and were considerably easier to design.

Following detailed design, coding will primarily involve transforming Ada psuedo-code into actual compilable Ada. As code is developed, testing will be done at the module interface level before integration begins. Integration will take place on a module by module basis, with testing following each step. After software integration is complete, the entire system will be tested during the king of actual field exercise for which the system was developed.

Implementation

The hardware architecture chosen to implement the C&C design is as shown in Figure 1. The LAN architecture is the key to flexibility and expandability. The Communications Network Controller organizes the network and routes messages between players and the applications software in C&C. Data received from players that represent a significant change will be broadcast over the LAN to all other C&C processors. Data generated within C&C that affects the exercise (e.g., initialization data, simulation data, and start/stop trial messages) will also be broadcast over the LAN. Both the Applications Processor and the Display Processor read the broadcast information from the LAN in processing exercise data according to their specific functions. All three processors communicate by sending messages between distinct nodes. Expandability is achieved by use of the LAN architecture, since other processors may be added to the LAN as needed. For example, additional Display Processors may be added to the LAN to accommodate specialized or expanded operator duties. The processors that execute the Communications Network Controller and the Applications Processor software are Data General MV/8000's. The computer chosen to execute the Display Processor software is a Data General MV/4000 with a GDC/2400 graphics display.

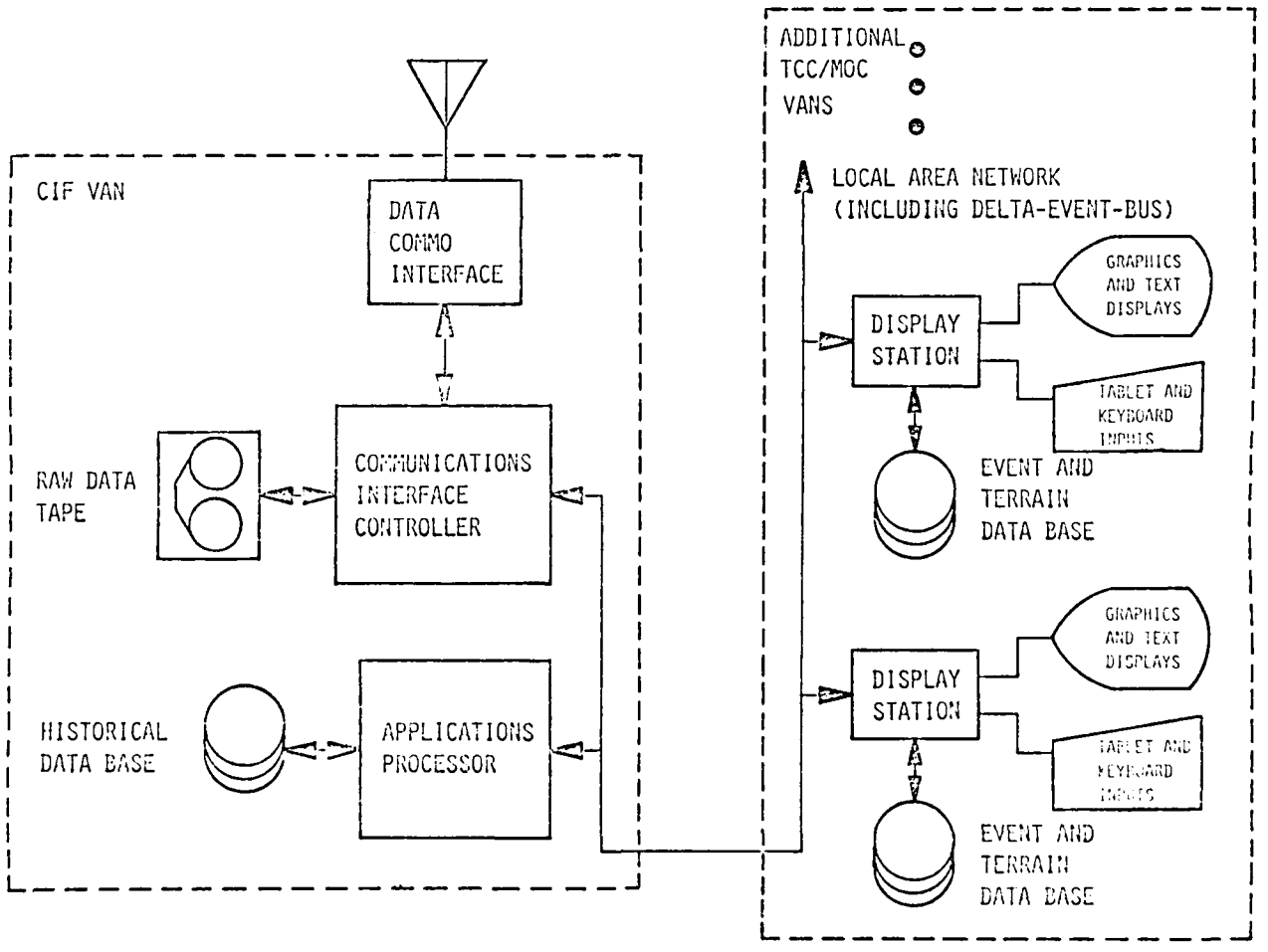


FIGURE 1: COMMAND AND CONTROL HARDWARE ARCHITECTURE

Functional assignments associated with each processor are arranged according to network control, applications execution, and human interface control. The network control functions are to 1) control the configuration and timing of the network in such a way that throughput is timely, 2) provide diagnostics in the event of a network failure and correct the problem if possible, 3) receive all communications from the network and generate output that indicates significant field activity, 4) transmit messages from the operator/controller to the player equipment, and 5) provide a raw data tape of all communications network messages. The communications messages data tape will provide a complete record of C&C input and output. The applications functions are required not only during real-time processing, but also in pre- and post-exercise processing. The real-time applications functions are to 1) create a sorted historical event log, 2) collect statistics needed to generate configuration and scoring reports, 3) route messages from the operator/controller at a Display Processor to player equipment, 4) assure player equipment acknowledgement of the message, and 5) provide processing algorithms for specific simulations, such as area effect weapons, alerts or alarms, and special exercise-dependent statistics. The Applications Processor history log data tape will allow complete reconstruction and permutation for post-exercise analysis. The applications functions performed in pre- and post-exercise mode are to 1) edit the player configurations and prepare initialization data, 2) create the terrain map image used by the Display Processors, 3) merge post-test late data to provide a complete time-ordered log of exercise history, and 4) replay the history log for post-test review and analysis. The functions performed by the Display processor are to provide 1) a man-machine interface through which the operator/controller can interact with the players in the field, 2) a color graphic representation of the field with appropriate symbols representing players, their movements, and their engagements, and 3) alphanumeric and graphic summaries of information generated by the Applications Processor relating to player performance and overall exercise evaluation. (An example of how this functionality was defined for the Display Processor using data flow diagrams is shown in Figure 2.) To create and edit the data flow diagrams, an Ada graphics editing program was written to support a machine readable database.

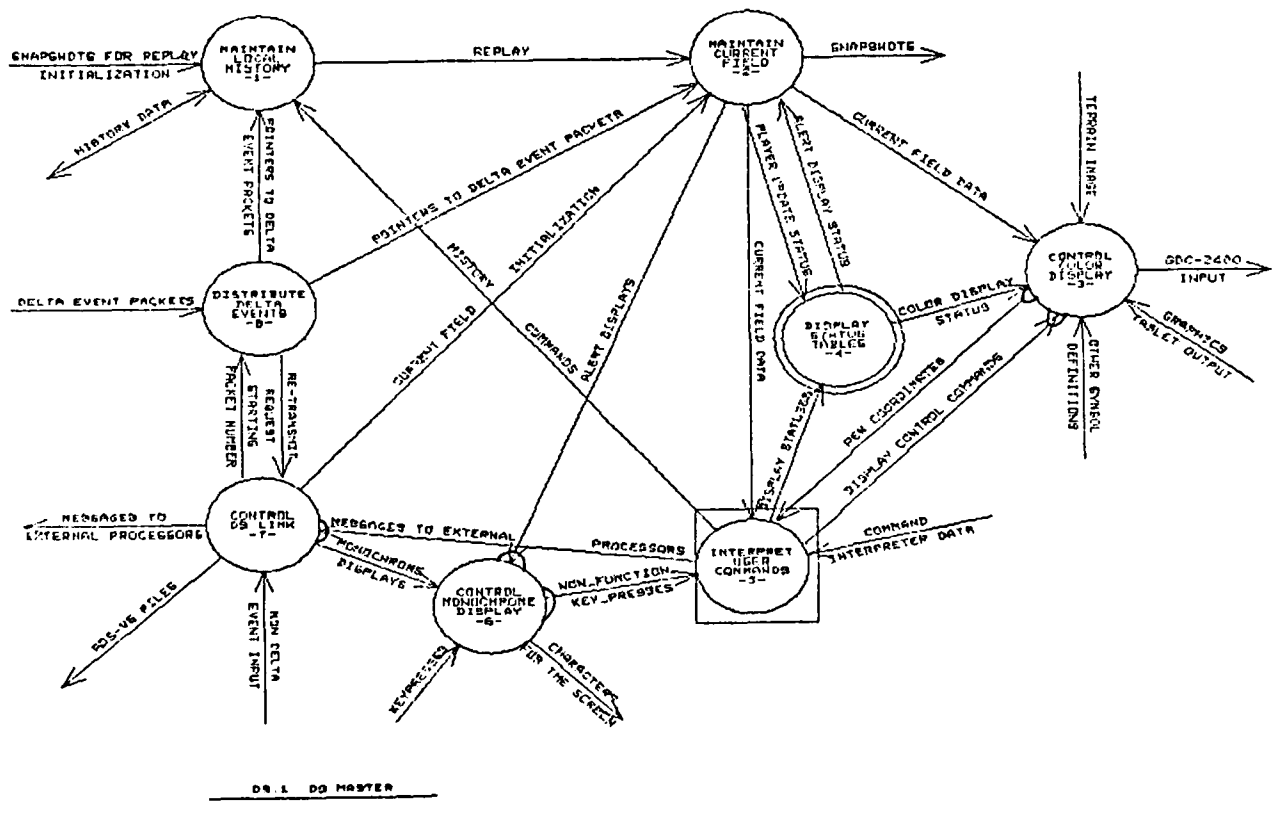


FIGURE 2: LEVEL 1 DATA FLOW DIAGRAMS FOR THE DISPLAY PROCESSOR

The current project schedule is a two-tiered incremental arrangement. Essential software needed to meet minimum performance criteria is to be developed first and will constitute a Limited Operating Capability (LOC). This system will then be augmented to develop increased functionality, until it constitutes Final Operating Capability (FOC). In the software development cycle, all phases beyond general design will be duplicated for LOC and FOC. The current status of the project is in the coding phase for LOC. LOC is scheduled to be completed in February of 1987, with FOC to follow one year later.

Advantages and Difficulties

The advantages of using Ada early in the design phase have been rewarding. The foresight and planning that went into Ada have allowed the development of tools that are reusable and expandable, avoiding extensive redesign. The modularity available in Ada contributes greatly to the maintaining of comprehensible compilation units. The software interface specifications required in Ada have also greatly aided in keeping separate software developers working toward common goals. Other notable advantages gained from using Ada are explicit multitasking and exception handling.

The design methodology was to a large extent refined as needs arose. The hardware design was intended to complement the use of Ada. This was accomplished by a balanced approach of top-down design and functional abstraction. The currently defined methodology is a process leading from high-level functionality to deeper and deeper levels of definition. Every step of the process follows from the previous one without duplication of effort and without large gaps in understanding.

Because of the flexibility of the hardware architecture and the modularity of the software, it is expected that the MAFIS C&C subsystem will be able to be easily tailored to other project needs. Portions of the MAFIS design can be used directly in other systems. The software functionality has been abstracted from hardware characteristics wherever possible and should result in easily transportable software parts. An ideal example of this would be the software to generate the map image from the Defense Mapping Agency's terrain database. The Government could save a significant amount of development expense if all Government software could be centrally archived for ease of access by other projects.

Difficulties encountered in the project have been associated with early implementations of an Ada compiler. Compiler shortcoming have forced programmers to develop work-around solutions in some cases. The overhead associated with current implementations of Ada was not anticipated when processor size was originally established and has impacted capabilities. In spite of these difficulties, project development has benefited overall from the use of Ada.

References

[1] BDM Management Services Company. MAFIS Software Management Plan:Volume I, Policies (October 1984) and Volume II, Procedures (April 1984). BDM/Austin.

[2] DeMarco, T. Structured Analysis and System Specification. New York: Yourdon Press, 1978.

[3] Yourdon, E., and L. Constantine. Structured Design. 2nd ed. New York: Yourdon Press, 1978.

[4] Gomaa, H. "A Software Design Method for Real-Time Systems." Communications of the ACM 27 (September 1984): 938-949.

[5] Privitera, J.P. Ada Design Language for the Structured Design Methodology. Ford Aerospace and Communications Corporation.

[6] Cherry, George W. Process Abstraction Method. Reston, Virginia: Cherry, 1984.

[7] Parnas, D.L. "On the Criteria To Be Used in Decomposing Systems into Modules." Communications of the ACM 12 (December 1972): 1053-1058.

MR. MICHAEL T. PERKINS
9020-II CAPITAL OF TX HIGHWAY, SUITE 600
AUSTIN, TX 78759

Mr. M. T. Perkins is Vice President, Advanced System Design and Integration, for the BDM Corporation. He received his BS in 1972 and Masters in Engineering in 1979 from Cal Poly San Luis Obispo. His graduate work focused on computer applications to communications and systems. He has spent the majority of his last 10 years with BDM designing implementing real--time, distributed processing systems for range testing and data collection. He participated in IEEE 730 Software Quality Assurance Standards. He is presently heading up the Mobile Instrumentation Field Instrumentation System (MAFIS) development and integration. In his spare time, he collects and thrashes BMWs.

DR. JOSEPH E. BOLGER
9020-II CAPITAL OF TX HIGHWAY, SUITE 600
AUSTIN, TX 78759

Dr. J. F. Bolger is a Senior Staff Member for the BDM Corporation. He received his PhD from the University of Texas in 1977, majoring in experimental nuclear physics. He did graduate work at the Meson Physics Facility in Los Alamos and as a post-graduate, he did experimental research in Switzerland at the Swiss Institute for Nuclear Research. All of his research required development and use of computer automated data acquisition and analysis systems. Currently, he is the Software Manager of the applications and human interface software development for the MAFIS Command and Control.

AN ADA* TASKING APPLICATION IN AN AIR DEFENSE SYSTEM

Christine Ausnit

SofTech, Inc.

Introduction

In designing the target tracking processes of an air defense system, the designer must choose an appropriate tasking structure. The Ada tasking features offer both multi-threads per task and single thread per task approaches to solving a given problem. In this case, there are several options available to the designer: a single task, one task per blip, one task per major function, one task per sector, and one task per track. This paper will discuss these strategies, their advantages and disadvantages.

Background: Air Defense System

An air defense system is a large, complex system consisting of a network of individual centers, each responsible for the defense of a particular geographical segment. Each center tracks aircraft within its area, communicating with other centers as well as with its own radar units, fire units and human operators. Its primary activities may be summarized as processing tracking information, evaluating threats, and assigning weapons. The design of the tracking subsystem is explored further.

Each center's tracking subsystem conducts a 360 degree radar scan of the sky at a fixed rate. During each scan, the radar detects reflections or blips from the objects, usually aircraft, flying through its area and records range, bearing, and rate of change information. This data is used in the processing in order to correlate or associate blips received on successive scans into the track (or path) of a single target. Furthermore, once during each scan, the subsystem attempts to extend the track by predicting the next position of the target, and this predicted location is fed back into the association algorithm. This prediction is computed whether or not a new blip was received; however, if no new blips are received over a specified period of time (or a specified number of radar scans) then the track is discontinued.

Alternative Design Approaches

The selection of a tasking structure for an embedded computer program is one of the most basic design decisions that must be made. This decision involves not only determining what entities to represent with tasks but also how many threads (in the sense of functionality) to include per task. In the case of the target tracking system, there exist both multi-threaded and single-threaded alternatives:

- single task handling all the work
- one task per blip
- one task per major tracking subfunction
- one task per scan sector (a sector being a pie slice of the whole scan)
- one task per track

The first choice, a multi-threaded one in that it embodies many actions within a single control element, can be eliminated as too general a design. Beyond some main program, it fails to identify the salient program or data structures. Furthermore it does not recognize the potential for conceptual concurrency (e.g. once a blip has been associated with a given track, then that track can be smoothed while other blips are associated with the remaining tracks).

The next choice, one task per blip, shows a single-threaded approach, but it can also be discarded as unworkable. Blips are the input to the system, not its mainstay. Logically, blips do not control the computation: they are data rather than actions. Moreover, they are

not persistent data: they record the position of some object at a particular instant. As the object moves and a new radar scan detects it, a new blip appears on the radar screen; the old one no longer exists. The overhead associated with the creation, destruction and decision-making of blips (i.e. deciding with which track to associate) would be excessive.

The third alternative, one task per each major subfunction, is a logical design which models the actual operation of the tracking subsystem. Because it is best used in combination with either the multi-threaded task per sector or the single-threaded task per track solution, only those two alternatives are explored in the subsequent sections.

Task per Sector

Sectors are arbitrary divisions of the radar scan, imposed by the software. Within each sector, the identical processing occurs: all the blips are associated with existing tracks, a resolution stage is performed to ensure that all tracks use different blips, these tracks are then smoothed, and the next point along the track is predicted. Track information is maintained in a database accessible by the entire system.

On first glance, it would seem that sector processing can be performed independently and therefore concurrently. Within a given sector, association must be performed before either smoothing or prediction. Association tries to find the best match between a single blip and some track in the database. Once the track has been isolated, it will no longer be considered as a possible match for other blips, and it can undergo smoothing and prediction at the same time that the other tracks are still under consideration by the association algorithm. Thus there is additional concurrency which can be exploited.

Problems arise, however, in the case of tracks nearing the edge of or crossing sector boundaries. Suppose a track in sector A should be associated with a blip in Sector B. In a track per sector scheme, this track does not have visibility to this blip because it lies in a different sector. Extra processing is required to handle these potentially "extra" blips that are not associated with a track in their sector before these blips can be assigned to new tracks. These additional computations render the design more error prone, less readable and less maintainable.

A final requirement is that sector processing for a given sector must be completed in the time the radar takes to sweep n-1 sectors. If the processing is not completed in this time, it must be stopped so that the processing task is then restarted with the fresh data from the new radar sweep of the sector.

The design of this solution is found in Figure 1, and the corresponding code is sketched in Figure 2. An array of tasks is declared with one task corresponding to each sector. The sectors are declared to be part of a controller task, whose function is to synchronize with the north sector pulse, stop the next sector's processing and provide an interface between the radar data and the current sector processing.

The procedure Associate and the tasks Smooth_Track and Predict_Next_Point are declared in the task body of the sector task. Were the two tasks declared directly inside the procedure's declarative portion instead, then the benefits of tasking would be lost. The procedure Associate could not complete execution until both its dependent tasks terminated. As Associate performs no further computing once it synchronizes with Smooth_Track, the effect of the entry call would be identical to that of a procedure call, and no concurrency would be achieved. The solution presented here declares the smoothing and prediction tasks outside Associate. Therefore, they are active and running even after Associate returns and while the parent Sector_Processing_Task checks for preemption before continuing to process the next blip. (Should either smoothing or prediction operate too slowly, a buffering task can always be introduced so that the entry call from Associate is immediately accepted by the buffering and does not need to wait for Smooth_Track or Predict_Next_Point to be ready to accept new tracks.)

Task per Track

Another, more elegant design for the tracking subsystem is based on assigning one task per track. Since the system may handle different numbers of tracks at different times, a linked list of track task objects will be used. (Otherwise an assumption about some arbitrary upper limit of tracks would have to be made.) Once per scan, each track is ready to receive a new blip association. Each track also awaits a

signal from a master controller in order to become active. A track may become discontinued because no new blips are associated with it during some specified number of consecutive scans. The master controller would allocate a new task if a new track is needed. (See Figure 3.)

The body of the track task is simple. Following the receipt of its identification number, the track waits to be associated with some blip. If a blip is found, the track gets updated. If no blip is found, the track is updated as is a counter of the number of missed blips. Should too many scans fail to yield an associating blip, the track task is ready to terminate. These two signals (data receipt and track activation) are the two entry points into the task. The track itself announces that it is ready to receive data by making an appropriate entry call. Similarly, once a particular associated track is out of the market for other associations, it calls the track smoothing and prediction operations.

The association function here is declared as a task. Its interface allows it to communicate with the radar in order to receive the blip data as well as to communicate with the track of its choice. Association chooses which track is the best match and calls that track's data receipt entry accordingly. The corresponding pseudocode is shown in Figure 4.

Smoothing is declared as a procedure rather than a task because the smoothing operation for any single track must complete before the next position for that track can be predicted. The prediction computation, however, can safely be declared as a task, as no further computations depend upon its execution.

Conclusion

This paper has discussed two different approaches to an embedded systems problem. Both designs addressed the possiblities for concurrency. The task per sector design is neither control nor data driven. The design is based on an arbitrary division of labor, prematurely constraining the choice of the overall software architecture. Parallelism exists to the extent that the similar processing is being done at the same time for as many sectors as there are in the radar, although sectors are never concurrently updated.

This design has all the disadvantages of approaching a concurrent problem from a linear standpoint. It is more difficult to maintain. Because of potentially delicate timing, algorithms that should be kept in a single unit may be split across several modules, detracting from the readability of the whole. It fails to model the functionality of the system: an air defense system is about tracking targets, not about radar sectors.

The task per track design, on the other hand, represents a data driven design. It is a much better abstraction to represent the functionality of the system, and it is consequently more readable and maintainable. Moreover, the mutually exclusive nature of tasks enforces protection of the track data. Only one track can be accessed or updated at a time; these track tasks form the database, obviating a database superstructure.

Although this design is better than a task per sector, it too may have potential timing problems, which would most likely be generated by the tasking overhead needed to handle the greater number of tasks and the dynamic allocation of tracks. The success of this implementation would depend in part on the run time scheduler and the existence of tasking optimizations [1].

Sectors are an implementation detail that are useful in the association algorithm itself in order to do bookkeeping on the incoming blips; however, this does not justify highlighting sectors at the top level of the design. In the task per track approach, the sectors are hidden in the association task.

The fundamental issue in this design is which entities to choose to model with tasks. Jackson's definition of entities is useful here [2]; an entity either performs or suffers actions in a time sequence. Applying this criterion, it is clear that a sector neither performs nor suffers actions. It is static and has no significant time ordering. Tracks, on the other hand, are associated with blips and predict new positions, both of which events are ordered in time. Blips are attributes of tracks: in a loose sense, it is the collection of blips that constitutes a track. This entity-action concept is especially appropriate in developing tasking designs because of the dual nature of tasks as program units and objects.

References

1Hood, P. and V. Grover, Ada Real Time Studies Report, SofTech, Inc., 1986.

2Jackson, M.A., System Development, Prentice-Hall, NJ 1983.

SofTech, Inc. Ada Software Design Methods Formulation Case Studies Report, 1982.

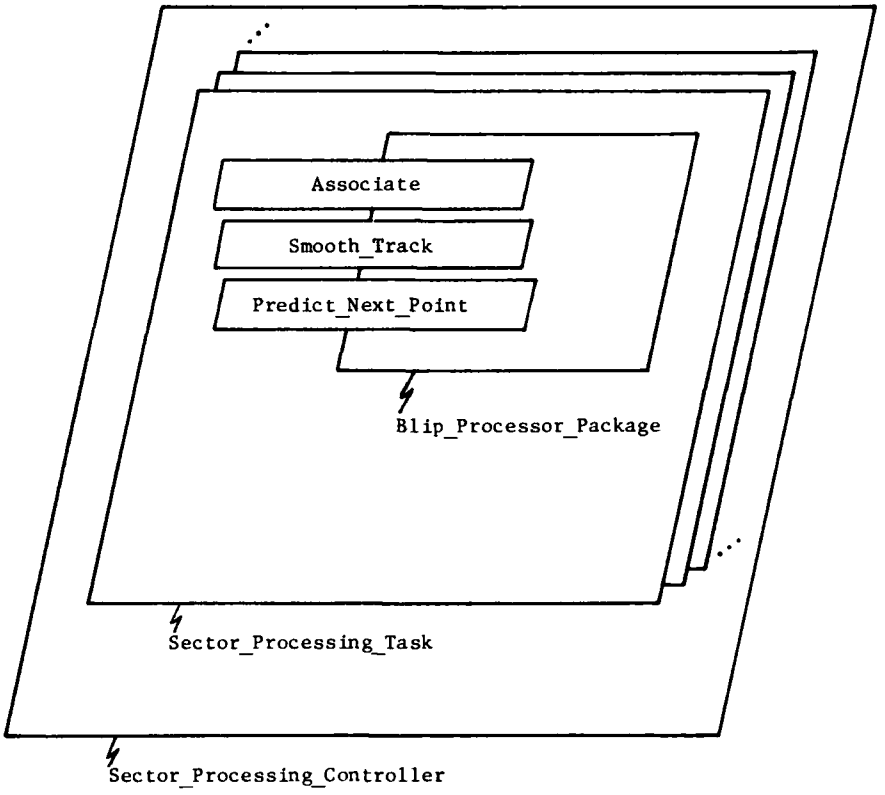


Figure 1: Sector Design

```
task Sector_Processing_Controller is
  entry New_Sector_Pulse;
end Sector_Processing_Controller;

with Blip_Package; use Blip_Package;   -- declarations for blips
task body Sector_Processing_Controller is

  task type Sector_Processing_Task is
    entry Get_New_Blip_Data (Blips : in Blip_Type);
    entry Stop_Processing;
  end Sector_Processing_Task;

  type Sector_Processor_List_Type is array (Sector_Count)
        of Sector_Processing_Task;
  Sector_Processor_List : Sector_Processor_List_Type;
  Sector_Data : Blip_Type;

  task body Sector_Processing_Task is separate;

begin
  -- ...
  loop
    Current_Sector := Sector_Processing_List'Last;
    for Next_Sector in Sector_Processor_List'Range loop
      accept New_Sector_Pulse;
      Sector_Processor_List (Next_Sector).Stop_Processing;
      Sector_Processor_List (Current_Sector).
                          Get_New_Blip_Data(Sector_Data);
      Current_Sector := Next_Sector;
    end loop;
    -- ...
  end loop;
end Sector_Processing_Controller;
```

Figure 2: Sector Controller (1 of 4)

```
separate (Sector_Processing_Controller)
task body Sector_Processing_Task is

  Local_Blip_Data : Blip_Type;
  Preempted       : Boolean;

  package Blip_Processor_Package is
    procedure Associate (Blip : in Blip_Records);
    task Smooth_Track is
      entry Get_Track (Track : in Track_Type);
    end Smooth_Track;
    task Predict_Next_Point is
      entry Get_Track (Track : in Track_Type);
    end Predict_Next_Point;
  end Blip_Processor_Package;

  package body Blip_Processor_Package is separate;
  use Blip_Processor_Package;

begin
  loop
    accept Get_New_Blip_Data (Blips : in Blip_Type) do
      Local_Blip_Data := Blips;
    end Get_New_Blip_Data;
    Preempted := False;
    for Current_Blip in Local_Blip_Data.Number_Blips loop
      Associate(Local_Blip_Data.Data(Current_Blip));
      select
        accept Stop_Processing;
        Preempted := True;
      else
        null;
      end select;
      exit when Preempted;
    end loop;
    if not Preempted then    -- processing finished early
      accept Stop_Processing;
    end if;
  end loop;
end Sector_Processing_Task;
```

Figure 2: Sector Controller (2 of 4)

```
with Track_Data_Package; use Track_Data_Package
separate (Sector_Processing_Controller.Sector_Processing_Task)
package Blip_Processor_Package is

  procedure Associate (Blip : in Blip_Records) is
    Track : Track_Type;
    Associated : Boolean := False;
  begin
    -- algorithm to associate and resolve (not shown here)
    if Associated then
      Smooth_Track.Get_Track (Track);
    else
      -- initiate new track (not shown)
    end if;
  end Associate;

  task body Smooth_Track is separate;
  task body Predict_Next_Point is separate;

end Blip_Processor_Package;
```

Figure 2: Association (3 of 4)

```
separate (Sector_Processing_Controller.
          Sector_Processing_Task.
          Blip_Processor_Package)
task body Smooth_Track is
  Local_Track : Track_Type;
begin
  loop
    select
      accept Get_Track (Track : in Track_Type) do
        Local_Track := Track;
      end;
    or
      terminate;     -- when there are no further tracks to smooth
                     --  and Associate has therefore stopped
                     --  making entry calls to Get_Track
    end select;
    -- do track smoothing
    -- update data base
    Predict_Next_Point.Get_Track (Local_Track);
  end loop;
end Smooth_Track;

separate (Sector_Processing_Controller.
          Sector_Processing_Task.
          Blip_Processor_Package)
task body Predict_Next_Point is
  Local_Track : Track_Type;
begin
  loop
    select
      accept Get_Track (Track : in Track_Type) do
        Local_Track := Track;
      end;
    or
      terminate;     -- when there are no further points to
                     --  predict and Smooth_Track has therefore
                     --  stopped making entry calls to Get_Track
    end select;
    -- algorithm to compute next point
    -- update data base
  end loop;
end Predict_Next_Point;
```

Figure 2: Association (4 of 4)

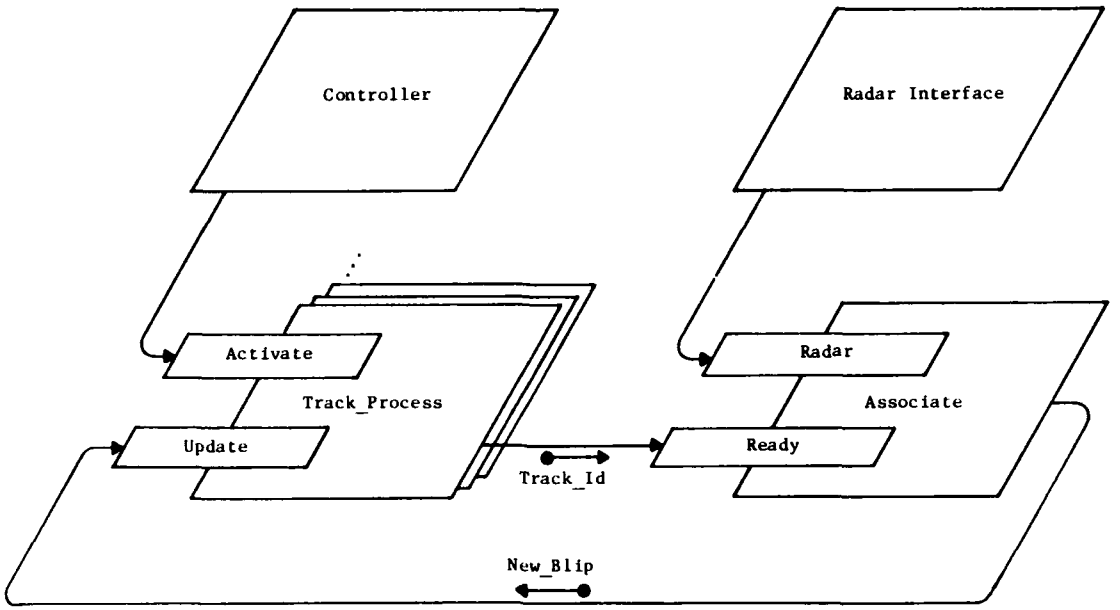


Figure 3: Tasking Design

```
type Track_Type;
type Track_Id_Type is access Track_Type;

task type Track_Process is
  entry Activate (Track_Id : Track_Id_Type);
  entry Update   (Blip     : Blip_Type);
end Track_Process;

type Track_Parameters_Type is
  record
    X, Y, Phi, Theta, Smooth_Index : Float;
  end record;

type Track_Type is
  record
    Ready      : Boolean;
    Process    : Track_Process;
    Parameters : Track_Parameters_Type;
    Next_Track : Track_Id_Type;
  end record;

Track_List, Track : Track_Id_Type;

task Associate is
  entry Radar (Blip     : Blip_Type);
  entry Ready (Track_Id : Track_Id_Type);
end Associate;

task Radar_Interface;    -- calls Associate with new blips
                         -- task body not shown
```

Figure 4: Task per Track (1 of 3)

```
task body Track_Process is

  My_Track_Id    : Track_Id_Type;
  New_Blip       : Blip_Type;
  Missed_Blips   : Natural;
  Miss_Threshold : constant Integer := <system-defined>;

  procedure Smooth_Track (Blip     : Blip_Type;
                          Track_Id : Track_Id_Type) is separate;
            -- not shown
  task Predict_New_Position is
    entry Get_Id (Track_Id : Track_Id_Type);
  end Predict_New_Position;
  task body Predict_New_Position is separate; -- not shown

begin -- Track_Process
  loop
    accept Activate (Track_Id : Track_Id_Type) do
      My_Track_Id := Track_Id;
    end Activate;
    select
      loop
        Associate.Ready (My_Track_Id);
        select
          accept Update (Blip: Blip_Type) do
                  -- received blip during this scan
            New_Blip := Blip;
          end Update;
          Smooth_Track (New_Blip, My_Track_Id);
          Predict_New_Position.Get_Id (My_Track_Id);
          Missed_Blips := 0;
        or
          Predict_New_Position.Get_Id (My_Track_Id);
          Missed_Blips := Missed_Blips + 1;
        end select;
      exit when Missed_Blips > Miss_Threshold;
      end loop;
    or
      terminate;
    end select;
  end loop;
end Track_Process;
```

Figure 4: Task per Track (2 of 3)

```
task body Associate is

  New_Blip          : Blip_Type;
  Best_Match, Match : Natural;
  Match_Threshold   : constant Integer := <system-defined>;
  Best_Track_Id     : Track_Id_Type;
  -- sector declarations
  function Degree_of_Fit (Blip     : Blip_Type;
                          Track_Id : Track_Id_Type)
                         return Natural is separate;
  procedure Initiate_New_Track (Blip : Blip_Type) is separate;

begin  -- Associate
  loop
    Track := Track_List;
    select
      accept Radar (Blip : Blip_Type) do
        New_Blip := Blip;
      end Radar;
      Best_Match := 0;
      while Track /= null loop
        if Track.Ready then  -- try to match blip to this track
          Match := Degree_Of_Fit (New_Blip, Track);
          if Match > Match_Threshold and
             Match > Best_Match then    -- best match so far
               Best_Match := Match;
               Best_Track_Id := Track;
          end if;
        end if;
        Track := Track.Next_Track;
      end loop;
      if Best_Match > Match_Threshold then
        Best_Track_Id.Ready := False;
        Best_Track_Id.Process.Update (New_Blip);
                  -- call track task
      else
        Initiate_New_Track (New_Blip);
      end if;
    or
      accept Ready (Track) do
        Track.Ready := True;
      end Ready;
    end select;
  end loop;
end Associate;
```

Figure 4: Task per Track (3 of 3)

FLOATING POINT COMPUTATION USING ADA'S MODEL NUMBERS

John J. Buoni and Richard L. Burden

Youngstown State University, Youngstown, Ohio 44555

The purpose of this paper is to point out the background of Ada's model numbers. By considering the Ada manual as a set of Postulates which an implementation of Ada must satisfy certain properties of model numbers are derived. It is the contention of the authors that only after one completely understands model numbers will the value of Ada's numerical approximation facilities be fully understood.

§1 Introduction

Ada defines the properties that approximates of real arithmetic computation must satisfy. Because real arithmetic is implemented on machines with different underlying hardware, the manner in which these properties are achieved is left to the implementation. The designers of Ada were faced with the problem of allowing different implementations and yet making the properties general enough for common numerical algorithms. The Ada design was based on W. S. Brown's work on floating point arithmetic.[2] However, differences between Ada and Brown's work abound since their goals were different. Brown was interested in producing a model of actual hardware; whereas with Ada, a machine independent implementation was required.

The purpose of this paper is to point out the background of Ada's model numbers especially Brown's work. It is the contention of the authors that only after one completely understands Brown's model will the value of Ada's numerical approximation facilities be fully understood.

Throughout this paper we shall adopt the notation

$$b\#0.d_1d_2\ldots d_n\#Ee \qquad (1.1)$$

where b will indicate the appropriate base, d_i for i=1...n will be digits in the base b, and Ee will mean that the base b is raised to the e power.

§2 Approximate Computation

Ada assumes that the arithmetic facilities are provided using binary numbers and allows the programmer to specify the minimal accuracy of a Real data type. For floating point this specification is an integer giving the minimal number of decimal digits of significance in stored values by the implementation and determines the floating point model numbers. Since a binary radix is assumed, then we define the floating point model numbers to consist of zero and all numbers of the form

sign*binary normalized mantissa*(2.0 ** exponent) (2.1)

where the mantissa length B and exponent range [-R...R] are determined from the number of decimal digits D by the formulas

$$B = D*\log(10)/\log(2) \simeq D*3.32$$

and

$$[-R..R] = [-4*B..4B].$$

Example 1. Suppose the Ada declaration

type F is digits 5;

is given. Then an easy calculation yields that $B = \lceil 16.6 \rceil = 17$ which is the predefined attribute F'Mantissa and that the exponent range is given by [-4*B..4*B] = [-68..68]. Hence, the smallest positive model number (the predefined attribute F'Small) is in the notation of (1.1)

$$2\#.1*E-68 = 2.0^{-69} \simeq 1.69*10^{-21}.$$

and the largest model number (F'Large) is

$$2\#.1111111111111111\#E68 = 2^{68}-2^{51} \simeq 2.95*10^{20}.$$

and the next model number greater than 1.0 is

$$2\#.1000000000000001\#E1 = 1.0+2^{-16}.$$

The predefined attribute F'Epsilon is the absolute value of the difference between the 1.0 and the next model number greater than 1.0; and in this case is about equal to $1.52*10^{-5}$ or in general 2.0**(-F'Mantissa + 1).

Remark: The fact that (F'Large)-1 > F'Small is useful in dealing with small numbers.

The advantage of the Model Number approach is that the model numbers are stored exactly, in the sense that if one assigns a model number to a variable, then one can test for equality and obtain the expected result. The safe numbers of Ada have the same number B of mantissa digits as the model numbers of that type and have an exponent range of -E..+E where E is implementation defined and at least equal to 4*B.[1]

Remark: The term normalized varies from architecture to architecture. The floating point architecture of the IBM 360/370 requires that the first (high order) hexadecimal digit be non-zero whereas the PDP-11 format adopted by the VAX-11 requires the first bit of the fractional part to be non-zero.

Since an implementation will typically have values which are not model numbers, the problem is to develop a model for the approximation of the numbers, some of these numbers are bounded by a model interval i.e. a real number interval whose endpoints are model numbers.

Definition 1. The model interval associated with a value that belongs to a real type is the smallest model interval that includes the value. Model intervals which contain no other model numbers

are called Atomic Model Intervals.

To illustrate this consider the following examples.

Example 2. Consider the model numbers whose mantissa length is 8 i.e. B = 8. Using hexadecimal notation we find that 16#0.A04 which is not a model number is bounded by the model interval whose endpoints are given by 16#0.A0 and 16#0.A1. Also we find that 0.1 which in hexadecimal is 16#0.1999...# is bounded by the model interval [16#0.198..16#0.19A].

Example 3. Consider the previous declaration of type F, i.e. type F is digits 5. With this type declaration, we have 17 binary places. The value 0.1 is bounded by the model interval [16#0.19999.. 16#0.1999A] where the hexadecimal numbers are used for convenience and are exactly equivalent to the normalized binary representation since the first three binary digits of the hexadecimal number 1 are zero. The width of this model interval is approximately $9.5367*10^{-7}$ which we saw to be less than F'Epsilon.

It is not hard to see that the Atomic model intervals vary in width. Indeed, the atomic interval [0.0..F'Small] has width of approximately $1.69*10^{-21}$ while the interval [1.0..1.0 + F'Epsilon] has width F'Epsilon which was found to be approximately $1.53*10^{-5}$. The following Theorem is then clear.

Theorem 1. The width of any Atomic interval is less than or equal to F'Epsilon.

Remark: The role that F'Epsilon plays is extremely important in the error analysis for model number computation. Hence the inequalities displayed in the following theorem are extremely important in scaling.

Theorem 2. $F'Small \leq (F'Epsilon)^2$,

and $F'Large \geq (F'Epsilon)^{-2}$.

These results lead us to the following theorem

Theorem 3. Let x_l and x_r be respectively the left and right endpoints of the atomic model interval containing the number x. Then

$$(1.0-F'Epsilon)*x \leq x_l \leq x_r \leq (1.0+F'Epsilon)*x$$

Hence, the relative error which we define as

$$\frac{|\ [x_l, x_r]\ - x\ |}{|x|} \leq F'Epsilon.$$

§3 Arithmetic Properties

Throughout the remaining parts of this paper we will indicate fl(x*y) to be the machine number result of x with y under the binary operation *. In a conventional model for floating point computation one begins by postulating the machine numbers (i.e. those numbers which are representable on a machine) as exactly the model numbers with floating point operations that are exact up to rounding or chopping. Then one obtains that

$$fl(x*y) = (x*y)(1+del) \text{ where } del \leq r|u| \quad (3.1)$$

where * varies over the binary operations; r varies from 1..4 depending on the type of arithmetic employed (round or chop); and u depends on the number of digits of accuracy.[4]

We wish to derive such a bound for computation of model numbers but first we must introduce some general properties.

Definition 2. Any machine number x is said to be F-bounded if and only if $|x| \leq$ F'Large. An interval X is said to be F-bounded iff every x in X is F-bounded. A machine number x is said to be in F'Range if and only if x = 0 or $F'Small \leq |x| \leq$ F'Large. Adopting the notation of Brown, for x any F-bounded machine number, we denote by x' the Atomic model interval containing x. Furthermore, if X is any F-bounded interval then X' is the smallest model interval containing X.

The Ada Postulates for floating point operations (+,-,*,/) are stated in Ada as follows:[1]

Postulate 1. The result model interval is the smallest model interval (of the result subtype) that includes the minimum and the maximum of all the values obtained by applying the (exact) mathematical operation, when each operand is given any value of the model interval (of the operand subtype) defined for the operand.

Postulate 2. The result model interval is undefined if the absolute value of one of the above mathematical results exceed the largest safe number of the result type. Whenever the result model interval is undefined it is highly desirable that the exception NUMERIC ERROR be raised if the implementation cannot produce an actual result that is in the range of safe numbers.

The floating point operations may be summarized as follows:

(1.) For each operand, a model interval of the appropriate type or subtype is obtained.

(2.) The mathematical operation is performed on the model intervals, obtaining a new interval.

(3.) The interval from the last step is expanded to a model interval. The model interval attained from the last step bounds the accuracy of the result.

Consider the following example:[5]

Example 4. Consider the following Ada declaration:

```
type F is digits 5;
x,y:F;
```

We wish to compute x*y, where x=0.1 and y=10.0.

Step 1: says that x is the model interval [16#0.19999..16#0.1999A] while y is the model number 16#0.A#E1.

Step 2: then gives the interval [16#0.FFFFA.. 16#1.00004].

Step 3: then gives the slightly larger model interval [16#0.FFFF8..16#1.0001].

Example 5. We wish to consider x+y, where x=1.0 and y=F'Small.

Step 1: x and y are model numbers so no new intervals are constructed.

Step 2: the interval at this step is the machine number 1.0+F'Small.

Step 3: from previous computations we see that this number is then expanded to the Atomic model interval [1.0..1.0+F'Epsilon].

Unfortunately, machine anomilies play an important role in any computation as well as the model number computation as the following example indicates.

Example 6. Consider a three decimal digit computer with no guard digit in its accumalator. We shall stipulate a mantissa length of 3. Such a machine would probably compute 10#1.00 * 10#.999 in the following manner.

Step 1: would normalize the numbers and then multiply in the following manner 10#.100E1 * 10#.999.

Step 2: would yield .099*10 or .990.

Step 3: would just return the result of Step 2 which is a seemingly unacceptable result because .999 (the actual result) does not lie in the model interval.

On the otherhand, with the mantissa length set to 2 affording us the luxury of extra machine precision in its model numbers. We find that the steps in that same calculation are:

Step 1: would compute the model interval .0999' which is [.99 1.00].

Step 2: would form the product of 1.00 times [.99 1.00] which would yield the interval [.99 1.00].

Step 3: would return the result as in Step 2 since it is a model interval. This model interval now does contain the answer of .999.

Remark: the same phenomena would result if we replaced the above problem with the subtraction problem 1.00 - .999 and leads one to conjecture the necessity of a guard digit in an Ada declaration. Additional insight into this phenomena will be given in Example 9.

Example 7. With the same computer in mind as in Example 6 and mantissa length set at 3 consider the computation of 1./x where x=9.0.

Step 1: would normalize the model numbers 1.0 and 9.0.

Step 2: would perform the reciprocal operation to yield the normalized result 10#.111E0.

Step 3: would return the result as in Step 3 since this result is a model number.

On the otherhand, if we had chosen the mantissa length to be 2 then the steps would proceed as follows:

Step 1: would normalize the model numbers 1.0 and 9.0.

Step 2: would perform the reciprocal operation to yield the normalized result 10#.111E0.

Step 3: would yield the model interval [.11.. .12].

The following result paraphrases the Ada Postulates 1 and 2.

Theorem 4. Let x and y be F-bounded machine numbers, and let * be any binary operator. Then fl(x*y) is an element of (x'*y')' provided x'*y' is F-bounded.

Some immediate consequences of these results are the following:

Theorem 5. Let x and y be model numbers and let * be any of the binary operations addition, subtraction and multiplication. If x*y is a model number then fl(x*y)=x*y.

Remark: Although the reciprocal of a model number is not a model number one finds that if a number is a power of 2 than it's reciprocal is also a model number.

§4 Error Bounds

We are now interested in deriving error bounds for various binary operations which is to say that all operations may be considered accurate to within F'Epsilon, whenever the operands themselves are F-bounded machine numbers with an F-bounded result. If the operands are model numbers whose product is also a model number then Theorem 4 supports this claim without any further qualification. However, if an operand x is not a model number, then the operation may effectively replace it with a different value $\hat{x}$ in x', and the relative error of the computed result will be small if the exact result is obtained from $\hat{x}$ rather than x. Since x and $\hat{x}$ lie in the same model interval x', then we may view $\hat{x}$ as being equivalent to x. Hence, the exact position of x and $\hat{x}$ are irrelevant.

Theorem 6. Let x and y be F-bounded real numbers and let * be a binary operation. In computing x*y, let $\hat{x}$ and $\hat{y}$ be the effective values in x' and y' for x and y respectively. Further suppose that (x' * y')' is in F'range. Then for every $\hat{z}$ in (fl(x*y))' there is a δ such that

$$\hat{z} = (\hat{x} * \hat{y})(1+\delta) \text{ where } |\delta| \leq \text{F'Epsilon} \qquad (4.1)$$

Proof. By Theorem 4, fl(x*y) is in (x'*y')'. If fl(x*y) is in x'*y' then by the definition of an interval operation there exist $\hat{x}$ in x' and $\hat{y}$ in y' such that $fl(x*y)=\hat{x}*\hat{y}$. If fl(x*y) is in (x'*y')'-(x'*y'), that is fl(x*y) is in u', where u is an endpoint of x'*y'. Now choose $\hat{x}*\hat{y}$ such that $u=\hat{x}*\hat{y}$. Then fl(x*y) is in $(\hat{x}*\hat{y})'$. Hence, for any $\hat{z}$ in (fl(x*y))' we find that $\hat{z}$ is in $(\hat{x}*\hat{y})'$. The result now follows from Theorem 3.

§5 Arithemetic Comparisons

In performing an arithmetic comparison, great care is required, since any error in either operand may reverse the result. Nevertheless, the result does convey information, which can be made precise by analyzing the possible error in each operand and then using the results of this section.

Postulate 3.[1] For the result of a relation between two real operands, consider for each operand the model interval (of the operand subtype) defined for the operand; the result can be any value obtained by applying the mathematical comparison to values arbitrarily chosen in the corresponding operand model intervals. If either or both of the operand model intervals is undefined (and if neither of the operand evaluations raises an exception) then the result of the comparison is allowed to be any possible value (that is TRUE or FALSE).

This postulate immediately yields the following theorem:

Theorem 7. Consider a comparison of two F-bounded machine numbers x and y. Let Z be the closed interval [x,y] if x < y or [y,x] otherwise. If there are at least two model numbers in Z, then the correct result is reported by Ada. If there is exactly one model number in Z, then Ada may report either the correct result or that x = y. Finally, if there are no model numbers in Z, then the report is implementation dependent.

Proof: Assume that Z = [x,y]. If Z contains at least two model numbers then Z must contain y_l (the left hand endpoint of y') and x_r (the right hand endpoint of x') with $x_r \leq y_l$. Hence the intervals x' and y' are disjoint and the result now follows from the Postulate.

If Z contains exactly one model number then we find that Z contains x_r which is equal to y_l. Therefore the postulate indicate that the correct result may be returned or that of equality if the two arbitrary points in the model intervals x' and y' that are chosen are x_r and y_l respectively.

Finally, if Z contains no model numbers then x' and y' must intersect at more than one machine number and hence the report is implementation dependent. This completes the proof of the Theorem.

Example 8. For the Ada declaration

type F is digits 5;

consider the comparison of the machine numbers x = .1 and y = .1 + F'Epsilon/16. Recall that x' is given by the model interval [16#0.19999..16#0.1999A] while y' may be found to be [16#0.1999A..16#0.1999B] in which case the mathematical result is returned or that of comparing the equal endpoints. Consider also the comparison of .1 with itself. In this case the interval Z = [.1]; hence does not contain a model number. Therefore, this result is implementation dependent.

Example 9. To illustrate the importance of "guard bits" consider approximating a root of $f(x) = x^3 - 1.73x^2 + .641x - .0584$ using a bisection search algorithm. The polynomial f has a root in the interval [.2,1] at $p \simeq .33378$. The algorithm begins with model intervals approximating .2 and 1.0, say a_o' and b_o'. At each step a model interval $p_i' = (\frac{1}{2}(a_i' + b_i'))'$. is formed from model intervals a_i' and b_i' where $(f(a_i'))'$ and $(f(b_i'))'$ are of opposite sign. Depending on the sign of $(f(p_i'))'$ new model intervals a_{i+1}' and b_{i+1}' are formed and p_{i+1}' computed. The procedure continues until the sign of $(f(p_i'))'$ cannot be determined, correctly, as positive or negative. The model interval corresponding to p_i' is then reported as the approximation. Results using N bit mantissa where N = 9,10,11 are given in the following table.

N:	Best Possible Model Interval
9	[.101010101, .101010110]
10	[.1010101011, .1010101100]
11	[.10101010111, .10101011000]
N:	Reported Model Interval
9	[.101001101, .101010000]
10	[.1010110100, .1010111001]
11	[.10101010011, .10101010111]
N:	Corresponding Decimal Interval
9	[.3252, .3281]
10	[.3379, .3403]
11	[.3328, .3337] .

If the number of decimal digits selected were 3, then minimally a 10 bit mantissa is required. This example illustrates the improvement obtained by using an 11 bit mantissa. The example further illustrates a source of possible confusion. If one has a machine with an 11 bit mantissa and selects 3 decimal digits, the error between the actual answer and our approximation is less than .001. If a machine with a 10 bit mantissa is used and 3 decimal digits are selected, the error is at least .004. One could be lead to false conclusions regarding the accuracy obtained using 3 decimal digits. The accuracy obtained involves an important trade-off between the number of decimal digits selected and the number of guard bits available. Subsequent work of the authors will investigate this further.

References

[1.] American National Standard. "Reference Manual for the Ada Programming Language", 1983.

[2.] Brown, W.S., A Simple but Realistic Model of Floating Point Computation ACM Transactions of Mathematical Software, v7, pp. 445-480, 1981.

[3.] Moore, R. E., "Methods and Applications of Interval Analysis" SIAM Studies in Applied Mathematics, Philadelphia, Pa., 1979.

[4.] Vandergraft, J. S., "Introduction to Numerical Computations", Academic Press, New York, N.Y., 1983.

[5.] Wichmann, B. A., "Tutorial Material on the Real Datatypes in Ada", Centacs Fort Monmouth, N.J.

J. J. Buoni, Professor
Youngstown State University
Youngstown, Ohio 44555

Richard L. Burden
Youngstown State University
Youngstown, Ohio 44555

DEVELOPMENT OF AN ADA* PACKAGE LIBRARY

Dr. Bruce Burton and Mr. Michael Broido

Intermetrics, Inc.
Aerospace Systems Group
5312 Bolsa Ave
Huntington Beach, California 92649

ABSTRACT

A usable prototype Ada package library has been developed and is currently being evaluated for use in large software development efforts. The library system is comprised of an Ada-oriented design language used to facilitate the collection of reuse information, a relational data base to store reuse information, a set of reusable Ada components and tools, and a set of guidelines governing the system's use. The prototyping exercise is discussed and the lessons learned are presented. Our experiences in developing the prototype library and lessons learned from it have led to the definition of a comprehensive tool set to facilitate software reuse.

INTRODUCTION

With the rising demand for cost-effective production of software, software reuse has become increasingly important as a potential solution to low programmer productivity. In the Ada programming language, explicit support is provided for software reuse through the "package" and "generic" language features. Unfortunately, the concept of Ada software reuse is not a panacea for our current software productivity problems. The notion of software reuse has been popular for decades. But implementing high degrees of reuse has usually failed, with the exception of some efforts in fairly narrow areas (business and compiler applications). The challenge then, is to recognize the contributions that the Ada language can make to a software reuse effort while at the same time identifying and resolving language-independent problems. Based on the promise of the Ada programming language we undertook the development of a prototype Ada package library.

The prototyping exercise included:

- an examination of the reasons for low software reuse in the past,
- identification of activities and tools which would support a reuse methodology that spans the software development life-cycle from requirements through maintenance,
- the development of a phased implementation plan for software reuse that defines a development path from prototype to an operational, multi-company, geographically distributed system,
- development of a prototype for that methodology,
- the development, acquisition, and evaluation of representative package entries, and
- an examination of user interface techniques that could be used to maximize communications between a reuse system and its users.

BACKGROUND

As discussed above, software reuse is not a new concept. Significant efforts have been underway since the early 1960's to improve software development productivity through reuse (consider the early observations of McIlroy about the benefits of reuse presented at the NATO Software Engineering meeting in Garmish in 1968) [STANDISH83]. An analysis of the problems attending reuse has led to the identification of several potential hindrances to reuse [STANDISH83, BROIDO85]. These impediments to reuse can be categorized as technical, economic, and political obstructions. Some typical problems that hinder reuse include:

* Ada is a trademark of the U.S. Department of Defense (AJPO).

- lack of universal standards for component composition, level of documentation, coding techniques, testing, etc.,
- difficulty in transferring an understanding of the purpose of a software routine from the author to the potential reuser,
- higher initial development costs and longer schedules,
- risk management issues such as warranty, liability, and accountability,
- the "not invented here" syndrome, and
- the lack of pride typically exhibited when reuse has been selected in a software development project over original development.

While the problems impeding reuse are significant, the large size and cost of a major software development effort provides substantial motivation to improve productivity through reuse. Although Ada provides a natural vehicle for encouraging software engineering reuse, the same technical and political obstructions that have limited reuse in the past are likely to once again impede the sharing of software engineering products across projects. The Software Technology Department within Intermetrics is actively investigating the problems that hinder reuse. We are determined to find solutions to these problems and to collect and reuse Ada packages.

Table 1. Overview of a phased development of a reusable package library.

Phase	Activity
1. Analysis and requirements Definition	- Identify characteristics of previous software libraries - Identify information unique to Ada packages necessary for the collection of Ada packages
2. Prototype software catalog	- Design and implement prototype Ada software catalog (ASCAT) - Catalog initial holdings - Evaluate ASCAT and user interface
3. Automate ASCAT/CMS interface	- Automate the interaction between the software catalog and the Configuration Management System (CMS)
4. Integrate support tools	- Automate examination of submitted Ada packages through incorporation of support tools to ensure adherence to submission standards
5. Expand the user community	- Focus of package reuse system will shift from a passive one where entries flow from users to software catalog to one where EMAIL would be used to provide automatic system/user communication
6. Automatic ASCAT/Software library interaction	- Fully automate the interaction between the CMS ASCAT and the program library
7. Provide multi-site and multi-company extensions	- Add provisions within the system to handle distribution, licensing, use restrictions

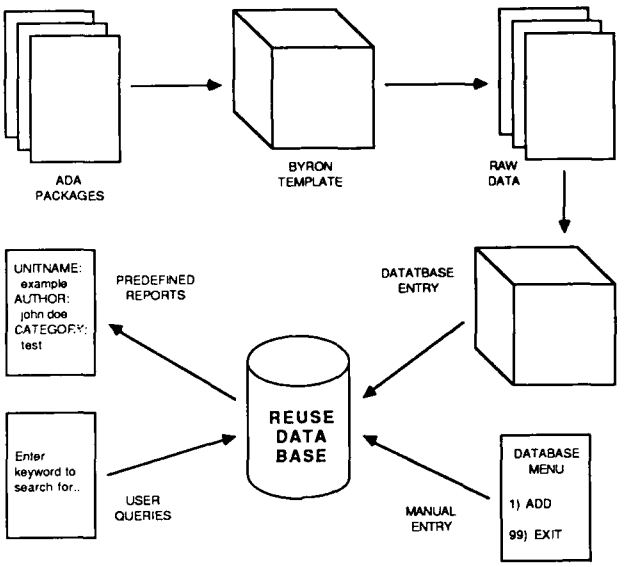


Figure 1. Reuse process overview.

APPROACH

Along these lines, we have defined a phased approach to the development of a reusable package library suitable for use on large Ada applications projects. Rather than define an elaborate reuse facility and implement the library in a single step, we are currently prototyping parts of this facility to investigate the potential utility of our approach. An overview of this phased development plan is shown in Table 1; a more complete view is offered in [BURTON85]. The initial effort on this project has been focused on the creation of an Ada Software CATalog (ASCAT).

An overview of the ASCAT portion of the Ada package reuse system is shown in Figure 1. The system has been implemented using Byron*, Intermetrics' Ada-based program design language, and a commercial relational database management system. Central to the system is the ability of Byron to support definition and use of user-defined keywords.

* Byron is a trademark of Intermetrics, Inc.

Software Classification and Data Element Selection

One key to the success of any reuse scheme is the types of classifications assigned to entries. The primary purpose of these classifications is to facilitate retrieval, but they may also be used to assist in defining storage strategies as well.

Selecting the classifications to be used is really a subset of a larger question: what data elements do we want to be able to retrieve about a particular entry? The list of storable elements seems in our opinion to be highly influenced by the size of the library (number of program units stored) and the degree of cooperation (or potential antagonism) among the users of the library. An initial cut at such a list was prepared [BROIDO85] from the perspective of our ultimate (multiple sites, multiple organizations, multiple usage types) system. Over 60 items which could potentially affect the suitability of an entry were named in seven major categories: identification (3 items), description (16 items), component parts (20), environment/usage (9), ordering information (7), and revision history (11). Even at this length, we recognize that there are undoubtedly many other items which could be added.

This list was far too large for our prototype, so we examined the context in which the prototype would operate. We characterized our initial environment as follows:

- All the users would be from the same company, although there would be several divisions using the common library. Thus, no restrictions on access would need to be supported.
- All initial entries would be written (when possible) in machine-independent Ada, so the compilation and execution environments would be well-defined.
- Source code would always be available, so users could do their own tailoring (no "black boxes"). Support in the form of corrections and training (other than by reading the source code) would not be provided.
- Emphasis was centered around the collection of reusable Ada packages rather than complete programs. Two factors influenced this decision. The first is that most of the packages we wanted to include already existed prior to the start of our efforts, and coherent design documents were not always available. The second factor was the widely distinct set of users we were addressing; they do not share the commonality of purpose which makes domain analysis an effective top-down approach. The decision to center our design on packages enabled us to define a standard header for each package, based on the requirements of our Byron program product. Formalized requirements and design documentation were not required.

  This decision causes the library to be more supportive of "bottom up" software construction techniques than most of today's top-down methods. The top-down methods reflect an attitude of defining what would be a perfect system and do not adequately recognize the influence of existing tools (including code) should have on requirements formulation in the presence of real cost constraints. (Note that the "object oriented design" strategies that are emerging with Ada reflect a tendency away from strict top-down methods.)
- No a priori naming conventions were established, although an informal guideline was prompted by the technical monitor of one of the contributing programs.
- Configuration management was not rigidly enforced, except within the rules imposed by Ada. In particular, no computerized list of outstanding users (people or programs) of the library routines was maintained.
- The programs which were intending to take advantage of the library provided no explicit funding for tool support or to ensure that any new packages created were generalized and otherwise suitable for future reuse. Package headers and other programmer-supplied information had to be easy (in both time and difficulty) for the programmers to supply.
- Various standards were established for the data items we would collect. Since we were attempting to catalog packages which had been previously created to support several different projects, It was necessary to retrofit many of the selected packages to include the required Byron comments. Part of our evaluation will be to try to identify the difficulties caused by "loose" definitions of essentially narrative fields (e.g., overviews). In addition, no common

methodology had been established, so the degree of formality and the list of available support items (repeatable test cases, previous sample output, user documentation, etc.) also varied considerably.

We filtered the original list down to the following data items for the database (others, such as the calling conventions and parameters, would be available from the source code if not given in the overview):

1. Unit name
2. Author
3. Unit size
4. Source language
5. Date created
6. Date last updated
7. Category code (see below)
8. Overview
9. Algorithm description
10. Errors/exceptions generated
11. Up to 5 keywords (for retrieval)
12. Machine dependencies (if any)
13. Program dependencies (if any)
14. Notes

Our retrieval strategy was based upon a combination of two alternate mechanisms. The first mechanism was the assignment of a hierarchical category code, with the hierarchy defined ahead of time and changeable only at well separated time intervals. This scheme is similar in concept to the ones used by Computing Reviews [ACM85] and the IMSL library [IMSL76]. But it was necessary to invent our own classification scheme since neither of those two was suitable to our purposes. Our scheme has the advantage that everyone knows what the codes are and can use an effectively finite procedure for searching the entries. Disadvantages include a growing list of vastly dissimilar "miscellaneous" entries and the inability of the original hierarchy designers to provide sufficiently discriminatory categories to provide effective retrieval (not too many or too few candidates).

For the second mechanism, we allowed the submitters to supply up to five keywords to be associated with each package. These keywords are not associated (as implicitly occurs within the hierarchy of categories), allow for overlapping topics (the packages do not conveniently fall into strict tree classifications), and can grow (without reprogramming or an all-knowing database administrator) with the needs of the projects they are created for. A scheme similar to this has been employed on NASA's COSMIC (Computer Software Management Information Center) system on complete programs, although the keywords allowed are suggested by the program authors and filtered by an acceptance team.

One of the authors is a member of the Applications Panel of the Department of Defense's Software Technology for Adaptable, Reliable Systems (STARS) Program. An important open issue surrounding the formation of a potential Ada package library to be available as GFE materials for DoD contracts is defining the quality of the entries. On the one hand, some people advocate including only items of the highest quality, with full DoD standard documentation and even formal independent validation and verification (IV&V) required on new entries. Others prefer to let a more flexible scheme apply, with a "trust level" associated with entries. This latter scheme encourages "promotion" of existing entries from "buyer beware" to higher trust levels; after all, using informally qualified designs and code and then adding formal testing and documentation can still take less time (and often risk) than inventing from scratch. For the prototype, we decided to let all submitted entries be accepted and then evaluate the impact of this decision.

Reuse Information Extraction Mechanism

Another critical phase in the development of an Ada package library involves the extraction mechanism used to collect reuse-oriented information. The extraction mechanism utilized in an Ada package library must eventually provide several different capabilities to insure efficient operation. These required capabilities include:

- support for automatic data collection,
- support for insuring standardization of data entries,
- support for assuring continuity and consistency of reuse information across the Software Development Life Cycle (SDLC),
- support for checking completeness and reasonableness (e.g., dates), and
- support for reuse information examination.

The reuse information extraction approach utilized in our Ada package library is detailed in Figure 2. An analysis of this figure reveals that each of the elements previously identified for data collection has been mapped into

predefined or user-defined keywords for the Byron design tool. A Byron template program was subsequently developed to automatically extract the reuse-oriented information. This information is placed into a file that can be directly processed into the ASCAT data base.

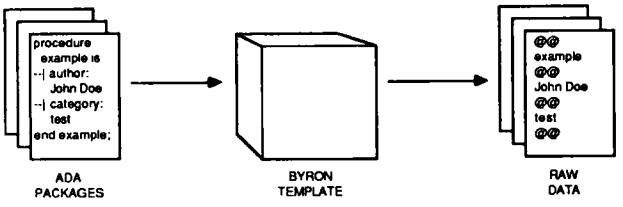


Figure 2. Extraction mechanism overview

The use of a Byron-oriented reuse information extraction mechanism provides most of the required capabilities enumerated above. This approach provides a means for automatic collection of data standardized in field name and format. Since the Byron design file is intended to transition into the implementation with reuse data intact, support is offered to assure information continuity across multiple phases of the SDLC.

While this extraction approach has many positive features, it is not without its shortcomings. The lack of predefined reuse attributes within Byron fails to support direct examination of reuse data items for completeness, consistency, and reasonableness. The inclusion of reuse-oriented information into the Byron-produced program library represents a simple potential improvement to our approach that could aid in the examination of the reuse data items.

Software Catalog Implementation

The software catalog for the reusable package library was implemented through the use of a commercial relational data base management package. The data definition capability used for field definition and the built-in data base programming language facilitated the examination of reuse data for limited correctness and consistency checking. The use of a data base also aided in the rapid development of an interface between the software catalog and potential Ada package users through the utilization of predefined reports and support for ad hoc user queries. Nonetheless, the user interface represents a weak link in our prototype package library. The present interface is very limited in the sense that it offers no context-specific support for communication between the reuse system and its users.

The present software catalog is limited in its interaction with the user. For example, consider the scenario of a software engineer performing an application software design of a routine that requires a sorting package. In the present system, the software engineer would need to: 1) exit the editor, 2) enter the software catalog data base system, 3) enter a query to identify the available sorting packages, 4) select the desired package, and 5) re-enter the editor and issue the necessary commands to draw the desired package (design/code) into the applications program design.

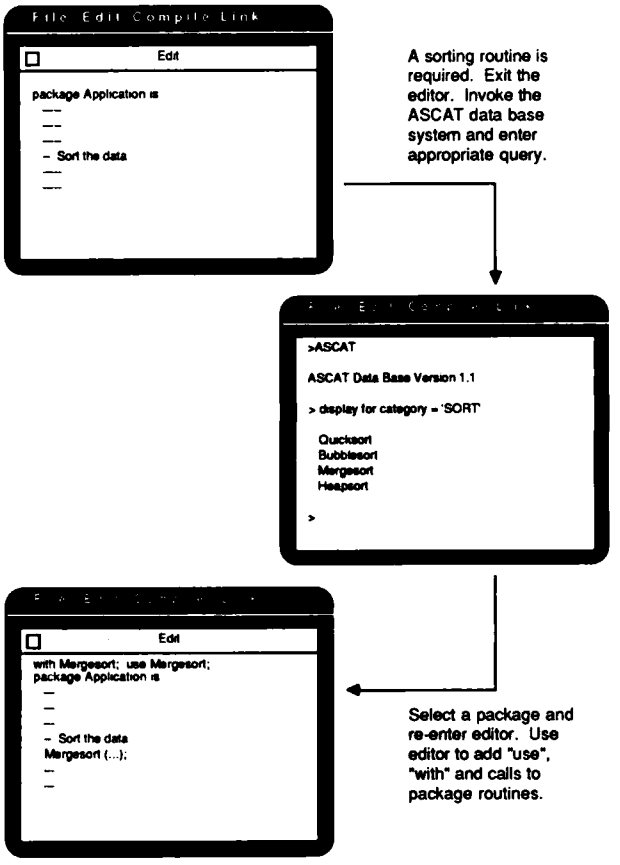


Figure 3. Current ASCAT operational scenario

This initial prototype software catalog can readily be improved to enhance the way in which it interacts with user. In Figure 3, the present mode of interaction is depicted. In Figure 4, another potential scenario is shown. In this scenario, a multi-window environment

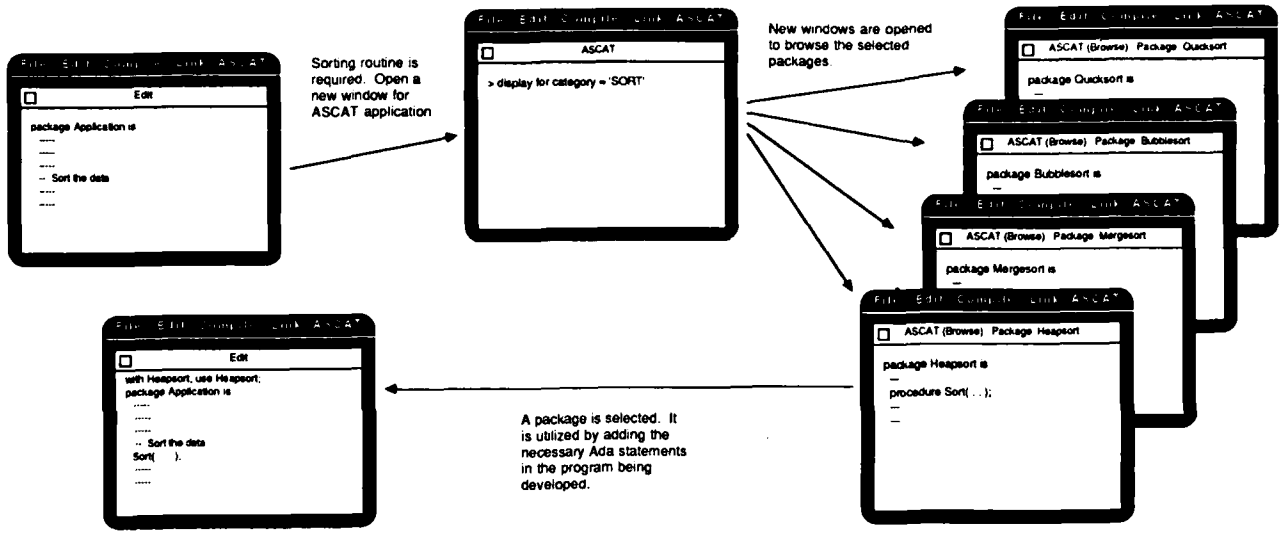


Figure 4. Improved ASCAT operational scenario

is used where the user may perform the software catalog inquiry and concurrently examine several promising packages without exiting the editor.

A third possible operational scenario of the software catalog is not pictured. In this third approach, the data base query language would be replaced by a natural language front-end, the software catalog search would be assisted by an expert system, and the multi-window approach would be supported by a language- and context-sensitive editor. The third approach is feasible with investigation into its implementation occurring in several current projects [ANDERSON85].

LESSONS LEARNED

The development, collection, evaluation, and cataloging of reusable components and tools undertaken in the development of an Ada package library has led to some interesting observations concerning Ada package reuse. Unfortunately, we do not yet have enough experience to evaluate the selected category scheme, keyword retrieval capability, or the list of collected data elements.

During the past year, we have developed a set of test and analysis tools written in Ada and intended for Ada software development efforts. The fixed-price nature of this contract and the fact that it represented the first major Ada development contract within our division motivated us to emphasize reuse of existing Ada packages as a cost and risk reduction measure. Based on the results of that contract, we found that reuse of existing generic support packages significantly improved our productivity, with over 33% of the code comprised of reused packages.

On the negative side, we found that several of the tools initially exhibited poor performance. In almost every instance, we found the general nature of the reused packages to contribute heavily to the performance problems. We also found that the generic Ada packages offered much more functionality than required in our application. The extra functionality resulted in a size penalty with respect to the executable code. The use of a performance analyzer and tailoring of the reused code for the current application substantially improved tool performance [RATHGEBER86].

We also studied the problem of composing reusable applications packages from existing reusable components. As part of an Air Force study, we compared the performance of two different implementations of reusable Kalman filter routines. One of the routines was written in Ada; generic Ada mathematics packages were heavily used in its development. The other routine was written in FORTRAN and specifically designed to solve a specific Kalman filter problem. A performance comparison of the generalized Ada package against the custom-tailored FORTRAN

routines showed the FORTRAN routine to exhibit significant speed advantages over its Ada counterpart. This performance difference is probably due to the relative immaturity of the Ada compiler used in this study and also to the generalized nature of the Ada packages. An important conclusion of the study is that the performance problems associated with including a generalized reusable Ada package into an applications program are substantially compounded when an entire system is comprised of reusable components which also consist of reusable components.

Although many of our lessons learned have negative implications for the use of Ada reusable packages, there is some light at the end of the tunnel. Reuse was a big aid in increasing our productivity in the development of Ada test and analysis tools. We also found that reuse can be successfully employed in the development of efficient Ada systems if sufficient thought is put into how the packages are to be reused and if the proper tools are available (e.g., such as a performance analyzer).

TOWARD A COMPREHENSIVE REUSE METHODOLOGY

A software development methodology which supports extensive reuse may be quite a bit different from the methodologies now popularly employed (top-down, structured designs, chief programmer teams, data flow analyses, HIPO charts, etc.). These systems are not well suited to achieving high levels of software reuse. These methodologies focus only on deriving systems almost exclusively from the requirements, without much regard for components which may already exist. The development of these systems depends largely on the individual participants to know when existing components can fully or partially satisfy the needs of the current project. When partial reuse is achieved (i.e., reuse with modification), the degree of reuse (and two-way traceability) is not recorded in any systematic way.

Reusability can be applied in every phase of the Software Development Life Cycle. New tools are needed in each one to support the activities involved.

Requirements Phase

Major new systems are rarely created from scratch any more. Major subsets of the requirements for new systems build upon the experiences gained with the old ones. As a minimum, we have come to expect high degrees of reliability and user friendliness in any new products.

Just as we express delivered products as complete programs, we tend to express requirements in single, monolithic specification documents. In order to support high degrees of reuse during the requirements phase, we need tools which allow us to express sets of requirements in small, reusable groups, just as we now compose programs from subroutines which are individually controlled. Refining requirements into sets in turn means that we need a new nomenclature: with the new identification scheme, traceability would not be tied to paragraph numbers which vary from product to product. But also in accordance with the subroutine analogy, composition paradigms are needed to establish compatibility (or at least minimize contradiction) among requirements sets.

Reuse also supports the rapid development of prototypes for evaluating user interface and time-line analyses.

Preliminary Design Phase

This is the phase during which the highest life cycle pay-off is likely to occur from improved levels of reuse. The overall system architecture and the interfaces between major elements are frequently the most stable parts of a system, and hence the ones around which the detailed changes are molded. Even in the face of implementation differences, new systems and upgrades tend to mimic the architecture of the original. (After all, look at the common functional partitioning among the compilers, operating systems, etc., produced by each of the mainframe and minicomputer vendors.)

The major contribution of a reuse methodology to the preliminary design phase is to help make the designer aware of the contents of the tool box that is available. Reinvention in software is often a waste of resources, and it is a major contributor to the low levels of productivity now achieved. As reusable sets of requirements are developed, designs which implement them (even if only to the level of Ada generic packages) can serve as significant building blocks. New design tools are needed which explicitly partition programs into reusable and application-specific parts. Design documents must also provide more of the designers' thought processes to future users: rationale for choosing one design over its alternatives should be documented. Also missing from most current design documents is a list of "invariants": those elements of a design which are "always" to be true, even in the face of the most likely requirements changes. Design tools which allow a user

to specify "just like that one, except ..." can be built on systems which support the decomposition of programs into reusable components with well-defined interfaces.

Rapid prototyping, as also described under the requirements phase, can help identify the proper sequencing of major elements so that architecture validation can be performed.

Detailed Design & Implementation Phases

Tools that can be used during the detailed design and implementation phases include context-sensitive editors which assist the user in selecting an appropriate package for inclusion (knowing such parameters as what has already been selected for this program, the intended levels of optimization and error checking, host/target computer selections, invocation of an Ada generic package, etc.). Other helpful tools include interfaces with the configuration management system to formally record inclusion of existing packages into a new program, logging modification histories so that an error found in one can be traced to all its "relatives", and performing static analyses to identify deviations from standards established for potential new entries for the reuse catalog.

When changes to library entries are made and formally approved, the configuration management system can aggressively (via electronic mail or other network communications) notify users that changes to library routines have been made and the users may wish to include the updates in their programs.

Integration and Test Phase

Reuse supports the integration and test phase by supporting the definition and generation of formal and informal test cases. It can also simplify performance tuning by allowing an analyst to select alternate algorithms from the stored library to adjust such parameters as execution speed, static or dynamic memory utilization, degree of internal checking that is performed, etc. It can also support the rehosting (if necessary) of programs from the development computer onto the intended target computer.

Maintenance Phase

Reuse during the maintenance phase is the best example of reuse, but it is so obvious that it is not even recognized as such. Rarely are programs discarded and reimplemented when the first bug is discovered. Regression testing represents reuse of the original qualification tests, and may be implemented using a standardized mechanism for preparing, executing and analyzing test cases.

CONCLUSIONS

In accordance with our previous plan, we have completed a prototype mechanism for extracting reuse information from packages developed in the normal course of business. We also have a primitive mechanism for entering that data in a catalog and searching the catalog for entries that are potentially useful on new projects. The approach centers on the design and implementation phases, since these are the ones to which reuse concepts may most readily be applied in the given environments.

We have confirmed with actual experience our earlier assessment that successful implementation of a reuse methodology requires thought, action and management direction and support throughout the software life cycle. This, however, may require a management reorientation to the view of software development as the acquisition of a long-lived corporate asset rather than as only the work required to produce the current deliverable [WEGNER84, YEH85]. Complementing the reuse efforts being conducted by the STARS office, which are targeted at long range objectives, our approach provides useful tools which can be utilized immediately.

We have achieved some success in applying software reuse. Effective use of the packages forced us to define subsets of them which subsequently required performance tuning. This points out the value of developing a comprehensive reuse methodology, with adequate support tools to facilitate the development of efficient systems comprised of reusable components.

The Ada language and the methodologies growing up around it provide a good start toward achieving larger scale reuse than we have achieved in the past. But they are not enough by themselves. Even with Ada, there are still plenty of obstacles to reuse. A management commitment and desire to improve productivity when coupled with a comprehensive reuse methodology and the proper tools offer substantial promise for improvement.

REFERENCES

ACM85 "Introduction to the CR Classification System," Computing Reviews, Vol. 26, No.1. Association for Computing Machinery, January, 1985, pp. 45-57.

ANDERSON85 Anderson, C.M. and Mc-Nicholl, D.G., "Reusable Software - A Mission Critical Case Study", AIAA Computer in Aerospace V Conference, October 21-23, Long Beach, California.

BROIDO85 Broido, Michael D., "Software Commonality Study for Space Station Phase B", Intermetrics Report IR-CA-029, Intermetrics, Inc., 29 May 1985.

BURTON85 Burton, B.A. and Broido, M.D., "A Phased Approach to Ada Package Reuse", STARS Workshop on Software Reuse, April 9-12 1985, Naval Research Laboratory, Washington, DC 20375-5000.

IMSL76 Reference manual, The International Mathematical & Statistics Libraries, IMSL, Fall, 1976.

RATHGEBER86 Rathgeber, R.L., "Technical Report on Ada Test and Analysis Tools", Intermetrics, Inc., Huntington Beach, California, In Preparation.

STANDISH83 Standish, T.A., "Software Reuse", presented at the ITT Workshop on Reusability in Programming, Rhode Island, September 7-9, 1983.

WEGNER85 Wegner, Peter, "Capital-Intensive Software Technology," IEEE Software, Vol. 1, No.3, IEEE Computer Society, July, 1984, pp. 7-45.

YEH85 Yeh, Dr. Raymond T., "Japanese and Brazilian Software Technology Initiatives". (Luncheon address). Published by the NSIA Software Committee in the Proceedings of the First DOD/Industry STARS Program Conference, 30 April 1985 - 2 May 1985.

AUTHORS

Dr. Bruce Burton is the manager of the Software Technology Department at Intermetrics Inc., where he has worked since 1981. He holds an M.S. in information and computer science and a Ph.D in physical chemistry from the University of California, Irvine. Dr. Burton's department is responsible for the investigation of software development problems that hinder the cost-effective construction of reliable software. The specific areas addressed by current research include Ada software reuse and real-time programming issues in Ada.

Michael D. Broido received the B.S. degree in mathematics from the California Institute of Technology in 1970, and the M.S. degree in computer science from the University of Southern California in 1973. He has been with Intermetrics since 1983 and has been involved in improving software engineering methods since 1976. He is a member of the Applications Panel of the STARS Program. His research interests include configuration management, software quality assurance, and performance improvement.

EXPERIENCE WITH THE INTEGRATION OF ADA* DESIGN METHODS

Paul L. Baker

Computer Technology Associates, Inc
McLean, VA 22102

ABSTRACT

A new software utility to manipulate data bases of digital images and maps is being designed in Ada using two design methods: 1) the Abstract Data Type method because it describes the application concepts well and because it works harmoniously with Ada, and 2) the Data Flow Diagram technique of Structure Analysis because it expresses the data flow architecture of our system clearly. The results of the two design techniques must be properly integrated to ensure a consistent design. Several integration techniques have been explored. All have been used manually, but could be supported by automated tools. The article contrasts two approaches to the transformation between data flow diagrams and Ada code and describes rules for connecting the code segments derived from the two different design methods.

1. Background

CTA has designed and is implementing a software system in Ada* to manipulate data bases of coordinate-referenced data, especially images. Funding for the work has been provided by Phase I and II Small Business Innovative Research (SBIR) grants through the Goddard Space Flight Center of the National Aeronautics and Space Administration (NASA-GSFC).

Even in the early phases of this work, the Ada* language played an important role by affording the means to document a key claim, namely that coordinate referenced data constitutes an Abstract Data Type that can be formally specified. This claim explains the major motivation for the project because it implies that general purpose software can be written to manipulate not just one image format but whole classes of image and cartographic formats. Typically, one finds many formats in use within the same agency; consequently, there is a practical significance to this claim.

This early application of Ada on the project employed the Abstract Data Type (ADT) method of design otherwise known as Object-Oriented Design[1]. Although this experience affirmed the value of this method, it did not appear to be appropriate for all of the system design work. Consequently, parts of the system were designed using the Data Flow Diagram (DFD) method of Structured Analysis[2].

The use of two design methods is effective, but it creates the possibility that the two design documents are inconsistent. Although the Abstract Data Types are written directly in Ada, the Data Flow Diagrams must be transformed into code, creating another possibility for inconsistency. Consequently, it became clear early that the consistent integration of design methods would be an issue for this project.

2. The Ada Design Methods

An Abstract Data Type (ADT) is a declaration of program structure that encapsulates a relatively complex data structure and procedural operations associated with the data. In the Ada community, this approach is also known as Object-Oriented Design[1]; and it is basically a construction method. If the type is constructed properly, it can be used without concern for its internal structure thereby freeing the attention of the designers for other matters.

---

*Ada is a registered trademark of the U. S. Government, Ada Joint Program Office (AJPO)

In this project, the ADT definitions were written in Ada concurrently with their exposition in English prose. The two documents express the same ideas; consequently, the Ada statements were organized in packages that corresponded one to one with chapters in the English exposition. Because the Ada ADT definitions are compilable, subsequent integration with implementation code is straightforward.

In practice, it is convenient to specify some of the next stage of design while completing the current stage. Traditionally, this is a role for PDL (Program Design Language). A number of Ada PDL's have been developed which provide syntax consistent with Ada[4] allowing the designer to intersperse PDL freely to elaborate on the definitions of the Ada ADT's. In our taxonomy, PDL is part of design but not a distinct method.

The highest level of any new system design serves to specify the virtual machine seen by the intended user. The design method used to express the design should be selected to portray the character of the virtual machine as clearly as possible. Unless the system is a very simple state machine, the Abstract Data Types cannot represent it clearly.

Because the system under development is basically a pipeline between an analysis station and a large data store, it is natural to portray the system design using Data Flow Diagrams[2] (DFD). Moreover, the DFD representations were familiar to both the project staff and outside reviewers. Consequently, DFD's could serve as a common basis for discussion of the design.

In summary, the Abstract Data Type method is perfectly adapted to the Ada language environment, but no single method is ideal for every view of the system. A logical method to complement ADT's is the Data Flow Diagram method. In previous nonAda experience within our company, DFD's and SREM[6] have been used about equally.

3. Integration Approachs

3.1 Assumptions

First let us establish a context for the discussion. Software is developed to satisfy a particular requirements specification. In our view, a system design is integrated when its parts are all mutually consistent and consistent with the requirements specification as well.

At the very least, integration requires maintaining the traceability of design elements to requirements specifications. For simple correspondence of the elements, a requirements traceability matrix is adequate. However, software development also requires composition and decomposition of elements, and these steps generate a need for structured documentation. Structured Analysis provides one documentation option; annotation embedded in structured computer language statements is another.

3.2 The Data Dictionary

A basic feature of the Structured Analysis method is the Data Dictionary which holds structured definitions of all the data items that pass between processes. The equivalent structure in Ada is the declaration section of a package where all the data types are defined. Normally, a Data Dictionary is organized alphabetically, while the Ada type declarations are not. Either alternative form can be taken as primary but it helps to be consistent. The pros and cons of the two options can be summarized as follows:

Options for Integrating
the Data Dictionary

Option 1: Data Dictionary is Primary

Pro: The Data Dictionary is easily read and updated.

Con: The Data Dictionary must be transformed to Ada

Mitigating Factor: Automated tool could be built.

Option 2: Ada Text is Primary

Pro: No transformation. Directly compilable.

Con: Not in alphabetical order. Hard to read.

Mitigating Factor: Automated documentation tool could scan the Ada text and build an alphabetic index.

Our project started with a Data Dictionary but shifted quickly to using all Ada because the job of maintaining consistency is too great without tool support.

3.3 The Data Flow Diagrams

Before Ada, the implementor had to invent a procedure hierarchy to represent a set

of DFD's. The Ada task construct revolutionizes the coding of DFD's because formal conversion to Ada is now possible without concern for the thread of control. If a diagram requires independent threads of control, Ada provides them through the tasking mechanism.

The hierarchical organization of Data Flow Diagrams documents the designer's intended system decomposition. A major choice to be made when coding these diagrams is whether to preserve this information in Ada and, if so, how.

The most considered transformation method known to this author is PAMELA, developed by George Cherry[3]. In PAMELA, processes that are not decomposed are associated with tasks while those which have a decomposition are associated with package constructs. Cherry argues that the compilable Ada code should correspond to the hierarchical design. Therefore, the hierarchy of the data flow diagrams is transformed into a nesting of packages within the declarative part of enclosing packages. That is, if processes B and C compose process A, then the package corresponding to A contains the packages corresponding to B and C.

Cherry also argues that information flow should be hierarchical; consequently, the packages which represent decomposable processes must contain procedures to communicate between tasks nested in different packages. The Ada procedures which define this information flow up and down the hierarchy serve as an interface definition that corresponds completely to the original DFD. Thus, complete traceability of the Ada code to the DFD is preserved.

On the other hand, hierarchical communication involves a heavy overhead for procedure calls. For example, if it is necessary for a sub-sub-process to communicate with a sub-sub-process elsewhere in the hierarchy, the task that initiates the communication first calls upward to a procedure in the package of its enclosing sub-process. This procedure in turn calls a procedure in the top-level package. The top-level process completes the call by first calling downward to the appropriate sub-process which in turn calls the desired sub-subprocess.

Actually, the Ada code derived from a DFD contains information that a conventional DFD does not; specifically, it tells which process initiates the data flow. Using PAMELA design rules, this information can be added as annotation to the DFD.

PAMELA is an excellent method, but its premise that communication should be hierarchical is questionable. Although it is reasonable to delegate authority and responsibility hierarchically, once a software process has been properly authorized to act, there seems no reason to forbid communication with its peers.

In view of this criticism, one should consider a simplification of PAMELA. One can eliminate the packages and their communication procedures by allowing direct calls between the tasks which implement the primitive processes. It is still possible to preserve information concerning the original DFD hierarchy as annotation by using structured comments surrounding the task declarations.

The main objection to the simplification concerns maintenance. Firstly, the flat task arrangement forces massive recompilation when parts are changed. Also, whenever a task declaration is changed, one must track down all references to that task entry point and change them. This replacement would be simplified if Ada permited task names as parameters of generics; but it does not. Generally speaking, Ada is weak in the area of reconfiguring task structure either to accomodate change or utilize parts from a stock of reusable software components[5].

In the absence of tool support, the simplified method was the obvious choice for this project. It is too tedious to construct all the interfaces of the hierarchy in Ada particularly since they do not contribute to the function of the system.

The pros and cons of these two options for transforming data flow diagrams can be summarized as follows:

Options for Transforming
Data Flow Diagrams

Option: PAMELA

Pro: Ada code is hierarchical.
Structure of the design is preserved in Ada code.
Effect of change is local.

Con: Transform of design to code is tedious and errorprone.
Hierarchy increases execution overhead.
Conditional calls are hard to implement.

Mitigating Factor: Automatic Design Tool exists to convert diagrams to Ada.

Option: Simplified, Flat Task Structure

Pro: No hierarchy to create execution overhead.
Manual transformation to code is simpler than PAMELA.

Con: Original design hierarchy is present only in Ada comments.
Effect of change can be global.

Mitigating Factor: Automatic tool could extract structured comments and verify against data flow diagram. Conversely a tool could create the code as suggested for PAMELA.

3.4 The Abstract Data Types

Structured Analysis is a form of top down design; that is, it progresses by decomposition rather than composition of elements. The Abstract Data Type method is more general; one may work the problem in either direction which encourages a middle-outwards development. Because of this basic difference in approach, the DFD part of a design will only mesh at its lowest level with ADT's. Part of the art of top down design is guiding the decomposition so that it meshes with the parts available for implementation. The situation is no different in this case.

When DFD's must be integrated with ADT's, the main decomposition criterion is that the decomposition should proceed sufficiently far that each primitive process is simple enough to identify with one operation of an abstract types. If decomposition is not carried far enough, a process may describe alternative and concurrent functional computations that are hard to capture as a single ADT operation.

Rules for Integration
of DFD's and ADT's

- data flow items are identified with abstract data types that are represented by type declarations in Ada.
- primitive processes are identified with an operation on an abstract type and are represented in Ada by an entry point. There should be a one to one relation of processes and operations. This requirement is the major restriction on the decomposition.
- data flow arrows are grouped together to define the operands of the operations. These groups of arrows are represented in Ada by the formal arguments of an entry declaration. If desired, the formal argument list can be made a named entity in the Data Dictionary.
- the Ada statements are annotated with comments to record their associations with data flow diagram entities.

The first rule is rather obvious. The second rule is more subtle because it makes an implicit assumption, namely that a DFD process identified with an ADT operation has the property that it executes once for each set of data flow items. In conventional Structure Analysis, one may never assume such a property for a process. This point underscores the need to develop the decomposition properly with an understanding of how it will be implemented by ADT's.

In general, the list of data flows connected to a DFD process is not identical to the parameter list of the corresponding ADT operation, because the operation may call on other processes in the DFD. The data flows involved in these calls also connect to the process. As a result, the DFD's containing primitive processes should be annotated to show how the flows are being grouped.

Manually, one can group the data flows by drawing a "cable-tie" around them. Manual cabling provides sufficient documentation; after all, the Ada code contains a specific, formal specification of these operations.

In an Ada environment, there is no special rule for files. Files are introduced in Structured Analysis because processes are assumed to have no memory. However, a file on a data flow diagram can be associated with an entry point in Ada. Naturally, the entry point must belong to a package that hides the state information represented by the file.

3.5 Final Thoughts

On the one hand, automated tools are highly desirable for dealing with DFD's; on the other hand, they may encumber the effort to integrate the DFD's with ADT's if the tool does not understand the cabling processes described above. If the tool produces skeleton task and procedure

statements that must be heavily modified manually to mesh with ADT's, its benefits may be dissipated.

Our discussion has concerned integration at the level of the output of design methods. Many investigators are interested in the integration at the level of the method itself. For example, R. J. A. Buhr has proposed a graphical design technique specifically for Ada that maps readily into Ada code[7,8]. Closer to home, a group in our sponsoring agency NASA-GSFC has developed a graphical method for dealing with Abstract Data Types[9]. This development is particularly interesting because it supports hierarchical, graphical decomposition in the manner that has proved so valuable with Structured Analysis. Like many software developers, we shall let our final judgement be swayed by the availablity of affordable, convenient, support tools.

4.0 Summary

In this project, two design methods were used, Abstract Data Types and Data Flow Diagrams. The results are integrated in the sense that elements of each design are connected though one or two of the following three methods.

First, different parts of the design can share the same data dictionary. The compositions expressed in a Structured Analysis data dictionary are isomorphic with Ada record definitions; therefore, it is simple to maintain consistency between different forms of the same dictionary.

Second, an element in one design method can implement the requirement stated by the specification of an element in another design method. For example, a primitive process in a data flow diagram can be implemented by an operation in an Abstract Data Type package.

Third, the elements of one design can be formally transformed into elements of another design. For example, Data Flow Diagrams can be formally transformed into Ada code.

Each of these integration methods is tedious, time consuming and prone to error. Consequently, tool support for the integration effort would be very desirable. Automated support for PAMELA is undergoing final tests[3]. However, simpler tools are also needed. For example, the order of Ada type definitions makes it difficult to trace the composition of a type without the help of an alphabetical index. Traceability matrices are notoriously difficult to compile and maintain; again a tool could extract them from the Ada annotation. Finally and somewhat unfortunately, Ada does not permit the parameterization of task calls. This restriction complicates the maintenance of code that makes calls into a package where revisions must be performed. Correct use of information hiding in Ada can prevent this problem, but the additional interfaces add overhead. A text preprocessor to parameterize the calls in a package could provide an alternative solution.

REFERENCES

1) see pg. 38, Software Eng. with Ada, Grady Booch, Benjamin/Cummings (1983).

2) see Part 2 of Structured Analysis and System Specification, Tom DeMarco, Prentice-Hall (1978).

3) seminar notes for Software Engineering with Ada, George Cherry, U. S. Professional Development Institute (1984). Contact George W. Cherry, P.O. Box 2429, Reston, VA 22090.

4) Survey of Ada-Based PDLs, January 1985, contact Dr. L. Lindley, Naval Avionics Center, Indianapolis, IN 46218.

5) "Structured Tasking in Ada?", P. H. Welch, Ada Letters, Vol. 5, No. 1, pg. 17 (July 1985).

6) "SREM at the Age of Eight", Mack Alford, Computer, Vol. 18, No. 4, pg. 36. (April 1985).

7) System Design with Ada, R. J. A. Buhr, Prentice-Hall, (1984).

8) "An Informal Overview of CADAE...", R. J. A. Buhr and G. M. Karam, Ada Letters, Vol. 4, No. 5, pg. 49 (March 1985).

9) preprint, Object Diagrams, Ed Seidewitz, NASA-GSFC, Greenbelt, MD 20771, (May 1985).

The author may be contacted at the following address:

Dr. Paul L. Baker
Computer Technology Associates
7927 Jones Branch Drive, Suite 600W
McLean, VA 22102
(703) 8482713

The author earned his BS and MS degrees in the field of Physics and his Phd in Astronomy. For 7 years, he conducted astronomical research specializing in the computer assisted analysis of 3d spectral line radio observations. Thereafter, he joined CSC (Silver Spring, MD) and, for three years, managed groups of programmers who analysed data from space physics and Landsat experiments. For the past 4 years, he has performed system analysis and design in his current position as Chief Scientist with CTA's Technologies Division. CTA is incorporated in the state of Colorado and has offices in seven states.

APPLYING THE SPIRAL MODEL: OBSERVATIONS ON DEVELOPING SYSTEM SOFTWARE IN ADA[1]

Frank C. Belz

TRW
Redondo Beach, CA 90278

Abstract

A new model of software development and enhancement, the *spiral model*, was introduced by Boehm in 1985; the model defines a risk-driven rather than specification-driven or prototype-driven approach to the software process. The introduction of Ada has established a risk laden transitional era especially in the development of Ada system support software, and many projects are taking a variety of risk-management approaches; the spiral model provides a way to view the successful approaches and may, as it matures, provide effective guidance for future projects. This paper describes the spiral model, some of the risks involved in developing Ada system support software, and a (hypothetical) application of the spiral model to a particular kind of such software: an *integration framework* for Ada project support environments.

1. Introduction

1.1. Overview In early 1985, Dr. Barry Boehm introduced a *spiral model* of software development and enhancement that provides a new framework for guiding the software process [Boehm, 1985]. Its major distinguishing feature is that it creates a *risk-driven* approach to the software process, rather than a strictly specification-driven or prototype-driven process. It incorporates many of the strengths of other models, while resolving many of their difficulties.

The spiral model is both *descriptive*, providing insight into the important processes that characterize successful software development projects in today's world of rapidly changing technology, and *prescriptive*, providing guidelines for better ways to organize the process of software development in such a rapidly changing world. The spiral model is in its early stages of development; there are many aspects of the model awaiting elaboration. Until the model reaches a useful level of completeness, its role as a prescriptive tool will be limited. The author, in collaboration with Dr. Boehm (and using the experience of other researchers, developers and managers), is attempting to lay the groundwork for such an elaboration of the model.

The introduction of Ada presents a rich resource in such an effort; the opportunities and risks abound in this transitional era, and many different approaches to risk management are being attempted. A particularly interesting area to observe is the development of Ada support software, especially persistent support systems whose services may have an execution lifetime much greater than that of a particular application program execution. Examples include operating systems, data base management systems, Ada Programming Support Environment portability kernels (KAPSES) and the integration frameworks for advanced Ada-based project support environments.

This paper, then, has three purposes:

1. Describe the spiral model and the context from which it grew; this introductory section excerpts and summarizes parts of [Boehm, 1985], presenting from that report: a brief historical review of software process models and the issues they address; a summary of the process steps involved in the spiral model; and a brief demonstration of the conditions under which the spiral model reduces to other useful process models.

2. Discuss some of the key risk factors involved in the development of persistent support systems developed in Ada; Section 2 provides that discussion.

3. Suggest a hypothetical scenario in which the spiral model approach is used in the development of an

[1]Ada is a trademark of the United States Department of Defense (AJPO)

underlying framework of an advanced Ada-based project support environment. Section 3 contains this scenario.

This report is, therefore, very much a preliminary view of work in progress, both with respect to the spiral model itself and with respect to the industry-wide introduction of Ada support services. Section 2 discusses a particular risk area in the development of new Ada systems: the design of the data management interfaces of support system software (such as operating systems, data base management systems and tool portability kernels). Section 3 describes a brief (hypothetical) scenario of the application of the spiral model to this risk area.

1.2. Software Process Models

The Waterfall Model

One of the earliest software process models, the *stagewise* model given in [Benington, 1956], recommended that software be developed in successive stages (operational plan, operational specifications, coding specifications, coding, parameter testing, assembly testing, shakedown, system evaluation).

The original treatments of the *waterfall model* given, for example, in [Royce, 1970], provided two primary enhancements to the stagewise model:

- Recognition of the feedback loops between stages, and a guideline to confine the feedback loops to successive stages, in order to minimize the expensive rework involved in feedback across many stages.

- An initial incorporation of prototyping in the software life-cycle, via a "build it twice" step running in parallel with requirements analysis and design.

The waterfall approach was largely consistent with the *top-down structured programming model* introduced in [Mills, 1971].

A *risk-management variant* of the waterfall model, discussed in [Boehm, 1975] and elaborated in [Boehm, 1976], expanded each step to include a validation and verification activity to cover high-risk elements, reuse considerations, and selective prototyping. Further elaborations of the waterfall model covered such practices as incremental development [Distaso, 1980].

Alternative Software Life-Cycle Models

Yet the waterfall model encountered a number of difficulties, leading to a number of alternative life-cycle models which do a better job of coping with these difficulties. For example:

- The waterfall model does not adequately address concerns of developing program families and organizing software to accommodate change. The Parnas *information-hiding approach* [Parnas, 1979] does an excellent job of addressing these concerns.

- The waterfall model assumes a relatively uniform progression of elaboration steps. The *two-leg model* [Lehman-Stenning-Turski, 1984], [Lehman, 1984] features separate processes of abstraction until a formal specification is achieved, followed by a set of formal deductive "reification" steps to proceed through design and into code.

- The waterfall model does not accommodate the sort of evolutionary development made possible by rapid prototyping capabilities and fourth-generation languages. Several *evolutionary development* models [McCracken-Jackson, 1982] and mixed models [Giddings, 1984] have been advanced to address this approach.

- The waterfall model does not address the possible future modes of software development associated with automatic programming capabilities, program transformation capabilities, and "knowledge-based software assistant" capabilities. The *automation paradigm* [Balzer-Cheatham-Green, 1983] provides an alternative life-cycle model and conceptual framework for incorporating these capabilities.

However, although each of these alternative approaches deals with some of the difficulties of the waterfall approach, each has its own set of difficulties and challenges to resolve. The information-hiding as an organizing approach has not yet been fully elaborated to see how it will cover such issues as prototyping. The two-leg model has challenges in accommodating software reuse, program families, and logical-physical design tradeoffs. The evolutionary development approach has challenges in scaling up to very large systems, ensuring process visibility and control, avoiding the negative effects of "information sclerosis,"[1] and avoiding the

[1] *Information sclerosis* is a syndrome familiar to operational information-based systems, in which temporary work-arounds for software deficiencies increasingly solidify into unchangeable constraints on evolution. A typical example is the following comment: "It's nice that you could change those equipment codes to

"undisciplined hacker" approach that the waterfall and other models were trying to correct. The automation paradigm has challenges in scaling up to very large systems accommodating program families, avoiding the effects of information sclerosis, and handling the boundaries between older, stable, but less powerful capabilities and new, unstable, but more powerful capabilities.

1.3. The Spiral Model

The spiral model of the software process serves as a significantly more robust foundation for a software development environment than previous models; it includes most previous models as special cases, and further provides guidance as to which combination of previous models best fits a given software situation.

The spiral model is illustrated in Figure 1. The radial dimension in Figure 1 indicates the cumulative cost incurred in accomplishing the steps to date; the angular dimension indicates the progress made in completing each cycle of the spiral. The trajectory of the spiral (working from the inside outward) tends to indicate the efficiency of the software development process:

- tightly wound, spring-like spirals indicate a low rate of cost increase as progress toward the product (or product component) is achieved;
- loosely wound spiral trajectories indicate higher cost for corresponding achievement.

The model holds that each cycle involves a progression through the same sequence of steps, for each portion of the product and for each of its levels of elaboration, from an overall concept-of-operation formulation to the coding of each individual program.

1.4. A Typical Cycle of the Spiral

Each cycle of the spiral begins with the identification of:

- The objectives of the portion of the product being elaborated (performance, functionality, ability to accommodate change, etc).
- The alternative means of implementing this portion of the product (design A, design B, reuse, buy, etc).
- The constraints imposed on the application of the alternatives (cost, schedule, interface, etc.).

The next step is to evaluate the alternatives with respect to the objectives and constraints. Frequently, this process will identify areas of uncertainty which are significant sources of project risk. If so, the next step should involve the formulation of a cost-effective strategy for resolving the sources of risk. This may involve prototyping, simulation, administering user questionnaires, analytic modeling, or combinations of these and other risk-resolution techniques.

Once the risks are evaluated, the next step is determined by the relative risks remaining. If performance or user-interface risks strongly dominate program development or internal interface-control risks, the next step may be an evolutionary development step: a minimal effort to specify the overall nature of the product, a plan for the next level of prototyping, and the development of a more detailed prototype to continue to resolve the major risk issues. On the other hand, if previous prototyping efforts have already resolved all of the performance or user-interface risks, and program development or interface-control risks dominate, the next step follows the basic waterfall approach, modified as appropriate to incorporate incremental development.

The spiral model accommodates any appropriate mixture of specification-oriented, prototype-oriented, simulation-oriented, automatic transformation-oriented, or other approaches to software development, where the appropriate mixed strategy is chosen by considering the relative magnitude of the program risks, and the relative effectiveness of the various techniques in resolving the risks. (In a similar way, risk-management considerations determine the amount of time and effort which should be devoted to such other project activities as planning, configuration management, quality assurance, formal verification, or testing).

An important feature of the spiral model is that each cycle is completed by a review involving the primary people or organizations concerned with the product. This review covers all of the products developed during the previous cycle, including the plans for the next cycle and the resources required to carry them out. The major objective of the review is to ensure that all concerned parties are mutually committed to the approach to be taken for the next phase.

The plans for succeeding phases may also include a partition of the product into increments for successive development, or components to be developed by individual organizations or persons. Thus, the review and commitment step may range from an individual walk-through of the design of a single programmer com-

make them more intelligible for us, but the Codes Committee just met and established the current codes as company standards." The greatest risk of information sclerosis occurs when the evolutionary prototype is placed in an environment that desperately needs improved capabilities.

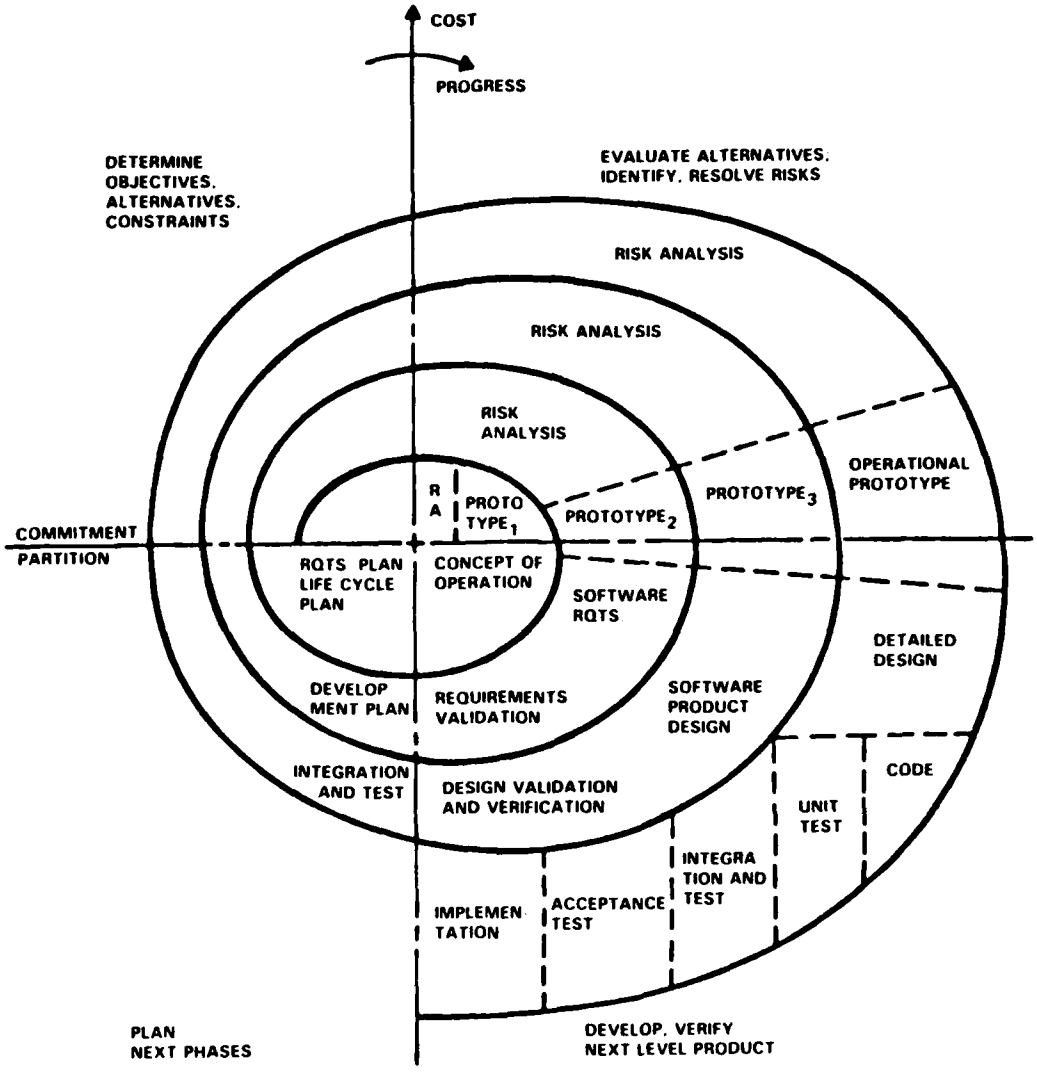


Figure 1. Spiral Model of the Software Process

ponent, to a major requirements review involving developer, customer, user, and maintenance organizations.

The spiral model applies equally well to development or enhancement efforts, each of which are initiated by a hypothesis that a particular operational mission (or set of missions) could be improved by a software effort. The spiral process then includes a test of this hypothesis: at any time, if the hypothesis fails the test, the spiral is terminated. Otherwise, it terminates in the installation of the new or modified software, and the hypothesis is tested by observing the effect on the operational mission.

1.5. Spiral Model Advantages

The primary advantage of the spiral model is that its range of options and risk-driven approach allow it to accommodate the best features of existing software process models, while avoiding most of their difficulties. In appropriate situations, the spiral model becomes equivalent to one of the existing process models. In other situations, it provides guidance on the best mix of existing approaches to be applied to a given project.

The primary conditions under which the spiral model becomes equivalent to other main process models are summarized below.

- If a project has a low risk in such areas as getting the wrong user interface or not meeting stringent performance requirements; and it has a high risk if it loses budget, schedule, and product predictability and control; then these risk considerations drive the spiral model into an equivalence to the waterfall model.

- If a software product's requirements are very stable (implying a low risk of expensive design and code breakage due to requirements changes during development); and if the presence of errors in the software product constitutes a high risk to the mission it serves; then these risk considerations drive the spiral model to resemble the two-leg model of precise specification and formal deductive program development.

- If a project has a low risk in such areas as losing budget and schedule predictability and control, encountering large-system integration problems, or coping with information sclerosis; and it has a high risk in such areas as getting the wrong user interface or user decision support requirements; then these risk considerations drive the spiral model into an equivalence to the evolutionary development model.

- If automated software development capabilities are available, then the spiral model accommodates them either as options for rapid prototyping or for application of the automation paradigm, depending on the risk considerations involved.

- If the high-risk elements of a project involve a mix of the risk items above, then the spiral approach will reflect an appropriate mix of the process models above. In doing so, its risk-avoidance features raise the probability that the difficulties of the other models will be avoided.

2. Ada Support Software: Risk Issues

There are special properties of support software, especially persistent support software, which affect the project risks when that software is being developed in Ada. In this section, a brief summary of a software designer's view of some key characteristics of such software (and some of their specific interactions with Ada) is followed by a summary of the management and technical risks that derive from these characteristics.

2.1. Characteristics of Persistent Ada Support Software

Persistent support software systems are generally designed to provide reliable resource management services (eg, data, process, i/o) according to policies which may restrict the provision of potentially sharable resources (eg, access control [mandatory and/or discretionary], processor allocation priority, device allocation/assignment). Persistence is a characteristic of such software when users of the software system[3] need to keep data over multiple "sessions"[4] and when multiple users need to access data maintained on shared hardware resources. The essential property of such software is that, when executing, its state has a longer lifetime than (ie, the value of its data is kept longer than) that of the software which is temporarily active on behalf of the user in the "session".

Examples of such software being developed in Ada include

- operating systems such as ASOS [Anderson-Hart, 1985],
- Data Base Management Systems such as TDBMS [Bamberger, et al, 1986], LDM, DDM, and Multibase [CCA,1985],
- portability kernels for APSEs such as prototype implementations of the facilities of the CAIS [KIT/KITIA, 1985] and the integrating framework of the ALS [ALS, 1985], both of which are based on the KAPSE model of [Stoneman, 1980], and
- a prototype implementation of the Ada/SQL support software to support the standard Ada-DBMS Interface reported in [Friedman-Brykczynski, 1986]; this interface permits the replacement of underlying DBMSs without affect the Ada programs using them.

---

[3] including both people in a timesharing environment, and sensors/effectors in a realtime control environment

[4] "threads" in a realtime control environment

These software entities need not be independent; most database management systems rely on operating system services in critical ways.

2.1.1. Data management support

In fact, if we focus on the critical aspect of the integration framework for an APSE (in anticipation of the next section), a key consideration, data management, can be described with a layered view to order a whole family of design issues. In [Friedman, et al, 1986], an 8-level reference model is proposed. In increasing order of complexity:

1. **Hardware** layer provides the basic storage resources.

2. **Operating System** layer provides file support (in the normal file-system sense) and multi-user access to files. (Supported by Ada I/O.)

3. **Record** layer provides external-internal data format conversion. (Supported by Ada I/O.)

4. **File** layer provides multi-user access to records, and access to data by key. (Supported by a file-management system.)

5. **Tuple algebra** layer provides specification of operations as database commands, allowing data management system to return subset of all data examined to the program. (This is the minimum layer for communication interface in a distributed storage system.)

6. **Relation** layer provides different user views of the data. (Typical Off-the-shelf DBMSs.)

7. **Object** layer provides invariant assertions about the data used for consistency, and possibly inheritance of date schema. (Research DBMSs.)

8. **Heuristic** layer employs probabilistic assertions about the data used for efficiency. (AI research areas.)

When designing the particular support software of interest in the next section, an integration framework for an APSE, it is critical to decide what capabilities will be provided at the interfaces. The proposed MILSTD CAIS specification probably resides at the File level for the most part with some functions at the Tuple algebra level. The requirements for the successor CAIS [KIT/KITIA, 1985 "RAC"] mandate an interface set at the Relation layer at least.

The more capabilities present at the interface, the more powerful the support service must be. Arguments in favor of a lower level include the fact the overall performance of the support software may be enhanced by virtue of its relative simplicity. On the other hand if the performance of interest is that of the overall system, and if achieving an effective portability kernel is critical, an engineering compromise may be necessary in which a carefully selected set of high level services are provided at the interface.

2.1.2. Ada Interaction Effects There are a number of special properties of Ada that affect the interfaces between such persistent support software and newly developed Ada programs.

- *The interfaces are closely tied to the Ada program construction mechanism.* The Ada language was carefully designed to permit programs to be constructed that rely on already existing and executing capabilities; these capabilities can be described using specifications of library units that are referenced with context specifications (**withs** and **uses**). The ways in which this process can occur depend on the model of program construction assumed by the Ada compiler.[ALRM, 1983]

  If it must be possible to change the capabilities of the support software in real time without bringing the system down, effective techniques for achieving this may involve both very sophisticated Ada programming techniques and special properties of Ada compilers.[Roubine, 1985]

- *Strong typing raises subtle interface design options.* Because of the ability (and mandate) to provide strong type checking across the interface, designers of the support software are provided with a new choice: data of predefined types can be the only data which cross the interface *or* the user defined types can be permitted at the interfaces. For most designers, the latter case has not heretofore been available except with weakly typed languages; Ada generics are necessary to achieve user defined types at the (Ada) strongly typed interfaces.

  Strong typing, enforced at compile time, provides the opportunity in some cases to eliminate runtime integrity checks; this is an incentive to permit user-defined types at the interface. However, compile time checks are limited (many conditions are not definable at compile time in Ada) and reliance on them may necessarily be augmented by other runtime checks. [Friedman, et al, 1986]

Maximal use of user defined types at the interface may require very sophisticated, even complex (or some say, arcane) use of Ada features including, at the least, nested generics. [Friedman-Brykczynski, 1986]

- *Processor management software, written in Ada, impacts and is impacted by the Ada Runtime Environment used by the Ada compiler.* Ada tasking semantics require that an unusual degree of control of processor resources must be maintained within the Ada RTE. For this reason it is particularly difficult to design interfaces that simultaneously use, say, the strong typing conventions, and permit applications programs to be compiled with different compilers.

- *Whether or not all concurrency in the system is modeled by Ada tasks has a dramatic effect on the character of the persistent support software.* If not, a normal alternative is to permit concurrently executing Ada programs. These two forms of concurrency must be carefully dealt with by the support software.

  For example, DBMS systems must be written with interfaces that acknowledge both forms of concurrency; this is in contrast to the current situation in which many preexisting DBMS's will only communicate with a single task at a time in an Ada program because, for the DBMS, the only identified source of concurrency is the executing program, not the tasks within a program.

  As another example, having multiple concurrently executing Ada programs provides for a clean way of separating the data-spaces of functions operating on behalf of separate users, but may require the definition of new inter-program communication protocols.

This is an incomplete, but representative list of issues that arise from the interaction of the nature of Ada and the intrinsic properties of persistent support software. Associated with these issues are significant risks.

2.2. Related Risks

The principal risk associated with the issues raised above is *lack of maturity and availability of critical resources*, such as people, including experts in the domain of the particular support software to be developed, Ada experts, and experts in both fields; and support software, including compilers and software construction aids. In most Ada development projects, these (as well as the perennial cost and calendar constraints) are the driving risks.

Domain expertise and Ada expertise both have to be present on the staff of a given development project, and though it is usually not mandatory, it is very helpful to have a single person with both.

Ada compilers are just beginning to exhibit the completeness and robustness required of the more ambitious uses of Ada such as that proposed in [Friedman-Brykczynski, 1986] and [Roubine, 1985]. The lack of the required compiler sophistication leads to more conservative design approaches. (Of course, in some cases, such as the requirement to support assurable multi-level security [Anderson-Hart, 1985], the requirements of the development itself may lead to even more conservative approaches.) In support software that must execute on a variety of different target systems [Bamberger, 1986], the principal of the least common denominator will dominate: for any particular Ada issue, the least capable compiler for a given target will drive the design approach.

The technical issues themselves pose substantial risks.

A fundamental aspect of the technical risk analysis is that this kind of support software is not amenable to the construction of tight requirements: for example, performance is important, but not precisely specifiable (usually), and sometimes subordinate to other concerns, such as verifiability. Unlike specific imbedded systems which have well-defined usage requirements based on the characteristics of the particular surrounding hardware/software system, the persistent support software must be able to accommodate several different applications (perhaps several different embedded system applications).

Failure to achieve satisfactory performance is a key risk; support software must lead to effective use of computing resources (both computational and storage). For some systems, internal simplicity may be critical [Anderson-Hart, 1985]; for others [KIT/KITIA, 1985], raising the level of the interfaces may provide an opportunity for optimization which will offset the added complexity.

Failure to provide an interface that is effectively usable by the designers of the user software is a significant risk. Simplicity of the user model and ease of use of the interfaces can determine whether or not the facilities provided are ever used, and if they are, the degree to which they can be used effic' ntly. For example, taking advantage of the Ada strong typing in the interface design in order to move the cost of some integrity checking to the compile time phase may have a signal effect on runtime performance, but if it leads to arcane usages of the Ada language it will impact the cost of all software develop-

ment depending on those interfaces, and it may even exceed the capability of many software development shops to use the resulting interfaces.

Being locked in to a particular compilation technology is a major risk in some cases. As indicated above, this risk is severe when concurrency management is involved in the support software, or when there are sophisticated requirements for adding new capabilities to the support software or for binding new programs to existing, executing support capabilities since these activities depend so heavily on the Ada program construction mechanism. This technical risk may be commercially critical and even where it is not, it may severely impact the ability of the persistent support system to upgrade with advances in compilation technology.

Being locked into a particular hardware technology may be a significant risk. Certainly when the support software of interest is the framework for an APSE, tying it to a single Instruction Set Architecture not only limits the conditions under which the framework (and therefore the APSE) can find acceptance but also limits the ability of the framework to migrate to new hardware as technology improves.

3. Applying the Spiral Model

Consider the following hypothetical situation: it has been determined by, say, organization **ABC**, that a new integration framework is needed for future automated environments that will support the development and enhancement of Ada-based systems. The environment to be developed using this framework will have to support the entire software process; it will have to be highly flexible, extensible and adaptable (to new underlying technologies, hardware and software).

How might the development of such a framework proceed according to the spiral model? Here is a partial scenario, in which the early stages of the project are described at a very high level.

Round 1. In this round organization **ABC** forms a small team of key advisors to analyze the feasibility of the goal. The following outline suggests the activities of the team.

Identify Goal: Obtain an environment framework with the (qualitative) properties described above.

Enumerate alternatives: (1) Obtain an already existing framework, eg: CAIS implementations, UNIX, PCTE, TRW's Productivity System, The Rational R1000, Arcturus, Toolpack, ... (2) Build a new framework using an already existing framework specification, eg MILSTD CAIS, PCTE, ... (3) Design and build a new framework.

Identify Risks: (1) Existing frameworks may fail on one or more technical risks such as those described above. (2) The specifications, even without the particular failings of their implementations, may fail the technical risks also. (3) The design, development and validation effort for a build from scratch may require excessive or unavailable resources.

Resolve Risks: Based on this analysis, the advisors conduct a systematic comparison of the existing candidate frameworks against the goals and the risk list, using interviews, demonstrations and technical analysis. This leads to the conclusion that no one framework satisfies the goals with satisfactory risk. A similar analysis of the specifications indicates that many features of the specifications are satisfactory, but there are essential technical risks that are not addressed. Finally, an analysis of the technical complexity and scope of a separate design and build concludes that

- The technical prerequisites for the essential component of a framework seem to be present. Merging them is a difficult technical exercise with uncertain results in the very near term, although it should be possible to develop a very strong framework in about 5 years.
- The development will require the combined efforts of several different kinds of expertise, unlikely to be found at any one source; however, there are clusters of researchers and developers that have already established informal technical exchanges on similar matters.

Plan for next phase: Organization **ABC** decides to stimulate the formation of a consortium of research and software development organizations. The analysis by the advisory team is used to stimulate membership and provide guidance to the consortium in its early deliberations.

Round 2. The consortium, over some time, actually forms. Its members include a large scale producer of software and user of environments, several smaller research groups specializing in program generation, program analysis and program testing and development process coordination tools; a small compiler vendor developing new techniques for incremental compilation of Ada programs. Two members have already constructed prototype environment frameworks. The consortium has few natural competitors and each member can immediately benefit from a framework based environment, so the incentives for cooperation are high.

The consortium identifies high risk areas: lack of clear assignment of responsibilities and contribution, lack of clear consensus on architectural and design assumptions and constraints, and lack of a short term approach to separate development of cooperating tools. A risk reduction strategy is agreed to: establish principles of cooperation and a management plan for the consortium,

formulate an initial concept of operation for the framework, and establish near-term interface definitions and rules of construction that enhance sharing of tool capabilities. These are among the tangible results of this Round; the group also establishes a Risk Management Plan to create visibility of the outstanding risks. Regular group meetings keep these issues under constant attention. New risks are surfaced: there is insufficient expertise in the consortium to solve the more subtle aspects of the object management support capabilities in the framework. The consortium agrees to search for a new member who can bring that expertise to the party.

The project plan is produced (consistent with the management plan and the principles of cooperation) in which each consortium member has a dual role: to develop particular capabilities (identified in the plan) such as basic program construction tools, or user-interface management systems, and to participate in consensus formation activities on the unpartitioned essential developments. Certain participants take responsibility for exploratory prototypes for critical areas, like the user interface. The consortium members commit to the plan for the next round.

Acknowledgement

I wish to thank Barry Boehm, who has provided not only the spiral model itself, but also extensive encouragement and support for the author's contributions. More directly, he has permitted the broad re-use of [Boehm, 1985] in the development of Section 1.

Frank C. Belz is currently Ada Support Technology Projects Manager for the Information and Systems Software Laboratory of TRW's System Development Division. His responsibilities include serving as a member of the Senior Advisory Board for the Army Secure Operating System (ASOS) Project and as advisor to the Common Apse Interface Set prototyping effort of the Ada Software Engineering (ASE) Project, participation in the TRW/DSG Quantum Leap planning effort, and participation in the Arcadia consortium.

Mr. Belz has previously served as Project manager of the Advanced APSE Prototype and KAPSE Standardization Support (AdaPAKSS) Project, and the Prototype Advanced APSE (PA-APSE) Feasibility Demonstration Project. He was the chief designer of TRW's AdaPDL IR&D Project, and has served on the staff of the Advanced Productivity Project. He has managed the Microprocessor Training Center, and has conducted research and development in microprocessor system development, automated software test measurement, software specification and verification, and formal programming language semantics. He has been intimately involved with the development of Ada, serving on the Air Force Ada Selection team, and as a member of the HOLWG Distinguished Reviewers (1979-80), and more recently on the Kapse Interface Team, and the Ada RunTime Environment Working Group (ARTEWG).

REFERENCES

[ALRM, 1983] Reference Manual for the Ada Programming Language, MILSTD 1815A, U.S. Department of Defense, Jan. 1983.

[AFSAB, 1983]. USAF Scientific Advisory Board, Report of the USAF/SAB Committee on the High Cost and Risk of Mission-Critical Software, J. B. Munson, chair, Dec. 1983.

[Alford, 1977] Alford, M. W., "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Trans. S/W Engr.* , JAN 1977, pp. 60-68.

[ALS, 1985] Turner, D.J., "The Ada Language System", *Proceedings of the Third Annual National Conference on Ada Technology*, Houston, Mar. 20-21, 1985, pp.82-86.

[Anderson-Hart, 1985]. Anderson, E.R., and R.M. Hart, "Why not UNIX? The Case for the Army Secure Operating System" *Proceedings of the Third Annual National Conference on Ada Technology*, Houston, Mar. 20-21, 1985, pp.225-229.

[Balzer-Cheatham-Green, 1983] Balzer, R., T.E. Cheatham, and C. Green, "Software Technology in the 1990's: Using a New Paradigm," *Computer*, Nov. 1983, pp 39-45.

[Bamberger, et al, 1986]. Bamberger, J., P. Ritter, and J. Wilson, "Tactical Database Management System - An Ada Technology Project for the US Army", *Proceedings of the Fourth Annual National Conference on Ada Technology*, Atlanta, Mar. 19-20, 1986.

[Benington, 1956]. Benington, H. D., "Production of Large Computer Programs," Proc. ONR Symposium on Advanced Programming Methods for Digital Computers, June 1956, pp. 15-27. Also available in *Annals of the History of Computing*, Oct. 1983, pp. 350-361.

[Boehm, 1975]. Boehm, B. W., " Software Design and Structuring," in *Practical Strategies for Developing Large Software Systems*, E. Horowitz (ed). Addison-Wesley, 1975, pp. 103-128.

[Boehm, 1976]. Boehm, B. W., "Software Engineering,"*IEEE Trans. Computers*, December 1976, pp. 1226-1241.

[Boehm, 1981]. Boehm, B. W., *Software Engineering Economics*, Prentice- Hall, 1981.

[Boehm, et al, 1984]. Boehm, B. W., M.H. Penedo, E. D. Stuckle, R. D. Williams, and A. B. Pyster, "A Software Development Environment for Improving Productivity," *Computer*, June 1984, pp. 30-44.

[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, Mar. 26-29, 1985.

[CCA, 1985] Smith, J.M., A. Chan, S. Danberg, S. Fox, A.Nori, "A Tool Kit for Database Programming in Ada", *Ada in Use*, Proceedings of the Ada International Conference, Paris, May 14-16, 1985, pp.41-57.

[Distaso, 1980]. Distaso, J. R., "Software Management- A Survey of the Practice in 1980." *IEEE Proceedings*, Sept. 1980, pp. 1103-1119.

[Friedman, et al, 1986]. Friedman, F., A. Keller, J. Salasin, G. Wiederhold, M. Berkowitz, and D. Spooner, "Reference Model for Ada Interfaces to Database Management Systems", *Proceedings, IEEE Conference on Data Engineering*, IEEE, 1986, pp. 492-506.

[Friedman-Brykczynski, 1986]. Friedman, F., W. Brykczynski, "Ada/SQL: A Standard, Portable Ada-DBMS Interface", *Proceedings, IEEE Conference on Data Engineering*, IEEE, 1986, pp. 515-522.

[Giddings, 1984]. Giddings, R.V., "Accommodating Uncertainty in Software Design," *Comm. ACM*, May 1984, pp. 428-434.

[KIT/KITIA, 1985]. Oberndorf, P.A., "Proposed MILSTD Common APSE Interface Set (CAIS)" and "Draft CAIS Requirements and Design Criteria (RAC)" in *KAPSE Interface Team Public Report*", Naval Ocean Systems Center Technical Document 552, August 1985.

[Lehman, 1984]. Lehman, M. M., "A Further Model of Coherent Programming Processes," *Proceedings, Software Process Workshop*, IEEE, Feb. 1984, pp. 27-33.

[Lehman-Stenning-Turski, 1984]. Lehman, M. M., V. Stenning, and W. Turski, "Another Look at Software Design Methodology," *Software Engineering Notes*, ACM, Apr. 1984, pp. 38-53.

[McCracken-Jackson, 1982]. D. D. McCracken and M. A. Jackson, "Life Cycle Concept Considered Harmful," *Software Engineering Notes*, ACM, April 1982, pp. 29-32.

[Mills, 1971]. Mills, H. D., "Top-Down Programming in Large Systems," in *Debugging Techniques in Large Systems*, R. Ruskin (ed), Prentice-Hall, 1971, pp. 41-55.

[Parnas, 1979]. Parnas, D.L., "Designing Software for Ease of Extension and Contraction," *IEEE Trans. S/W. Engr.*, March 1979, pp. 128-137.

[Roubine, 1985] Roubine, O. "Programming Large and Flexible Systems in Ada", *Ada in Use*, Proceedings of the Ada International Conference, Paris, May 14-16, 1985, pp.197-209.

[Royce, 1970]. Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings, WESCON*, August 1970.

[Stoneman, 1980], "Requirements for Ada Programming Support Environments, Stoneman", Department of Defense, 1980.

[Taylor, et al, 1985]. Taylor, R.N., L. Clarke, L.J. Osterweil, J.C. Wileden, and M.Young, "Arcadia: A Software Development Environment Research Project", *U.C. Irvine Information and Computer Science Technical Report*, Nov. 27, 1985.

Experience Collecting and Analyzing Automatable Software Quality Metrics for Ada*

J. A. Perkins, D. M. Lease, and S. E. Keller

Dynamics Research Corporation
60 Concord Street, Wilmington, Ma., 01887

Abstract

Metrics researchers are currently in the early stages of validating the relationships between metrics and the quality problems encountered by users and developers of software. In order to establish these relationships, large amounts of data defined for validating specific metrics must be collected. Before performing such costly validation, we believe the metrics should be evaluated with respect to whether they reflect our current understanding of quality principles. Our preliminary attempt at validation focuses on a human vs. automated approach to analyzing an existing Ada program. The program consists of fourteen "packages" and approximately 150 "procedures" and "functions". Segments of this code were selected and analyzed with respect to the software quality sub-criteria of flow simplicity, limited visibility, and error prevention and detection. The study focuses on disagreements between human and automated analysis, and attempts to explain those discrepancies and suggest possible ways to improve both measurement techniques and the quality of the software program analyzed.

Keywords

software metrics, software quality, software measurement tools, Ada, software training

1 INTRODUCTION

In the future, software metrics will provide a basis for making scientific predictions of software project parameters. Time to completion of a project, additional spending required to increase product quality by x amount, and prediction of problems before they are out of control are examples of parameters critical to the successful management of software. Metrics relating cost to quality will support the isolation of cost-drivers involved in software development and the evaluation of cost-benefits of alternative resource allocation strategies [Dunham83].

The Ada language provides a syntactic and semantic richness that makes it possible to collect meaningful data by performing a static analysis on existing Ada code,and then defining metrics in terms of this data. These metrics have two main purposes. The first is the improvement of quality of existing code. Secondly, these metrics can form the basis for evaluating the current status of an Ada programmer and pinpointing specific areas where that programmer needs additional training. This can also lead to an evaluation of training methodolgies.

DRC has developed a tool called ADAMAT** (Ada Measurement and Analysis Tool) that performs this static analysis on Ada code. ADAMAT consists of three separate tools that together provide extensive insight into the makeup of an Ada program. The first tool is the automated data collection tool. The data collection tool performs a static analysis of Ada code to collect information about that code. Secondly, a quality analysis component, based on the formal definition of the metrics hierarchy, provides both an interactive analysis of the metrics and a method for pinpointing specific problem areas of the software. Thirdly, a report generator creates a complete report, based on all defined metrics, for any user-determined set of "packages" or subprograms. Each of the tools is implemented in Ada. (For further information on how ADAMAT is implemented, see [Keller85].)

* Ada IS A REGISTERED TRADEMARK OF THE U.S. GOVERNMENT Ada JOINT PROGRAM OFFICE (AJPO)

** ADAMAT IS A TRADEMARK OF DYNAMICS RESEARCH CORPORATION

The metrics themselves exist in a hierarchy based on the McCall metrics framework, tailored to the Ada language. At the lowest level in the framework are data-items, collected by the automated data collection tool directly from Ada source code. Examples of data items are: 1) maximum level of nesting, 2) objects local to a module, and 3) number of "out" parameters in a "procedure". Metric-elements are low-level metrics defined only in terms of these data-items (metric-elements will often be referred to simply as metrics hereafter). Examples of metric-elements are: 1) nesting, which is the inverse of the maximum level of nesting, 2) local "types" referenced, which is the percentage of "types" local to a module that are referenced by that module, and 3) "out" parameters with values, which is the percentage of "out" parameters given a value on all possible execution paths of a "procedure". These metrics are then combined to form software quality sub-criteria, such as flow simplicity, limited visibility, and error prevention and detection. Software quality criteria, such as simplicity, modularity, and anomaly management are then based on these sub-criteria.

The metric-elements are defined by a numerator and a denominator, both non-negative, with the numerator less than or equal to the denominator, rather than a value in the interval [0.0,1.0]. The value of a criterion, sub-criterion, or a metric is the ratio of the sum of all the numerators of metric-elements, to the sum of all the denominators of metric-elements making up that metric. The metric-elements themselves are broken into two distinct categories, absolute and relative. Absolute metrics are measurements of the absolute amount of a software characteristic. Many of the traditional metrics fall in this category. Examples are: 1) the number of lines of code, 2) number of operators, and 3) the number of branches. Absolute metrics support comparisons between problem spaces, comparisons of interest to acquisition managers. The value of most absolute metrics is calculated using one as the numerator, and N+1 as the denominator, where N is the number of occurrences of the language feature [Keller85]. A score other than 1/1 (i.e., a denominator greater than one) for an absolute metric indicates the presence of language features or constructs that cause the code to move away from the desired goal.

An example of an absolute metric within our study is module_exits, which falls under the sub-criterion flow simplicity. The denominator for module_exits is the number of return statements in a "procedure", or the number of return statements minus one in a "function" (a "function" is required to have at least one return). This metric is defined as an absolute metric because program flow is made more complex by the use of multiple exits from a module.

Relative metrics, on the other hand, are metrics that measure actual quality relative to an ideal or potential quality. These metrics address quality within a problem space or problem solution. Examples are: 1) the percentage of non-complex boolean expressions, 2) the percentage of composite "types" which are "private" "types", and 3) the percentage of globals referenced by a subprogram that are declared in the "body" of the "package" containing the subprogram. Relative metrics support comparisons within problem spaces and solutions, comparisons of interest to development managers and training managers. The value of most relative metrics is given by a numerator A and a denominator P, where A is the number of actual occurrences and P is potential number of occurrences of a language feature [Keller85].

An example of a relative metric within our study is lm_types, which falls under the sub-criterion limited visibility. Lm_types has the number of locally defined "types" that are used to declare an object as a numerator, and the number of locally defined "types" as a denominator. Locally defined "types" should always be used to declare a local object within a subprogram, or the "type" should not have been declared within that module.

Metrics researchers are currently in the early stages of validating the relationships between metrics and the quality problems encountered by users and developers of software. In order to establish these relationships, researchers must collect large amounts of data defined for validating specific metrics. Before starting such a costly validation effort, we believe the metrics should be evaluated to determine whether they actually reflect our current understanding of quality principles. To perform this study, we evaluated the Ada code making up the "report_generator" described above.

The investigation focused on determining the answers to the following questions:

1) How well does the automatable measure of quality compare to the quality measurements obtained by human analysis?

2) How well do the metrics detect quality problems in the software.

3) How well do the metrics detect the need for training in specific features of the Ada language?

4) How can the results of questions 1-3 be used to improve the quality and validity of the metrics themselves?

2 METHOD

As was mentioned above, the data we examined was from code for the "report_generator" for the ADAMAT tool. Because of the sheer number of subprograms in the "packages", we selected one module from each "package" for examination. In some cases, the selected module picked was the "package" itself, which was considered to be the "package" without the implementation of the subprograms declared in the "package" specification. Of the fourteen "packages", we selected the package itself in six cases and a subprogram in eight cases.

2.1 Method Of Automated Analysis

The investigated metrics constitiuted three sub-criteria of the current ADAMAT metrics framework. The first sub-criterion, flow simplicity, is part of the criterion simplicity. Secondly, limited visibility is part of the criterion modularity, and thirdly, error prevention and detection is part of the criterion anomaly management.

The flow simplicity sub-criterion attempts to measure the complexity of the flow, both from a traditional and non-traditional point of view. Traditional metrics such as level of nesting and number of branches were collected. We also considered other causes of flow complexity, such as multiple exits from "loops" or subprograms, branch constructs that can exit to more than one location, and branches that can cause program flow to go backward. We also considered a nested branch to be more complex than a non-nested branch.

Under flow simplicity, a total of nine metrics were collected. The metric number is an abbreviation that can be used to access values for this metric from the tables. These metrics were:

FS1) Branch_constructs is an absolute metric. Its denominator is the number of branch constructs plus one. The following are branch constructs: "if", "elsif", "raise", "goto", "return", "loop", "exit", and "case".

FS2) Module_exits is an absolute metric. Its denominator is the number of exits from a module (since every module has at least one exit, it is unnecessary to add one to the denominator). For a "function", the number of exits is the number of return statements. For a "procedure" and a "package body", the number of exits is the number of return statements plus one.

FS3) Level_of_nesting is an absolute metric. The denominator is given by the maximum level of nesting caused by an "if", "loop", or "case" statement ("else" and "elsif" are part of an "if" statement, and any deeper nesting within them is counted) plus one. Therefore, code without nesting will have a denominator of one.

FS4) Non_back_branching_constructs is an absolute metric. The denominator is the number of branch constructs that can result in the return of program control to a line of code that has already executed. The only branch constructs that can cause return of control to a line already executed are "goto's" or "loops".

FS5) Branch_and_nesting is an absolute metric. Its denominator is the sum of the level of nesting over all branch constructs plus one. For example, a branch construct at the second level counts as two, a branch construct at level three counts as three, etc. The level of nesting is said to be one if no nesting has occurred. Therefore, the first line of code is at level one, and the denominator will be one if no nesting occurs. Nesting is defined as in (FS4).

FS6) Multiple_exit_loops is a relative metric. The numerator is the number of "loops" with only one exit. A "for loop" or "while loop" has one exit if no "exit" statement, "raise" statement, or "goto" which leaves the "loop" is within the "loop". A "basic loop" has more than one exit if more than one "exit" statement, "raise" statement, or "goto" which leaves the "loop" is found in the "loop". The denominator is the number of "loops".

FS7) Structured_branch_constucts is a relative metric. The numerator is the total number of structured branch constructs. Any branch construct from (FS1) above is structured except "goto", "raise", "return", or "exit name", where "name" is the name of a outer level "loop". These control constructs are non-structured because they cause a jump in program control from the current location to a location other than the next line of code or end of the current construct. The denominator is the number of branch constructs as defined in (FS1).

FS8) For_loops is a relative metric. The numerator is the number of "for loops". The denominator is the total number of "loops".

FS9) Multiple_exit_location_loops is a relative metric. The numerator is the number of "loops" where control cannot be given to a statement other than the one immediately following the "end loop". Control can be given to a different location if the "loop" contains a "raise", "goto", or "return" statement, or if the "loop" is nested inside another "loop", with an "exit name" statement within the inner "loop".

The limited visibility sub-criterion attempts to measure how well items that are not needed within units are hidden from the units. Examples of items that we believe should be hidden if they are not used are: user-defined "types" and "subtypes", variables, constants, "exceptions", "procedures", and "functions". If these items are made visible when they are not intended to be used, program reliability may suffer from unanticipated usages.

Under limited visibility, ten metrics were collected. These were:

LV1) Withed_packages_referenced is a relative metric. The numerator is the number of visible "packages" that were referenced in any of the following ways: via a subprogram call, use of an object, raising or handling an exception, or using a "type" that resides in the "withed" "package". The denominator is the number of "packages" made visible via "withs", either by a "subprogram", or the "package" "body" or specification.

LV2) Lm_types is a relative metric. The numerator is the number of "types" declared local to the module that are referenced by that module. If the module is a subprogram, the "types" local to it are those "types" and "subtypes" declared within the module. If the module is a "package", the "types" local to it are all those "types" and "subtypes" declared within the "package" specification, or within the declarative region of the "package" "body". A "type" is defined to be referenced if the name of the "type" appears in any statement within the module. The denominator is the number of "types" local to the module.

LV3) Lm_objects is the same as Lm_types, except objects local to a module are used instead of local "types". Objects are variables,"constants", and "exceptions". An "exception" is used if it is "raised" or handled.

LV4) Lm_operators is the same as lm_types and lm_objects, except user-defined operators are counted. User-defined operators include functions and procedures.

LV5) Lmp_types is a relative metric. The numerator is the number of "types" declared local to a module's "package" that are referenced by that module. The denominator is the number of "types" declared local to a module's "package" For a subprogram, a "type" is local to its "package" if it is declared either in the "package" specification or in the declarative portion of the "package". For a "package", there are no "types" local to the module "package", unless the "package" is nested within another "package". In that case, any declarations within the outer "package" specification that are visible to the nested "package" are the "types" local to the "package".

LV6) Lmp_objects is the same as Lmp_types, except objects are counted.

LV7) Lmp_operators is the same as Lmp_types and Lmp_objects, except user-defined operators are counted.

LV8) Emp_types is a relative metric. Its numerator is the number of "types" made visible to the module via "withs" that are referenced. Its denominator is the number of "types" made visible to the module via "withs".

LV9) Emp_objects is the same as Emp_types, except objects are counted.

LV0) Emp_operators is the same as Emp_types and Emp_objects, except user-defined operators are counted.

The error prevention and detection sub-criterion does not attempt to predict the number of errors that will be encountered when code is executed. Rather, it attempts to measure how well features of Ada were used to prevent or detect possible errors. Examples of this kind of prevention/detection include insuring that subexpressions are constraint-checked, insuring that "out" parameters are always given values, and using default initializations for variables. Metrics measuring the proper use of "exceptions" were not collected for this study. Since the proper use of "exceptions" limits their use to situations where an error has already occurred, "exceptions" are addressed under the separate sub-criterion error handling, which resides with error prevention and detection under anomaly management.

Under error prevention and detection, five metrics were collected. These were:

E1) Default_init is a relative metric. The number is the number of locally declared variables which are given a default initialization. The denominator is the number of locally declared variables.

E2) User_types is a relative metric. The numerator is the number of user-defined "types" used by the "procedure". A "type" is defined to be used if its name appears in any statement within the module. The denominator is the total number of "types", both user-defined and system-defined, used by the "procedure".

E3) And_then_or_else is a relative metric. The denominator measures the number of times that an index into an array was range checked in a compound boolean expression. The numerator measures the number of times that the compound boolean expression used an "and then" or an "or else" construct to avoid the possible raising of CONSTRAINT_ERROR. For example,

```
a: array(1..10) of integer;
begin
loop
if (i in 1..10) and (a(i) > 0) then ...
```

may "raise" CONSTRAINT_ERROR even if i is not in the legal range, because the order of evaluation for the boolean operands is not defined: however, this cannot happen in the following case.

```
a: array(1..10) of integer;
begin
loop
 if (i in 1..10) and then (a(i) > 0) then ...
```

E4) Out_params_w_values is a relative metric. Its numerator is the number of "out" parameters guaranteed to have a value assigned into them on all paths if no "exception" is "raised". The denominator is the number of "out" parameters. "In out" parameters are not considered to be "out" parameters for this metric because they must have a value when the procedure call is made, or the program is erroneous.

E5) Constraint_checking is a relative metric. The numerator is the number of sub-expressions which are constraint checked. The denominator is the total number of sub-expressions.

2.2 Method Of Human Analysis

As our baseline for comparison, we chose a human analysis of the modules we examined. Due to the difficulty of giving absolute metric scores to the modules, we opted for a ranking system. For each sub-criterion, the modules were given a ranking from one to fourteen, with one being the best score and fourteen the worst. This human analysis caused us to limit ourselves to the small number of "procedures" that we analyzed. To accurately rank the code, the total number of ranked modules had to be small. However, within each "package", the most involved "procedure" was selected for rating. In some cases, no "procedure" involved any code other than simple variable assignments, so we opted instead to investigate the "package" itself.

Definitions of sub-criteria

The human analysis of the modules is based on the following definitions of the sub-criteria:

Flow Simplicity:

The control flow of the module is easily understood for any combination of the possible input values to the module.

Limited Visibility:

The visibility of the module is such that the visible software items are limited to items appropriate for that module.

Error Prevention and Detection:

The implementation of the module is such that all errors that can occur during elaboration or execution of that module are prevented and/or detected.

These definitions were developed to provide direction for the human analysis at the intuitive level without actually specifying what features of the language should be investigated or how much any positive or negative factor of the software should be weighted.

The human analysis involved four separate passes over the set of modules. Each pass involved investigating the set of modules with respect to flow simplicity, then limited visibility, and then error prevention and detection. On the first pass, the modules were not investigated in any particular order. On the second and third pass, successive pairs of modules were investigated according to the rankings established on the previous pass. The ranking of the modules was modifed where appropriate. On the fourth pass, the distinguishing factors that dictated the rating were identified. These distinguishing factors were then used to establish a final set of rankings.

The distinguishing factors dictating the ranking of the modules by human analysis for each of the sub-criteria are as follows:

Distinguishing Factors for Flow Simplicity:

FSF1) the number of branches (ie. "if", "case", and "loop"),

FSF2) the number of back branches,

FSF3) the number of "loops" containing a single statement,

FSF4) the number of "loops" with multiple exits,

FSF5) the number of returns from the module, and

FSF6) the number of levels of nesting.

Distinguishing Factors for Limited Visibility:

LVF1) the number of sets of operators "withed" that are visible,

LVF2) the number of global variables that are visible,

LVF3) the number of sets of operators "withed" that are unused, and

LVF4) the number of global variables visible that are unused.

Distinguishing Factors for Error Prevention and Detection:

EF1) the number of instances where variable declarations are not isolated to a "loop" (where possible),

EF2) the number of instances where array slices are not used (where possible),

EF3) the number of unrelated "types" declared in the specification of a "package",

EF4) the number of instances where "and then" is not used when required, and

EF5) the number of variables declared in the specification of a "package".

2.3 Guidelines For Implementation

Prior to design and implementation of the "report_generator", we established a list of implmentation guidelines. A discussion of the Ada-specific guidelines for each of the selected criteria aids in understanding the results of both the human analysis and the automated analysis.

This list does not contain Ada-specific guidelines concerning flow simplicity, but does contain Ada-specific guidelines about limited visibility and error detection and prevention.

The Ada-specific guidelines relating to limited visibility are as follows:

LVG1) Every "package" specification should conceptually represent a single object.

LVG2) Every "type", object, and operator declared in a "package" specification should be referenced by some other "package" of the "report_generator".

LVG3) A library unit should be "withed" only if some "type", object, or operator in that "withed" unit is to be directly referenced.

LVG4) No "package" should be created that merely subsets the set of visible "types", objects, or operators of another "package" specification.

The Ada-specific guidelines relating to error prevention and detection are as follows:

EG1) No variables should be declared in a "package" specification.

EG2) Every variable should be declared using a user-defined "type", or be of "type" boolean, and each constant should be declared using a user-defined "type" or anonymous type.

EG3) Every variable should be initialized when declared.

EG4) The short circuit operators "and then" and "or else" should be used only when the semantics of the compond boolean expression dictates the order of evaluation.

EG5) Every "procedure" should initialize all "out" mode parameters at the beginning of the sequence of statements in a body of code.

The rationale for most of the above guidelines is obvious. The rationales for guidelines LVG4 and EG5 are given below.

Rationale for LVG4

Although creating multiple views of a "package" is a useful method of limiting visibility, this benefit can only be realized if the the different views are placed in separate library units. Given the limited Ada environment available to us (specifically, the TeleSoft Version 1.5 and 2.1 compilers on a VAX/VMS operating system), we decided that the increased compilation difficulties that we encountered by using more library units outweighed the benefits of multiple views.

Rationale for EG5

We believe that initializing "out" mode parameters at the start of each "procedure" will reduce the likelihood that undefined values are returned in cases where defining an "out" parameter is not part of the the functionality. For example, a "procedure" that performs a search often returns two values, where one value is the location of the object, and the other value is a flag that indicates if the desired object was found. In the case where the object is not found, the value representing the location is unimportant, but failure to define this value results in an erroneous program.

The above guidelines are a subset of the guidelines which governed the design and implementation of the "report_generator". Most of the other guidelines are traditional guidelines that apply to programming in any standard Von Neuman language.

3 RESULTS

3.1 Results Of Automated Analysis

The automated metric scores of the selected modules for flow simplicity, limited visibility, and error prevention and detection are shown in Table 1. The metric-element scores of the modules for the sub-criteria are shown in Tables 2, 4, and 5. The metric scores for limited visibility are broken into "local_to_module", "local_to_package", and "external_to_package" in Table 3.

Module Name	Flow Simplicity		Limited Visibility		Error P & D	
	N/D	Score	N/D	Score	N/D	Score
type	5/5	1.00	49/84	0.58	83/85	0.98
time	5/5	1.00	14/100	0.14	8/8	1.00
set	5/5	1.00	10/92	0.11	8/9	0.89
module	5/5	1.00	14/97	0.14	5/5	1.00
format	4/4	1.00	7/10	0.70	8/11	0.73
set_format	5/5	1.00	3/11	0.27	11/11	1.00
next_file	6/9	0.67	8/123	0.07	7/8	0.88
find_file	9/36	0.25	18/145	0.12	9/14	0.64
read_module	15/29	0.52	16/120	0.13	14/16	0.88
module_order	9/17	0.53	28/164	0.17	13/24	0.54
next_trace	7/14	0.50	9/143	0.06	10/11	0.91
store_trace	16/55	0.29	54/232	0.23	17/33	0.52
make_report	9/17	0.53	14/16	0.88	2/4	0.50
fetch_level	13/31	0.42	19/199	0.10	8/14	0.57

TABLE 1. SCORE BY SUB-CRITERIA FOR ALL MODULES

Module Name	FS1	FS2	FS3	FS4	FS5	FS6	FS7	FS8	FS9
type	1/1	1/1	1/1	1/1	1/1	0/0	0/0	0/0	0/0
time	1/1	1/1	1/1	1/1	1/1	0/0	0/0	0/0	0/0
set	1/1	1/1	1/1	1/1	1/1	0/0	0/0	0/0	0/0
module	1/1	1/1	1/1	1/1	1/1	0/0	0/0	0/0	0/0
format	1/1	1/1	1/1	0/0	1/1	0/0	0/0	0/0	0/0
set_format	1/1	1/1	1/1	1/1	1/1	0/0	0/0	0/0	0/0
next_file	1/2	1/1	1/2	1/1	1/2	0/0	1/1	0/0	0/0
find_file	1/6	1/2	1/4	1/2	1/14	0/1	4/5	0/1	0/1
read_module	1/5	1/1	1/3	1/3	1/7	2/2	4/4	2/2	2/2
module_order	1/3	1/1	1/2	1/2	1/4	1/1	2/2	0/1	1/1
next_trace	1/3	1/1	1/3	1/1	1/4	0/0	2/2	0/0	0/0
store_trace	1/9	1/1	1/5	1/2	1/27	1/1	8/8	1/1	1/1
make_report	1/3	1/1	1/2	1/2	1/4	1/1	2/2	0/1	1/1
fetch_level	1/5	1/1	1/3	1/3	1/9	2/2	4/4	0/2	2/2

TABLE 2. FLOW SIMPLICITY METRIC-ELEMENT SCORES

Module Name	Local to Module		Local to Package		External to Package	
	N/D	Score	N/D	Score	N/D	Score
type	49/84	0.58	0/0	1.00	0/0	1.00
time	1/15	0.07	0/0	1.00	12/84	0.14
set	4/7	0.57	0/0	1.00	5/84	0.06
module	5/12	0.42	0/0	1.00	8/84	0.10
format	7/10	0.70	0/0	1.00	0/0	1.00
set_format	0/0	1.00	0/0	1.00	2/10	0.20
next_file	3/3	1.00	2/18	0.11	2/100	0.02
find_file	7/8	0.88	0/21	1.00	8/113	0.07
read_module	3/3	1.00	2/14	0.14	9/101	0.09
module_order	12/12	1.00	2/5	0.40	9/142	0.06
next_trace	3/3	1.00	4/33	0.12	1/105	0.01
store_trace	17/17	1.00	20/44	0.45	15/163	0.09
make_report	3/3	1.00	0/0	1.00	10/12	0.83
fetch_level	6/6	1.00	1/12	0.08	8/176	0.05

TABLE 3. LIMITED VISIBILITY SCORES BROKEN DOWN BY LOCATION OF TYPES, OBJECTS AND OPERATORS

Module Name	LV1	LV2	LV3	LV4	LV5	LV6	LV7	LV8	LV9	LV0
type	0/0	29/29	20/54	0/1	0/0	0/0	0/0	0/0	0/0	0/0
time	1/1	1/1	0/1	0/13	0/0	0/0	0/0	6/29	6/54	0/1
set	1/1	3/3	1/1	0/3	0/0	0/0	0/0	5/29	0/54	0/1
module	1/1	2/2	2/2	1/8	0/0	0/0	0/0	5/29	3/54	0/1
format	0/0	7/7	0/3	0/0	0/0	0/0	0/0	0/0	0/0	0/0
set_format	1/1	0/0	0/0	0/0	0/0	0/0	0/0	0/7	2/3	0/0
next_file	1/2	0/0	3/3	0/0	0/2	1/4	1/12	1/30	1/55	0/15
find_file	3/3	0/0	7/8	0/0	0/0	0/4	0/17	3/30	1/55	4/28
read_module	2/2	0/0	3/3	0/0	1/1	1/3	0/10	1/30	4/58	4/13
module_order	5/5	0/0	12/12	0/0	0/1	0/1	2/3	0/31	1/59	8/52
next_trace	1/2	0/0	3/3	0/0	1/4	2/3	1/26	1/29	0/57	0/19
store_trace	2/8	0/0	17/17	0/0	2/3	0/2	18/39	15/38	0/61	0/64
make_report	1/1	0/0	3/3	0/0	0/0	0/0	0/0	2/3	0/1	8/8
fetch_level	4/5	0/0	6/6	0/0	1/3	0/2	0/7	3/33	0/59	5/84

TABLE 4. LIMITED VISIBILITY METRIC-ELEMENT SCORES

Module Name	E1	E2	E3	E4	E5
type	0/0	29/31	0/0	0/0	54/54
time	0/0	7/7	0/0	0/0	1/1
set	0/1	8/8	0/0	0/0	0/0
module	0/0	5/5	0/0	0/0	0/0
format	0/2	7/8	0/0	0/0	1/1
set_format	0/0	0/0	0/0	0/0	11/11
next_file	0/0	1/2	0/0	2/2	4/4
find_file	0/4	3/4	0/0	2/2	4/4
read_module	0/2	2/2	0/0	1/1	11/11
module_order	2/11	10/12	0/0	0/0	1/1
next_trace	0/0	2/3	0/0	2/2	6/6
store_trace	0/15	16/17	0/0	0/0	1/1
make_report	0/1	2/3	0/0	0/0	0/0
fetch_level	1/5	4/5	0/0	0/1	3/3

TABLE 5. ERROR PREVENTION AND DETECTION METRIC-ELEMENT SCORES

3.2 Results Of Human Analysis

The final rankings of the selected modules for flow simplicity, limited visibility, and error prevention and detection based on these distinguishing factors are shown in Tables 6, 7, and 8 respectively. The horizontal lines in these tables separate modules into groups according to the distinguishing factors. The lower the module is in the table, the lower the ranking of that module according to the human analysis.

Although the relationship between the distinguishing factors and the ranking of the modules is illustrated in Tables 6, 7, and 8, further discussion is in order.

For flow simplicity, the two primary factors affecting the ranking are the number of branches (FSF1) and the number of back branches (FSF2). However, the other three factors (FSF3, FSF4, and FSF5) did have an effect.

"Read_module" is ranked higher in the human analysis than "fetch_level" despite "read_module" having more branches and the same number of back branches as "fetch_level". "Read_module" has two "for loops" that contained a single statement. The purpose of these two "loops" is to perform a copy of one array into another. In fact, the "loops" could be replaced by array slices. These simple "loops" were considered less of a factor in increasing the difficulty of understanding the flow than other branches. "Read_module" is considered to be only sightly worse than "next_trace" which has two branches and no back branches.

"Find_file" is in a lower grouping than "fetch_level" because "find_file" has both multiple returns from the module and multiple exits from a "loop" (a single "return" inside a "basic loop" is the reason for both of these conditions). In fact, "find_file" is grouped with "store_trace" which has more branching and deeper nesting.

For limited visibility, the ranking of the modules can be reduced to two major considerations: the number of conceptual objects available to the module and the number of the conceptual objects available to the module that are not used. The conceptual objects are the sets of operators made visible to the module by "withing" and the set of variables made visible to the module by declarations in the "package" containing the module.

For error prevention and detection, the ranking of the modules can be reduced to two considerations. The first consideration is whether any unnecessary errors occur or any unavoidable errors fail to be detected when the present code is executed. The second consideration is whether any modification can be performed to the code of the module to aid in preventing the introduction of errors into the code during maintenance.

All the modules in group 2 of Table 6 except "next_trace" fail to reduce the scope of variables where possible. Each of these modules contains a "loop" that has the following property:

> The "loop" contains variables declared outside the scope of the "loop" whose values depend only on the current instance of the "loop".

Declaring these kinds of variables within the "loop" would eliminate the possibility that their values are used across "loop" instances or outside the scope of the "loop".

"Next_trace" implements the copying of arrays using "for loops". The use of array slices to perform these copies would decrease the likelihood of inadvertent changes to the intended functionality.

"Type" is a "package" that declares many "types" in the "package" specification. Although these "types" are either discrete "types" or "string" "types", the "types" are unrelated. Separating these "types" into individual "packages" would remove the need for users of these "types" to have visibilty to other unneeded "types" in "package" "type".

"Next_file" is ranked low with regard to error prevention and detection because of the difficulty of understanding the simple functionality provided. The intent of "next_file" is to return the next file in a list of files. At one point in "next_file", a check is made to see if the value of the input parameter references an existing file and to check that the reference is not to the last file in the list. The check is performed by calling a "function" called "validate_file_reference", which is declared in the specification of the "package" containing "next_file", and then by checking that the value is less than "list_tail", a variable declared in the "body" of that "package". This check is written as follows:

```
if     validate_file_reference(file_reference)
   and file_reference < list_tail
```

Although the intended functionally suggests the potential need for the short circuit operator "and then", examination of "validate_file_reference" shows that the above check is functionally equivalent to the following:

```
if     list_head <= file_reference
   and file_reference =< list_tail
   and file_reference < list_tail
```

This, of course, can be reduced to either of the following:

```
if     list_head <= file_reference
   and file_reference < list_tail
```

```
 if file_reference in list_head..list_tail-1
```

Rewriting the check in "next_file" as indicated above would significantly decrease the difficulty of understanding the simple functionality that is being performed.

"Format" is a "package" that declares variables in the "package" specification. The users of "format" are supposed to be able to read the desired form for the report being generated, but are not supposed to be able to modify that form. Having access to the variables declared in the "package" specification provides the undesired capability to modify the form of the report. Since the contents of the format variables are read from an external file, a constant declaration is not appropriate. However, declaring the variables in the "body" of "format" and providing only "functions" in the specification of "format" would eliminate the possibility of users changing the desired form of the report.

Flow Simplicity

Module Name	Distinguishing Factors		
type	0 branches		
time	0 branches		
module	0 branches		
set	0 branches		
format	0 branches		
set_format	0 branches		
next_file	1 branch	0 back branches	
make_report	1 branch	1 back branch	
module_order	1 branch	1 back branch	
next_trace	2 branches	0 back branches	
read_module	4 branches	2 back branches	2 simple loops
fetch_level	2 branches	2 back branches	
find_file	3 branches	1 back branch	2 returns
store_trace	8 branches	1 back branch	deep nesting

TABLE 6. FLOW SIMPLICITY RANKING FACTORS

Limited Visibility

Module Name	Distinguishing Factors	
type	0 withs	
format	0 withs	
time	1 with	
set	1 with	
module	1 with	
make_report	1 with	1 set used
set_format	1 with	2 withed variables used
read_module	2 withs	1 global variable unused
next_trace	2 withs	1 set unused
find_file	3 withs	All global variables unused
module_order	5 withs	All global variables unused
fetch_level	5 withs	1 set unused
next_file	2 withs	1 set & 1 global variable unused
store_trace	8 withs	6 sets unused

TABLE 7. LIMITED VISIBILITY RANKING FACTORS

Error Prevention and Detection

Module Name	Distinguishing Factors
time	
set	
module	
set_format	
make_report	loop isolation
store_trace	loop isolation
module_order	loop isolation
find_file	loop isolation
fetch_level	loop isolation
next_trace	loop isolation
read_module	array slices
type	unrelated type declared in specification
next_file	and then
format	variables declared in specification

TABLE 8. ERROR PREVENTION AND DETECTION RANKING FACTORS

4 DISCUSSION

Our discussion of the results consists of comparing the human and automated analysis, using the metric scores to pinpoint quality problems and training deficiencies, and suggesting changes to the metrics framework.

4.1 Comparison Of Human And Automated Analysis

Comparisons of the rankings obtained through the human analysis and the automated metric analysis are shown in Tables 9, 10, and 11. The rankings for flow simplicity compare favorably, but the correspondence between the rankings for the other two sub-criteria is not readily apparent.

For flow simplcity, the rankings are similiar because the metric-elements used to define the sub-criteria and the distinguishing factors used to produce the human rankings are similar. Both use the concepts of branches, back braches, returns from module, exits from "loops", and level of nesting. The human analysis uses each of these in an absolute sense whereas the automated analysis uses some in an absolute sense and others in a relative sense.

Two of the metric-elements, branch and nesting (FS5) and structured branch constructs (FS7), are not specifically mentioned as distinguishing factors but are clearly related to the distinguishing factors (FSP1, FSP2, FSP5) used. In fact, the count for branches in the human analysis corresponds more to a count of the number of uses of structured branch constructs ("if", "case", and "loop") than to a count of the number of branch constructs ("if", "case", "loop", "exit", "return", "raise", "goto"). The only metric element not clearly related to a distinguishing factor is "for_loops" (FS8) and the only distinguishing factor not related to a metric-element is simple "loops" (FSP3).

For limited visibilty, the relationship between the rankings is obscured by the number of data-items used and the means of ranking the modules based on the metric scores. The human ranking is based on only two factors, the number of conceptual objects visible and the number of conceptual objects not used. Moreover, the human ranking uses these factors in an absolute sense. A few of the metric-elements (LV1, LV6, LV9 and LV0) capture most of the information considered in the human analysis. None of the other metric-elements used to defined limited visibility were considered as factors in the human analysis. Also, all of the metric-elements constituting this sub-criterion were interpreted in a relative sense.

For error prevention and detection, any correspondence between the two rankings is coincidental. The distinguishing factors used in the human analysis have no correlation to the metric-elements used in the automated analysis. In general, the combination of the simple functionality of the "report_generator" and adherence to the guidelines outlined in Section 3.3 prevented any of the software principles measured by the metric-elements of this criteria from being a distinguishing factor in the human ranking of the modules. The paragraphs to follow examine why each of the metric-elements does not have a corresponding distinguishing factor.

Default_init(E1) measures the proportion of variable declarations that are initialized as part of the declaration. The human analysis found that the only variable declarations that violated guideline EG3 are variables that are clearly defined by a call to a "procedure" before being read. The human analysis also revealed that the "types" used in the declaration of these variables have no values that correspond to undefined or null. Based on these two factors, default initalization of these variable is considered unnecessary and the use of initialization would be misleading.

User_types(E2) measures the proportion of the "types" used that are user declared. The human analysis found adherence to guideline EG2. In fact, there are only two system-supplied "types" used, namely string and boolean, and these are used only occasionally.

And_then_or_else(E3) measures the potential need for the short-circuit operators. Only "next_file" has this potential need and the automated analysis did not capture the potential problem due to the form of the expression of the "if".

Out_params_w_values(E4) measures the proportion of "out" parameters that are defined for all paths. Adherence to guideline EG4 results in all "out" parameters having this property.

Constraint_checked(E5) measures the proportion of sub-expressions that are constraint checked. Almost all of the sub-expressions are constraint checked due to the simp icity of the numeric processing and the array indexing needed to perform the functionality of the "report_generator".

Flow Simplicity

Module Name	Human Analysis Rank	Metric Score Rank
type	1	1
time	1	1
module	1	1
set	1	1
format	1	1
set_format	1	1
next_file	7	7
make_report	8	8
module_order	8	8
next_trace	10	11
read_module	11	10
fetch_level	12	12
find_file	13	14
store_trace	14	13

TABLE 9. FLOW SIMPLICITY RANKING

Limited Visibility

Module Name	Human Analysis Rank	Metric Score Rank
type	1	3
format	2	2
time	3	8
set	3	11
module	3	7
make_report	6	1
set_format	7	4
read_module	8	9
next_trace	9	14
find_file	10	10
module_order	11	6
fetch_level	11	12
next_file	13	13
store_trace	14	5

TABLE 10. LIMITED VISIBILITY RANKING

Error Prevention and Detection

Module Name	Human Analysis Rank	Metric Score Rank
time	1	1
set	1	6
module	1	1
set_format	1	1
make_report	5	14
store_trace	5	13
module_order	5	12
find_file	5	10
fetch_level	5	11
next_trace	5	5
read_module	5	7
type	12	4
next_file	13	7
format	14	9

TABLE 11. ERROR PREVENTION AND DETECTION RANKING

4.2 Pinpointing Quality Problems

Although ranking the overall quality of the software system is an important exercise, the ability to locate specific problems is equally important. The paragraphs to follow indicate how the automated metric scores for the set of modules can be used to pinpoint problem areas.

The metric scores of the modules for limited visibility are extremely low. In fact, an examination of the scores for the metric "external_to_package" indicates that 10 of the 14 "packages" have more than 70 "types", objects, or operators that are not used. Examination of metric-elements LV8 and LV9 indicate that in all of these cases, many of the unused externals are "types" or objects. The common denominator for all of these modules is the "withing" of "package" "type". The metric-elements LV2 and LV3 show that "package" "type" declares 29 "types" and 54 objects. The metric scores point out the need to reorganize the contents of "type" into several "packages".

An examination of the scores for metric-element E1 indicates that default initialization does not accompany most variable declarations. The reasons for this have been discussed in the previous section. The metric scores indicate the need to investigate whether the "types" in the "report_generator" can be re-defined so that default initialization is meaningful for all local variable declarations (including variables that are defined before used).

An examination of metric-elements LV8, LV9, and LV0 indicate that 6 of the 12 modules which have access to "withed" information use the "types" and objects but not the operators from the "withed" "packages". An examination of the data items indicated that in each case the objects are constants. The metric scores point out the need to allow access to the "types" and constants declared in these "package" specifications without providing access to the operators declared in that specification. We concluded that more than one view of a "package" may be required for some of the "packages" in the "report_generator".

An examination of metric-elements LV6 and LV7 indicates that two of the three modules that use operators declared "local_to_package" do not use the variables declared in that "package". The metric scores reflect the need to further layer the architecture of the "report_generator".

Although an examination of the scores for metric-element LV9 for "package" "set_format" does not indicate the presence of the two variables declared in the "package" specification of "format", the broader definition of limited visibility that we use for ADAMAT would indicate this problem. In the overall framework, metric-element LV9 is not a metric-element but a metric. LV9 is defined to be the sum of the usage of external constants(LV91) and the usage of external variables(LV92). For "set_format", LV91 has the score 0/1 and LV92 has the score 2/2 which results in LV9 having the score 2/3.

4.2.1 Modifications To The Report Generator

The "report_generator" is presently being modified based on both the automated and human analysis. The paragraphs that follow discuss these changes.

Three changes are made to each of the "types" contained in "package" "type". First, each of the "types" is moved into a separate "package". Second, each of the "types" is defined so as to contain a null or undefined value. Third, each of these "packages" is given three views. For example, "sort_type" will be "packaged" as follows:

```
package sort_value_type_package is

  type improper_sort_value_type is
    (undefined_first_sort_value,
     module_order,
     ...
     metric_value,
     undefind_last_sort_value);

  type sort_value_type is
    range module_order .. metric_value;

end package sort_value_type_package;

with sort_value_type_package;
use sort_value_typepackage;
package sort_package is

  type sort_type is private;
  undefined_sort: constant sort_type;
  default_sort: constant sort_type;

  ...

  function succ_sort
    (sort: sort_type)
  return sort_type;

  ...

private

  ...

end sort_package;
```

```
with sort_package;
package sort_type_package is

  subtype sort type is
    sort_package.sorttype;
  undefined_sort: constant sort_type
    := sort_type_package.undefined_sort;
  default_sort: constant sort_type
    := sort_type_package.default_sort;

end sort_type_package;
```

"Packaging" the "types" in this manner directly addresses three of the problems pinpointed by the automated analysis. Unnecessary "types" no longer need to be visible to "packages" that need only a few of the "types". Operators for a "type" no longer need to be visible in cases where only the "type" and associated constants are required. The undefined value can be used to initialize variable declarations in cases where the use of another value would be misleading. The undefined value is also useful for initializing "out" parameters.

Both "store_trace" and "module_order" are "packaged" separately from operators that are used local to a "package". For example, "store_trace" is placed in a "package" that "withs" the "trace_package", where "trace_package" is the "package" currently containing "store_trace".

"Packages" "format" and "set_format" are merged. The variables in the specification of "format" are placed in the corresponding "package body". "Functions" which provide read only access to the desired format of the report are declared in the "package" specification.

User-defined "types" are defined for those cases where type string is currently used. "Type" "boolean" will continue to be used in the "report_generator".

The check in "next_file" is clarified by using the membership operator. Variable declarations are isolated within "loops" where possible.

4.2.2 Effect Of The Modifications On The Metric Scores

The modifications will result in changes to the metric scores for the "report_generator". The new scores for limited visibility and error prevention and detection are the most interesting, since the modifications to the sub-criteria have the greatest impact on these scores.

For error prevention and detection, the problems detected by automated analysis are 1) the lack of default initialization and 2) the use of the "types" "string" and "boolean". The modifications raise the scores for metric-elements E1 and E2 to near 1.0.

For limited visibility, the modifications directly address the problems detected by the metrics. Enough of the modifications have been completed to allow collection of new metric scores for 6 of the orginal modules. The new metric-element scores for limited visibility are shown in Table 12.

Examination of Table 12 indicates that 1) all but one "package" "withed" is used, 2) the scores for "external_to_package" improved for all modules except "store_trace", 3) based on the scores for "local_to_package", further layering may be required for "find_file", and 4) the "local_to_module" scores are low for all the modules that are "packages" and high for all the modules that are subprograms.

Module Name	LV1	LV2	LV3	LV4	LV5	LV6	LV7	LV8	LV9	LV0
set	2/2	1/1	0/2	0/5	0/0	0/0	0/0	2/2	6/6	0/0
module	3/3	1/1	0/2	0/7	0/0	0/0	0/0	3/3	9/9	0/0
time	6/6	1/1	0/2	0/13	0/0	0/0	0/0	6/6	12/12	0/0
make_report	4/4	0/0	3/3	0/0	0/0	0/0	0/0	3/3	3/6	8/8
find_file	3/3	0/0	1/1	0/0	0/0	0/3	3/6	1/2	1/4	2/7
store_trace	19/20	0/0	2/2	0/0	1/1	0/0	0/2	2/19	2/98	38/124

TABLE 12. NEW LIMITED VISIBILITY METRIC-ELEMENT SCORES

4.3 Identifying Training Needs

In general, relative measures are useful for indicating the need for training in specific features of the Ada language. The following paragraphs examine the relationship between the metric scores and the need for training.

For flow simplicity, the metric-element scores over the set of modules for 3 of the 4 relative metrics are excellent. The metric scores for FS6, FS7, and FS9 indicate there is no need for training in the area of structured programming. The score for "for loops"(FS8) indicates that only 3 of the 8 "loops" are "for loops". Although there are advantages to using "for loops" over other "loops" when possible, the score of 3/8 does not warrant the need for training. In fact, this metric, despite being a relative metric, may not be an appropriate metric for measuring the need for training.

For limited visibility, the overall metric score is low for most of the modules. An examination of the scores for "local_to_module", "local_to_package", "external_to_package" indicates that in general, the "local_to_module" usage is good, the "local_to_package" fair, and the "external_to_package" usage poor. Clearly, training in how to limit visibility to unneeded external information is required.

For error detection and prevention, the overall metric score is high for most of the modules. In fact, 3 of the 5 metrics have excellent scores. No conclusion can be reached concerning the need for training in the area of the short-circuit operators. The scores for default_init(E1) are extremely low. The need for training in the area of default initialization of variable declaration appears warranted.

4.4 Modifications To The Metrics

Improving the metrics for each of the three sub-criteria is a primary goal of this study. The following paragraphs discuss the possible modifications to the metrics based on the analysis of Section 5.1, 5.2, and 5.3.

A major difference between the human analysis and the automated analysis rankings is that the human analysis involves interpreting the information in an absolute sense, whereas the automated analysis combines both an absolute and relative interpretation.

We have determined that both forms of measurements are useful. Therefore, three measurements based on the numerator and denominator of relative metrics are in order. The first is the current measurement (good/total), to provide a measure of the proportion of times a specific principle is followed. The second is 1/total, providing an absolute measure of complexity with respect to a specific principle. The third is 1/(total-good), which is an absolute measure of the number of violations of a specific principle.

For example consider metric-element LV1. LV1 is currently defined to be the proportion of "withed" "packages" that are referenced by the module. Under the proposed scheme, LV1 would have three definitions. The first is as currently defined, the second is the number of "packages" "withed", and the third is the number of "packages" "withed" that are not used.

The "report_generator" is being modified to allow the user of ADAMAT to rank the modules according to any of these interpretations.

Notice that absolute metrics have the same meaning under any of the three intrepretations.

The human analysis indicates that the following four metrics are candidates for addition to the metrics framevork:

1) The proportion of locally declared variables that cannot be isolated in a "loop" contained within the module.

2) The proportion of "for loops" that cannot be eliminated through the use of array slices.

3) The number of "if" constructs not using a short-circuit operator when required.

4) The proportion of "packages" that do not contain variable declarations.

The use of the metric scores to locate quality problems in the "report_generator" indicates that the following metrics are candidate new metrics:

1) The proportion of modules that use types and/or objects from a "withed" "package" without using any of the operators in that "package".

2) The proportion of modules declared in a package that use other operators from the package without using any of the variables declared in that package.

The new scores for the modified modules indicate that the definition of "local_to_module" may need to differ depending on whether the module is a "package" or a subprogram.

We are investigating the feasibility of incorporating each of these modifications into our current metrics framework.

5 CONCLUSION

Our analysis indicates that the metrics for the three sub-criteria flow simplicity, limited visibility, and error detection and prevention are useful in pinpointing quality problems in existing software and for identifying specific features of the Ada language where training is required.

The comparison of the human and automated analysis suggests three interpretations of the metrics in our framework.

The identification of the distinguishing factors that determined the human ranking and identification of deficiencies in the analyzed software indicates the need for additional metrics in our ADAMAT framework.

Each of the above conclusions is encouraging. However, our most encouraging finding was that the investigation of the scores for existing metrics lead to the discovery of new software principles that can be used as the basis for improving the exisitng metrics framework.

6 REFERENCES

[Dunham83]
Dunham, J. R., Kruesi, E., "The Measurement Task Area", IEEE Computer, November 1983, pp. 47-54

[Keller85]
Keller, S. E., Perkins, J. A., "An Ada Measurement and Analysis Tool", Annual National Conference on Ada Technology 1985, pp.188-196

7 ABOUT THE AUTHORS

Mr. John A. Perkins is a member of the Software Research and Development Group at Dynamics Research Corporation. He has a Bachelor of Science degree in Mathematics from Purdue University and a Master of Science degree in Mathematics from the University of Illinois. Mr. Perkins has been involved in the deveolpment of translators for multi-processor scientific computers and in the development of an attribute grammar-based translator-writing system.

Mr. Damon M. Lease is a member of the Software Research and Development Group at Dynamics Research Corporation. He has a Bachelor of Science degree in Mathematics/Computer Science from Bucknell University. Mr. Lease has been involved in Ada programming and the development of a set of Ada programming guidelines for the Air Force.

Mr. Steven E. Keller manages the Software Research and Development Group at Dynamics Research Corporation. He has a Bachelor of Science degree in Biology from the University of Colorado, and Master of Science degrees in Computer Science and Biology from the University of Oregon. Mr. Keller has been involved in the development of compilers for multi-processor scientific computers and was the principal investigator for the development of an ordered-attribute-grammar-based translator-writing system.

THE TECHNOLOGY LIFE CYCLE AND ADA

MIGUEL A. CARRIO, JR.

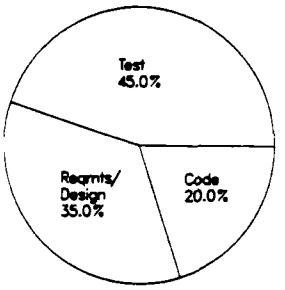
TELEDYNE BROWN ENGINEERING

ABSTRACT

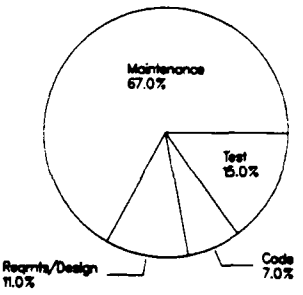
The rate of change of technology development continues to increase significantly while the system and software life cycle's rate of change that assimilates the technology continues to atrophy. Major technology changes are occurring approximately every 2 to 3 years, while system life cycles for complex and large embedded computer systems are in excess of 12 years, and in some cases as long as 18 years. Automated tools and paradigms, coupled with modern methodologies and technologies, and evolutionary development approaches necessitate a reexamination of classical or traditional life cycle models used to build systems. An examination of the activities that occur during the maintenance phase is required to properly identify and separate true maintenance from pseudo-maintenance activities. Furthermore, renewed and more extensive emphasis is required on the early life cycle requirements and design phases as a consequence of the emerging technologies.

BACKGROUND

The rapid infusion of software technology into embedded computer systems and the dependency on this technology to serve as the solution medium has itself created a higher level of complexity whose interrelationships and transformation algorithms are still not completely understood. This situation is further obfuscated by the lack of design discipline and the esoterism associated with computer languages and code understood by a relative few. What has resulted, despite the emergence of information identifying the contribution that coding activities represent as a function of the life cycle, (less than twenty-percent in most cases) Figure 1, is a misfocus of the problem. The misfocus occurs in part as a consequence of the late detection of software design errors (Figure 2), resulting in maintenance activities being the costliest phase in the life cycle. Noiseux's reference indicates that pseudo-maintenance can be as high as 83% of maintenance costs. From a consumption of resources and productivity view, one is led to believe that concentration of solution resources should be focused on the maintenance or support end of the life cycle to effect the most savings. Despite the expenditure of large quantities of resources in maintenance, the last twenty years has yielded few productivity gains, little discipline or significant documentation that was of use in design maintenance and fault correction insight.



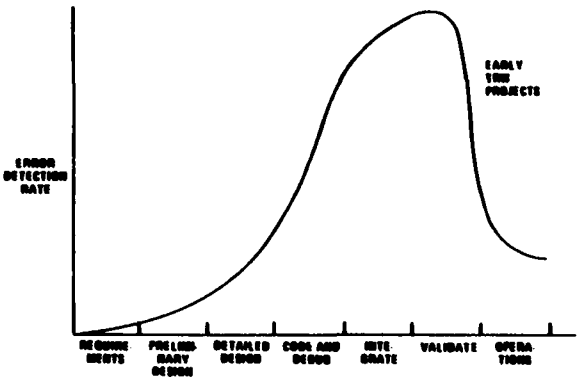
Software Development Life Cycle (Except Maintenance)



True effort required for many large-scale systems

FIGURE 1

CATCHING SOFTWARE ERRORS LATE: PROJECT RESULTS *



*PRESENTED AT 1976 AIAA SOFTWARE MANAGEMENT CONFERENCE

FIGURE 2

CURRENT PARADIGM, PROBLEMS AND MISCONCEPTIONS

The current life cycle is essentially comprised of three major phases - a conceptual and definition phase, a development phase, and a deployment and operational phase. In the conceptual and definition phase (i.e., the initial or early life cycle phase), requirements are identified, an intended performance envelope is stated, and statements of needs are written. During this initial and formative phase the prime individuals involved are the users and the keepers of doctrine. The development phase primarily consists of the design, code and test activities concerned with the system implementation. The development community, as the activities imply, are in turn supported by the systems, hardware and software engineers/designers; programmers and analysts; and test and quality assurance specialists, to name a few. A major difference between development and requirements type individuals is that the latter represents the user and functional expertise, while the former is more technical specialist driven. The deployment and operational phase also consists of the important maintenance and support activities required to complete the final or last phase of the life cycle.

For the most part, it is during the deployment and operational phase that users and developers meet and formally interact. The two initial major life cycle phases, requirements and development, are the fiercely guarded domains of the user and developer, or the customer and builder.

It is interesting to note that in the world of high technology and complex computer systems* that the relationships and interactions between the key individuals representing the different communities of interest are difficult to grasp and understand. However, a simple analogy is required to insure clarity since the problem of understanding life cycle phase relationships and responsibilities remains cloaked due in part to the alphabet soup and acronyms that specific communities use**. The problem is further exacerbated when the different Government, industry and academic communities are brought together to work on a project.

The analogy is one where an individual (homeowner) desires to build a home by engaging a builder (contractor). The prospective homeowner is isolated for the most part from the various developer-subcontractors (i.e., architect, plumber, electrician, mason, carpenter, etc.). Initial meetings between the two are to select type of house, appliances, color schemes, etc. and contract finalization. During construction, a few limited interactions, equivalent to development design reviews, are made to satisfy contract construction draw schedules of monies for completion work (e.g., basement, first floor, roof) to reimburse the builder. The corresponding quality assurance or acceptance personnel are provided by the customer or his bank in the form of inspectors providing certificates of specific work completion, and the ultimate acceptance certification, the certificate of occupancy.

The problems of communicating requirements and information is no less difficult or different between high technology users and specialists than the homeowner with his builder. Unless a homeowner can understand architectural blue prints, electrical and plumbing diagrams, he is at the same loss that a user or functional expert is, in the event communications are attempted with a software engineer or programmer (i.e., unless the user understands the specific coding language and data flow diagrams). In plain English, unless the user has previously built his own home, he is at a communications loss, caveat emptor. Similarly, a builder may encounter his own difficulties because he built a house that did not meet the full expectations of a buyer who wasn't sure of what he wanted or where to locate certain partitions, and selected tile colors and patterns he would have subsequently preferred changed.

Thus, one key and fundamental problem that has been identified across the life cycle, that continues today, is that of communications, of understanding and insuring the integrity and accuracy, of the initial set of requirements. The passing of requirements information becomes a significant issue that is commonly overlooked since users and developers assume that because they all speak the same language, English, that requirements accuracy and specification clarity are all guaranteed and that the language is sufficient to insure the intended product. *Would a builder consider building a house with* only an english written description, without architectural renditions and blueprints? The question that must be repeatedly asked is: Is the as specified system and intended performance, the same as the as designed, the same as the as documented, the same as the as tested and the same as the as built? Under the current life cycle and paradigms that are used to build systems the answer is no.

In addition to the problem of communicating requirements, a number of other issues and concerns centered on the existing life cycle exist. Figure 3A represents the life cycle currently used, with Figure 3B representing a simplified version of Figure 3A, that will be the one used as a reference throughout this paper, unless otherwise stated. Under the current paradigm used to develop systems, when design specifications are passed to programmers in the coding phase, it is at this point, and not prior, that the specification is converted into an implementation. Additionally, the programmer by default as a result of a lack of a viable communications link back to the user or systems engineer, is "licensed" to interpret requirements and

*Other equivalent terms are embedded computer systems, mission critical computer systems, weapons systems, battlefield functional systems, real time systems, target or applications.

**Department of Defense uses either Milestone 0, I, II, III or conceptual, design, development, deployment phases respectively. Some DoD services use system & performance specifications, functional and product specifications, while others call them A level specs, B level and C level specifications respectively, ad nauseum.

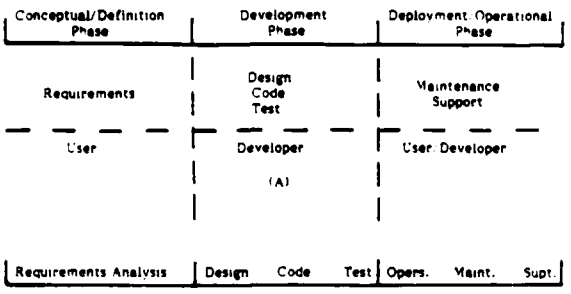


Figure 3

TRADITIONAL/CURRENT LIFE CYCLE

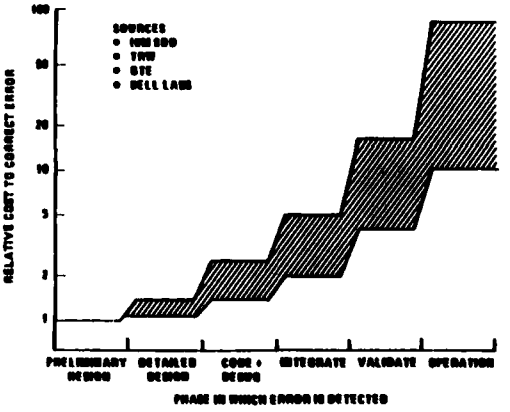
implement them based upon his understanding of the intent of the requirements using his own formal syntax and semantics (i.e., using the coding/programming language). This issue of interpretation is not really challenged until much later, in the test phase, when an attempt at reconciliation of the as implemented versus as documented design is made. At this point in the life cycle (testing phase) it is quite costly to correct mistakes since considerable amounts of resources, have been expended in time, people and costs. From design to the system integration phase, several years may have intervened, requiring hundreds of specialists, producing thousands of pages of documentation. Software errors are traditionally detected very late in the life cycle as Figure 2 illustrates. Similarly as Figure 4 illustrates the associated costs of detecting errors late in the life cycle are significant. Fewer errors can cost more to correct the later they are detected.

Once design is implemented via code and the coding specifications exist, from this point on in the life cycle, it is the code that is maintained, changed and supported in an attempt to insure conformity between the implemented system and intended requirements. Additionally once long lead procurement items and hardware (LLPI/H) are delivered to the contractor, as a result of commitments made early in the design phase, many requirement that cannot be accommodated by the LLPI/H's are "passed over" to the software side of the house for accommodating implementation. The latter's disruptive effect on schedules and costs is futher masked by the valid hardware/software tradeoff task found in major systems developments, (i.e., true apriori tradeoffs vs. "passed over" requirements). Thus, the present paradigm supports the maintenance of code and not the maintenance of design specifications. This leads to the situation that under the present paradigm the more maintenance that is performed on an existing system, the more structure is destroyed and the less insight into intended performance and requirements results. Instead of graceful system degradation or evolutionary transitioning into an enhanced performance envelope, systems are discarded, and replaced by completely new systems with little commonality and reusability between them.

Many systems' lifetimes are reduced by the incomprehensible and voluminous specification documentation produced that stifles the ability to gain design insight, traceability and correlation of functionality with the same. Insight into development and design methodology, and derived requirements are also inhibited. The few viable prototypes that are created during initial design by systems engineers to provide insight, are discarded and never used to assist in the evolutionary and iterative processes of design and requirements synthesis. Further insight into current versus automated paradigms can be derived from references on Balzer and Sievert.

Current thinking and paradigms of the last 25 years centered on life cycle phases, relationships and technology that were stable and simplistic. Systems were manageable because they were small in size (e.g., program sizes of several thousand lines of codes); design and programming teams consisted of fewer individuals, computers had not entered the world of multiprocessing, microprocessing, virtual memories, concurrent tasks, relational data bases, and technology was slow to change. Today, systems comprised of a half-million to a million lines of codes, designed by teams of individuals numbering 50-100 are common.

CATCHING SOFTWARE ERRORS LATE: THE COST



PRESENTED BY MR. JACQUES GANSLER, DEPUTY ASSISTANT SECRETARY OF DEFENSE FOR MATERIEL ACQUISITION, AT THE 1977 AIAA SOFTWARE MANAGEMENT CONFERENCE.

MALIGNED MAINTENANCE

The ratio of software to hardware dependency and functionality has shifted significantly in the last 30 years as illustrated in Figure 5. Present embedded computer systems are primarily dependent on and impacted most by software. Since software is labor intensive and complex, the larger the system, the greater the resources required to support and maintain the software activities. When problems occur in an embedded computer system, the last to be detected, most costly and severe are the software problems. Why are the software problems so costly to resolve and detect? A fundamental answer is that we collectively (industry and government) are not as knowledgeable as we think we are. The latter is not as negative as it sounds. With the technology and corporate history about 25 years old, and with individual system acquisition times of 10-15 years, all this means is that we are still learning about the new technology, and have not had sufficient data points to make many intelligent decisions. However, the documented mistakes of the past[16] together with the emergence of new tools, different life cycle models and software paradigms, coupled with innovative and formal methodologies reveal that there are better and cost effective ways to build systems in a disciplined and predictable manner.

Furthermore, examining maintenance activities in greater details reveals that most of the activities occurring are not true corrective maintenance activities (i.e., fixing latent defects, software bugs and errors arising from code). The majority of activities occurring in this phase can be classified as *pseudo-maintenance* activities. References by Boehm and Glass shed further light on this subject. The activities found in the maintenance phase consist of the following:

a. Making enhancements to the system performance envelope, this is commonly referred to as adding bells and whistles or gold plating. The core requirements are not impacted but features are added (e.g., modifying a screen format or adding color graphics to it). Increased scope.

b. Changing the baseline performance by adding new or substituting other requirements relative to those implemented. This primarily results when the user is not sure of his initial requirements set. Increased scope.

c. Changing the baseline performance envelope to improve response times in excess of requirements or in anticipation of a future change in doctrine-optimization. Increased scope.

d. Increasing the baseline requirements in order to transition or evolve to an intended or objective baseline. Intentional and planned for increase in scope.

e. Corrective maintenance, in the true sense, to sustain an existing system and keep it in an operational state by repairing defects arising from its use within its original performance envelope. No change in scope.

Conditions (a) through (e) are reflected in the following five expressions (1) through (5) respectively:

Where performance (P) is represented by:

T_i - original timing constraints

t_i - new timing constraints

F_i - original function set

f_i - new function set

R_i - original requirements envelope

R_s - substituted requirements envelope

r_i - new or additional requirements set

Subscript i represents the various iterations or versions that result in satisfying and sustaining a level of performance. This is part of the localized tuning process.

(1) Enhancement
$P_i \Rightarrow [T_i], [F_i], [R_i]$
$\searrow [f_i]$

(2) New or Substituted
$P_i \Rightarrow [T_i], [F_i], [R_i]$
$\searrow [R_s, r_i]$

(3) Optimization
$P_i \Rightarrow [T_i], [F_i], [R_i]$
$\searrow [T_n]$

(4) Transition or Evolutionary
$P_{i_{Baseline}} \Rightarrow [T_i], [F_i], [R_i]$
$\searrow$
$P_{i_{objective}} \Rightarrow [T_i+t_i], [F_i+f_i], [R_i+r_i]$

(5) Corrective Maintenance
$P_i \Rightarrow [T_i], [F_i], [R_i]$

Activities (a) through (d) and 1-4, represent pseudo-maintenance activities, while (e), (5) represent true or corrective maintenance in the classical sense. However, the pie charts or bar graph representations found in texts referencing major life cycle activities showing relative weights of phases, for the most part just identify maintenance as a whole thereby generating a composition misconception.

Most of the activities occurring in the maintenance phase, over 80%[9], are of the psuedo-maintenance type. This results as a consequence of the following:

a. The problems generated by using the current paradigm where as intended is different from as implemented design. Effort is thus expended to correct this.

b. Very late detection of complex problems and unclear design, obfuscating traceability and methodology.

c. Lack of a usable communications and graphical language understood by users and developers - the communications problem.

d. Lack of early working prototypes that can be maintained and evolved into end-implementable systems, instead of throw-away prototypes.

e. Documentation that precludes corporate design histories and does not provide insight into design and development methodologies.

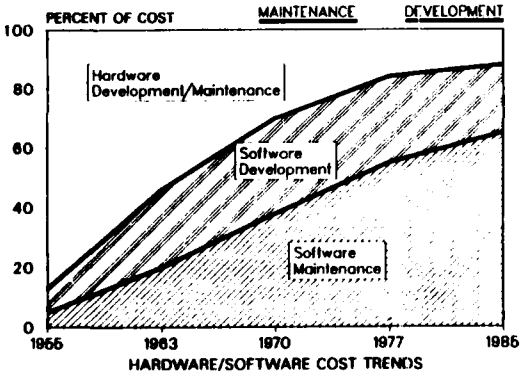
f. Lack of automated tools to assist in harnessing the new technology that enables the compatibility of innovative concepts. These tools should also enable deriving productivity synergism from the merging of innovative concepts and technology.

g. Lack of an awareness of the changes being wrought by the emerging technology, an awareness to change and adoption to it.

NEW APPROACH TO LIFE CYCLE

By identifying new components to the life cycle and addressing these areas via new tools, paradigms and approaches, the pseudo-maintenance issues identified in (a) through (g) can be resolved today. However, it is felt that one of the biggest impediments to getting a grasp on managing the development of large complex systems and software productivity is item (g), an awareness of the impact that emerging technology can have and a willingness to change, to adopt, to experiment using the new tools and paradigms.

The technology life cycle of Figure 6 shows a modified life cycle, that can be used to support automated paradigms and more accurately reflect technology changes that impact how systems are built.



HARDWARE/SOFTWARE COST TRENDS

FIGURE 5

If advantage is to be taken of working prototypes based on design that can be maintained and evolved into end-item systems, then a greater emphasis must be placed on the early or initial life cycle phases. Specifically, a synthesis phase must be identified where the following areas are addressed:

Prototyping - Using an automated paradigm[1], early design driven prototypes are established and maintained for the duration of the life cycle. This enables a strong traceable link between requirements, design, implementation and maintenance. A corporate history and process insight is available.

Reusability - Designing for reusability must be addressed at this point since costs and resources required are greater than the single context or system used in a single application (point system). Tools that support reusability must be invoked at this early phase to be effective and to insure productivity.

Artificial Intelligence/Knowledge Bases - The emerging technology will enable the development of intelligent tools supported by a knowledge base that will provide the ability to identify minimum functionality for generic systems and assist in the allocation of functions, software and design reusability. AI/KB tools supported by automatic code generators will enable concise and consistant transformations of requirements to design to implementation in significantly less time.

Generic Instantiations - Once a system or class of systems is developed using these innovative concepts, tools and paradigms, subsequent evolutionary derivative systems and subsystems can be efficiently and rapidly instantiated enabling a higher degree of verification, validation and a shorter deployment time This activity may consume what is commonly referred to as rapid-prototyping using very high order or Fourth Generation Languages. Functional instantiation and completeness would be enabled earlier in the life cycle process instead of the meaningless functional and physical configuration audits that are required at the end of development.

Data Base Applications/Support Environment - With systems becoming larger and more complex, requiring extensive support in their post-deployment period, data base management of voluminous information becomes a critical function. Separation of the data from its applications enables a concentration on synthesis. The migration of data between the corporate, host & target development, and applications data bases can then be focused on relative to such functions as access, distribution, fusion and transportability.

Technology Insertion - Under an automated paradigm, using modern methodologies and an evolutionary development scenario using automated tools, the ability to assimilate technology rapidly is made possible. This insures that developed systems can take full advantage of what is available on the market in a cost-effective manner.

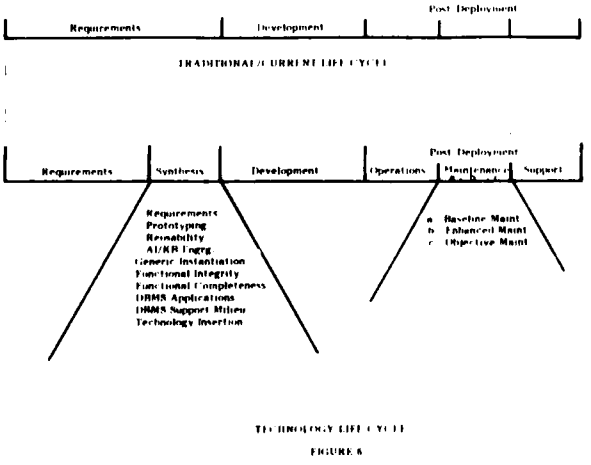
Automated Tools - Systems have become so large, complex, and labor intensive that adoption of automated paradigms and modified life cycles while essential is insufficient. Substantial increases in productivity can only occur when innovative concepts, paradigms and methodologies are fully supported by automated tools that in turn incorporate formalisms and graphics. Formalisms in the sense that a methodology supported by a rigid syntax and semantics is embodied within the specific design tool environment and is machine processable and executable. Furthermore, the user interface to such a design environment shall be extremely friendly and graphical so as not to burden an already technology-overburdened architect.

In conjunction with the synthesis phase of the life cycle, the maintenance phase would be divided into three maintenance parts: baseline, enhancement and objective maintenance phases. The baseline maintenance would correspond to the traditional or corrective maintenance performed on the original or baseline system, while the other two consist of all of the pseudo-maintenance activities. Enhanced maintenance would comprise all of the pseudo-maintenance activities except for the unique activities associated with transitioning to an objective system.

TAGS[R] Technology represents such an example of the automated paradigm coupled with a formal methodology, graphics, and documentation aides, automated on an engineering design workstation. The point being that such automated tools with automatic code generation back-ends are appearing and available on the marketplace.

LIFE CYCLE EQUIVALENT MODELS

An equivalent representation of the Figure 3B and 6 life cycles is shown in Figure 7.



TECHNOLOGY LIFE CYCLE

FIGURE 8

Figure 7A illustrates that the traditional life cycle is a unidirectional flow model that cannot support evolutionary development, and only supports the maintenance iterations associated with the code and coding products. Figure 7B is the technology life cycle representation allowing bidirectional flow of requirements information, reusability, iterative maintenance of design and evolutionary development.

The technology life cycle thus enables the assimilation of Ada[R] with its different forms (e.g., Ada Program Design Language). The concepts espoused by Ada Technology such as data abstractions, packages, tasking, generics, and types can easily be accommodated and are preferred under the new paradigms. Module functionality and interfaces can be effectively addressed and referenced in design considerations without concern for implementation details of such. As these concepts are elevated to higher levels of abstractions and addressed throughout the life cycle, the seeds of common denominators and better communications are ultimately planted leading to a greater understanding between users and developers. This is essential for the passing of requirements in a uniform and consistant manner.

NEED FOR CHANGE

The traditional life cycle has been slow to change. It is essential that the life cycle be viewed from a different perspective, and that it too be subject to modification to accommodate the technology changes imminent over time. The rapid technology changes of the the last 25 years have focused attention on the specific activities occurring within it. As a result, much attention has been focused on the coding phase (i.e., Ada Language initiatives), on deployment (i.e., Post Deployment Software Support Concept), on software development (i.e., Software Technology for Adaptable, Reliable Systems (STARS) Program), on environments (i.e., Software Engineering Environment (STARS-SEE)), but until recently very little on the broader issues of the life cycle.

If significant productivity gains are to be made in developing and fielding systems then a closer look at those areas mentioned must be made. An understanding of the relationships and transformations between phases is key to transitioning between them. Significant resources, and much dialogue is expended on integrated battlefield management concepts and open system interfaces, but very little on integrating methodologies with automated paradigms and evolutionary or reusability concepts where significant benefits are to be accrued. Congress debates over systems and their numbers to be acquired for the military, but very little if any debate is ever heard over how to build these systems in timely cost effective ways, or for that matter, identifying long term development strategies in light of short term fiscal policy and rapidly changing technology.

The entire design review process must also be examined. Preliminary and critical design reviews should be adhered to as intended, to signal major design stabilization points, and not to satisfy a contractual schedule or appease management. In light of the synthesis and pseudo-maintenance phases discussed, new milestone reviews should be identified (e.g., upon completion of a working prototype or simulation run). A reusability and data base management review should also be established. Depending on the type of development, design reviews should be flexible and custom tailored for the specific system, yet rigid and formal enough to provide the insight into system progress and maturation. Automated program management tools supported by knowledge bases can be most effective in providing a program manager insight into the system development and its specific life cycle.

A properly tuned and understood life cycle can support system development so as to conserve resources that can be used elsewhere to address the larger multi-context issues of reusability and evolutionary development where significant payoffs are to be realized. In the long term, adoption of new paradigms and technology life cycles would result in the disappearance or transformation of pseudo-maintenance and the requirements phase into a new combined process capable of synthesizing systems in times less than that associated with the technology.

The commitment of resources to and emphasis on the initial phases of the life cycle should be viewed as viable investments. Risks are optimally minimized, productivity yields maximized, and software development with its schedules and cost can be disciplined and most importantly - predicted.

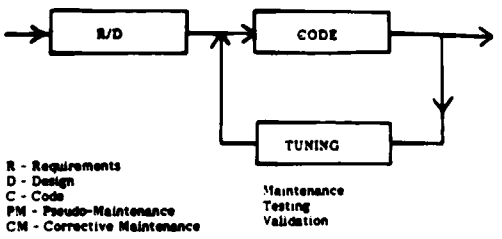


FIGURE 7(A)

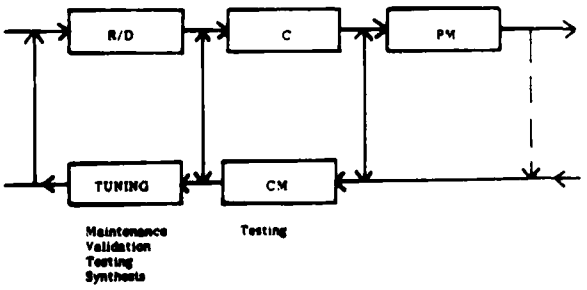


FIGURE 7(B)

REFERENCES

1/ Balzer, Cheatham, Green - Computer, Vol. 16, No. 11; Nov. 83, pp. 39-45. Subject: Software Technology in the 1990's: Using a New Paradigm.

2/ Balzer, Robert - Proceedings of COMPSAC 84 Conference on Computer Software & Applications; Nov. 84, Chicago, Ill.; IEEE# 0730-3157, pp. 471-475. Subject: Evolution as a New Basis for Reusability.

3/ Bersoff, Henderson, Siegel - Software Configuration Management; Prentice-Hall, Inc., 1980.

4/ Boehm, B. - Software Engineering Economics; 1981, Prentice-Hall, Inc.

5/ Booch, G. - Software Engineering with Ada; Benjamin/Cummings Publishing Co., Inc., 1983.

6/ Clapp, J. - Proceedings of COMPSAC 84 Conference on Computer Software & Applications; Nov. 84, Chicago, Ill., Library of Congress No. 83-640060, pp. 479-480. Subject: Software Reusability: A Management View.

7/ CODSIA, DoD Management of Mission-Critical Computer Resources, Mar. 1984; Council of Defense & Space Industry Association's (CODSIA) to Under Secretary of Defense Research and Engineering, Volume 1, Task Group Report.

8/ Gilb, T. - Software Engineering Notes, ACM; Jul. 1985, Vol. 10, No. 3, pp. 49-62. Subject: Evolutionary Delivery Versus the Waterfall Model.

9/ Glass, R.G. & Noiseux, R.A. - Software Maintenance Guidebook; 1981, Prentice-Hall, Inc.

10/ Goguen, J.A. - Computer; Vol. 19, No. 2, Feb. 86, pp. 16-28. Subject: Reusing and Interconnecting Software Components.

11/ Jones, G. - Proceedings of COMPSAC 84 Conference on Computer Software & Applications; Nov. 84, Chicago, Ill., Library of Congress No. 83-640060, pp. 476-478. Subject: Software Reusability: Approaches and Issues.

12/ Roman, G.C. - IEEE Computer Magazine; Vol. 18, No. 4, Apr. 85, pp. 14-23. Subject: A Taxonomy of Current Issues in Requirements Engineering.

13/ Sievert, M. - Computer; Vol. 18, No. 4, Apr. 85, pp. 56-65. Subject: Specification Based Software Engineering.

14/ Spinrod, M. & Abraham, L. - Software Engineering Notes, ACM, Vol. 10, No. 3, Jul. 85, pp. 47-49. Subject: The Wild-West Life Cycle.

15/ STARS Applications Workshop: Software Applications & Reusability Proceedings; Monterrey, CA., Nov. 1985.

16/ U.S. Army In-Theater Post Deployment Software Support Study, Phase 1 Report, May 1983, Contract DABT60-82-C-0047, Teledyne Brown Engineering.

17/ U.S. Department of Defense - Reference Manual for the Ada Programming Language; MIL-Standard 1815A, Feb. 1983.

18/ Zelkowitz, Shaw, Gannon - Principles of Software Engineering and Design; Prentice-Hall, Inc.

Mr. Miguel A. Carrio, Jr. is presently Manager of Advanced Technology Programs at Teledyne Brown Engineering's Washington, D.C. facility. Mr. Carrio's past responsibilities at TBE have been as program manager of both the Defense Communications Agency WWMCCS Advanced Area Research & Development Programs' Independent Verification & Validation/Testing & Prototype (IVV/T&P) efforts; and the DoD Joint Tactical Fusion Program IVV/T&P. Prior to joining TBE he was a branch chief at the Army's Communications Electronics Command-Software Technology Center with responsibility for such programs as the highly successful and first validated Ada Compiler (NYU Ada/Ed), Ada Design Methodology Efforts and Ada Education training programs.

Mr. Carrio's experience in systems and software engineering spans 23 years. Mr. Carrio has a BS Physics, LIU, and an MS Engineering from Fairleigh Dickinson University.

SESSION IV: ADA IMPACT ON SECURE OPERATING SYSTEMS

CHAIRPERSON

M. Margaret Zuk, Group Leader

MITRE Corporation

Use of the Ada language for secure systems introduces many complex issues. The elaborate Ada Runtime Support Library, the unpredictable behavior of Ada programs possible with different compiler implementations, and the complexity of the language itself are concerns that secure system designers and verifiers face.

This panel will focus on the impact that Ada has on the design and implementation of secure systems. An overview of secure system design and the techniques that are used to verify secure systems will be presented. System designers will then discuss current work in this area, and share their experiences with the use of Ada.

Margie Zuk is a group leader in the Trusted Computer Systems department at MITRE, Bedford. Her group provides security system engineering support to the Army and to the DoD Computer Security Center's Office of Application Systems. She has been involved with Ada since 1981, and serves as chairman of the Ada Verification Workshop's Secure Systems Working Group.

Margie holds a Masters degree in Computer Science from Stevens Institute of Technology and a BA in Math from the college of Mt. St. Vincent. Prior to joining MITRE, Margie worked for Bell Laboratories and Ford Aerospace.

SESSION IV: ADA IMPACT ON SECURE OPERATING SYSTEMS

PANELIST

Dr. Richard A. Platek, President

Odyssey Research Associates, Inc.

The state of the art in formal Ada specification/verification is reviewed including current European work. Several problems are surfaced and solutions proposed. In particular, methods for dealing with Ada's so-called non-predictability and complexity are presented. The first is handled through the use of non-deterministic post-conditions for Ada program units. The second is handled using the method of Clusters which the author and his colleagues are developing.

Richard Platek is President of Odyssey Research Associates (ORA) of Ithaca, NY and a member of the Department of Mathematics, Cornell University. He has a B.S from M.I.T. and a Ph.D. from Stanford University, both in Mathematics. ORA is concerned with the applications of formal methods to the development of trusted software with special emphasis on secure systems. Their activities span the spectrum from security engineering on systems which are currently being fielded to the development of advanced, knowledge-based software engineering tools for the inferential development of provenly correct systems. Prof. Platek is Chairperson of the SIGAda Formal Methods Committee.

SESSION IV: ADA IMPACT ON SECURE OPERATING SYSTEMS

PANELIST

Eric R. Anderson, Army Secure Operating System Project Manager

TRW DSG

This talk addresses some of the criticisms that have been leveled against Ada's suitability for use in trusted computer systems. It treats the use of Ada for both untrusted applications programs and security kernel implementation. (It does not directly address the use of Ada for other trusted software, but much of what is said about its use for security kernels applies to trusted software as well.) The paper concludes that each Ada criticism is either unfounded or poses a problem that has a solution; thus, Ada is indeed suitable for trusted computer system implementation.

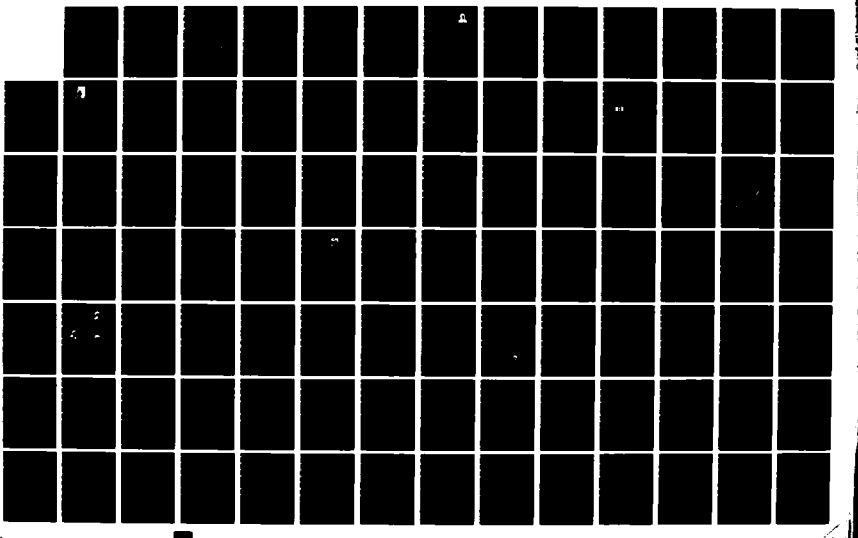
Mr. Anderson has 17 years of experience at TRW, both as a software manager and developer. His areas of expertise include project management, real time operating systems, programming languages, and computer security.

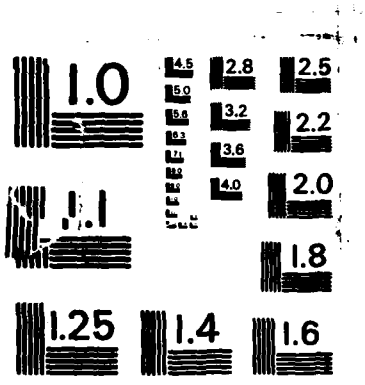
Mr. Anderson is the project manager of the Army Secure Operating System (ASOS) project. In the Concept Definition Phase, he was the chief designer of both the Dedicated Secure Operating System and the Multilevel Secure Operating System, and designed the Task Management portions of both operating systems. He previously managed the "Security Kernel for Secure Operating Systems" IR&D project. He was a subproject manager on the TDP project operating system and a work package manager on the MIFASS project real-time operating system. He was the project manager of the Kernelized Relational Information and Storage System (KRISS), and a work package manager of the Kernelized Secure Operating System (KSOS).

Education: A.B., Computer Science, University of California, Berkeley, 1969. M.S., Computer Science, University of Southern California, 1972.

AD-A167 802 PROCEEDINGS OF THE ANNUAL NATIONAL CONFERENCE ON ADA (TRADE NAME) TECHNOL (U) ARMY COMMUNICATIONS-ELECTRONICS COMMAND FORT MONMOUTH NJ 2/2

UNCLASSIFIED 1986 F/G 9/2 NL





SESSION IV: ADA IMPACT ON SECURE OPERATING SYSTEMS

PANELIST

W.E. Boebert, Chief Scientist

Honeywell Secure Computing Technology Center

There are three sets of issues which must be fused in the use of Ada in secure systems. The first set deals with the use of Ada as a programming language for secure applications whose operations are controlled by a Trusted Computing Base (TCB). The second set deals with the use of Ada in the implementation of "trusted processes", and the third deals with the use of Ada to implement the reference monitor subset of the TCB. The positions taken by the Secure Ada Target project on these three sets of issues will be described.

Earl Boebert is currently the Chief Scientist at Honeywell's Secure Computing Technology Center, where his prime responsibility is acting as Principle Investigator on the Secure Ada Target project. Prior to this, he held a variety of technical and managerial positions at Honeywell. Before joining Honeywell in 1966, he was an EDP officer in the United States Air Force, and before that, a programmer and machine operator at the Stanford Computation Center.

SESSION IV: ADA IMPACT ON SECURE OPERATING SYSTEMS

PANELIST

Steven R. Hart - Staff Engineer

M/A-COM Linkabit

SYNOPSIS: An overview of some of the problems associated with the use of Ada to develop trusted computing systems is presented. A methodology and generic software architecture is given which allows full Ada to be used as the development language for applications programs in a secure environment, based on standard reference monitor concepts. The methodology allows for bi-directional portability; that is, programs running under the secure environment can be easily ported to any other machine, and externally generated programs may be brought into the secure environment.

Steve Hart is a Staff Engineer at M/A-COM Linkabit where he works on problems in computer security, satellite networking, and software engineering. His research interests include development of combinatorial solutions to multichannel scheduling and formal techniques for verification of specifications and protocols. He received a B.S. in Mathematics in 1977 from the University of Nevada, Las Vegas, and a M.A. in Mathematics in 1980 from the University of California, San Diego.

DEVELOPMENT OF A CORPORATE ADA TRAINING CURRICULUM

Linda F. Blackmon

General Dynamics
Ft. Worth, Texas

Abstract

This paper discusses the process followed to define a large corporation's need for Ada training and the establishment of a corporate mandate to develop and deliver an effective Ada training program to all divisions. The management process involved in obtaining cooperation from all levels of staff in the development of a corporate-wide project is emphasized, along with a description of the courseware design and development methodology used to develop the curriculum and the challenges in the design of a technology curriculum for an industrial setting as opposed to an academic environment.

Management Strategy

Perspective

There are two extremes in the business world: the small, individually owned and operated company which focuses on one or two products and/or services, and the large, multi-national, highly diversified corporation. Such a corporation serves as an umbrella organization to a host of independent divisions and companies, any one of which might be classified as a sizable company if it operated on its own, and none of which are necessarily functionally related to the others. Each of these extremes has its own unique decision-making process and method for facilitating change within the organization.

In the small company, the owner-manager is ultimately responsible for the final decision; employees may or may not be invited to participate in the process. Once the decision has been made, change is usually mandated and any employee who is unable or unwilling to cooperate will be asked to find other employment. Large corporations, on the other hand, are not able to effectively institute change by fiat from the highest levels; it is necessary to involve the mid and upper level management of each of the component divisions/companies in the decision-making process and in the delivery of the new methodology or product once the decision has been made. It is not, of course, necessary for the division-level management itself to participate in the process directly; they will most often delegate that responsibility to senior level technical or lower-level management personnel.

General Dynamics falls into the category of a large, diversified corporation. Composed of thirteen divisions and companies, with goods and services ranging from the production of military products (tanks, fighter planes, and nuclear submarines) to the mining of coal and lime, this corporation practices matrix management to effectively produce goods and efficiently deliver services to a broad range of customers. The majority of products and services produced are targeted to military use, and the DoD has mandated that Ada will be the language used in the development of all embedded and mission-critical systems. It was clear that a commitment to a well-defined, coherent plan to train software engineers in Ada was essential.

In such a corporation the top-most level of management at the corporate headquarters must use their broad-range, "big picture" point of view to identify possible corporate-wide needs. Identification must be followed with a process which involves the division-level people in the refinement of the needs statement, the promulgation of appropriate solutions, and the facilitation of those solutions at the appropriate levels in the divisions. Top management is, in effect, more dependent on relationships than on authority to assure effective execution of their requests; authority impacts more and more individuals as the management hierarchy is descended. The larger the corporation the more possible it is for lower level employees to subvert or sabotage the wishes of upper-level management if they feel that the decisions made are not in their best interests or that they were not represented in

the decision-making process. The answer to this problem lies in the utilization of the team approach, making it possible for representatives from each of the involved groups to have input into the decision-making process and the facilitation of the changes.

A Working Model

The production of a well designed, effective curriculum requires:

1. Expertise in Instructional Design Methodology

2. In-depth understanding of the target population needs and environment

3. A functional command of the subject matter to be covered

The resources for the first two requirements were available in the divisions; subject matter expertise was found by hiring an outside consultant. The design team is composed of the Corporate Coordinator, who serves as a facilitator-moderator-manager for the project; representatives from the Computer Related Training Department and the major production locales; and the Subject Matter Expert (SME). Each member of the design team is in close touch with product production to be sure that the design of the curriculum remains focused on the needs of the corporation.

The initial objectives of the design team were to:

1. Refine the overall curriculum requirements and produce a detailed specification of need;

2. Define the focus for the corporate training effort (not all needs could or would be met by the corporate effort; some would have to be addressed locally);

3. Set priorities for both immediate and long-term development efforts.

Once these initial objectives were accomplished, the team moved to the second set of goals, that of establishing a set of standards for the development of the courses (including formatting standards for such products as the Student Guides, Instructor Guides, Lab Guides, Presentation Visuals, and other support materials), and for configuration management of the completed curriculum.

The final (and on-going) mission of the design team is to participate in the development of and approval of the detailed design for each course; to participate in and monitor the first and second pilot offerings of each course; and to recommend and approve any changes generated from those pilots.

The responsibilities of the design team also include the constant monitoring of the training needs of the corporation and the need to be alert to changes which require modification of the overall curriculum and/or reshuffling of priorities for course development.

Design, Development, Delivery, and Maintenance of the Curriculum

Focus of the Design

When designing for an industrial rather than an academic setting, considerations include:

1. Time: courses must be fast-paced and intensive; a two-semester course at a university may be compressed into two weeks in industry;

2. Audience: mixed audiences are common, with experience ranging from none (the new graduate) to the experienced programmer/analyst;

3. Reinforcing exercises and examples: courses should include as much hands-on laboratory practice as possible, with the labs targeted to actual production needs. One of the objectives of the design team is to build into the materials a set of usable models for the software engineers to apply on the job when the training is completed. The examples and exercises model the concepts underlying the applications but are presented as "fantasy games" or "case studies" in order to focus learning on the conceptual level rather than on a current specific application.

Development Tools

Two main tools are used in the production of the course materials:

1. Information Mapping Technology (1): Information mapping technology structures material in a top-down fashion; each topic is decomposed into concepts, and each concept is presented in terms of a set of information blocks. Materials structured in this way are easy to read and to follow during class and serve as an excellent reference for the students when they return to their workplace.

2. A PC-based authoring system which allows the presentation visuals to be produced using color and animated graphics. This method of production of presentation visuals for the classroom has many benefits: most authoring systems are easy to learn and can reduce the time spent on development of these types of teaching aids; when dealing with computer related subject matter, it is easy to simulate screens; and, finally, learning is facilitated when color and animation are used to emphasize teaching points.

Instructional Design Methodology and "Courseware Engineering"

The primary basis for the design and development of the courses is the standard Instructional System Design model: Analysis, Specification of Objectives, Design of the Course, Development of Materials, First Pilot, Revise, Second Pilot, Final Revision, and Turnover. Within this model, this project has designated the student guide as the primary document from which all other materials flow. Within this process, which closely parallels the traditional software development cycle, software engineering principles such as abstraction, modularity, information hiding, and configuration management may be applied. The development of teaching materials using such principles may be termed "Courseware Engineering".

To communicate the flavor of Courseware Engineering, the following is a broad outline of the rules used to generate the basic design of the course, how that design is translated into the Student Guide, and a brief description of information which whould be included in the configuration management system of a course, as expressed in a Course Standards document.

1. The Design Process

A detailed outline of the course must be developed at the beginning of the design phase. The outline should break the material into the following levels:

SECTIONS: The broadest subdivisions of the course, sections correspond to chapters of a book.

TOPICS: General divisions of the sections, topics usually correspond to the objectives for the course.

CONCEPTS: The smallest "chunk" of information presented to the student, concepts are the building blocks of topics.

The process of outlining the course represents the first cut at "chunking" the learning materials. A further refinement now occurs, as the CONCEPTS are analyzed and divided into even smaller units, known as "blocks".

Logically group and order concepts to generate lecture segments that do not exceed thirty to forty-five minutes.

Once the student guide outline has been completed, decisions must be made on the details of the classroom presentation which includes the placement of exercises, distribution of handouts and other Job Aids, and the placement and types of presentation visuals. The outline is now STORYBOARDED to include these materials.

EXERCISES:

Each logical concept group is followed by a "cookbook" exercise or a set of reinforcing questions.

Each topic is followed by a "reinforcing exercise" which is a problem requiring the student to apply the concepts of the topic.

Exercises are intended to be executed in-line during the lecture, but can be done as a group if the classroom does not have individual terminals and a traditional lecture-lab format must be used.

2. The Development Phase

Student Guide

The student guide flows from the outline. The student is presented an overview of each SECTION and TOPIC while the concepts are detailed, using information mapping technology.

Each concept is now defined, described and illustrated using graphics and/or examples and non-examples.

3. Course Standards Document

a. Course Description

   1) Goals and objectives, stated in behavioral terms
   2) Delivery Modalities

b. Target Audience

   1) Intended audience
   2) Prerequisites for the course

c. Environment specifications describing the hardware and software tools required for delivery and maintenance of the course.

d. Configuration management specifications detailing the edit controls and file management guidelines for the course.

Delivery and Course Maintenance Strategy

Course maintenance is controlled through two primary mechanisms:

a. Standardization of production methodology, i.e., a single PC-based authoring system and word processing package will be used for all of the courses.

b. Establishment of "course managers", each of whom has responsibility for the ongoing maintenance of one or more courses in the curriculum. Once a course has been released for general delivery, it is the responsibility of the course manager to keep track of recommended revisions and to provide current materials to the Ada instructors. When a sufficient number of non-critical revisions have accumulated, or on the occasion of a critical revision (a technical error is discovered, or the language itself changes), the course manager is responsible for calling in the design team to assist with the necessary modifications, as needed.

Closing Summary

Corporate philosophical and financial committment to a large scale training effort in a new technology, such as Ada, assures that at least a minimal level of expertise will be available to all groups in the company. The creation of the product must involve a bottom-up approach to generate a sense of ownership on the part of the users. Continued involvement of the actual practitioners of the technology in the ongoing training efforts enriches all programs within the company and encourages development of new applications and tools.

An effective, efficient curriculum can be developed and delivered for a large corporation by:

Professionally managing the curriculum development

Obtaining the best technical expertise available in the field

Involving the user community in all phases of curriculum development and delivery.

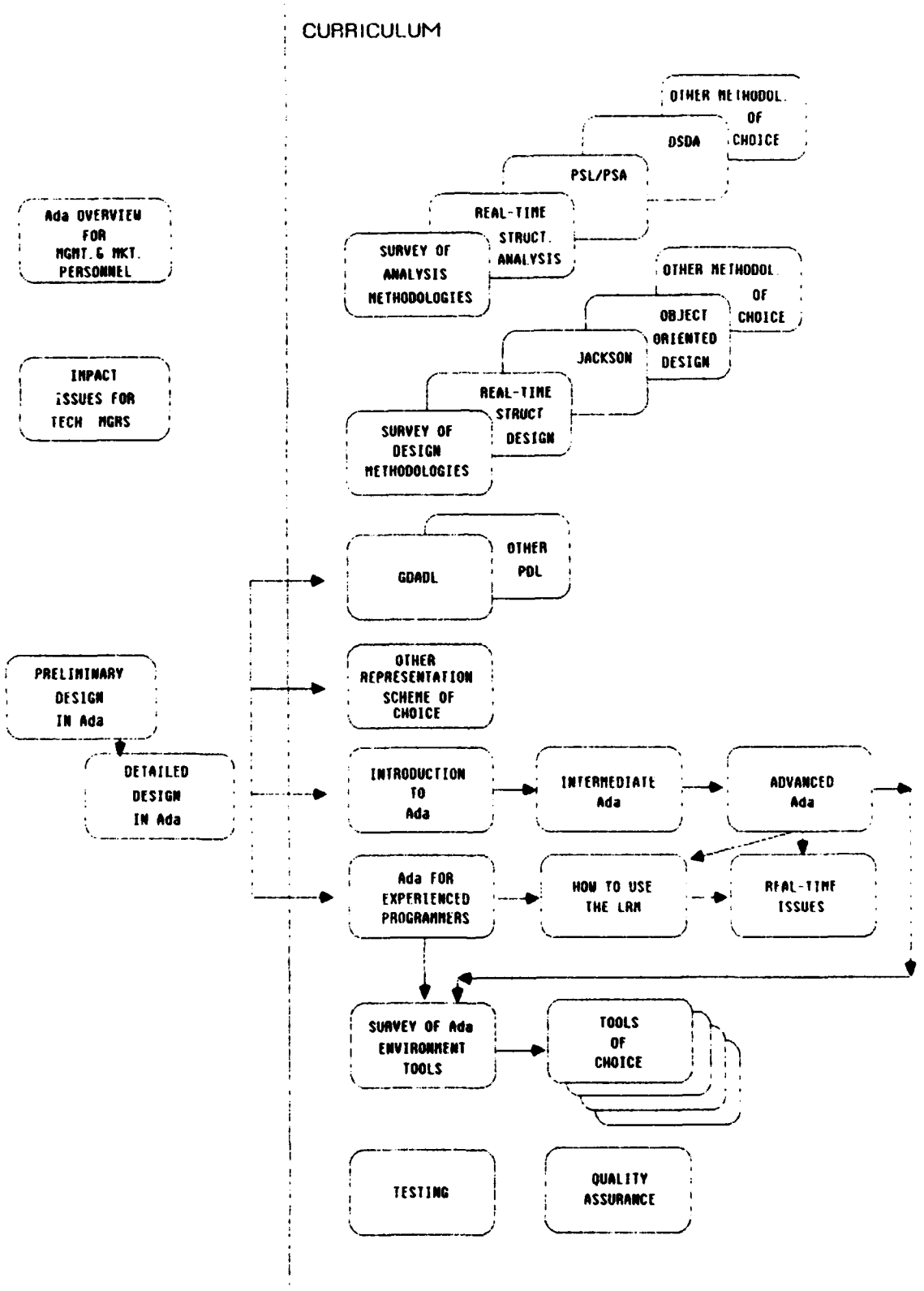
Bibliography

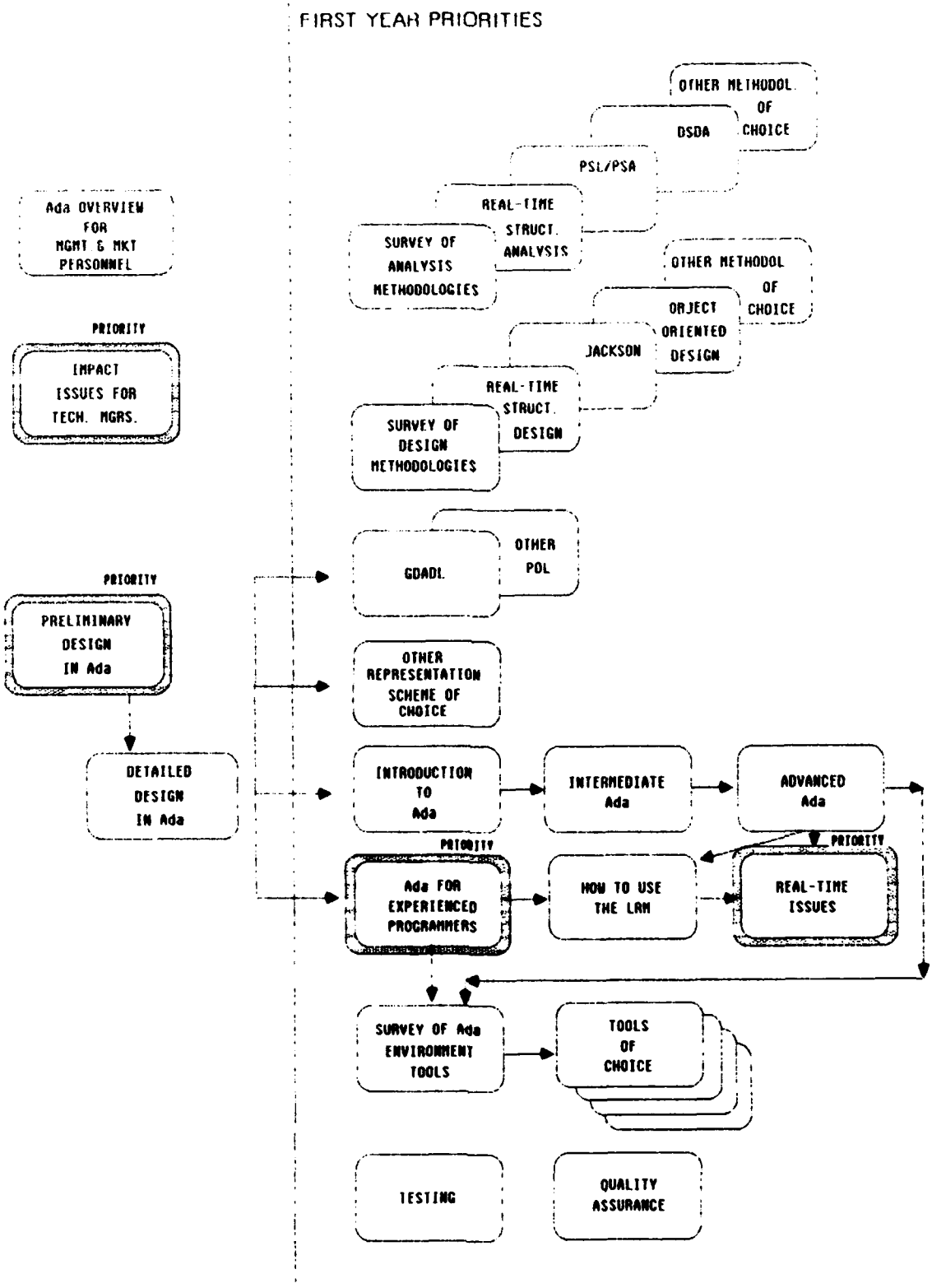
(1) Horn, R.E. "How to Write Information Mapping", Information Resources, Inc., Lexington, Mass.

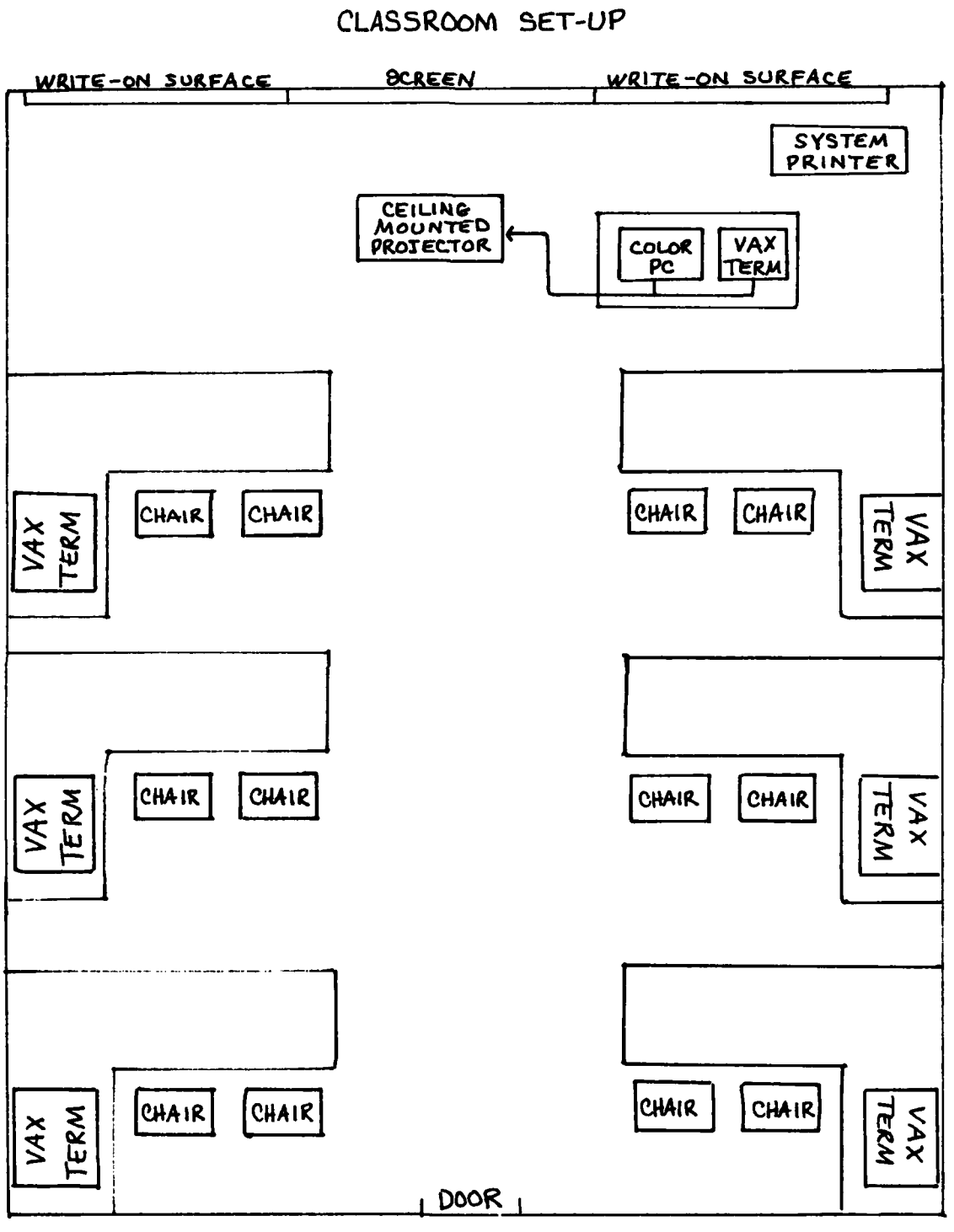
Olympia, P.L. "Information Mapped SAS: Teaching SAS for Retention", University of the District of Columbia.

Olympia, P.L. "Information-Mapped Chemistry", Journal of Chemical Education, Vol.56, p. 176, 1979.

Ms. Blackmon is the Coordinator for the Corporate Ada Training Project at General Dynamics. She also serves as Project Lead for Scientific and Engineering Training at GD's Central Center in Fort Worth, Texas. Prior to joining the General Dynamics Computer Related Training staff, Ms. Blackmon was a Software Engineer in the Advanced Computer Systems Laboratory at Texas Instruments and an Instructor in Computer Science at Tarrant County Junior College, Fort Worth.


CURRICULUM
OTHER METHODOL. OF CHOICE
DSDA
PSL/PSA
REAL-TIME STRUCT. ANALYSIS
SURVEY OF ANALYSIS METHODOLOGIES
Ada OVERVIEW FOR MGMT. & MKT. PERSONNEL
OTHER METHODOL. OF CHOICE
OBJECT ORIENTED DESIGN
JACKSON
REAL-TIME STRUCT DESIGN
SURVEY OF DESIGN METHODOLOGIES
IMPACT ISSUES FOR TECH MGRS
GDADL
OTHER PDL
PRELIMINARY DESIGN IN Ada
OTHER REPRESENTATION SCHEME OF CHOICE
DETAILED DESIGN IN Ada
INTRODUCTION TO Ada
INTERMEDIATE Ada
ADVANCED Ada
Ada FOR EXPERIENCED PROGRAMMERS
HOW TO USE THE LRM
REAL-TIME ISSUES
SURVEY OF Ada ENVIRONMENT TOOLS
TOOLS OF CHOICE
TESTING
QUALITY ASSURANCE


FIRST YEAR PRIORITIES
OTHER METHODOL. OF CHOICE
DSDA
PSL/PSA
REAL-TIME STRUCT. ANALYSIS
SURVEY OF ANALYSIS METHODOLOGIES
Ada OVERVIEW FOR MGMT & MKT PERSONNEL
OTHER METHODOL OF CHOICE
OBJECT ORIENTED DESIGN
JACKSON
REAL-TIME STRUCT. DESIGN
SURVEY OF DESIGN METHODOLOGIES
PRIORITY
IMPACT ISSUES FOR TECH. MGRS.
GDADL
OTHER PDL
PRIORITY
PRELIMINARY DESIGN IN Ada
OTHER REPRESENTATION SCHEME OF CHOICE
DETAILED DESIGN IN Ada
INTRODUCTION TO Ada
INTERMEDIATE Ada
ADVANCED Ada
PRIORITY
PRIORITY
Ada FOR EXPERIENCED PROGRAMMERS
HOW TO USE THE LRM
REAL-TIME ISSUES
SURVEY OF Ada ENVIRONMENT TOOLS
TOOLS OF CHOICE
TESTING
QUALITY ASSURANCE


CLASSROOM SET-UP
WRITE-ON SURFACE
SCREEN
WRITE-ON SURFACE
SYSTEM PRINTER
CEILING MOUNTED PROJECTOR
COLOR PC
VAX TERM
VAX TERM
CHAIR
CHAIR
CHAIR
CHAIR
VAX TERM
VAX TERM
CHAIR
CHAIR
CHAIR
CHAIR
VAX TERM
VAX TERM
CHAIR
CHAIR
CHAIR
CHAIR
VAX TERM
DOOR

IDENTIFIERS

Definition

An identifier is used as a name for an Ada entity. An identifier is a sequence of characters that stands for whatever data may be associated with it when the program is run.

Rules

1. An identifier is composed of
   - o the letters A through Z (upper or lower case)
   - o the digits 0 through 9
   - o underlines
2. An identifier may be as long as the length of a line.
3. Identifiers are not case sensitive.
4. The first character of an identifier must be a letter.
5. An underline must be embedded between two non underlines.

Examples

```
Current_Altitude
Maximum_Height
Front_40
FRONT_40
Stack_Size
StackSize
Maximum_Altitude_OF_CURRENT_Target
MAXIMUM_Altitude_of_Current_Target
MaximumAltitude_of_Current_Target
```

Non-Valid Examples

```
4_July          (must start with letter)
_July_4         (must start with a letter)
Grand_Total_    (underline must be embedded)
Hello__There    (underline must be embedded between
                two underlines)
Raise_10%       (% not a legal character)
```

Comment

The identifiers Front_40 and FRONT_40 represent the same data item. The identifiers Stack_Size and StackSize represent distinct identifiers.

Choose names that reflect the problem domain. Do not use cryptic identifiers like X, P, CC, etc.

Related Concepts

reserved words

EXAMPLE OF "INFORMATION MAPPED" CONCEPT

USING STRUCTURED TECHNIQUES TO TEACH REAL-TIME EMBEDDED COMPUTER APPLICATIONS

Ruth S. Rudolph

Computer Sciences Corporation
Moorestown, New Jersey 08057

Abstract

The task of teaching programmers to work with real-time embedded computer systems is extremely complex. The number of topics to be discussed and the difficulty of the concepts involved produce poorly organized, ineffective coursework. Use of Ada concepts to organize the material and its presentation offers a solution. Since Ada is designed to handle complexity, its problem-solving approaches can also be effective in teaching complex concepts. The success of this effort (applying Ada technology to such a curriculum) may serve as the basis for designing new teaching methods in the future. Ada constructs for concurrency offer excellent tools for understanding that aspect of real-time systems. This initial attempt assumes that the students had no prior knowledge of Ada language. Nevertheless, the description of concepts in an Ada-like fashion still clarifies the material. If the class has some background in Ada, this new method can be even more effective.

Introduction

An Educational Fairy Tale

The curriculum listed the course as CMS-2, a high-order language, but the course content also included an introduction to the hardware, assembler language, and operating system, and alluded to the life cycle of the project. These topics seemed inseparable in a real-time embedded computer application. The students liked the class with one reservation: the presentation seemed to lack continuity. It was disjointed. "Why," they asked, "does the instructor GO from one topic TO another and never really complete any single discussion?" And so it went, year after year, until one day the instructor read a book by Grady Booch called Software Engineering With Ada. Then the instructor had a dream.

In the dream was shown a course that consisted of neatly-wrapped packages. Each package contained little windows, and in each window was a sign that corresponded to an identical notation in the course outline. Each package contained detailed information about a topic, but the packages were tied together in a precise way. The ribbon around each package NEVER became tangled with any other.

The next day the instructor repackaged the CMS-2 course. The topics carefully were developed into individual self-contained units that could be taken off the shelf. The relationships between the packages were described using standard interfaces. The students did not have to be confused by complex detail because all they could see was what appeared through the windows, and yet all the relationships in the application were clear...and the instructor taught happily ever after.

This fairy tale is fanciful perhaps, but it illustrates an approach to presenting a coherent picture of all the aspects of a large, real-time system to a class whose only experience with computer applications has been writing a small mathematical problem in a high-order language. This is a dilemma that has been growing for the last 20 years. In seeking a solution, an insight was gained into how the system initially was put together.

"Structured" Teaching or a Go-to-Less Course

The designers built a dedicated operating system that now looks very similar to an Ada package. A number of services for the user (applications programmer) were specified. Each specification included a predefined packet (similar to a procedure parameter list in Ada) that served as a template for the user. When the service was requested, the user placed the real values in the packet. All the implementation details of the service were hidden from the user. However, if the request for service was unsuccessful, a flag was set in the user's local scope to indicate it. This is somewhat like raising an exception in Ada since the reason for failure was clearly identified.

Furthermore, implementation of the services was designed in a modular fashion; a look inside at the details reveals an interesting and innovative organization of procedures. The procedures, themselves, were packaged by functionality. This system was built before "software engineering" had become a "buzz" word. The system designers intuitively applied techniques that have since become accepted practice.

Armed with an understanding of Ada, software engineering techniques, and the advantage of hindsight, the relationships between the applications and the operating system can be neatly described in an Ada-like fashion. Ada terminology can be applied aptly to pieces of this system even though the designers had never heard of Ada and, in fact, wrote in assembler language.

The instructor's problem was to explain to the class how an individual application module fits into the system. This required explaining some very difficult concepts and contructs. The student had to understand relative addressing and relocatable programs; the reuse of a fixed number of base registers; the relationship between the high-order code and the efficiency of the resulting machine language; the trade-off between memory and time; and the relationships between the operating system, the application programs, and the hardware.

This was an overwhelming teaching task. There are two aspects to the solution:

1. The topics are limited to details essential to make the discussion coherent. To accomplish this, complex ideas are defined as primitives. The underlying reasons or implementation details are hidden from the student. (Examples follow.)

a. Data definitions in this system can affect the efficiency of execution because of hardware addressing techniques. The complexity of the explanation is distracting and confusing. The solution is to provide user "defaults" for variable declarations which will guarantee efficiency. This is similar to providing user-defined default parameter values in Ada procedure calls. Such defaults are identified as red flags for the class to focus attention on a preferred (but not essential) way of doing things.

b. Data structures can be defined by either the user or the compiler. In addition, the user can select one of two formats for the structure. The preferred choice generates object code that is much more efficient than any of the alternatives. Exceptions to these choices are raised but not explained.

c. The system requires use of fixed-point numbers instead of floating point, and considerations of significance, precision, and accuracy are sometimes new to the student. Awareness of this is essential. A generic approach is used to present this in an understandable way.

2. The relationship between the operating system and the application programs is described in terms of formal interfaces, originally defined by the user but enforced by the operating system. A library package or service provided by the operating system serves as an environment in which a particular program executes. The user sees only the specification of the operating system services. The specification formally describes the interface between the service and user. The result is as if the user were looking through a window. All of the implementation details are hidden and only the necessary information is in view. In turn, the application program may hide details from others as well by packaging related resources. (Examples follow.)

a. A module may be scheduled at a predefined number of standard entry points and only at these points. This is presented as a generic package that contains a group of procedures.

b. A program's data and instructions are divided into two modules to expedite program execution. This is made possible through a single high-level directive to the compiler. The physical base registers assigned to the modules are identified as actual parameters when the linker is invoked.

c. A module requests communication with the operating system to be served or to exchange information in a specified manner. In this case the implementation is viewed as a task. The application module calls the operating system and waits for a response.

d. A module provides parameterized data to the operating system as specified and points to it in a predefined manner. The notion of access types that create an object that holds the parameters helps explain a high-level language function.

e. The operating system returns messages to the modules and points to them in a predefined manner. Access and task types help demonstrate this complex relationship.

f. A library of functions is made available to any user who requires it through a standard interface and a local work area. The library is a package of functions.

This new approach to the course provides the instructor with a collection of teaching resources. These, in turn, are carefully organized into library-like units. Since the makeup of each class is always different, and the classes themselves are not homogeneous groups, another problem in presentation is solved. Now the scope of the course work can be varied in response to the needs of each particular group. In the future, any package from the library can be presented in a standalone fashion for an appropriate audience. Then the hidden details can be explained in a relevant way. Neither the class nor the instructor speak of Ada, but its problem-solving techniques are used to develop the course structure and content. This endeavor demonstrates two things:

1. The ideas in Ada have been used intuitively and effectively in developing real-time embedded systems for at least the last 20 years.

2. The Ada constructs can be used to effectively describe such systems without any reference to the Ada language and in the absence of its use as a program design or implementation language.

Concurrency in Real-time Systems

There is another aspect of embedded systems that deserves analyzing. Real-time systems must deal with problems of concurrency whether the implementation is logical or physical. The operating system handles these issues and therefore they are hidden from the user. However, the application, once again, must make use of specified services to avoid problems such as mutual exclusion and deadlock in real-time programming. The issues of concurrent programming require a different set of problem-solving skills than those required for sequential programming because of the difficulty in ensuring correctness. Mutual exclusion and absence of deadlock present some unique problems in guaranteeing correct implementation. Since Ada supports concurrency, the language provides an excellent vehicle with which to capture the solutions to system concurrency problems and present them in an Ada-like fashion as a springboard for teaching concurrency to applications programmers.

Summary

The Ada "effort" is now 10 years old, and many feel it has become a mystique that will never be a reality. The use of Ada to improve presentations dealing with real-time embedded computer systems may prove to be effective in accomplishing a variety of things:

1. Formal expression of intuitive implementations of difficult concepts.

2. Organized presentation of abstract and complex notions.

3. Introduction to a state-of-the-art method of problem solving for sequential and real-time programs.

4. Easing of the Ada training burden by introduction of Ada-like approaches in non-Ada environments.

5. Exposure of faults in current design caused by limitations in computer architecture and high-level languages.

The Ada language provides both structure and technique for more effectively and efficiently teaching complexity. A new teaching method has evolved on the basis of the instructor's knowledge of the Ada language, which has improved this course (CMS-2) greatly. There is every reason to expect similar results when this method is applied to other courses. Ada concepts provide a valuable tool, even in a training setting.

References

1. ADA83 - Reference Manual for the Ada Programming Language, ANSI/Military Standard MIL-STD-1815A, United States Department of Defense, January 1983.

2. BEN82 - M. Ben-Ari, Principles of Concurrent Programming, Prentice-Hall International, 1982.

3. BOO83 - Grady Booch, Software Engineering with Ada, Benjamin/Cummings, 1983.

4. DEI84 - Harvey M. Deitel, An Introduction to Operating Systems, Addison-Wesley, 1984.

5. PY81 - I.C. Pyle, The Ada Programming Language, Prentice-Hall International, 1981.

About the Author

Ruth S. Rudolph, training coordinator for the Tactical Systems Center at Computer Sciences Corporation, Defense Systems Division (DSD), Moorestown, New Jersey, is responsible for developing the internal technical training courses given within the Center. Included in this technical training is an Ada curriculum, which she designed. For the last 4 years Ms. Rudolph has taught Ada courses throughout DSD.

The Implementation of a Graphics Package in Ada

Benjamin J. Martin Bennett Setzer Reginald Walker

Atlanta Univ. Kennesaw Col. Atlanta Univ.
Atlanta, GA Marietta, GA Atlanta, GA

This paper is a report on a project to develop a graphics package for the Zenith Z-100 computer system. The project began with implementing a subset of the CORE system in Pascal. Subsequently, the system was converted to Ada. Current work concerns extending the system. This report is mainly concerned with the rationale for using Ada to implement both the original and the extended graphics systems. We conclude that Ada is clearly superior to Pascal in defining a graphics system. Some minor concerns have arisen due to the size of the Ada compiler.

This paper is a report on a project to develop a graphics package for the Zenith Z-100 computer system. The overall project can be divided into these phases:

-- Implementation of a subset of the CORE graphics system in Pascal.
-- Implementation of more sophisticated graphics drivers for the Z-100.
-- Porting the original system to other versions of Pascal.
-- Converting the Pascal system to Ada.
-- Development of a different graphics model.
-- Implementation of the new model in one or more target languages.
-- Porting to different micro-computer systems.

This report is mainly concerned with the rationale for using Ada to implement both the original and the extended graphics systems. We conclude that Ada is clearly superior to Pascal in defining a graphics system. We have some concerns about setting up a system that is convenient for the student to use. The Janus Ada compiler is large and complex to use if a hard disk is not available.

This report is organized into these sections:

-- Rationale and history for the project as a whole.
-- Comparison of the Ada and Pascal implementations.
-- An extended graphics model.
-- Implementation considerations for the extended model.

RATIONALE AND HISTORY

The origin of this project was the need for an inexpensive graphics package that could be used to support courses in computer graphics and computer vision. We are using the word "package" loosely here, to mean a group of routines. The hardware to be used consisted of over a dozen Zenith Z-100 micro-computers in the Micro-computer Laboratory at Atlanta University. Available packages were too expensive or were hosted by unsuitable languages, such as BASIC. Since Pascal was the major language used throughout the computer science curriculum at Atlanta University, we wanted Pascal, or a similar language, to be the host language.

A further problem with existing packages was that most were device drivers. That is, they implemented graphics primitives, but provided no more. It was finally decided to implement a portion of the CORE graphics system using Pascal. This was because of availability: [Harrington] contains a very detailed description of a CORE subset, including pseudo-code routines; and, Pascal was available for the micro-computers. This implementation was carried out by G. Payne and completed by him in 1985 (see [Payne]). An advantage of developing the system locally was that the actual algorithms used would be

available for study in courses.

At this point three issues arose that prompted us to develop this package further:

-- Because of limitations inherent in Pascal, the package did not represent the best software design practices. We felt that this would result in students having difficulties using the package.
-- Several faculty at Atlanta University became heavily involved with programming in Ada and proposed that Ada become the basic language used in the computer science curriculum. Also, Ada became available for use in the curriculum with Janus Ada.
-- Research efforts in computer vision and artificial intelligence were moving forward. To support this, a graphics system with a dynamic and hierarchical concept of a graphic image was needed.

Two groups began work developing the package further. Martin and Walker undertook to convert the CORE package into Ada. Setzer began the development of a more efficient low-level interface to the hardware and began designing the new graphics model. We eventually combined efforts. The following phases of the project are complete: the CORE subset has been implemented in Pascal and converted to Ada; better drivers for the Z-100 have been coded and tested. At this time, we are working on designing and implementing the new model.

In summary, we came to realize that our educational and research objectives required a refinement of both the specifications of the graphic system and the implementation.

-- The package itself needs to be a good example of software design, specifically, demonstrating modularity, information hiding, and security.
-- The specification of the package should present a clean interface to the user.
-- The package should support a dynamic, hierarchical concept of a graphical image.

IMPLEMENTATION COMPARISON

The subset of the CORE system supported includes 2 dimensional pictures, segments, filled polygons, and transformations. No windowing or clipping and no mapping to viewports is supported by the subset implemented. We felt that this was adequate as a starting point for the system. Also, the decision to pursue a completely different design was made soon after Mr. Payne completed this implementation, which also delayed consideration of any extensions.

The Pascal version was implemented in MicroSoft Pascal. Although this version of Pascal supports separate compilation, this is not a standard feature of Pascal. Other versions of Pascal do not have this capability at all or support it in a different manner. Since we eventually wanted to use versions of Pascal that would not support separate compilation, such as Turbo Pascal, we decided not to use this feature. This dictated including the actual graphics system source in a program using it. We note that most Pascal compilers allow the compile time inclusion of source text. We emphasize, in this section we are discussing Pascal without separate compilation.

The Ada version was implemented in Janus Ada by converting the Pascal routines. In the Ada implementation, the routines are organized in a package with the work areas hidden within the package body. The package can then reside in a library. An application then needs only an appropriate 'with' clause to access the graphics routines.

We observed three major problems with the Pascal implementation. Each problem could cause difficulties for students using the Pascal version. The Ada version effectively solves these problems. The problems are

-- The necessity of global variables in Pascal for certain data.
-- The inability, in Pascal, to hide the actual type of certain data.
-- The necessity, in Pascal, of including the entire source code for the graphics package at each compilation.

In the Pascal implementation, an important part of the included source is the definitions of work space for the graphics system. This includes storage for segments as well as status information, such as the current pen position. The organization of the system is such that all the procedures act on this body of data. Because of the number of data, it is inconvenient to pass them as parameters. Also, the data must remain between invocations of the system. For these reasons, Pascal global variables are used for these work spaces.

The global variables in the Pascal

implementation represent its most serious difficulty. First, status and other work area data can be freely manipulated by the user. The dangers of this are well known. Second, there could easily be conflicts with the names of a user's global variables. Besides being an annoyance, this could cause considerable confusion to a student trying to use the system. Searching for the second definition of a variable, when one definition of it is hidden in an include file, can be frustrating.

In the Ada implementation, work areas are hidden within the body of the graphics package. This makes this data unavailable to the user, except through the exported routines. This also makes the variable names invisible to the user's program.

Pascal does not allow the protecting of a data type by defining a type for it. For example, a segment name in the CORE system is simply a small integer. It seems, however, to make little sense that segment names can, therefore, be operands in arithmetic operations and can be the results of arithmetic operations. However, Pascal does not protect from such misuse. In Ada, making a segment name a private type prevents this.

The necessity of including the complete source code of the graphics system into a program presents two problems. The first is simply a question of efficiency and convenience. Object code is generally more compact than source code. This implies less disk space used, and generally quicker processing of programs. The size of the system as distributed to others is particularly a problem with the large Pascal compiler.

The second problem was observed when, in a programming, course students were assigned a problem involving a set of procedures provided by the instructor. The procedures were in source form. A significant number of students modified the provided procedures to conform to their program designs although that had been strictly forbidden. These students were quite dismayed with the evaluation of their programs, since the programs were run with the instructor's original set of procedures. It was quite difficult to convince some of these students that the problem was in their programs and not in the procedures provided by the instructor.

The point is similar to the previous problems, given enough users, there will be misuse. Since, in Ada, only the compiled package would be distributed, the students will not be able to manipulate the source. This will eliminate one cause of problems.

As can be seen, one of our major concerns is security. Simply put, with numerous students using a system, every possible misuse will appear. Protecting the package from the student and vice-versa can considerably lower the number of possible problems. In turn, this will benefit the student in the learning process.

To conclude this section, we note that Mr. Payne did a very good job of implementing the CORE subset. The desire to modify and extend his work was due to problems with Pascal and with the CORE system itself. As will be seen, our design to remove some of the problems with the CORE system would present more difficulties in using Pascal.

AN EXTENDED MODEL

In response to new research and educational needs, we developed several criteria for a graphics model. We use the word "model" to refer to the way the graphics system looks to the user, independent of the implementation. A model needs to meet these criteria to satisfactorily support our work. The effect of these criteria is to move towards a more object or data oriented view of graphics and away from a procedural view. Our criteria were as follows.

-- Pictures, transformations, windows, and display devices are objects that can be manipulated by the system.
-- These objects can be manipulated, combined, and brought into relationships with one another by operations defined by the system.
-- Pictures can have hierarchical structures of unlimited complexity.
-- Such objects can be created or destroyed. Memory usage should substantially reflect the number of active objects and their complexity.

The CORE system did not meet these requirements in several regards. For example, in CORE a geometric transformation is applied to a segment by defining the transformation and then drawing the segment. The drawing commands are suitably transformed and the result saved as the segment definition. Thus, the relationship between a transformation and the segment

to which it is applied is implicit in a time sequence of events. Other than flow of control, there is no evidence in a program of this relationship. In our model transformation and a picture are combined by a binary operator to result in a new picture. The new picture is the original picture, transfomed.

One important class of operations on pictures is that they be combinable into larger pictures. This should be done while still preserving the picture/sub-picture relationship. There are at least two advantages to this. First, sub-pictures can be specified and manipulated, independently of the super-picture. Second, analysis of digitized images leads to a hierarchical description of the scene. See, for example, [Ballard], especially chapters 10 and 11. Another discussion of the use of hierarchies in modeling can be found in [Foley], Chapter 9. Structured pictures can better represent the results of this analysis.

In the CORE system, pictures are represented more by procedures than by data. The hierarchical relationships of several procedures involved in drawing parts of one picture can represent the structure of that picture. This is, however, relatively static. We need to be able to dynamically change the structure of a picture.

In summary, the extended system can be described as defining several data types. Implicit in these definitions will be the desired operations. Such a description is more intuitive for students: objects are being manipulated with certain results. A functional description, with little emphasis on side effects, is easier to teach and easier to verify.

IMPLEMENTATION CONSIDERATIONS

In this final section we present reasons for and against using Ada to implement the extended model. The positive reasons can be summarized as follows:

-- Greater security problems in the extended system.
-- The ability of Ada to represent data types.

The negative reasons reduce simply to the difficulty in using the Ada compiler.

Any graphics system including the ideas of the preceeding section will, of necessity, represent pictures by means of linked structures. For example, two-pointer nodes would suffice to represent an arbitrarily complex hierarchical structure. Pointers are hazardous to deal with, even in a well regulated environment such as Ada. Keeping pointers hidden away within a package can prevent numerous problems for student and other users.

The Ada package is an ideal way to implement a data type. All structure irrelevant to the data type definition can be hidden from the user. Both type definitions and operations can be exported to the user. Thus, the added types and operations can appear to be language extensions. This gives a very clear picture to present to students, greatly easing the teaching problems.

The major objection that we can see to using a graphics package in Ada is due to the compiler. The Janus Ada compiler is a large program, requiring the student to manipulate at least two floppies for the compiler. It is also fairly slow. In fairness, however, both of these points are true of the Microsoft Pascal compiler. A large computer version of Ada using a micro-computer as a graphics display device may turn out to be one way to solve this problem. Also, on a system with a hard disk drive, the compiler size would not be much of a problem.

We are naturally drawn to Ada by its expressive power and standardization. We do feel that some effort will be needed to set up an Ada system that is convenient for students to use. Save that one concern, using Ada offers great benefits in the classroom.

REFERENCES

[Ballard] Ballard, Dana H. and Christopher M. Brown. COMPUTER VISION. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[Foley] Foley, J. D. and A. Van Dam. FUNDAMENTALS OF INTERACTIVE COMPUTER GRAPHICS. Addison-Wesley, Reading MA. 1985.

[Harrington] Harrington, Michael. COMPUTER GRAPHICS, A PROGRAMMING APPROACH. McGraw Hill, 1985.

[Payne] Payne, Gregory. "Implementation of the CORE Graphics System on the Zenith Z-100", Thesis, Atlanta University, Atlanta, GA.

EXPERIENCES OF PASCAL TRAINED STUDENTS IN AN INTRODUCTORY ADA COURSE

Robert C. Mers

North Carolina Agricultural and Technical State University

Abstract

The author has designed and taught a Programming in Ada course at North Carolina A. & T. State University in Spring and Fall semesters of 1985. This course assumes a thorough knowledge of all Pascal features, including records and pointers, and attempts to give a complete overview of the Ada language in one semester. Hands-on experience is an integral part of the course. In this paper the content of the course, including programming assignments, environment, and resources, is described. A quantitative measure of student achievement in Fall 1985 is taken and an analysis is given. Observations of the degree of difficulty students have learning various Ada features and suggestions for improvement in the course are made.

1. Introduction and Goals

Several papers presented previously at these conferences have dealt with the contents of a first programming course in the Ada language (e.g. Richman[3] and Rudd[4]). This paper describes the content of such a course at North Carolina A. & T. State University, but it also addresses the degrees of difficulty that Pascal trained students have learning various Ada features. Both noncognitive and cognitive instruments are used to measure students' competencies in these Ada constructs. The statistical results from these instruments, along with the instructor's direct experience with the students, are used in improving the course to challenge and yet enable these students to more effectively learn Ada features. This paper has been written to provide direction in designing a first Ada course which is geared to students trained in a least two semesters of Pascal.

The goals of this Programming in Ada course are to give students a complete overview of the Ada language, provide hands-on experience programming various constructs of the Ada language, and create awareness of development and controversies of Ada and software engineering and design techniques using Ada. Important in achieving these goals is development of competencies in advanced features of Ada such as package design, separate compilation, generics, exception handling, and tasking.

2. Course Content and Methodology

(a). Syllabus

The topics of the course were covered in the below order in both Spring and Fall 1985 with some minor permutations.

The first part of the course is an overview of the language prior to nitty gritty details, the purpose being to give students a feeling of the spirit and style of Ada. History and rationale of the Ada language is given along with a brief description of its major goals and features of Ada (see Barnes[2]). A fairly detailed treatment of packages, subprograms, generics, and TEXT_IO is given so that students can understand simple Ada code and the use of the TEXT_IO package in this code.

At this point the rudiments of the language are covered. A section is now done on the predefined types and operations as well as type, variable, and constant declarations. Detailed discussion of enumerated types, subtypes, and derived types is covered immediately after control structures rather than now in order to get the students programming as soon as possible. Since meaningful programming requires knowledge of conditional and loop structures, the IF, CASE, and LOOP statements are covered next. Basic loops and EXIT statements are mentioned as well as the familiar WHILE and FOR statements. At this point the transition is made to user defined types. Derived types, subtypes, and enumeration types are now presented, as well as attributes, overloading of literals, and general type conversion. To complete the emphasis on modularization,

the features of subprograms such as named and positional notation, recursion, default parameters, overloading of subprogram names and operations, and scope and visibility are covered next.

The course now makes the transition from simple to compound data types. DECLARE blocks are introduced here because of their application to creation of array objects at run time. Both constrained and unconstrained array types are introduced, as are array attributes, assignment using aggregates, slices, array operations, and strings. Records without discriminants are covered in detail.

Emphasis is now shifted from programs as single compilation units to use of packages external to the program and separate compilation. Compilation units are covered and more detail on packages is given. Then abstract private and limited private types are presented with rationale, their position in the package, and examples.

The advanced typing features covered next build on previous treatment of types. Discriminated records and access types are discussed. Students are then introduced to user defined floating point and fixed point types, including their hardware implementation.

Exception handling, detailed treatment of generics, and tasking are covered last since these concepts are most foreign to the students' previous computer science experiences. Predefined exceptions, exception handling and propagation of exceptions are covered in some detail, but detailed treatment of generics and tasking get short changed due to time constraints. However, the rudiments of parallel processing, caller and called tasks, task rendevous, entry and accept statements, and select statements are covered.

(b). Environment

The compiler used is the New York University Ada Ed for the Digital VAX 11-780 Computer, both provided to North Carolina A. & T. State University by the U. S. Army Center for Tactical Computer Systems. In Spring 1985 version 1.1 of Ada Ed was used. The slow compilation and execution time of this version made it difficult to complete more than a few programming assignments and caused a long turn around time (30 minutes per run) even on batch mode. In summer of 1985 version 1.5 was made available, and since then the turn around has improved to the range of 5 to 10 minutes.

(c). Program Assignments

Due to the nature of the compiler described above and the fairly extensive programming background of the students, the instructor's methodology has been to use relatively few medium length programming assignments rather than give many short programming assignments. The contents and objectives of the programming assignments are indicated below.

Program 1. This is a rather simple program involving use of the predefined scalar types and IF and WHILE control structures. The objective is two-fold, to get used to the VAX system as well as becoming familiar with TEXT_IO and the rudiments of Ada.

Program 2. In this program the students are doing Input and Output on enumerated types as well using the FOR and CASE statements.

Program 3. This program requires the students to use unconstrained arrays and create array objects at run time. Use of attributes is emphasized. Students do matrix computations and provide error handlers when operations are undefined.

Program 4. Here the students design a package and use its resources from a main procedure. Separate compilation is required. One semester a package of trigonometric functions was designed; another time the package consisted of complex number operations.

Program 5. This program focuses on variant records and/or linked lists involving these records. The students have to observe the strict rules of discriminated records and get used to the non pointer notation of Ada.

(d). Resources

A primary text is used, and students are provided a copy of the Reference Manual for the Ada Programming Language[1]. However most of the material presented in class consists of the instructor's lecture notes compiled from sources listed in the bibliography including SoftTech notes for the CENTACS Summer Faculty Research Program of 1983[5]. Barnes[2] was the primary text in Spring 1985, but it was replaced by Young[6] in Fall 1985. This author feels that Young's text is the best current comprehensive elementary book on the Ada language from the standpoint of completeness, style, consistency in clarity of presentation, and applications.

3. Analysis of Student Experiences

(a). Method of Data Collection

To analyze quantitatively the degrees of student difficulty in various areas of the Ada language, both cognitive and noncognitive instruments were administered to the Fall 1985 Ada class at the end of the semester. The noncognitive instrument was a questionnaire on 25 topics from the Ada language. These were briefly described, and the students were asked to choose a response for each from the choices "Quite Difficult," "Somewhat Difficult," and "Not Difficult." Questions with the same responses were asked regarding the Language Reference Manual[1], textbook, and instructor's lectures.

The cognitive instrument was the final examination of the course, consisting of 25 questions, one in each topic from the noncognitive questionnaire. Student performances in each area were recorded. Numbering of the noncognitive questionnaires enabled the instructor to match these with the corresponding final examinations.

Each student was asked to give the overall GPA (grade point average), grade in introductory Pascal (C260) and grade in advanced Pascal (C265) in order to determine what effect these variables might have on student performance on the various topics of Ada.

The 25 topics questioned were (1) subprograms, (2) packages, (3) generics, (4) predefined types and operations, (5) enumerated types, (6) subtypes and derived types, (7) basic control structures, (8) basic LOOP and EXIT statements, (9) unconstrained arrays, (10) simple records, (11) aggregate assignment, (12) attributes, (13) overloading, (14) named notation, (15) default values, (16) separate compilation, (17) private types, (18) real types, (19) discriminated records, (20) access types, (21) TEXT_IO, (22) type conversion, (23) exceptions (24) blocks, and (25) tasking.

The population was a set of students from the C290 (Ada) class. 15 students completed the noncognitive questionnaire; 16 took the final examination; and 14 participated in both.

Copies of both the cognitive and noncognitive instruments are available upon request.

(b). Analysis of the Noncognitive Instrument

A Chi Square test of proportionality was completed for each of the 28 items (the 25 listed above plus the questions on the Reference Manual, textbook, and lecture notes). With a 20% chance risk of rejecting a null hypothesis (proportion of students having problems same as those not having problems), we conclude that students perceive themselves having problems with survey items 12 (attributes) 17 (private types), 24 (block structures), and 25 (tasking). A regression analysis was also performed to determine if the variation in student self perceived problems (measured by the noncognitive variable) can be attributed to GPA and grades in C260 and C265. The multiple correlation between the noncognitive items and GPA and C260 (Introductory Pascal) grade is 0.452. The proportion of variation in student perceptions of Ada problems accounted for by the joint knowledge of GPAs and C260 grades is 20%, and this contribution is not statistically significant at the 0.05 level (i.e., by replicating the experiment 100 times, we are guaranteed that 95 of such experiments will lead to a conclusion that the variation in student perceived problems explained by the GPA and C260 grade is not significant). The C265 (Advanced Pascal) grade was found to be unrelated to student perceptions of Ada problems.

(c). Analysis of the Cognitive Instrument

Again a Chi Square test of proportionality and regression analysis were used. The correlation between GPA and total score on the cognitive test (final exam) is 0.510. The proportion of the variation in the Ada final test scores accounted for by knowledge of GPA is 26%. Although this contribution of the total variance explained is not statistically significant at the 0.05 level ($f = 4.216$ with 1,12 degrees of freedom), the GPA appears to be a determinant factor of problems that students will face in higher level computer science courses including Software Engineering Using Ada.

Given that the knowledge of GPA has already been used to explain the variation in the C290 (Ada class) cognitive test scores, 4% is the additional variation in the test scores explained by grades in C260 and C265. This additional variance is not statistically significant at the 0.05 level, indicating that grades in C260 and C265 do not have much in common with performance in C290.

The Chi Square proportionality test was performed on the number of scores below the average versus the number of scores above the average on each item. With a 10% chance of making a wrong inference, we conclude that students have problems with cognitive items 13 (overloading), 21 (TEXT_IO), and 24 (block structures). Although not statistically significant, students appear to have problems with item 25 (tasking).

(d). Degrees of Consistency

It is clear from the above analyses that there is consistency between a student's perception of having problems and the student actually having problems in the areas of block structures and tasking Students thought that they had problems with private types and attributes but did not have problems on the cognitive test. But for overloading and TEXT_IO students had problems on the cognitive test but did not perceive themselves to have problems.

Reasons for the inconsistency could be (1) students' lack of clarity about their strengths and weaknesses and (2) the varying levels and kinds of questions asked on the cognitive instrument. These questions included True-False, short essay, listing, short computations, fill in the blank, writing declarations, and writing short coding segments. The greatest discrepency was on the item TEXT_IO. Students felt very comfortable with it, having used it in programs all semester. However the students had trouble with the TEXT_IO question on the cognitive test, which was "Name and describe 4 procedures in TEXT_IO other than GET or PUT." The author was stunned that students would OPEN, CREATE, and CLOSE files and do NEW_LINE and SET_COL commands all semester and yet not be able to name them on an exam!

4. Conclusion and Recommendations

In addition to above mentioned topics in which students have difficulty, the author has found that students have difficulty on topics such as access types discriminated records, separate compilation, and exception handling, although this is not statistically shown by this study. In general, Pascal trained students have had the most difficulty on the Ada features that are most unlike Pascal. Therefore the author feels that a stronger emphasis on Ada features unlike Pascal and a more rapid treatment of Pascal like features will more successfully fulfill the goal of giving Ada students a deeper knowledge of all the major Ada features in a one semester course.

The background of the students, teaching experience by the author, upgrading of the North Carolina A. & T. computer science program, and students' varying abilities to learn various Ada constructs are all factors in the development and delivery of this course. The level of the students seems to be improving as the computer science program matures. Grades in the Fall 1985 course were considerably better than in the Spring 1985 course. Both courses covered essentially the same topics although slightly more time was spent on elementary topics (beginning through simple records) in the Fall as a response to poorer student performance in the spring. About ten of the fifteen weeks of the course were spent on elementary topics.

The instructor, in consultation with the Department of Computer Science is in the process of upgrading this course. Beginning in Spring 1986 only six to seven weeks will be spent on elementary topics and far more time spent on advanced topics and the software engineering aspects of Ada. The students should be able to absorb those constructs that duplicate Pascal more quickly and more completely understand packages, separate compilation, advanced types, exception handling, generics, and tasking because more time will be spent on these topics.

The programming assignments will be adjusted accordingly. Only the first program will be restricted to discrete types. The second program will focus on run time array processing, and the third will involve package design. Remaining programs will involve advanced types, generics, exception handling, and possibly tasking. Perhaps individual or team projects will be assigned emphasizing software engineering techniques and design.

In addition the course title and number will change from C290, Programming in Ada, to C490, Software Engineering Using Ada. This change is in the spirit of the ACM (Association for Computing Machinery) recommendations for accreditation, which deemphasize a proliferation of elementary language courses and emphasize rigorous computer science courses as electives in the undergraduate major.

References

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A,1983.

2. John Barnes, Programming in Ada, 2nd Edition, Addison-Wesley, 1984.

3. M. Susan Richman, Teaching Ada as the Student's First Programming Language, Proceedings of the 2nd Annual Conference on Ada Technology, 1984.

4. David Rudd, Teaching Ada at Hampton Institute, Proceedings of the 2nd Annual Conference on Ada Technology, 1984.

5. Softech Inc.: CENTACS Summer Program, U. S. Army (CENTACS), 1983.

6. S. J. Young, An Introduction to Ada, 2nd Edition, John Wiley, 1984.

Acknowledgements

The author wishes to acknowledge the help of Mr. Amos Olagunju of the Department of Mathematics and Computer Science, North Carolina A. & T. State University, in the development of this paper and analysis of the questionnaire. He also thanks the C290 class of Fall 1985 for participation in the questionnaires.

Dr. Robert C. Mers
Department of Mathematics and Computer Science
North Carolina A. & T. State University
Greensboro, NC 27411
919-379-7823

Dr. Robert C. Mers is an Assistant Professor of Mathematics and Computer Science at North Carolina A. & T. State University. He received the Ph.D. degree in Mathematics from the University of Colorado at Boulder in 1975. In 1983 he was a participant in the U. S. Army CENTACS Summer Faculty Research Program, consisting of 10 weeks of Ada Language Training. He is also a 1984 graduate of IFRICS (Institute for Retraining in Computer Science) at Clarkson University.

THE DEVELOPMENT AND IMPLEMENTATION OF AN ADA* TUTORIAL SYSTEM

James E. Walker

Prairie View A&M University
Department of Mathematics and Computer Science
Prairie View, Texas 77446

ABSTRACT

This paper describes how good instructional development techniques can be maximized when integrating Computer Assisted Instruction (CAI) strategies with those of Computer Managed Instruction (CMI). The Ada programming language will be used to implement the system. This project is referred to as a system because of the CAI/CMI integration. Instead of just teaching information to students via a computer, students are provided a custom-designed learning program that is managed by the computer. The computer may instruct a student to refer to the Ada Reference Manual, review a tutorial, take a test, or continue with the next level of instruction. As a student completes each level of the tutorial, the student's achievement and progress are recorded. By accessing the records of a particular student, a teacher can determine what success the student is having and can identify any problem areas that might exist. The teacher can also determine the effectiveness of the learning program itself.

INTRODUCTION

Because of the diversity of student's background, the Ada Tutorial System (ATS) provides an ideal situation for the use of the individualization based on rate-of-progress. Each student can proceed through a given program at a rate that is commensurate with past job experience. The ATS is an information system designed to facilitate the management of instruction and individualized instruction in particular. It provides the automated data collection, data processing, and reporting capability needed to cope with the managerial demands of individualized instruction. The ATS frees the instructor from much of the low-level clerical work inherent in modern training curricula while providing the tools necessary to manage instruction.

Phase 1: Design

Competency Analysis

Competency analysis is a particularly critical component of instructional development. It allows the course developed to break up what must be learned into manageable-sized segments, and it ensures that the ATS addresses all that needs to be taught. Competency analysis involves breaking down large tasks or educational goals into smaller subtasks so that learning can be operationalized. It is important that the course developer knows what components make up a skill so that each component can be taught and behavior can be shaped with the result that the student can perform the whole task correctly and reliably. Since a competency is a set of performance objectives, it can be used as an indicator of what a person can do.

The development of this component was simplified by using the competency level segments that were predefined by the Ada textbook that is currently being used to teach Ada at Prairie View A&M University. Because of the magnitude of the Ada language, the ATS will only cater to the "Pascal Subset" and packages. The method of learning used to acquire each competency will be cognitive domain.

Design Evaluation

In the previous stage the designer ascertained, in a sense, the destination of the ATS. Now the designer must decide how to determine when that destination has been reached.

In the cognitive domain, short answer, multiple choice, true/false, and matching are suitable for testing depending on the performance level of the objective. The ATS is particularly well suited to the evaluation of competencies in the cognitive domain because of it's capability for providing testbanking functions.

During this stage of design, testbanks are created. A testbank is a set of questions or test items that measure the students' ability to perform one or more competencies. It is critical that test items be matched to the type of learning and performance level of the associated objectives. As mentioned earlier, ATS will implement the cognitive domain method of learning. Fig. 1.1 shows the Flow of the ATS.

*Ada is a registered trademark of theU. S. Government, Ada Joint Program Office (AJPO).

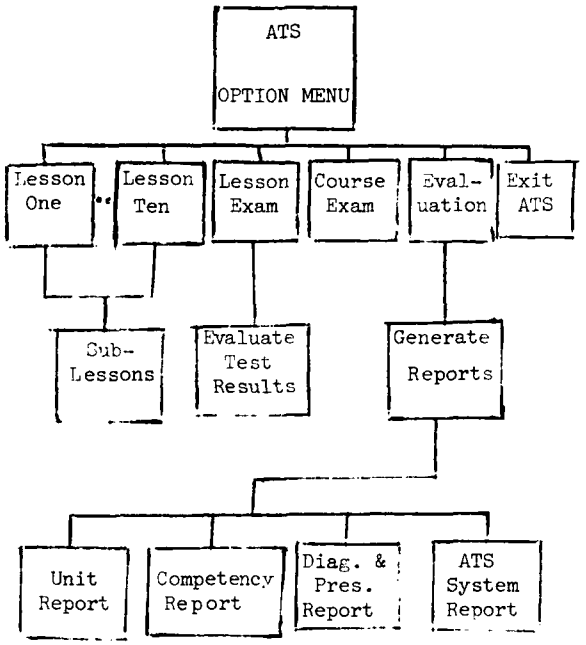


Figure 1.1

Phase 2: Development

During the development phase, decisions are made about how the outcomes identified in the design phase will be attained. It is during the development phase that the instructional activities are assembled in a way that will enable students to achieve the competencies identified.

Hardware/Software Requirements

Because of the campus wide availability, the IBM PC was chosen as the host computer for ATS. Each PC has a minimum of 256K, color graphics capabilities, and a dot matrix printer.

The ATS software package was written in Microsoft Basica because of it's many available functions.

Delivery

The instructional delivery system consists of the instructors and facilities used to train the students. The goal of the instructional delivery component is to train the students totheir maximum potential within a minimal time frame and with the available resources. The contents of each module in the ATS are as follows:

I. ATS MENU

- A. Lessons (In Levels)
- B. Exams (In Levels)
- C. Course Exam
- D. Evaluations
- E. Exit ATS

II. Lessons

- A. Introduction
- B. Names in Ada
- C. Predefined Data Types
- D. Constants and Expressions
- E. Control Structures
- F. User-Defined Types
- G. Arrays, Strings, and Records
- H. Subprograms
- I. Formatted Input and Output
- J. Packages

III. Exams (Tests For Each Lesson)

IV. Evaluations (Analyze Test Results)

Instructionally Related Functions

Data Collection: To support the instructional management process, a wide variety of data must be collected by the ATS. The data defining each student's instructional history are generated as the student passes through the curriculum defined by the design phase.

Diagnosis and Prescription: From a student's point of view, diagnosis and prescription are the core of the ATS. The diagnosis function is used to determine the basis of the students' observed performance. A mechanism was implemented in the ATS to diagnose students' performance at the unit of instruction.

Reporting" Because the ATS is basically a management information system, reporting is a crucial function. Each of these persons, from his or her own point of view, is interested in the instructional and other data stored by the ATS. As a result a variety of reports are needed: A Student Competency report, Diagnosis and Prescription report, and a System-Related report.

Set Progression or Achievement Criteria

Part of the development phase required the ATS developer to establish measures for progression decisions and testing functions. It is at this stage where decisions are made regarding when students will be tested and on what portion of the subject matter. Also, standards for achievement must be set. The ATS developer must decide the standards by which the student will be judged to have achieved competency in a particular area.

The measure for progression in the ATS is for the student to make a grade of 90 or better. There are tests for each lesson and the questions are equally weighted. The highest possible score for each test is 100.

Set Remedial Strategies

In order to achieve an individualized learning system, the ATS developed must address the needs of students who have difficulty achieving the prescribed competencies or who do not have the

appropriate entry level skills and knowledge. Toward this end, the ATS developer must establish remedial strategies so that the learning is maximized for all students.

The ATS handles remedial strategies by prohibiting the student from continuing to the next lesson until the recommended competencies for that particular lesson have been achieved. Each time a student logs into the system a database is searched to locate that students location in the course.

Phase 3: Implementation

To explore fully the management of instruction via the ATS conceptualized in the preceding phases is a very large task. Therefore, we will look at the operation of the ATS from the point of view of the student and the instructor. The intent is to present the flavor of the system in a concise manner.

The Student

Upon logging into the ATS the student is presented with the ATS Option Menu. If the student chooses to view a lesson then he/she is presented with a menu of competency level lessons. If the student chooses a lesson with sub-lessons, then the student may select which sub-lesson he/she wants to view. Once the student has viewed each sub-lesson with a comfortable level of assurance, then he/she will be eligible to take a competency level exam. If the student receives a grade of 90 or above on the competency level exam then he/she may proceed to the next competency level tutorial. If the required score was not achieved then the student may be directed to additional reading material, asked to review a certain sub-lesson, or to review the entire competency level lesson.

When the student has successfully completed the entire curriculum presented by the ATS then he/she will receive a transcript report verifying this accomplishment.

The Instructor

Because the level of instructional management related to behavioral objectives has been automated by the ATS, the instructor can focus on higher levels of management. The ATS puts at the instructor's fingertips a large number of reports, which can be used in many ways. Because the instructional process is highly individualized, the instructor can monitor individual students. He or she can use the Unit Report to see where the students stand relative to the curriculum. The Performance Profiles can be used to check progress. By using the computer to generate performance profiles, the instructors may discover areas in which they might change the ATS to improve learning technology.

Phase 4: ATS Evaluation

Feedback is an essential component of the instructional development process. During the evaluation phase, the ATS developed draws on many sources of data in order to judge the success of the Ada Tutorial System he has created. The most potent indicator of success is whether students have indeed learned what was intended.

Based on feedback, the course developed identifies areas of the ATS to be improved, and the cycle begins again, at the design phase.

References

1. Computer Based Training Systems, 1984.
2. O'Neil, Harrold F.: Computer Based Instruction, Academic Press, 1981.
3. Reference Manual for the Ada Programming Language, ANSI/MIL STD 1815A, 1983.
4. Siad, Sabina: Ada: An Introduction, CBS College Publishing, 1985.

Biographical Sketch

Mr. James E. Walker is an instructor of Mathematics and Computer Science at Prairie View A&M University, Prairie View, Texas. Mr. Walker is a graduate of Prairie View A&M University where he received his BS/MS in Mathematics. Mr. Walker is currently the secretary for the Conference on Software Technology.

The Use of Computer-Assisted Instructions in the Areas of Reinforcement and Testing for the L202 Module (Basic Ada Programming) of the US Army's Ada Training Curriculum

P. Caverly, R. Canavan, P. Goldstein, and K. Pastuzyn

Ada Technology Center
Jersey City State College, NJ 07305

Abstract

The Ada Technology Center at Jersey City State College has received a contract from CECOM to produce Computer-Assisted-Instruction materials for use in reinforcement of concepts and testing of students taking the L202 course (Basic Ada Programming) of the U.S. Army's Ada Training Curriculum.

Based on our experience of teaching the L202 course to hundreds of students of varying backgrounds, we feel that reinforcement will be particularly valuable to them both while they are taking the course and after they have returned to work. The testing capability will be of special interest to managers who will be able to evaluate the achievement of those employees who take the course. The materials are being developed using Digital's Dimension Authoring System which runs under the VAX/VMS operating system.

Introduction

This paper describes our approach to the cost effective and efficient use of CAI for support of Ada Technology training programs. It is our experience after training over a thousand government and insdustrial programmers, scientists and engineers that a multi-media approach is most efficient and cost effective.

The U.S. Army has spent a great deal of money to develop an Ada Curriculum which contains software engineering methodology courses and hands-on Ada language courses. The training in the software engineering courses is currently delivered by lecture and some problem sets, while the hands-on Ada language courses are delivered by lecture and lab work in computer program development. These courses are very intensive presenting a great amount of material over short periods of time. This means that students need to apply the concepts presented almost immediately. Before they can gain any proficiency in a topic, a new topic is already being covered. After completing a course and returning to their work site, many students do not have any means or resources to enhance their understanding of what they have learned. To help them overcome this limitation we are developing the means by which students can reenforce their understanding and test their knowledge at their own pace. We are doing this via an automated portable medium. We are also providing a testing capability for managers so that they can assess the expertise of course attendees.

CAI for Reinforcement and Self-Testing

For our initial effort, we have chosen to develop CAI Reinforcement and testing materials for L202, Basic Ada Programming, the first hands-on course in the Ada language in the U.S. Army's Ada Training Curriculum.

We are developing our CAI materials using Digital Equipment's DIMENSION Authoring System (DAS). DAS is a complete system for writing and delivering computer-aided-instruction lessons, and has the capability for various types of record keeping and report generation on students activities. DAS runs under VMS on VAX computer systems and is therefore portable over VAX systems.

We have zeroed in on the difficult aspects, such as

- Strong Typing
- Generic I/O Packages
- Structured Programming
- Packages
- Separate Compilation and Library Facilities

The combination of instructor, student guides, hands-on experince, self-paced automated reinforcement and self-testing provides greater opportunities for in-depth learning.

Testing

Testing, on the other hand, is for the manager. There has been wide concern about the expertise of professionals who

have taken the intensive Ada courses. Managers have been sending their people to Ada courses at considerable expense and time away form the job, yet without any means of judging the level of competency of the attendee once he returns from the course.

By providing the manager with a CAI test package consisting of a set of graded exercises on the facilities of the language, the manager can get a grasp on student competency level and make a vale judgement to see if the student can go on for further studies in Ada technology, or should revisit the reinforcement self-testing portion. We are providing these two packages under this contract effort, but there has to be further investigation to develop norms for the testing phase.

Conclusion

A great deal of time, effort and expense is involved in creating good CAI lessons and tests. When CAI is used as the sole delivery medium, the results can be disappointing. In most circustances students learn best when exposed to a variety of teaching approaches, and this is especially true in learning a programming language, particularly one like Ada since the philosophy on the use of Ada differs substantially from most other general purpose programming languages. Hence, we find that our approach in which CAI is used to "add value" to training is efficient and cost effective.

IMPLEMENTATION OF AN ADA* REAL-TIME EXECUTIVE - A CASE STUDY

James D. Laird
Dr. Bruce A. Burton
Mary R. Koppes

Intermetrics, Inc.
Aerospace Systems Group
5312 Bolsa Avenue
Huntington Beach, California 92649

ABSTRACT

Current Ada language implementations and runtime environments are immature, unproven and are a key risk area for real-time embedded computer systems (ECS). This study provides a test-case environment in which the concerns of the real-time, ECS community are addressed. A priority driven executive is selected to be implemented in the Ada programming language. The model selected is representative of real-time executives tailored for embedded systems used in missile, spacecraft, and avionics applications. An Ada-based design methodology is utilized, and two designs are considered. The first of these designs requires the use of vendor supplied runtime and tasking support. An alternative high-level design is also considered for an implementation requiring no vendor supplied runtime or tasking support. The former approach is carried through to implementation.

INTRODUCTION

Since the inception of the common DoD High Order Language (HOL) effort in the mid-70's, the Ada programming language has remained a cornerstone of the government effort at producing software in a cost-effective manner. Validated Ada compilers are becoming available on a variety of different computers with at least 17 validated compilers now available and more slated for validation during the current year. There are currently 37 different defense programs using Ada, and this number is anticipated to exceed 120 during the next four years[1]. While this progress is encouraging, the success of the Ada language in meeting the needs of specific applications will hinge on the consideration of the potential risks that face the implementors of a given system. This process of risk identification should be followed by development of risk minimization and avoidance strategies tailored to meet the needs of the system. The emphasis of this paper is in the area of technical risk identification and resolution for real-time ECS applications. While the Ada programming language is intended for real-time applications, current compilers and runtime systems are unproven for these types of programming efforts. Consequently, the impact and implications of using the Ada language and Ada-oriented methodologies in embedded real-time development efforts should be assessed. While it is necessary to examine how well and to what extent the built-in real-time features of the language meet the needs of ECS applications, additionally, we must re-evaluate the standard approaches to solving real-time problems in light of the new capabilities and assess the impact, if any, on the way we design and implement these solutions in software.

SCOPE

Perhaps the major consideration with regard to the use of the Ada programming language for real-time ECS applications is the cost of doing so in terms of memory and processing overhead. The relative costs associated with the use of Ada and its real-time features is especially relevant to small embedded computer system applications given the physical and temporal constraints imposed on these types of applications. The determining factor in the decision to utilize a particular high order language (HOL) feature is often the efficiency of its implementation. It is important to know what the utilization of Ada with its real-time tasking primitives, representation specifications, exception handling, and various other features translates to in terms of program size, speed, and efficiency. The ability to selectively include runtime support and

* Ada is a Registered Trademark of the U.S. Government (AJPO)

its resultant overhead for these features on an "as needed" basis is another important consideration. During the course of this investigation, answers to fundamental questions such as these were sought. In addition, other issues specific to real-time ECS applications were examined and addressed as they naturally developed within the context of the implementation of the case study executive. In addition to runtime support issues, Ada specific solutions and strategies were sought and implemented with regard to issues such as shared data, adaptability to hardware, effective deadlock, task management, maximization of concurrency, and reliability.

BACKGROUND

It is important to stress the significant conceptual differences between the two approaches investigated with regard to this case study implementation of a priority driven Ada executive. Figure 1 serves to illustrate the alternative approaches and concepts and their implications for the developer of an Ada executive.

The terms O.S., executive, and runtime support or system (RTS) are often used rather loosely when ECS topics are discussed. The ambiguity of this terminology in the ECS environment is primarily due to the overlap in functionality provided by different implementations for different applications. An application residing on a bare machine may interface with software providing minimal scheduling and memory management. This software is often referred to as an "executive" or runtime kernel whereas the same services provided on another system may be obtained from software referred to as an O.S. The primary difference in terminology is attributable to the variety and nature of the services provided by the support software in question. The more minimal the services provided, the more likely that the terms runtime support, runtime kernel, or executive will be applied. True operating systems in the strict sense are distinguished by two major factors. They are typically developed independently of any compiler/applications software and are acquired independently rather than as a part of a given compiler system or package. The other major distinction is in the comprehensiveness of the services provided by an O.S. for the target machine; services that may be targeted and utilized by a variety of differing applications and tools as well as many different compiler systems. The minimal runtime support for applications developed under a single

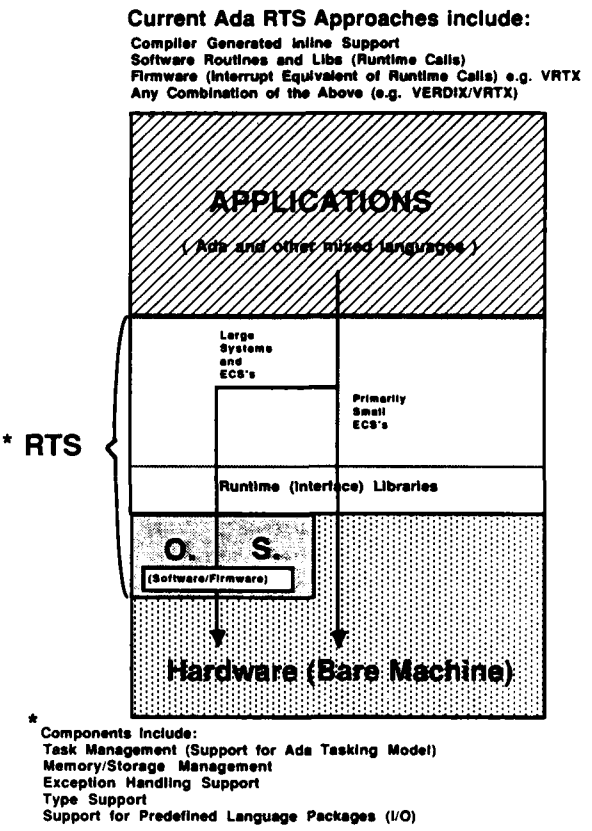


FIGURE 1.
RUNTIME SUPPORT (RTS) APPROACHES

compiler system may interface to, and utilize, the comprehensive services provided by an O.S. Therefore, the RTS for an ECS can be thought of as providing the minimal required subset of O.S. services needed for a given application. As stated, this minimal subset can be provided by direct access to the underlying machine or through the utilization of the services provided by an underlying comprehensive O.S. The former case is the most typical for embedded computer systems. The term "executive" is most often used to refer to that part of the RTS that performs the basic scheduling and memory management. Other portions of the RTS may include I/O control, timer/clock management, and a certain amount of systems level runtime error and interrupt trapping.

The RTS or runtime environment of an ECS is the combination of hardware and software that supports the execution of application programs and the programming language features utilized to develop those programs. As illustrated in Figure

1, this support can be implemented in hardware, microcode, through direct calls to an O.S., through the use of runtime support libraries, or by compiler generated (in-line) code. The operating system and RTS needs of small embedded computer systems are typically modest. All that such small ECS targets usually require is an "executive" consisting of little more than a basic scheduler, memory manager and some type of I/O manager or controller. Obviously, different applications may have specific needs relative to memory management, I/O, or clock services which will be reflected in the "executive/O.S" software.

APPROACH

This paper addresses two basic options or approaches to the implementation of an Ada executive and briefly discusses ongoing as well as proposed work in a third area of related investigation. The first of these approaches is explored in depth (through to implementation) and consists of a combination of a "pseudo executive" or scheduler at the applications layer in concert with vendor supplied executive software at the runtime system level. The obvious benefits of such an approach - imposing an additional layer of control upon the runtime system scheduling mechanism - include ease of portability, and relative target independence with respect to the underlying scheduling algorithm at the RTS layer. These benefits as well as the tradeoffs in overhead and consistency from implementation to implementation will be discussed in detail. Another option is explored at a high level only. This alternative, termed the bare machine approach, is consistent with the traditional approach to avionics-based executives and is considerably more limited in scope than the first in the sense that it assumes no underlying vendor supplied runtime support. This executive performs all necessary support for the execution of user jobs or "tasks". However, this approach is significantly more restrictive than the first with respect to the nature of what constitutes a "task" as well as to the use of certain Ada language features involving both the Ada tasking model and dynamic memory management and certain other real-time aspects of the language. The final option is considered only in terms of current and ongoing investigative work and proposed future studies based upon the results of past investigations. This approach diverges from the others in that it proposes a complete migration to the runtime system layer in order to probe the issues of efficiency and risk reduction for real-time Ada applications. This option emphasizes the tailoring and optimization of the executive functions provided at the RTS layer.

A multi-phased approach beginning with a requirements specification was utilized for the design and development of the priority driven executive. The functional capabilities that were to be provided were extracted from an existing avionics executive implemented in a combination of FORTRAN and Assembly language. It was determined that these same functional capabilities would be provided within the executive being implemented in the Ada language.

Figure 2 is a schematic representation outlining the functional interaction of the major components of the FORTRAN/Assembly model utilized. This representative model of a priority driven executive exerted its control over the user task states through the creation and management of task control blocks. These contained fields of information specific to each user task transition between the active and inactive state. The type of scheduling mechanism utilized within this system was a non-preemptive, voluntary context switching algorithm. Ultimately, the actively executing user tasks were responsible for initiating the scheduling of other user tasks and, in many cases, themselves. This was accomplished through explicit calls to the scheduling primitives provided by the executive and consisted of both time and signal dependent scheduling. Based upon the

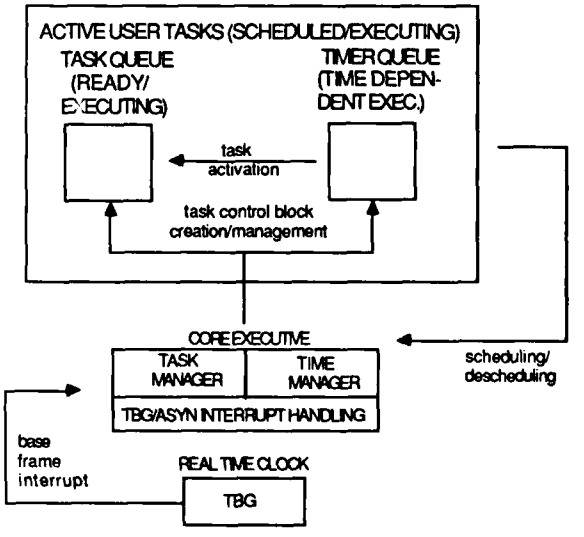


FIGURE 2.
FORTRAN/ASSEMBLY EXECUTIVE FUNCTIONAL SCHEMATIC

receipt of either a time-based or signal-based event, the executive managed the task state transitions between inactive, ready for execution, and executing.

This model was selected primarily for its representative features as a real-time, multi-tasking, priority driven avionics executive and for its relative small scale. Using this model, an Ada equivalent was developed to provide as much, if not the same, functionality available in the FORTRAN/Assembly language implementation. While equivalent in functionality provided, the Ada equivalent constituted a complete re-design utilizing Ada concepts and features where possible. For this reason, the Ada executive posed some unique problems from the outset with respect to use of the new Ada concepts and features such as the Ada tasking model. These issues will be addressed in some detail in the RESULTS section of this paper.

The Ada priority driven executive was to provide facilities for the creation of active tasks via a scheduling mechanism. The scheduling mechanism would provide time-dependent scheduling capabilities, precision timing of task activation as measured by time base generated (TBG) epochs, and signal dependent scheduling capabilities. The Ada priority driven executive would perform prioritized tasking and would have the option of enabling and disabling interrupts. The capability to directly connect to a real-time clock interrupt would be provided. In the absence of such a facility, the real-time clock interrupt would be simulated with the smallest granularity possible. In short, the Ada priority driven executive was required to be a real-time, multi-tasking process manager with interrupt handling and both cyclic and asynchronous scheduling capability.

Integral to the design of the Ada priority driven executive was the selection and application of a state-of-the-art, Ada-based design methodology. A somewhat novel design approach was selected that was based upon Object Oriented Design[2] with enhancements and modifications specific for real-time embedded systems[4]. The methodology derived was termed Real-Time Object Oriented Design (RTOOD) and drew upon another real-time, systems-based design methodology called Design Approach for Real-Time Systems (DARTS)[5]. The steps utilized in this hybrid methodology are outlined in Figure 3.

Using Object Oriented Design constructs often referred to as

I. Definition/statement of the problem

II. Informal strategy (Modified specification)

III. Identify objects and attributes

IV. Identify Operations

V. Identify concurrency *(DARTS)

Decomposition into tasks/packages based on:

The asychonous nature or major transforms
-- sequential vs. concurrent --

specifically:

- I/o dependency
- time critical functions
- computational requirements
- function cohesion
- temporal cohesion
- periodic execution

VI. Establish the interfaces

VII. Implement the operations

* (DARTS)
Design Approach for
Real-Time Systems

Figure 3.
REAL-TIME OBJECT ORIENTED DESIGN (RTOOD) METHODOLOGY

Booch-o-grams[6], a high level schematic depicting the major Ada program units required for implementation of the Ada priority driven executive is presented in Figure 4. This high-level overview establishes the necessary relationships between the major components of the system in terms of visibility among program units as well as defining the interfaces through which they communicate. The design arrived at and presented here is a top-level abstraction only. It was necessary to iteratively apply the design methodology from the highest level of abstraction downward to arrive at a complete definition of the required components.

Similarly, a high level design was developed for the alternate approach - termed here "the bare machine approach" - to the development of an Ada executive. This high-level abstraction is shown in Figure 5 and represents the more restrictive traditional approach to implementation of a bare machine executive. The model represented implements its own concurrency through the executive while disallowing the use of the Ada tasking model per se as well as any difficult, and potentially risk-prone, dynamic storage management. The potential benefits and risks of each of these approaches was examined with the former approach being carried through to implementation and limited utilization.

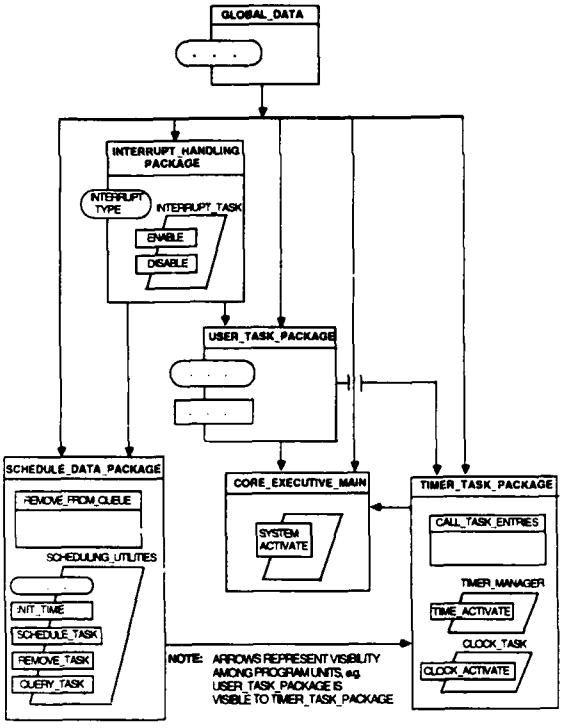


FIGURE 4.
HIGH LEVEL RTOOD SCHEMATIC
EXECUTIVE WITH UNDERLYING RTS

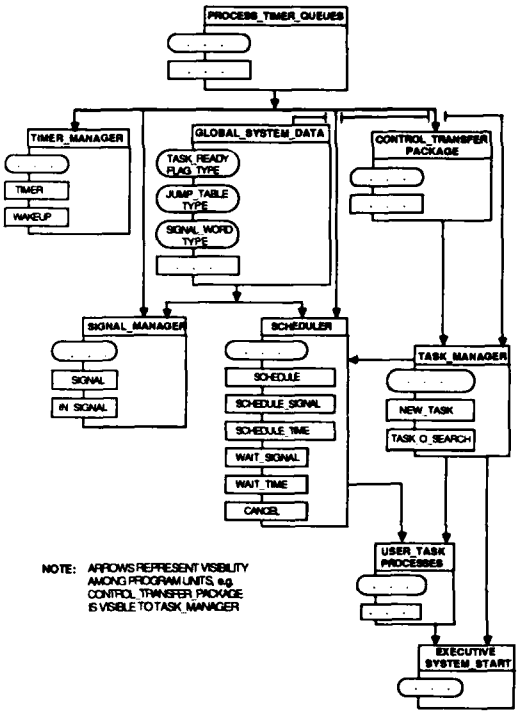


FIGURE 5.
HIGH LEVEL RTOOD SCHEMATIC
"BARE MACHINE APPROACH"

An important function of this case study was to provide an evaluation of the tools and methodologies that would be utilized in an actual project consisting of the specification, design, and implementation of a real-time application. The Byron™* program development language and toolset developed by Intermetrics was utilized as a component of the overall design methodology. The Byron Program Design Language (PDL) is based on the Ada language and provides a mechanism to associate textual information with Ada language constructs. The Byron toolset facilitates program design and development through the provision of documentation support and analysis tools.

The approach to testing and analysis of the executive under development considered several issues. Functionality as well as program sizing and overhead were considered. An examination was made of various runtime system parameters such as task scheduling, interleaving, and prioritization.

RESULTS

I. ADA EXECUTIVE WITH VENDOR RUNTIME SUPPORT

The capabilities of the FORTRAN/Assembly language implementation and the Ada language implementation are summarized in Table 1. The Ada language version consists of two major components - the program code and the vendor supplied runtime system. In both implementations the scheduling primitives are provided by the executive, but the ultimate responsibility for cyclic/acyclic task scheduling lies with the user (application) tasks. Note, however, that the task interleaving and task waiting in the Ada language version is strictly under the control of the Ada runtime system and not under the control of the executive as in the FORTRAN/Assembly implementation.

* Byron™ is a Registered Trademark of Intermetrics, Inc.

TABLE 1. FORTRAN/ASSEMBLY VERSUS Ada IMPLEMENTATION			
CAPABILITY	FORTRAN/ASSEMBLY EXECUTIVE	Ada EXECUTIVE	Ada RUNTIME SYSTEM
CYCLIC/ACYCLIC TASK SCHEDULING	Provided	Provided	
TASK DE-SCHEDULING	Provided	Provided	
TASK INTERLEAVING	Provided		Provided
TASK WAIT	Provided		Provided
PRIORITIZED TASKING	Provided	Provided	Provided
TBG INTERRUPT HANDLING	Provided	Provided	

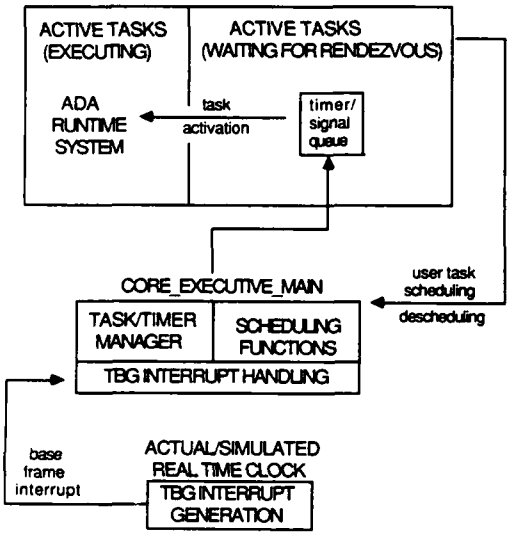


FIGURE 6.
ADA PRIORITY DRIVEN EXECUTIVE
FUNCTIONAL SCHEMATIC

Furthermore, although tasking could be prioritized dynamically (changed) in the FORTRAN/Assembly implementation, priorities at the runtime system level are static in the Ada language version.

Functional Summary Figure 6 depicts the major functional components of the Ada equivalent prototype developed for the case study investigation. The major distinction between the Ada implementation and the FORTRAN/Assembly model depicted previously in Figure 2 involves the interaction of the Ada runtime system with the priority driven executive functions. While the FORTRAN/Assembly model managed all state transitions for user tasks from inactive to executing and all information associated with these state transitions, the Ada implementation utilizes the Ada runtime support system (for the tasking model) to manage the active processing phase of any user task as well as the body of information associated with a tasks' active execution. Specifically, the Ada runtime system manages the interleaving or time-slicing of concurrently executing user tasks and is responsible for management of the associated task activation information. The start of a user tasks' scheduled execution phase is strictly under the control of the Ada priority driven executive at the applications layer, yet, the management of the transfer of control between any number of concurrently executing user tasks is by definition under the control of the vendor supplied Ada runtime system. In addition, the Ada language specification dictates the enforcement of critical regions (non-interruptable sections of code) with respect to the acceptance of task entry requests and subsequent processing. The enforcement of these critical regions in conjunction with the priority-based scheduling through the Ada executive imposes upon the Ada runtime system an additional level of control via a predetermined algorithm for transfer of control (interleaving) among concurrently executing program units.

To satisfy the requirement for a cyclic capability, the executive was required to have some method for specifying fixed-rate scheduling. This was provided on two levels. In keeping with the scheme utilized in the original model, the facility for scheduling a task for execution is provided. Active tasks currently executing may therefore utilize this facility to re-insert themselves into the schedule for future execution, or this may be done by some other active user task. This requires some hypothetical scheduling scheme among the user tasks. In the original model a voluntary, non pre-emptive scheduling scheme was utilized among the user tasks that enforced the notion that no transfer of control or context switching among tasks could occur unexpectedly. Bearing in mind that within an Ada environment the underlying operating or runtime system utilizes another level of scheduling for the interleaving of currently active tasks, a task prioritization scheme among these tasks is then required to enforce the notion that a particular task is incapable of having its scheduled execution interrupted once it begins. In short, we have a scheduling scheme at the user task level to specify fixed-rate triggering of a tasks' processing and the Ada pragma "PRIORITY" enforced at the underlying operating or runtime system level to ensure uninterrupted completion of that processing.

The major potential point of failure with respect to this type of approach to

task scheduling at the applications level is at the underlying runtime system level. The issue is one of consistency from implementation to implementation with respect to time slicing of concurrently executing processes of equal priority. While fixed rate triggering of task execution can be guaranteed via a combination of algorithmic control, prioritization, and interrupt handling through the "psuedo executive", no such guarantee can be made with respect to the method of time slicing utilized by the underlying runtime support for concurrent tasks of equal priority. This will vary from implementation to implementation although adhering to the so-called "FAIR" requirement dictated by the language specification. Given the stringent nature of typical ECS performance and reliability requirements, this potential inconsistent behavior across implementations could pose a significant risk.

Static prioritization of Ada tasks may be a problem in some instances of task scheduling or interrupt handling since external events often dictate a need to dynamically change priorities. The Ada rendezvous occurs in a first in, first out manner using a queue structure for multiple entry calls issued for any given task entry point (ACCEPT statement). There is no way to reorder and influence the position a calling task may occupy in such a queue. It is possible that with dynamic task prioritization this could be programmer controlled although it is not clear whether task prioritization is used for the queueing order of simultaneous entry calls in a given implementation. Presently, if it is desired to hold or halt the acceptance of an entry call that has been issued and is queued, there is no recourse for doing so (cleanly) aside from termination through the ABORT mechanism and re-invocation.

Efficiency: Space and Time The FORTRAN/ Assembly language implementation used as a model in this case study was coded in a little over 1 K (bytes) of memory and accounted for somewhat less than two percent of the entire system. While the entire Ada system consisted of just over 700 lines of code, the space requirements varied with respect to the host machine. The Ada version required anywhere from 27 K to 38 K bytes of memory for the applications code alone. The runtime kernel on one machine imposed an additional penalty of 200 K bytes to utilize the Ada tasking model. It should be noted, however, that the executive was developed for functional realism and was not optimized for minimal program size. The runtime kernels were large, as much as 200K bytes, but the runtime kernels were intended for a main-frame environment, not a typical ECS application. The significant lessons learned were in what options were available to optimize the size and speed of the executable image. Significant savings - approximately 100K - were available via a selectively loadable tasking kernel in at least one implementation while other options resulting in savings were no runtime checking (1-2K savings), and no debugging instrumentation (5K savings). In one particular implementation, the option for space optimization was offered yet yielded no appreciable difference in the size of the executable image. While there is no strict linear relationship with respect to overhead between host and ECS environments, the significant savings realized through configurability within the host environments has significant positive implications for ECS environments where efficiency constraints are paramount.

The granularity of clock services available was insufficient to perform significant timing analysis at the time of the case study investigation. However, parallel investigations within the same environments at a later date revealed data significant to the type of real-time processing utilized within the prototype executive. It was found that the total storage penalty to include a minimal exception handling capability within each Ada program unit was on the order of 4-5 percent of the total program storage while the cpu overhead to invoke an exception handler ranged from 30-500 microseconds. This represents an acceptable cost in either a host mainframe or embedded environment.

The overhead in terms of time to utilize the rendezvous mechanism within the host environment was rather high, being approximately 11-12 milliseconds. Given the relatively rapid frame times of many real-time applications (on the order of 40-100 milliseconds), a feature that uses approximately one tenth of the frame time poses serious risk[3]. However, based upon current investigations with Ada for embedded 16 and 32 bit targets, the case can be made that this is a problem somewhat localized to the mainframe environment.

Maintaining Concurrency While there are a number of methods available to trigger the processing of a task, each has a relative cost in terms of efficiency. Many of these methods can be very expensive in terms of memory utilization and rely heavily on efficient garbage collection by the underlying

runtime system to ensure adequate storage availability. The danger of exceeding maximum storage capacity is always present and, therefore, many of the available methods are entirely unsuitable for small to moderate size machines.

One acceptable technique is to make task processing dependent upon a rendezvous that is placed inside a loop. The processing is triggered by the rendezvous acceptance. When the processing is complete, the task merely loops back to the accept statement and goes to sleep until a subsequent entry call is made. This assumes that an unmet rendezvous constitutes a sleep state. The assumption is that it should be a sleep state to prevent waste of valuable CPU time, but this may be implementation dependent. There is an additional problem with this approach. The Ada rendezvous mechanism is defined in such a way that a calling program unit is suspended until the processing that is contained within the ACCEPT-DO END block of the task being rendezvoused with is completed[7]. This is comparable to the synchronous behavior of a procedure call. The acceptable solution to this problem is to place any processing in a called task after an empty ACCEPT-DO END block (the DO/END would be optional). By placing the processing segment following the point of rendezvous, and not within it, the task that issued the entry call can rendezvous quickly and continue executing, concurrently, with any processing that has been triggered in the called task. This was the method utilized successfully in the case study.

Resource Contention Resource contention among user tasks with regard to the scheduling facilities was detected in the design phase as well as at the coding stage. In an Ada tasking environment, contention for shared data and processing resources seems to dictate a tasking solution in the form of a monitor task that encapsulates the resource in question. The use of flags or a semaphore system has the inherent danger of collision on the flag and does not seem satisfactory in a heavily task oriented system. The tasking approach therefore seems to be the most satisfactory solution.

A problem was encountered when the executive tasks prioritization levels were equal to or lower than the least urgent user task. This introduced the problem of a highly active user task locking out the executive tasks and affecting the stability of the timing cycle. A satisfactory solution to this problem is to make the executive tasks' prioritization levels the highest in the system and to introduce enough delay to allow the user tasks sufficient processing time.

II. THE BARE MACHINE APPROACH

The alternate design approach proposed in this study for the Ada priority driven executive (see Figure 5) is intended for a bare machine environment with no resident operating system nor any vendor supplied Ada runtime support. The design of such an executive raises some important issues with respect to what must be provided to support the execution of an Ada application on such a bare target. When operating within such an environment, the implications of the traditional model of an executive, such as the original FORTRAN/Assembly language implementation used as a basis for this study, must be considered. This approach differs greatly from that which utilizes an underlying runtime system. This approach implies that beyond the generation of native machine instructions from the HOL by some generic translator or compiler, it becomes necessary to provide programmer supplied support for any HOL language features not directly implementable through primitives on the bare hardware. It therefore becomes the task of the runtime supervisor or executive software to provide this underlying support for things such as concurrency or multi-tasking, I/O, dynamic storage and memory management to name a few. In addition, this executive must *not*, in turn, rely on some underlying support for its own execution.

The design of this executive was purely hypothetical and no specific embedded target was selected. For that reason, only a high-level design was iterated. Currently, typical vendor supplied Ada runtime support packages facilitate things such as: system elaboration or initialization, task communication and scheduling, exception handling, interrupt, I/O, and type support. The amount of overhead varies with each vendor's implementation. The design proposed here is for an Ada executive function that would minimally support the execution of other Ada software constituting jobs or "tasks". However, the Ada tasking model is not supported by the proposed subset Ada implementation for a bare ECS target. As in the traditional model, concurrency is achieved via the executive utilizing a non pre-emptive, voluntary context switching mechanism. Control over scheduling is therefore explicit and known to the programmer. In addition, any dynamic data or storage management is restricted to that which supports the execution of the executive functions only. It must be noted that the notion of an "all Ada executive" at

this level is fallacious. A certain amount of privileged accessing of register and stack contents by the executive functions to facilitate the basic context switching and memory management would be required. This is not directly achievable from within the Ada language. Therefore, a component of the executive software (e.g. the Control_Transfer_Package) would by necessity be implemented in a lower level programming language. In current commercial Ada runtime systems for embedded targets such as the 1750A, this accounts for approximately two percent of the supplied runtime support. Ada packaging concepts facilitate the encapsulation and isolation of such machine context sensitive components.

The rationale for the approach to concurrency presented is straightforward. While explicit context switching can be considered risky, it has certain potential benefits. It avoids the necessity of excessive locking since the programmer knows exactly when context switches are to be performed. Another benefit is realized when a high priority event occurs that must be handled rapidly as is the case in many real-time systems. While handling such an event, it may be deleterious to release the processor. Finally, the avoidance of unnecessary context switches and/or checking results in greater efficiency[8]. Admittedly, however, it becomes necessary to question the feasibility and advantages of using Ada without its tasking features and other real-time components versus using any other high-level programming language. It should also be noted that, with some re-working of the design, there is nothing to explicitly prevent the use of the Ada tasking model and rendezvous concept, provided that the necessary runtime support is supplied at an acceptable cost in memory overhead and execution efficiency. This is the motivating concept driving our current and future investigations with respect to Ada real-time systems and will be discussed in the following section.

Current and Future Investigations The rationale for an approach such as the bare machine option is that given the present state of tasking support in an environment that supports full Ada tasking, exception handling and other HOL features, the resultant program size may be unsuitably large for an embedded application. While the bare machine approach represents one available option, an additional alternative exists that holds some promise for the design of compact, efficient real-time systems and is the focus of our current and future investigative work. This consists of a migration to the RTS layer in pursuit of optimization and risk reduction at this level while maintaining the complete (or nearly complete) functionality of the language. The focus is on tailorable, configurable runtime support for the design of efficient real-time systems in Ada. It is highly likely that the full functionality of the traditional model of a priority driven executive can be achieved in this manner by minimizing the role of a programmer supplied executive and relying on the efficient implementation of the Ada tasking model at the operating or runtime system level. While it may still be necessary to provide customized runtime/executive support, this can be provided primarily through tailoring of existing systems at the RTS level to meet specific performance requirements rather than exerting additional control at the applications layer.

CONCLUSION

Many issues of concern exist due to the immaturity and quality of Ada language implementations and uncertainties regarding performance. The performance of the code generated by early compilers may be poor and may result in poor system performance. In addition, although several of the issues that face developers of real-time ECS applications in Ada are design issues or primarily resolved through good programming technique, many issues remain that pose risk to the development of real-time systems in Ada. Current design techniques for real-time applications in Ada are inadequate, and some issues can be resolved by development of a comprehensive design method for real-time systems. Furthermore, the development of programming techniques or strategies, and the education of programmers can aid the elimination or minimization of many concerns.

The unique constraints imposed upon real-time embedded computer applications often require that specific solutions and strategies be utilized. Implementation languages, in turn, must be sufficiently flexible and powerful to accommodate these solutions in the most efficient manner possible. We have identified a number of key risk areas and issues for real-time ECS applications and have explored these issues, and solutions, within the context of a specific Ada language application. With respect to the issues that were successfully addressed within the scope of this case study, the following conclusions can be made.

Current runtime support required for implementation of the Ada tasking model is generally high in memory

utilization and execution overhead. However, as Ada language systems mature and currently available optimizing technology is employed, large runtime overhead with respect to memory utilization and execution speed should certainly become less of an issue. This is in fact the case with some of the Ada language systems currently under development. Current investigations with a variety of differing compiler systems and runtime environments for 16 and 32 bit embedded targets have revealed that kernel runtime systems currently exist that appear to be providing the minimal, configurable support necessary to accommodate Ada language features in a timely and efficient manner. Standardized kernel runtime support on the order of 2K provided by minimal system service interfaces is currently available (e.g. VRTX) and can be targeted and utilized efficiently by Ada compiler systems for a variety of embedded targets. In addition, preliminary analysis and timing studies with Ada language systems for embedded targets such as the AIE 1750A cross compilation system have indicated that the basic language features are being implemented in an efficient manner. Basic context switching times on the order of twenty microseconds and general code expansion ratios on the order of four to six are encouraging for the development of compact, efficient real-time applications in Ada. In addition, hardware architectures optimized to execute Ada code and that implement Ada language features in hardware at runtime are being developed.

Problems remain with the non-support among many Ada implementations of certain real-time features of the Ada language. A case in point is the vectoring of interrupts to task entries via the Ada representation specification. This continues to be a concern to the real-time applications community although it is somewhat localized to the mainframe environment. Additional problems are rooted in the language specification itself (MIL STD 1815A) which fails to provide certain features desirable in typical real-time systems. While alternatives exist, this lack of certain explicit language primitives poses unique problems for many types of real-time applications. Specifically, the lack of explicit language primitives to allow dynamic "disconnection" and "connection" to interrupts without the termination or creation of a program unit (task) and the inability to utilize dynamic task prioritization are of major concern to ECS developers. Furthermore, the lack of precision in the specification of exact delays as well as the lack of alternatives or ability to time-out during initiated rendezvous' is an impediment to the development of efficient, reliable real-time systems in Ada.

There is a continuing need for a clear, concise design methodology for real-time embedded Ada applications that includes a criteria for the identification of concurrency and a graphic means of depicting concurrent relationships with timing and synchronization information at any given point in the system. While helpful, the hybrid method utilized during this case study falls short of fulfilling such a broad requirement.

The difficulties encountered during the course of the case study investigation in assessing Ada real-time features in a host machine environment were significant and underscored the need for further study of the problems and issues encountered in real-time applications. We are currently continuing our real-time investigations to evaluate the effectiveness of Ada language systems for real-time embedded applications within realistic host and target environments. This work is being carried out with a focus on the 1750A and 68000 compiler and runtime environments. A comparative analysis of runtime characteristics and performance among various Ada compiler systems and sample runtime environments is ongoing. We are also identifying and utilizing state of the art real-time Ada evaluation, test, and simulation tools in the effort to analyze the performance characteristics of Ada applications in realistic target environments. The focus of our initial case study was at the applications level although an alternative was proposed for a prohibitively restrictive Ada executive that fulfilled a subset of the runtime responsibilities to support the execution of concurrent Ada programs. The current approach calls for migration to the RTS level to investigate optimization and tailoring of existing systems to allow efficient use of the Ada tasking model and other real-time features within realistic target environments. It is in this manner that we will attempt to address and seek additional information and solutions to those issues left unanswered in our preliminary Ada real-time investigations.

ACKNOWLEDGMENTS

The authors wish to acknowledge the support and advice of the personnel at Intermetrics, Inc. in the preparation of this manuscript.

REFERENCES

1. Judge, J.F., "Ada Progress Satisfies DoD", Defense Electronics, June 1985.

2. Booch, Grady, Software Engineering with Ada, Benjamin/Cummings, Menlo Park, California, 1983.

3. Davis, R., "FDA Program Conclusions", Intermetrics Inc., Huntington Beach, California, August, 1985.

4. Laird, James D., "Implementation of an Ada Real-Time Executive: A Detailed Analysis", Intermetrics Inc., Huntington Beach, California, March, 1985.

5. Gomaa, H., "A Software Design Method for Real-Time Systems", Communications of the ACM, Vol. 27, No. 9, September 1984.

6. Temte, Mark, "Object Oriented Design and Ballistics Software", ACM Ada Letters, Vol. IV, No. 3, November/December, 1984.

7. United States Department of Defense, Reference Manual for the Ada Programming Language MIL STD 1815A, Ada Joint Program Office, March, 1983.

8. Binding, Carl, "Cheap Concurrency in C", ACM SIGPLAN NOTICES, Vol. 20, No. 9, September 1985.

BIOGRAPHIES

James Laird is a member of the software engineering staff for the Software Technology Department at Intermetrics' California Division. He has been active in the Ada IR&D investigations for the NASA Space Station Pre-Award and Phase B activities under the Space Systems Software Group. He previously worked in the area of software simulation and testing for the Rockwell B1-B project. His interest areas include real-time programming with Ada and software simulation. He is a member of ACM and IEEE Computer Society

Bruce Burton is the manager of the Software Technology Department at Intermetrics, Inc. He holds an M.S. in Information and Computer Science and a Ph.D in Physical Chemistry from the University of California, Irvine. Dr. Burton is interested in the real-time programming area and in the field of software reuse.

Mary Koppes is a Software Engineer for the California Division of Intermetrics, Inc. and is currently working in the Space Systems Software Group on the Space Station Phase B Definition and Preliminary Design contract. Ms. Koppes has also contributed to the Ada research and development work. Ms. Koppes received her B.A. in Mathematics/Computer Science from the California State University, Long Beach.

PRACTICAL EXPERIENCES OF THE ADA LANGUAGE FOR REAL-TIME EMBEDDED SYSTEMS DEVELOPMENT FOR THE DEFENCE-RELATED MARKET

Dr Mel Selwood

Plessey-UK Limited, England

Summary

This paper describes some of the experiences gained to-date from an Ada research programme, undertaken within the Plessey Company in the U.K., by the author and his team. This programme is investigating the cost-effective and beneficial introduction of the Ada language for Defence-related (mostly real-time) software applications. Particular emphasis is placed upon minimising the risks and maximising the benefits for large and / or embedded microprocessor-based systems. Within the context of this largely practical work programme, the paper identifies a number of key concerns within the team (and it is suggested within industry at large) in making the transition to Ada. Also, some suggestions for improving the application of current Ada compilers and tools is provided to the vendors of these products.

Introduction

Clearly, in order to obtain the longer term benefits claimed for the Ada language, it is first necessary for any commercial organisation to be able to bid for and implement Ada projects both profitably and at minimum risk. This can only be done with confidence (at least for fixed-price contracts) when that organisation has already implemented "representative" Ada projects, so that it can draw upon real experience of the technical issues involved. Only then can it, for example, reliably establish criteria for estimating project timescales and costs, and define appropriate standards and procedures for Ada-based developments.

In general, for large and complex (especially real-time embedded microprocessor based) systems, for which Ada is intended, the risk of using Ada ahead of gaining such real experience may well be too great. Further, although the introduction of Ada may go hand-in-hand with improved software engineering practices, it would be inappropriate to ignore the often large existing investment in non-Ada software and systems products (and the associated manpower skills, working practices, equipment etc.) in favour of some totally new, unproven development scenario.

Instead, the situation demands an optimum (transition) solution for which on the one-hand changes to the status quo are minimised, while on the other hand the potential benefits of Ada and associated development methods are maximised.

This paper reports on some of the practical experiences gained to-date from an Ada research programme which was set up to address this problem.

Ada Research Programme

The programme was set up as a 'virtual' project, involving a hybrid mix of practical study activities and real Ada software developments. This project is ongoing and involves multiple study teams with distributed interests, thereby ensuring a broad approach to the problem.

Given that the coding phase of projects typically equates to only 10% of the overall development effort, it is clear that the real value of Ada will come not from the direct characteristics of the language itself but from the catalytic (secondary) influence upon the software engineering methods employed. Thus, the following issues are being examined:

i. the appropriate time-frame for introducing Ada,
ii. the impact of Ada upon the overall development lifecycle,
iii. the necessary standards and procedures (Project Management Baseline),
iv. the anticipated effort / cost - time profile for Ada projects,
v. constraints upon the design and implementation of products, and their expected characteristics,
vi. operational aspects (eg appropriate development environment).

Within this context, the programme focusses upon the production of Ada demonstrators for gaining practical experience of Ada, to demonstrate representative Ada-based software systems actually working, and to allow ancillary studies eg into code size and performance issues. The work is complemented by a series of across-application studies to check the consistency of the results and their applicability to varied applications.

Three demonstrators are currently being worked upon:

A Digital Telephone Exchange Demonstrator

This system implements the complex multi-tasking activities associated with setting up, supporting, and terminating, one or more concurrent two-party telephone calls from *Digital Voice Terminals* (DVTs). The system supports a variety of facilities eg system configuration, abbreviated dialling, diversion of calls, call pre-emption etc.

The system is a conversion of an existing product which was developed using the "Modular Approach to Software Construction, Operation and Test (Mascot)", and implemented using the Coral 66 language. The majority of the new system preserves the existing Mascot design (excluding the Mascot 'machine' concept, which supports primitives such as operations on control queues for synchronising access to shared data areas). The required system functions were then manually re-implemented using the full features of the Ada language.

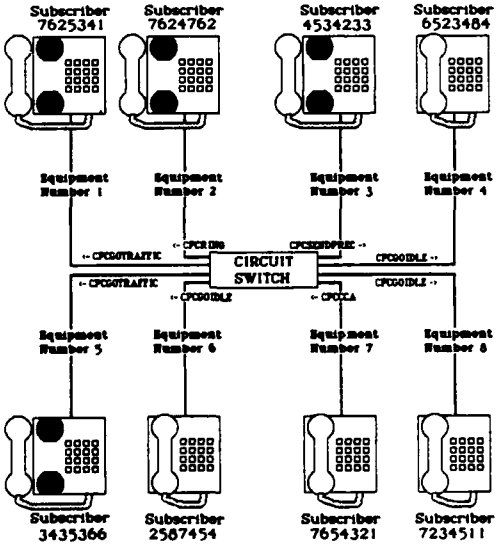


Figure 1 - Typical Screen Snapshot

In contrast, the man-machine interface (MMI) sub-system is both newly designed and implemented and this serves as a test case for studies into Ada design methods.

The demonstrator involves three platforms:

i. an entirely DEC Vax-based Ada system with software emulating the function of the real (DVT) hardware - a typical screen snapshot is shown in figure 1.
ii. the Vax-based Ada system linked via an RS232 interface to an existing hardware rig supporting real DVTs,
iii. the Ada software at ii. re-ported to run on an Intel 80286-based target.

A Sonar System demonstrator

This system is based upon an existing Pascal implementation and demonstrates a typical naval surface ship sonar data processing and colour display function. It presents both Active and Broadband Passive sonar data in grey-scale format, updated in real-time, together with automatic target detection and tracking functions. Operator interaction is via 'soft' keyboards presented on the sonar data displays, and manipulated using a special purpose five-button keypad. A sonar cursor is also provided, controlled from a rolling ball.

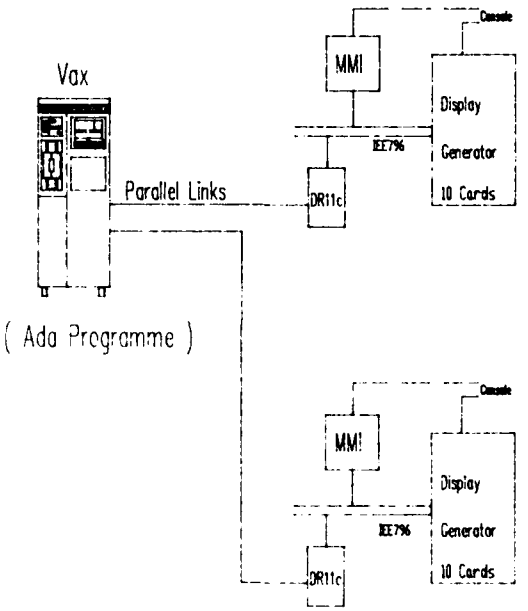


Figure 2 - Platform 1 Sonar Demonstrator

There are two platforms (figures 2 and 3). In the first the Ada software runs in a Vax which is directly coupled via twin parallel interfaces to two IEE796 microprocessor busses, on which are situated multi-plane colour graphics display generators and man-machine interface (MMI) cards. All intelligence is deliberately removed from the IEE796 cards, and accordingly the Vax-based Ada software deals with the display and MMI hardware at the lowest level of bit manipulation.

In the second platform, the majority of the demonstrator is reconfigured for a multiple Intel 8086 microprocessor system whilst the remaining Ada control module runs in either Vax or IBM PC-AT machines.

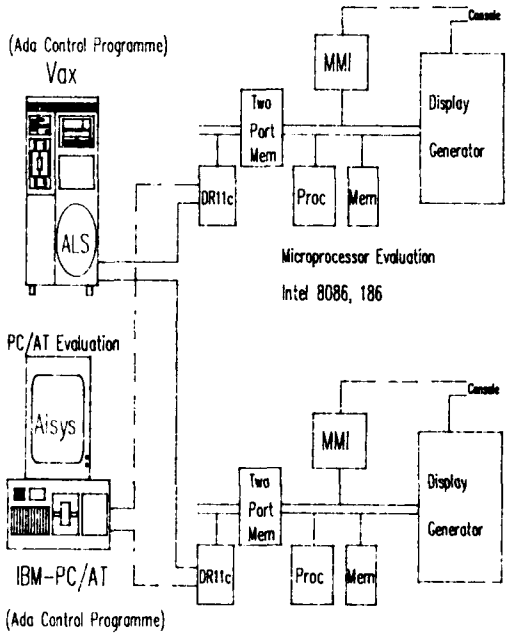


Figure 3 - Platform 2 Sonar Demonstrator

An Engine Monitoring System demonstrator

This demonstrator is based on an existing product[2] which performs a range of engine life count calculations, incident / exceedance monitoring and vibration analysis, for the Rolls Royce Pegasus engine. It provides a platform for investigating the use of Ada in high performance-critical applications; the majority of complex algorithmic calculations being carried out in real-time.

The initial investigations involve re-implementing in Ada that software which is used to assess the low cycle fatigue damage on the major components of the engine.

There are two phases to the work. The first involves an examination of the existing design and Pascal implementation, and re-implementing the system in Ada to run on a Vax. In the second stage the system is re-targetted to run in the real Motorola 68000-based hardware configuration.

RESULTS AND DISCUSSION

Digital Telephone Exchange Demonstrator

Development of this demonstrator commenced using the Telesoft Ada compiler V2.1, and later the Karlsruhe Ada compiler V1.1. However, the performance of these products was below expectation and a change was made to the newly arrived DEC Ada compiler V1.0, with which the implementation of the first (Vax-based) platform was satisfactorily completed. The second platform is still under development but is expected to be completed very shortly.

For the third platform, involving an Intel 80286 embedded microprocessor target, it is planned to use the Verdix/VADS system (running under VMS) since the DEC Ada compiler does not currently support code generation for non-DEC microprocessor targets. This work should allow comparison between the DEC and Verdix products.

The following list identifies some of the operational issues which have been raised by the demonstrator experience:

i. The security and robustness of the Ada library management facilities.
ii. The ease with which software developed using one compiler can be recompiled under another.
iii. The robustness of the compiler, and the existence of any implementation constraints.
iv. The development machine resources required.
v. The speed of compilation.
vi. The extent of interference by the operating system in the execution of the Ada software.

Thus, taking point v. as an example: for both the Telesoft and Karlsruhe Ada compilers, in the particular application used in the demonstrator exercise, the CPU time required to build the first demonstrator platform (before the system was fully coded) was in excess of two hours - the elapsed time being substantially longer. Considering only the MMI sub-system (ca. 25% of the total system), this meant that

the "(re-) build and run - analyse and debug" development iteration could be performed on average two or at most three times per normal working day (depending on the extent of re-compilation required).

When the DEC Ada compiler was used in the same application, the entire system was built in 16 minutes CPU time (typically two hours elapsed time) on a Vax-11/785 with 8 Mbytes main memory. This turn-round time compared favourably with experiences of traditional languages eg Coral, and led to much greater overall productivity.

The platform 1 software comprises some 40 packages, including 34 Ada tasks, and is implemented in approximately 25,000 lines of code (including comments). This represents a substantial working example of a complex Ada system, albeit a Vax-based implementation.

Sonar Demonstrator

The first (Vax-based) platform was implemented using both the Karlsruhe and DEC Ada compilers. In transferring to the DEC Ada compiler it was noted that successful re-compilation and build occurred without needing any code changes at all. Despite the inevitable differences in run-time characteristics, this is an optimistic sign for portability of Ada across different projects.

The first platform is implemented in approximately 8,000 lines of code (including comments). This compares with approximately 5,500 lines of Pascal code in the original implementation. However, this smaller size can be attributed to the reduced number of facilities for error recovery and reduced program robustness, rather than to any verboseness of Ada (except where enforced by the strong typing features of the language).

The speed of progression of the Sonar 'ping-front' from the bottom to the top of the display screen provides a simple measure of the net relative performance of the software. Such measurements show that, in the applications used, for single-threaded (non multi-tasking) versions of the platform, the DEC Ada-based system slightly outperforms the DEC Pascal version. An extended multi-tasking DEC Ada implementation is showing comparable performance to its single-threaded counterpart.

For the second platform (multi-Intel 8086 microprocessor configuration) the SofTech ALS system is being used. However, to meet the operational requirements of the system, the target hardware is not based on standard Intel boards. Accordingly, certain non-trivial problems are involved:

i. Support for the different hardware components employed.
ii. The size of the run-time support (RTS) system - this currently exceeds the amount of the processor card on-board memory.
iii. Alterations to the standard RTS to cater for the different configuration of the target hardware.

This work is ongoing but is already highlighting a number of important issues eg the impact on validation.

Engine Monitoring System (EMS) Demonstrator

In the past for this application the Structured Analysis / Structured Design method[4,5] has been used. For Pascal, this has often involved considerable pre-implementation 'engineering' of the design. However, for Ada the information hiding features (primarily) appear to permit a more optimal mapping with consequential improvements in the software structure. This area of investigation is still at an early stage, but the initial results look encouraging.

Comparing functionally equivalent single threaded (non multi-tasking) Vax-based versions of the software implemented respectively in (DEC) Pascal and (DEC) Ada, shows a decreased run-time performance in the latter case, contrary to the results from the Sonar Ada platform. A number of implementation changes to the Ada version have been made in an attempt to explain this difference, but so far these have not substantially altered the results.

The reduced Ada performance for this application may be due to the significantly higher degree of numerical processing involved, and this is being investigated. Work is progressing on the implementation of a driver/display unit (to display the low cycle fatigue results) for which no such numerical calculations are involved. This should provide an opportunity for further comparison.

For the second stage of the work, it is planned to use the Verdix/VADS cross-development system and this should provide both size and performance data for the 68000-based target configuration in the near future.

Ancillary studies

Apart from the demonstrator projects, a number of additional studies are being carried out in the following key areas:

The design of Ada-based systems: A number of methods eg Structured Analysis / Structured Design[4,5], Mascot[1], the Structured Systems Analysis and Design

Method (LSDM/SSADM)[6], and object-oriented approaches[7,8] are being reviewed.

In the design and implementation of the MMI sub-system for the Digital Telephone Exchange Demonstrator, using a Mascot-like approach, it was found that for example:

i. Significant effort was needed to design and efficiently package the data types and objects (Mascot provides insufficient support).
ii. Packaging structures were initially adopted which were subsequently found to be non-optimal. Thus, while undesirable sharing of data objects was avoided, the first implementation required excessive sharing of data type definitions.
iii. Although the strong typing of Ada generally led to a much more straightforward mapping between the design and code, this was at the expense of some awkwardness in the processing of the data.
iv. Using 'with' alone, rather than 'with' and 'use', for referencing other packages was found to be much clearer and less error-prone (for this large scale development). This contrasts with the (implied) recommendations of most Ada textbooks whose examples are rather simple.

The following points were also observed:

Mascot segregates processing units (activities) and intercommunication data areas (IDAs), the access procedures for which encapsulate the more complex inter-task communication and synchronisation aspects. This is useful when the implementation teams are of mixed ability. In Ada, inter-task communication is an implicit part of all of the applications software. The full impact of this upon large systems developments has yet to be established.

A direct mapping of a Mascot design to Ada usually leads to all activities being implemented as (active) Ada tasks. However, since Ada assumes a synchronous tasking model (rendezvous) the implementation of the IDAs leads to two possibilities: (1) treating them as decoupled (asynchronous) inter-task communication mechanisms, and hence coding them as (passive) Ada tasks, or (2) effecting a synchronous inter-task communication by means of a rendezvous. In the first (more general) case, the overall system performance depends even more heavily upon the efficiency of Ada tasking.

It is important in defining the application boundary for an Ada task, to bear in mind that it is not possible to alter the priority of an Ada task dynamically (at run-time). Thus, care has to be taken not to group functionally related processing activities within a single task if the functions are inherently not of equivalent priority.

The MMI sub-system has now been re-designed and re-implemented based upon the findings of the review into this and other methods. This serves as a model example from which a reasonably optimal design and implementation code of practice is being derived for future applications.

Program testing: Most Ada compiler vendors are supplying symbolic debuggers for use within the program debugging stage of software development. However, most Defence-related projects involve a high degree of stringent testing (verification and validation against the requirements and design) and this aspect appears to be receiving scant commercial attention.

Accordingly, the suitability of using commercial symbolic debug facilities as the basis of a more sophisticated test harness is being investigated. To-date a prototype test tool has been produced which involves lexical analysis of the software under test and the automated production of command files to drive the DEC symbolic debugger. Further work is being carried out to investigate the representation of the complex real-time behaviour of Ada systems by advanced graphical means.

During this work numerous instances have been encountered of having to gather data about the software under test which must clearly have already been obtained during the course of Ada compilation. However, this information is not made externally visible by the compiler. It is felt that there is immense scope for Ada compiler suppliers to collaborate with industry to help overcome this sort of problem.

Configuration Management: Because of the inherent complexity of most Defence-related programmes, and the fact that they require multiple development teams, strong emphasis is placed on the need for efficient tool-based configuration management methods to support Ada projects. Of particular interest is support for software re-use across projects and in devising suitable schemes for linking existing or future configuration management databases to Ada system build facilities.

Current investigations are looking at the use of the DEC, SofTech and Verdix products for this purpose. The results are expected to form the basis of a future paper.

Run-time support: The provision of efficient run time support for embedded microprocessor-based Ada systems is crucial

to the use of Ada for such applications. To-date studies in this area have been frustrated by the frequent lack of detailed data from the compiler vendors about the characteristics of the run-time support systems to be supplied. Again, this is a problem area which could benefit from further collaboration between the Compiler suppliers and Industry. Some of the issues of importance are:

i. the functionality, size and performance of the RTS, and the 'hooks' provided for use by applications software,
ii. the advantages and disadvantages (eg for portability) of using proprietary RTS systems eg Intel's iRMX, and the Hunter & Ready VRTX system, as well as Ada specific products,
iii. the interaction between the underlying Ada compiler technology and the RTS system in the context of the often non-standard hardware configurations used in embedded microprocessor applications.

At the present time software that can be used to 'bench-mark' the commercially supplied run-time support systems is being developed.

Conclusions

The research programme being carried out by the Plessey Company represents a major initiative to examine the key issues associated with the transition to Ada. In common with views expressed elsewhere[9], Ada is regarded as much more than another programming language and is expected to provide a new and real opportunity to catalyse substantial improvements in industry's software engineering capability. In particular, by allowing the unification of working practices, Ada is expected to increase the opportunities for much greater levels of software portability and re-use, software reliability, and overall productivity.

Industry is keen to take advantage of these benefits but clearly any significant advance requires the availability of (cross-) compilers and support tools appropriate to the systems to be produced. Such compilers need to be not only technically compliant with the Ada Language Reference Manual but also of high performance and operationally efficient.

This is still an area of concern and despite the (increasing) number of validated Ada compilers available it is thought that there is much to be done before Ada can really be put to effective use for embedded microprocessor applications for which Ada has most to offer.

However, for the complex and real-time applications described here, the experience gained to-date gives cause for optimism. Thus, in the development of the Vax-based platforms the DEC Ada compiler appeared to be well engineered and operationally efficient. In these particular circumstances it resulted in comparable run-time performance in at least one application to Pascal, and although it currently involves increased code sizes when the run-time allocation of storage is taken into account, it is thought likely that this situation will improve in future products.

At the same time, the fact that the research programme has slipped in time-scales due to the non-availability of high performance cross-compilers and tools to support representative embedded micro-processor configurations is a cause of concern. Such delays could frustrate industry in bidding for and implementing these types of application. A further concern is that there often seems to be insufficient documentation concerning the detailed compiler characteristics eg resource requirements, availability of intermediate compiler outputs, run-time support features etc. This is an area which is just beginning to receive greater attention in the Ada community and needs to be encouraged.

As a general conclusion, it is felt that to-date there has been considerable emphasis (perhaps not surprisingly) on producing validated Ada compilers, but much less on providing genuine support for real Ada developments (whether this be appropriate compilers, support tools or in devising effective working methods for use with Ada). Clearly, if Ada is to be put into real service and provide the benefits that industry is expecting this balance has to be redressed at the earliest opportunity.

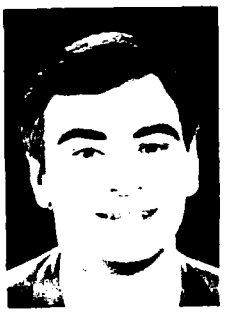
Acknowledgements

This paper is produced as a consequence of an in-house (private venture) funded Ada research programme carried out by the Plessey Company in the U.K. The author would like to record his thanks to the Company for supporting the production of this paper and its presentation at the 4th Annual National Conference in Atlanta, and to those people engaged on the programme who have provided the results presented.

References

1. Official Definition of Mascot, The Mascot Suppliers Association, U.K., 1980.
2. Cooper G A, Carter C B, Hess A, "AV-8B/GR Mk 5 Engine Monitoring System", 21st Joint Propulsion Conference, California, 1985.
3. Slape J, "Experience in the Use of Ada for a Digital Switching System", Ada UK Conference, 1986.
4. Yourdon E and Constantine L, Structured Design, Prentice-Hall, 1979.
5. DeMarco T, Structured Analysis and System Specification, Prentice-Hall, 1979.
6. Learmonth and Burchett's Structured Design Method (LSDM), Learmonth and Burchett Management Systems, London.
7. Booch G, Software Engineering with Ada, Benjamin/Cummings, California, 1983.
8. Berard E V, An Object-Oriented Design Handbook for Ada Software, E.V.B. Software Engineering Inc., 1985.
9. Walsh T J, "Transitioning to Ada: The challenge for Software Engineering", Proc. 3rd Annual National Conference on Ada Technology, Texas, 1985.

The Author



Dr Mel Selwood is the Project Manager for the Plessey Company's Ada Research Programme which has been carrying out the work described in this paper and may be contacted at Plessey Defence Systems, Abbey Works, Titchfield, Fareham, Hampshire, U.K. (Telephone +44 329 43031).

After graduating with B.Sc Honours in Applied Chemistry, he expanded his interests in Physical Chemistry and Computing to study solid state decomposition reactions and numerical methods of kinetic data analysis, for which he was awarded a Ph.D. in 1978. With this firm technical and computing background, he has since been actively involved in the development of several large command and control systems for the U.K. Navy and Royal Air Force, with duties including real-time software design and implementation, training, software management and development support. Prior to his current position, he led a Software Technology Group looking into improved methods and tools to support software systems development and has a strong personal interest in improving the software engineering principles and practices used within industry.

Tactical Database Management System -

An Ada[1] Technology Project for the US Army[2]

Judy Bamberger, Phil Ritter, Jackson Wilson
TRW Defense Systems Group
Redondo Beach CA

The Tactical Database Management System (TDBMS) is a prototype of a state-of-the-art database management system being developed in Ada for an Army laboratory responsible for developing, testing, and evaluating new hardware and software designed to meet the information management needs of battlefield automated systems. There are three major portions to the TDBMS contract:

- The database management system itself, supporting system maintenance programs, and front end programs to provide a variety of ways the user may access the database;
- A test bed in which to run experiments;
- A number of studies that emphasize areas of future research.

This paper presents an overview of TDBMS and then concentrates on two issues we have faced in the development of TDBMS: (1) using an Ada-based program design language (PDL); and (2) selection of an Ada compiler.

Background

As a part of its initiative to upgrade the technology of all areas of its battlefield management strategies, the Army has recognized that more sophisticated data management capabilities are required. An integrated database management system (DBMS) will provide uniform and consistent control of the data as opposed to each application having its own private files, with the data widely dispersed so that there is little or no attempt to control it in a systematic way. The Tactical Database Management System (TDBMS) project is one attempt to improve overall Army effectiveness by developing a prototype relational, distributed database management system that is implemented in Ada, designed to meet the Army's requirements for information processing in battlefield situations. TDBMS comprises a state-of-the-art database management system developed in Ada on: Sun workstations running under Unix[3], VAX[4] 11/780s running under VMS[5], and IBM-PCs running under PC-DOS. The database management system is coupled with a test bed in which to examine and explore simulated battlefield data management issues. In addition, several studies will be performed to explore issues related to the Army's battlefield modernization, including security of database management systems, the suitability of using Ada for a fielded DBMS, and characterization of battlefield data flow scenarios.

Current off-the-shelf, commercial DBMSs fall short of meeting the Army's needs due to the lack of sensitivity to battlefield requirements, including poor response time, complexity of the user interface, and large storage requirements.

TDBMS addresses these issues in the following ways:

TDBMS is a **database management system.** A database management system (DBMS) is used to: define data, organize data, store data, access data, and modify data. A DBMS supports the following characteristics:

- Centralized control of data;
- Consistency and uniformity of the data;
- Data independence of the applications programs; and
- Protection of the data.

By centralizing control of the data, redundancy of stored data is itself controlled, and inconsistency of stored data can be avoided. A single point of control (which may, in practice be one or more individuals) may be established; this is called a database administrator (DBA). Security restrictions may be more readily controlled, and data integrity may be maintained by the DBA who has the central responsibility for the entire database. From the DBA's viewpoint, requirements can be addressed from a

[1]Ada is a registered trademark of the US Government, Ada Joint Program Office (AJPO)

[2]This work is being funded under contract number DAAB07-84-C-K578 for the US Army, CECOM, COMM/ADP Center, Information Processing Technical Directorate.

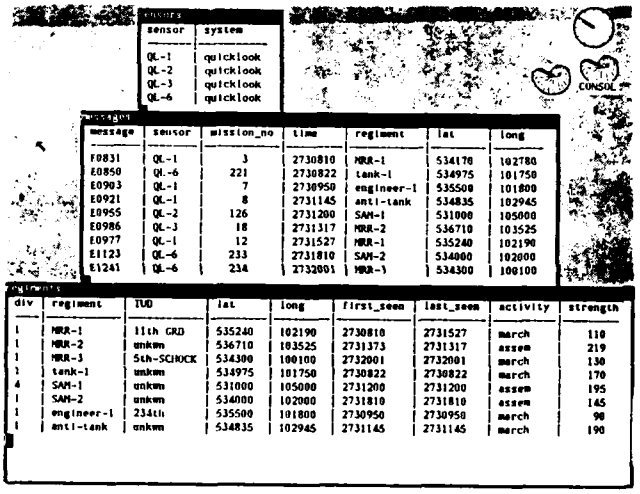
[3]Unix is a trademark of Bell Laboratories.

[4]VAX is a trademark of Digital Equipment Corporation.

[5]VMS is a trademark of Digital Equipment Corporation.

general view for the good of the system as opposed to addressing each requirement from an individual user's standpoint. Thus, data independence becomes possible, allowing the various applications of the data to be immune to any changes in storage structure or access strategy.

TDBMS is a **relational** *DBMS.* Relational database technology has been maturing over the past decade. Much has been published about the definition and theory comprising the relational calculus and relational algebra on which relational DBMS technology is based [Codd 70], [Date 81], [Ullman 82]. All types of relationships (one-to-one, many-to-many, n-way, and reflexive) can be naturally represented in a relational database. Information is represented as *relations* (i.e., tables), regardless of the kind of relationship being described. This provides a more uniform representation of the information within the database to the user. Figure 1 illustrates an example of the kinds of relations that could be found in a tactical database.



sensors

sensor	system
QL-1	quicklook
QL-2	quicklook
QL-3	quicklook
QL-6	quicklook

CONSOLE

messages

message	sensor	mission_no	time	regiment	lat	long
E0831	QL-1	3	2730810	MRR-1	534170	102780
E0850	QL-6	221	2730822	tank-1	534975	101750
E0903	QL-1	7	2730950	engineer-1	535500	101800
E0921	QL-1	8	2731145	anti-tank	534835	102945
E0955	QL-2	126	2731200	SAM-1	531000	105000
E0986	QL-3	18	2731317	MRR-2	536710	103525
E0977	QL-1	12	2731527	MRR-1	535240	102190
E1123	QL-6	233	2731810	SAM-2	534000	102000
E1241	QL-6	234	2732001	MRR-1	534300	100100

engineers

div	regiment	TUD	lat	long	first_seen	last_seen	activity	strength
1	MRR-1	11th GRD	535240	102190	2730810	2731527	march	110
1	MRR-2	unkwn	536710	103525	2731373	2731317	assem	219
1	MRR-3	5th-SHOCK	534300	100100	2732001	2732001	march	130
1	tank-1	unkwn	534975	101750	2730822	2730822	march	170
4	SAM-1	unkwn	531000	105000	2731200	2731200	assem	195
1	SAM-2	unkwn	534000	102000	2731810	2731810	assem	145
1	engineer-1	234th	535500	101800	2730950	2730950	march	90
1	anti-tank	unkwn	534835	102945	2731145	2731145	march	190

Figure 1. Example Tactical Database.

Since all types of relations are represented in a similar manner, query optimizations may be performed in a uniform manner. TDBMS uses a data manipulation language called TDL, which is based on SQL, a *de facto* industry standard. Figure 2 presents a sample TDL query against relations found in Figure 1, with the result of the query also given.

TDBMS is a **distributed** *DBMS.* The TDBMS prototype DBMS consists of two portions: a *front end* and a *back end.* The front end programs are provided for Sun workstations, terminals tied to VAX VMS, and IBM-PCs. The front ends communicate with a back end by means of a network connecting the two[5]. The back end

[5]The term "network" is used here in a general sense; it covers point-to-point and broadcast instances.

```
select m.sensor, m.message, m.regiment
    from messages m
    where
        ( m.time between 2730800 and 2731800 )
    and
        ( m.sensor = "QL-1" or m.sensor = "QL-2" ) ;
```

example1

sensor	message	regiment
QL-1	E0831	MRR-1
QL-1	E0903	engineer-1
QL-1	E0921	anti-tank
QL-2	E0955	SAM-1
QL-1	E0977	MRR-1

Figure 2. TDL Query and Resulting Relation.

comprises one or more network-connected, dedicated machines. The back end program (the DBMS itself) runs on a Sun workstation. TDBMS supports the capability to query relations that reside on multiple back end machines. The specific location of the data may, at the user's option, be either transparent or visible to the front end user. Relations may be replicated across one or more back end machines. This replication may be either transparent or visible to the front end user. By supporting a distributed database management system, TDBMS responds to the Army's need for a database that is survivable in harsh environments, when the failure of a single node of the database must not impact continuance of overall DBMS support. Figure 3 presents a pictorial representation of TDBMS.

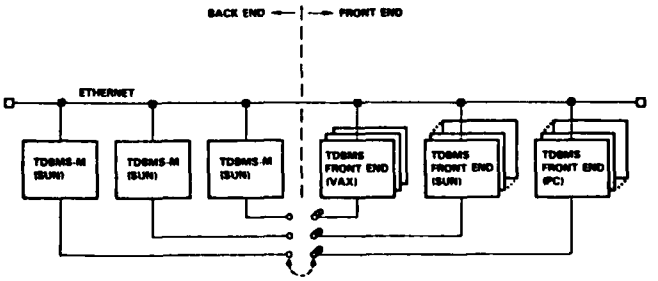


Figure 3. TDBMS is a Distributed DBMS.

TDBMS is **implemented in Ada.** The Army has taken a positive stand with respect to Ada - that it is to be the implementation language of choice for all Army software. This has multiple benefits to the Army. With respect to the TDBMS project in particular, this requirement ensures that:

- A generally useful tool, a DBMS, is provided in a language that is available on multiple host/target pairs, so the Army may export database technology to any number of other Army projects.
- TDBMS can be ported to militarized hardware in the future. Since Ada is a DoD standard language, one can assume with a high degree of confidence that the next generation of Army standard computers will have Ada compilers targeted to them.
- TDBMS is compatible with other on-going technology efforts sponsored by the Army. This includes the Army Secure Operating System (ASOS) project, which is building a multilevel secure operating system on which application programs may be run.
- TDBMS also provides a focus for discussion and evaluation of the merits and demerits of the Ada language, Ada compilers, and other Ada-related support software for systems development.

Thus, implementing TDBMS in Ada fully supports the Army's Ada initiative.

TDBMS is designed to meet Army requirements. By implementing TDBMS in Ada, as discussed in the previous paragraph, the requirement of *portability* has been addressed. This statement by no means implies that just because a program is written in Ada, it is then automatically portable. However, with the strong backing of DoD for the availability of Ada on a wide variety of hosts and targets, the compiler support is anticipated to be available. The validation of these compilers provides a high degree of assurance that the Ada source code processed correctly by one compiler will also be processed correctly by any compiler.

TDBMS is readily *modifiable and extendible.* TDBMS was designed in a modular, building block fashion that easily permits the addition or replacement of single modules. Figure 4 illustrates some of the modules that compose TDBMS.

For example, the prototype TDBMS uses a recursive decent parsing algorithm. This is known to be not as efficient as other parsing techniques, yet the error detection and recovery provide for a more robust parser. Should the Army decide at some point that greater speed is required, the current parser building block could be replaced with a new, table-driven parser.

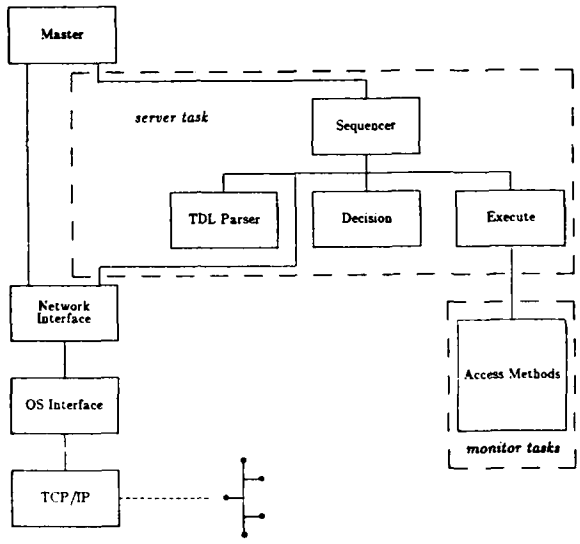


Figure 4. General Back End Design.

The Army has an obvious requirement for *survivability* in its databases. It is toward this end that the need for distributed databases, and integrity checks implemented by *triggers* and *command procedures*, have been included as a part of TDBMS.

The Army has a requirement for *high performance.* To this end, several internal optimizations have been made. In addition, much attention has been paid to using efficient algorithms to access information within the database. Much research has been performed in the area of query optimization [Hanani 76], and the use of B-trees for indexing [Comer 79]. The TDBMS test bed will be used to measure certain characteristics of the DBMS during simulations of "real world" scenarios, after which, the Army may decide to further tailor and enhance the basic capabilities of TDBMS.

There is also the need for *security.* The prototype TDBMS will support *access* security; that is, that database users must have the appropriate authorizations (database access, read, write, execute) on the appropriate relations (the data dictionary, relations, views, command procedures) in order to perform certain classes of database manipulations (select, update, insert, delete, invocation of a command procedure). Multilevel security is not a requirement on this phase of TDBMS; however, this is a major concern to the Army when TDBMS is ultimately fielded. One of the studies for the TDBMS contract identifies requirements for making TDBMS multilevel secure at some future date [Garv85]. TRW is performing additional research and development on the design and implementation of a secure database management system using TDBMS as a baseline.

Using an Ada-based program design language (PDL)

Both flow charts and program design languages (PDLs) are often used for the representation of system design. Flow charts, more traditionally used, provide a graphical representation of system flow. PDLs, on the other hand, describe the system in a textual format. The formatting (indentation, keyword high-lighting) of a PDL captures the information that is provided via arrows and different shapes of figures in flow charts. The Army is requiring its new projects to represent their system design in an Ada-based PDL. We used TRW's Ada PDL [Ada PDL 84].

Several issues came to the forefront as we progressed:

- The kinds of people who were best at using Ada PDL;
- The degree of formality we required;
- The degree of detail we required;
- Changes required when we moved from design to code; and
- Implications for our customers.

Each of these are discussed below.

People Impacts. The qualities we found in those people who had the easiest time of doing the system design and representing that design in Ada PDL included:

- Excellent abstraction capability;
- Experience in the application; and
- Familiarity with Ada used as a PDL.

We found this last point to be of relatively little importance compared to the second, and the first point to have the most significance of all the characteristics. Our design team was composed of one senior technical lead, two more senior people, with the remainder of the team relatively young, inexperienced, and well-educated in computer science fundamentals. Among us, we knew multiple programming languages; only a few of us knew Ada. We found that those team members who had the best underlying conceptual models of how systems work had the least problems understanding and using Ada PDL to represent the design. We found those individuals who were best at viewing complex systems as layers of controlled abstractions produced the best, most understandable design.

Degree of Formality. The top-level design of TDBMS [System Specification 85] was represented textually. Each of the requirements of TDBMS was identified and described in English. These requirements were then iteratively refined into a number of functional units [Part 1 Rationale 85], which we designed as Ada PDL packages. As we progressed through the design phase, we added more details to the package specifications, we identified capabilities that were required across the system and added those to the system-wide utilities units, and we added more details about how each of the functional units were to behave. This process is not unique to those situations where Ada PDL is being used to represent design. However, we did make some decisions on our use of Ada as a PDL that we felt aided this process.

We made a decision early on in the design phase to write our inter-unit interfaces in "pure" Ada; i.e., for the appropriate level of detail, all semi-colons, commas, and parentheses would be required. Inter-unit interfaces were defined as those package specifications that were used by more than a single functional unit. By requiring a high degree of formality in the definition of our interfaces, we were able to discuss issues in a concrete, unambiguous framework. This helped us control and verify our inter-unit interfaces.

For the design of bodies, we left the degree of formality up to the individual designer. TRW's Ada PDL supports the use of formal Ada constructs and less formal, English-like "design narrative". We found a high degree of design narrative used throughout the body design modules. The use of design narrative in the body design modules enabled us to capture the sense of the design without being entrapped by the details of an implementation language.

Degree of Detail. The question of "where does design stop and coding begin" becomes even more of a problem when the design and implementation languages are the same (or extremely close). When designing in any Ada-based design language with the intent of implementing in Ada as well, this distinction is not clear.

Because we required "pure" Ada to be used for our inter-unit interfaces, and because many of these interfaces were complex data structures that can be represented quite naturally using Ada's data structuring capabilities, we often found a subprogram body that was to validate certain fields of the input parameter and return some function of that input parameter could be fully and accurately written in three or four Ada (PDL) statements. Such subprograms were transformed into code in a matter of minutes.

The degree of detail of the design varied from designer to designer, and even within pieces of design produced by a single designer. In some cases, the choice of the algorithm and the data structure to be used to implement a given system function were obvious, given known system requirements and constraints. These areas were not described in as much detail as were those areas that were deemed the highest risk either by the project as a whole or by the individual. Those high risk areas were designed in substantial amount of detail, perhaps even looking

more like code than like design. It was often the case where the designer would write a short piece of English-like design narrative to describe the function to be performed, question whether or not that design narrative correctly described the required functionality, and then provided a short series of Ada statements (including subprogram calls) to prove to her/himself that the design narrative was essentially correct in the first place. Thus the design became more detailed than was required.

Moving from Design to Code. We had a relatively painless time moving from design to code. The degree to which this has to do with using an Ada PDL, using Ada as the implementation language, with the design itself, or the quality of the design and implementation team is not discussed here. What is of interest is those areas, and they fall into a few categories, where we needed to make "significant" changes when implementing the Ada PDL design.

We found that we used **limited private** in design in places where we could not when we implemented in Ada. The TRW Ada PDL processor does not check whether or not a data object is **(limited) private**, so we were passing aggregate structures containing **limited private** components as output parameters. As a result, we had to "demote" some of our **limited private** types to simply "private" types at implementation time. However, we required that the intent of the type being **limited** be identified clearly in the comments in the code.[6]

Due to storage allocation issues, we found that we had to reorder some fields in some aggregates. This was not a major issue, just one of which to be aware.

We identified and implemented a number of abstract data types. These included an 8-bit type holding values 0 .. 255. We built this on top of an existing 8-bit integer type, and provided a complete set of operators so we could manipulate objects of that type as naturally as any other integer type. We also required a variable length string capability. We identified the need for this during the design phase, and we designed assuming that we would have a fully variable length string capability available to us at implementation time (which we built). We also identified the need for a generic storage manager[7] early on during design. We provided a detailed specification of the capability at design time, and dealt with the detailed design as implementation time neared.

These examples illustrate the different points at which we needed different capabilities than those we identified at design time. Due to the sophistication of the initial design, making these changes and additions were clean and painless.

Impacts on the Customer. This area does not get much discussion, yet it is one of the most important aspects before the Army Ada initiative can be deemed successful. Both the customer and the contractor must identify that there is an impact when using an Ada-based PDL on the customer. In the near term, this impact is a major one - one of education.

Since design has been traditionally represented using flow charts and data flow diagrams, customers reviewing such a design have developed standards for recognizing good (and bad) designs. The customer receiving a design in Ada PDL must be sufficiently knowledgeable in software engineering, design techniques, and Ada PDL to recognize whether or not the design is a good design. This is no different from when a customer has to learn to read flow charts and data flow diagrams. However, the number of usable primitives in an Ada-based design language are far greater than the number of primitive shapes and kinds of arrows in traditional flow charts.

In addition, the customer is required to understand which Ada PDL constructs best lend themselves to being used in a "good" representation of a design. Where encapsulation is required, packages are the feature to expect to be used. Where encapsulation and parallelism are both required, tasks are the feature to expect. An encapsulation should support the complete abstraction of the object or type being encapsulated: a full set of operations, a full set of state functions, a full set of I/O and error handling capabilities. To support robust error handling, exceptions specific to each unit should be exported in a package specification. Generics should be used for those areas where common functionality is required over a variety of types of data structures. Libraries of utility subprograms should be provided for software components that are required by a number of functional units. No more information should be made visible at any given level than is required by the users of the software at that level. That is, global data should be minimized; state information and local variables should be hidden and declared at the level where they are actually used.

The customer must be able to recognize a good design in Ada PDL, not just the fact that an Ada-based PDL has been used.

[6] In this context, we had some discussion as to whether or not this is only an idiosyncrasy of the Ada PDL processor, the design of the Ada language, the manner in which we represented our design, or any or none of the above. There were constructs that we used during the design phase to more naturally and logically represent the *intent* of the design that cannot be exactly replicated in ANSI-standard Ada. Whether or not this is an acceptable use of an Ada-based design language is an issue often discussed with religious fervor.

[7] Support for the (de)allocation of a variety of types of data.

Selection of an Ada compiler

There are several major points that we considered when selecting an Ada compiler for use on TDBMS:

- Whether or not a compiler is available on the hardware on which we need to develop Ada code;
- Whether or not the compiler is validated; and
- Whether or not the compiler is suitable for use for our application.

What is presented in this section is not the only considerations that we had when selecting an Ada compiler, but some major areas that impacted us directly and how we chose to deal with them.

There are several Ada compilers currently available for multiple hosts and targets [Shugerman 85]. Many of these compilers are validated [Arpanet 85]. But what does "validated" really mean to a project?

Validation

The official validation policy is currently under going clarification. Previous incarnations of the policy and procedures for complying with the policy have been summarized in [Hook 85], [Knoop 85], and [Kopp 85]. Compiler users have long been concerned with the validation policy and the requirements of controlling and baselining support software changes in a real project environment. As a result, an embedded computer industry Ada Compiler Validation Working Group (AVWG) was formed, which meets with those responsible for making and enforcing the validation policy. The AVWG report is published as [AVWG 85].

Quite briefly, "The objective of the validation process is to certify Ada compilation systems that conform to the language standard." [Hook 85]That is, the intent of the validation process is to stamp out all dialects - no *validated* Ada compiler shall accept a subset or a superset of the language as defined in ANSI/MIL-STD-1815A [ALaRM 83].

It is quite important to projects using Ada compilers today to recognize what validation does *not* guarantee.

- Validation does not indicate the suitability of a compiler for a particular purpose.
- Validation does not replace a set of application-specific requirements.
- Validation does not measure the performance of a compiler.
- Validation does not evaluate any other component of the programming environment. [Kopp 85]

Again, the goal of validation is to *stop language proliferation*, nothing more.

While TDBMS was not required to use a validated Ada compiler, we chose that route for one major reason. In order to be validated, a compiler is required to pass a substantial test suite (currently some 2000+ tests). While the test suite does not (and cannot) test every combination of language features, it does serve to provide a significant measure of the capability of a compiler to correctly process the entire Ada language.

Compiler Suitability

Performance. There are a number of measures of suitability of a compiler. Obvious among them is compile-time speed. We had a choice of two compilers for the Sun workstation at the time we needed to commit to a single compiler. Both compilers were validated at the time. We had a substantial piece of Ada code developed internally that was similar to the TDBMS application, and we attempted to compile it using each of the two compilers. One compiler was quite slow (about 150 - 200 lines per minute), and, in fact, was unable to correctly compile our test program, without our having to make significant changes to the Ada code. The other compiler was significantly faster (about 400 - 600 lines per minute on our sample code), and was able to successfully detect a number of Ada coding errors that were not detected on a previously-used, unvalidated compiler or on the first of the two candidate compilers. Only a few, relatively minor changes to the Ada code were required to avoid the compiler bugs on the second candidate compiler. The advantage of having benchmark programs similar to the application in question cannot be overemphasized when evaluating a compiler.

However, in other types of projects, and in future work on TDBMS, the run-time performance of the generated code is of at least as much importance to the compiler selection process. For Increment 1 of TDBMS, this was not an issue. We did note, however, that our test code executed at a reasonable speed. Had performance been an issue, we would have needed to determine the tolerances and required performance levels prior to running our test programs. It would be at this point that we would possibly have had to make a trade-off between the compile-time speed and the run-time speed.

Tasking Support. TDBMS uses tasks, both dynamically spawned tasks and statically initiated tasks. We needed to ensure that the compiler supported these language features. Since we had decided to select a validated compiler, we got this guarantee "for free". However, since performance is not of primary importance during Increment 1 of TDBMS, we did not evaluate in detail the

speed with which context switches occur during rendezvous. Again, when speed becomes a factor in our development, this issue must be examined. We did create a test program that simulates some of the functionality of the TDBMS back end to examine how the tasking scheduler works. *It should be noted that writing code based on the knowledge of how the task scheduler works produces "erroneous" Ada programs; i.e., programs that are written with that knowledge may not be portable.* During design, we assumed that all tasks would be scheduled "fairly", that rendezvous would be performed within a tolerable period of time, and that no entries would be starved. The tasking test program that we wrote bore this out, so we put no special priorities, guards, or timing controls in our programs. The advantage is that we have cleaner, simpler code; the disadvantage is that there is a chance that there may be some problems porting the existing TDBMS to other machines. So, for Increment 1, we did rely on the knowledge that the task scheduling algorithm implemented by the compiler that we are using does, in fact, rely on that algorithm being "fair". *We explicitly called this out in our project documentation.* However, in some portions of our code, notably those portions that we deemed the highest risk in porting to a new compiler, we chose to implement our tasking using a "pure Ada model" (i.e., not depending on the knowledge of how the task scheduler works).

One additional concern with tasking has to do with storage allocated for dynamic tasks. TDBMS is a program that will run, conceivably, infinitely. When a user decides to establish a session with TDBMS, the user indicates that on a TDBMS front end machine, and a back end *server task* is dynamically spawned to handle the user's session. When the user indicates that the current session is to be terminated, the dynamically spawned task is terminated as well. Since it is anticipated that a substantial number of these server tasks may be initiated and terminated over time, it is of major importance to us that all storage space used by each dynamically activated task be reclaimed when that task terminates. If not, there is obviously some limit on the number of user sessions that can be permitted over time, and our current implementation, which assumes this storage is reclaimed, must be adjusted to take this into consideration.

Chapter 13. There are language features in Ada that are not required to be provided, even in validated compilers. And the presence or absence of these features may be of just as much importance to the compiler selection process as any other issue. These language features are defined in Chapter 13 of [ALaRM 83] and include:

- Interface to subprograms written in other programming languages;
- Unchecked programming, including unchecked conversion and unchecked deallocation; and
- Representation, length, and address clauses, which address how types and objects are to be mapped onto the underlying machine.

Two of these language features are required by TDBMS: (1) interface to the C programming language, to access the TCP/IP network and device driver primitives; and (2) unchecked programming, to move conceptually untyped data into a typed location, and vice versa.

All instances of interfacing to C routines are isolated into a single unit - the OS Interface unit. The OS Interface unit presents an Ada interface to all who use it. Internally, the subprogram bodies perform transformations on parameters in accordance with the limitations of the interface as defined in Appendix F of the vendor's compiler documentation, call the C subprograms, and transform the output parameters into a format compatible with the Ada interface. Since the TDBMS project chose to use off-the-shelf TCP/IP interfaces and not develop such an interface as a part of the project, we required minimal support for interfacing with C routines.[8] However, the need for interfacing with device drivers is present in many kinds of systems programs, so the project must ensure that this capability is supported to some extent by the compiler.

There are two places where TDBMS is required to effectively "ignore" the strong typing enforced by the Ada programming. As a systems program, the DBMS portion of TDBMS is required to read untyped bytes off the disk, and to format them according to certain byte patterns that are known only after the information is read off disk. The second place where the strong typing must be ignored is in transmitting information over the network when communicating among TDBMS machines; information is placed on the network as a string of (untyped) bytes; it must be read off and then reformatted according to byte patterns that are known only after the information is received at the other end. Both of these two situations are similar, and the unchecked programming, an optional feature of Ada, is required in both instances. Unchecked conversion is not a required feature of Ada, and TDBMS had to ensure not only that the compiler supported it, but also to what extent it was supported. We discovered, by trial and error, that there were well-known (but poorly documented) limitations to the cases where unchecked conversion would "do what we thought it should do". It must be remembered, when selecting a compiler for a project where unchecked programming is

[8] It should be noted that an Ada interface to TCP/IP is to be provided as part of the NOSC tool set, and will be in the public domain. This capability was not available when we required it, however, nor will it support our needs once it becomes available, as it implements only a subset of the TCP/IP capabilities that we require.

required, to have a complete understanding of the capabilities and limitations of this feature.

Two other portions of Chapter 13 that would have saved us substantial time and effort had they been present are the representation and length clauses and unchecked deallocation (i.e., garbage collection). Representation and length clauses could have been judiciously used early on in the coding phase to help simplify some of the manipulations and conversions of data that we must perform. Since they were not present, there are some "unusual" pieces of code that implement, in a non-direct manner, the movement of data from one strictly and tightly defined memory location to another. The presence of unchecked deallocation would have greatly simplified the writing our own storage allocator.

Other Environment Tools. Ada compilers are only a single component of the tools required over the software life-cycle. However, they are intensively used during the code, unit test, and integration and test phases of a project. A fully developed software engineering environment is many years and many millions of dollars off in the future. However, the bulk of our work was done on the Unix operating system which provides a rich complement of tools that can be combined to build a wide variety of life-cycle support tools.

Some tools are still provided as a part of the compilation environment. The importance of a symbolic debugger cannot be overlooked. Regardless of how careful programmers are, they still make errors. A symbolic debugger in our environment has been quite helpful in assisting us locate certain errors. A symbolic debugger that is robust and complete would be even more help.

Library management and configuration management tools are also important on any project where more than a single person is involved. With the basic library management capability supplied with the Ada compiler, we have found that we could build more sophisticated library management and configuration management tools in the Unix environment, where we are doing the vast majority of our development. By having the Ada compilation system provide only the minimal support for library and configuration management, we are able to tailor our programming environment to the way that TRW best does business, instead of having another view of project management foisted upon us and being, perhaps, incompatible with TRW standards and policies.

Vendor Support. This area is often overlooked in evaluation of a compiler, yet it is extremely important. Vendor support is required in two major areas: documentation and maintenance. Without proper documentation, any tool is virtually useless, and Ada compilers and environment tools are no exception. However, without timely maintenance support, the compiler may be useless anyway. Once we have identified a bug, we report it to our vendor. As of yet, we have not encountered any "show stoppers" - i.e., we have always found a work around. However, this may not always be the case. We have found it invaluable to foster a good working relationship with our vendor such that we can keep the lines of communication open. This includes providing them access to our source libraries (having had them sign non-disclosure agreements, of course). We anticipate that because of this good relationship, should we ever encounter a "show stopper" bug, we will receive prompt attention and timely response.

Summary

The TDBMS project is a research project aimed towards developing a DBMS that addresses the specialized requirements of a tactical database. The TDBMS design was driven by the following tactical requirements:

- Maintenance of system survivability and reliability by the hardware and software;
- Flexibility of real-time performance factors dependent upon the particular tactical situation;
- Built-in data security; and
- Portability and adaptability without performance degradation.

These driving requirements are what separates this prototype TDBMS from other commercial DBMSs.

Upon completion of this phase of the TDBMS contract, the Army will receive a sophisticated database management system with a variety of user interfaces, a test bed in which to use and tune the DBMS, and a number of studies exploring issues for future research in the areas of database security, the applicability of Ada to database management systems and system software, and battlefield information requirements. Both the Army and TRW will have gained substantial experience at evaluating several Ada compilers and other support software, using an Ada-based design language, procuring Ada systems, and implementing major system components in Ada. For the Army, this experience will not only result in obtaining state-of-the-art products that can be used in a variety of applications programs, but it will enhance the Army's capability to be a wise and educated consumer of software and Ada products.

Using any new technology has benefits as well as pitfalls; Ada (both for design and implementation) is no exception. The early days of Fortran compilers and Fortran-based design methodologies were traumatic. History has shown that the introduction of other new languages and

methodologies has also met with a difficult initiation period, as individuals, academia, corporations, and the services come up on the learning curve. The same can be said for the current time period with respect to Ada. The Ada community has seen a substantial investment on all parts toward increasing the quality of tools, personnel, education, and software products. With this kind of impetus, and the backing of the DoD, we feel that quantifiable benefits of Ada will be seen by the end of the decade.

References

[Ada PDL 84] Bamberger, Judy.
Ada PDL User's Manual.
TRW, Redondo Beach CA, 1984.

[ALaRM 83] US Department of Defense, Ada Joint Program Office.
Reference Manual for the Ada Programming Language ANSI/MIL-STD-1815A).
United States Government, 1983.

[Arpanet 85] Public Arpanet account: ada-information; file: validated-compilers.
Matrix of Validated Compilers.
Arpanet , 1985.

[AVWG 85] Hilfinger, Paul N., *et al.*
Report of the Validation Committee on Validation Policy.
1985
Available on the Arpanet.

[Codd 70] Codd, E. F.
A Relational Model of Data for Large Shared Data Banks.
Communications of the ACM 13(6), June, 1970.

[Comer 79] Comer, Douglas.
The Ubiquitous B-Tree.
Computing Surveys 11(2), 1979.

[Date 81] Date, C. J.
The Systems Programming Series: An Introduction to Database Systems.
Addison-Wesley Publishing Company, Reading MA, 1981.

[Garvey 85] Garvey, Cristi.
TDBMS Security Study.
Prepared for US Army Communications-Electronics Command, Ft. Monmouth, NJ, 1985.
TRW, Defense Systems Group, Tactical Database Management System, CDRL Item F005, Contract No. DAAB07-84-C-K578.

[Hanani 76] Hanani, Michael Z. and Ben-Zvi, Yaacov.
Queries for reports from a large file: analysis and fast response.
Management Datamatics 5(1), 1976.

[Hook 85] Hook, Audrey, A.
Ada Validation Process, Policies and Procedures.
AdaJUG/SIGAda Joint Meeting , 1985.
Presentated as part of the Policy Committee.

[Knoop 85] Knoop, Patricia A.
Ada Validation Procedures.
AdaJUG/SIGAda Joint Meeting , 1985.
Presentated as part of the Policy Committee.

[Kopp 85] Kopp, Major Allan H.
Polic*man - Ada Validation Policy.
AdaJUG/SAE-AE-9E (1750 User's Group) Joint Meeting , 1985.
Presentated as part of the Government Corner.

[Part 1 Rationale 85] TRW.
Prototype Database Management System Part 1 Specification Rationale (draft final).
Prepared for US Army Communications-Electronics Command, Ft. Monmouth, NJ, 1985.
TRW, Defense Systems Group, Tactical Database Management System, CDRL Item F003, Contract No. DAAB07-84-C-K578.

[Shugerman 85] Shugerman, Marvin.
Ada Implementation Analysis.
TRW Internal Document , 1985.
A collection of who is doing what, and anticipated availability.

[System Specification 85] TRW.
System Specification (draft final).
Prepared for US Army Communications-Electronics Command, Ft. Monmouth, NJ, 1985.
TRW, Defense Systems Group, Tactical Database Management System, CDRL Item B001, Contract No. DAAB07-84-C-K578.

[Ullman 82] Ullman, Jeffrey D.
Computer Software Engineering Series: Principles of Database Systems.
Computer Science Press, Rockville MD, 1982.

Phil Ritter is a member of the technical staff in the Information Processing Department at TRW. He has worked on the design and development of language processing and database management projects both at TRW and at the University of California at Irvine. He is currently the lead designer on the TDBMS project. He received his BS in Computer Science from UC Irvine in 1984.

Judy Bamberger is a member of the technical staff in the Information Processing Department at TRW. She has designed and implemented several key portions of the TDBMS system. Her prior experience on other Ada projects at TRW has helped her serve as the Ada and training focal point on the project. She received her BS in Mathematics, French, and Education from the University of Wisconsin - Milwaukee in 1974, and her MS in Computer Science from UCLA in 1985. She is currently Vice-Chairperson of the Ada-JOVIAL Users Group (AdaJUG).

Jackson Wilson is a staff engineer in the Database Management and Support Software Department at TRW. He has managed the design and development of database management software on several projects in the Defense Systems Group. He is responsible for system engineering on the TDBMS project. He received his BA in Mathematics/Computer Science from UCLA in 1972, and his MS in Computer Science from UCLA in 1975.

A Practical Approach for Translating FORTRAN Programs to Ada[1,2]

V. Santhanam

Computer Science Department
Wichita State University, Wichita, Kansas

Abstract

Attempts to translate FORTRAN programs to equivalent Ada programs using automatic translators date back to 1983. While a number of translators has been constructed to date, few have attempted to provide maintainable Ada code at the output. This paper describes a translation approach which emphasizes the quality of output code. The goal is to produce Ada code that can be subsequently maintained and retargeted easily. This goal is achieved by abstracting portions of the input source code that do not lend themselves to simple transliteration and then reconstructing an equivalent output code. The approach uses a combination of optimization techniques and knowledge from user-supplied directives. The conclusion drawn from this work is that while table-driven transliteration schemes may have failed to yield acceptable translations, a more sophisticated translator based on the abstraction and reconstruction approach can be devised to produce maintainable Ada code at the output.

Introduction

Numerous pieces of time-tested software written in FORTRAN exist today within the defense industry. With the Department of Defense moving toward mandating the exclusive use of Ada in future software systems, much of the existing software must be either scrapped or reimplemented in Ada. When reimplementation in Ada is appropriate, potential savings from avoiding manual recoding and subsequent retesting warrant a closer look at the possibility of automatic translation. However, if automatic translation is to be acceptable as a broad reimplementation strategy, the quality of Ada output must be higher than if one simply needed an equivalent Ada program.

An early attempt to translate FORTRAN programs to Ada using a language transformation tool is described by Slape and Wallis [1]. The report identifies valid mappings of major FORTRAN constructs to equivalent Ada constructs. It also notes two major difficulties of automatic translation: (1) there are constructs in FORTRAN that do not readily translate to Ada (e.g., COMMON and EQUIVALENCE), and (2) translated code often contains unidiomatic uses of Ada constructs (i.e., the translation, though valid, does not have the "native Ada" style). Several other reports have also discussed the importance output style and the difficulties of attaining that with simple transliteration systems [4,5].

This report presents an approach that reduces the problem of the inadequate output style which characterizes statement-by-statement translators. Using a combination of analysis techniques germane to optimizing compilers and a limited amount of user input, the style of output code is improved significantly. User input is accepted in the form of *annotations* embedded within the FORTRAN source. The approach is backed by the detailed design of a *translation system*, which is depicted in figure 1. The dialect of FORTRAN addressed in this report is ANSI FORTRAN-77 [6].

The Approach

Much of the difficulty reported in the literature stems from the fact that the attempts to date have concentrated on a statement-by-statement translation of FORTRAN subprograms in isolation. While FORTRAN compilers indeed work on individual subprograms one at a time, Ada compilers do not. It is, therefore, not appropriate to translate subprograms in isolation. In the approach presented here, FORTRAN subprograms are translated in *groups*. The group must form a functionally complete set. That is, every subprogram that is called from one or more subprograms under translation is itself under translation, or it has been described by a user-supplied interface annotation. The translator not only checks the validity of subprogram interface but also determines the correct parameter-passing modes for each subprogram.

The functionally complete group of subprograms at the input yields an Ada package at the output. This is in contrast with the Slape-Wallis converter

1. Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

2. This research was supported by Boeing Military Airplane Company, Wichita, Kansas.

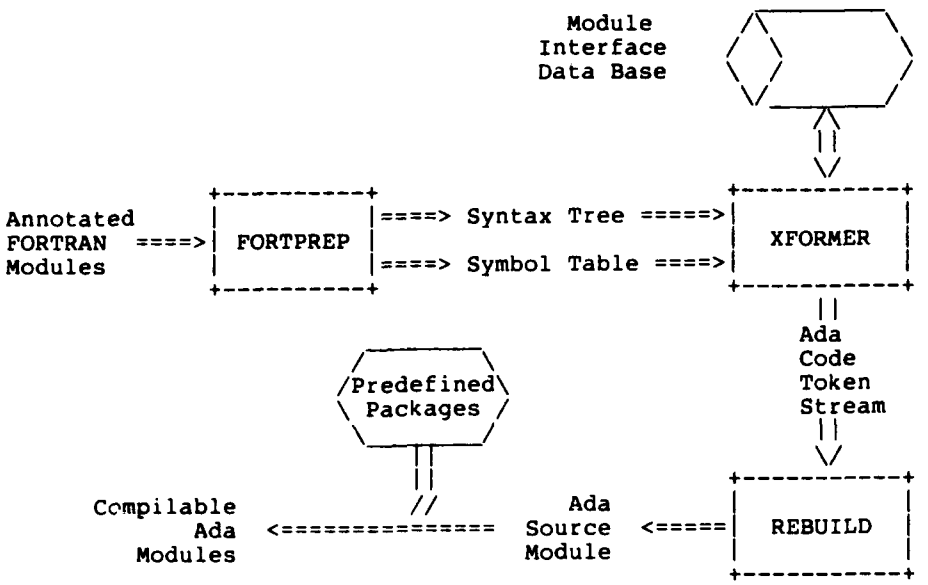


Figure 1. A Translation System based on the Present Approach

[1], which generates a package for each subprogram. (Wichmann and Meijerink [5] argue that such a translation is not an idomatic use of the Ada package facility). If the group of modules includes the main program an Ada (main) procedure is also produced at the output. A separate pass of the translator is used to generate packages corresponding to COMMONs if any.

The packages and the main procedure can, of course, be compiled separately. Each compilation unit 'WITH's a predefined translator environment package and other packages as necessary. The predefined environment package would include the type declarations corresponding to FORTRAN data types and the declaration of all intrinsic functions.

Language Issues

Several FORTRAN constructs are known to present difficulties in translation. This section presents the methods employed in the present approach for handling the translation of such constructs.

Subprogram Interface

FORTRAN-77 rules for interfacing with subprograms are not as rigid as those laid down by Ada. For example, a formal one-dimensional array parameter may be associated with a multidimensional actual parameter array. To compound the problem, most FORTRAN compilers do not check subprogram interfaces, thus allowing more freedom than permitted by the language. The following discussion pertains only to the problems of translating legal FORTRAN-77 programs.

Array parameters. When the dimensionalities of a formal parameter array and the corresponding actual parameter array differ, the following technique is used to translate the program: both the formal and the actual parameter arrays are *linearized*, i.e., translated to one-dimensional vectors. The translator is directed to linearize an array (formal or actual parameter) through a user-supplied annotation. When an array is linearized, reference to its elements is through a single subscript. Converting multiple subscript references to single subscript references is straightforward. Other approaches, such as UNCHECKED_CONVERSION to convert the actual parameter to the same type as that of the formal parameter, are not likely to work. This is because the mapping of array elements to storage is rigidly specified in FORTRAN-77 (the infamous column-major order), whereas it is implementation-dependent in Ada.

Another potential problem with array parameters is that the bounds of the actual parameter array index need not match those of the formal parameter array index. Fortunately, free conversion is permitted in Ada between the types of two arrays of the same dimensionality and the same number of components in each dimension. Thus, an explicit type conversion is generated whenever this situation occurs:

```
SUBROUTINE SORT(A)
INTEGER A(50)
:
END
:
INTEGER X(0:49)
:
CALL SORT(X)
```

translates to

```
type ARRAY_A_IN_SORT is INTEGER_ARRAY_1(1..50);
procedure SORT( A: in out ARRAY_A_IN_SORT );
:
end SORT;
:
X: INTEGER_ARRAY_1(0..49);
:
SORT( ARRAY_A_IN_SORT(X) );
```

In addition, FORTRAN-77 allows the actual parameter to be an array element, provided enough elements follow it to match the size of the formal parameter array. The situation is handled in the translation by slicing the actual parameter array before applying the type conversion.

Variable number of parameters. FORTRAN-77 does not permit variable number of parameters for user-written subprograms. However, there are a number of intrinsics in the language that can be referenced with a variable number of parameters, e.g., the function MAX0. The desired effect can be obtained in Ada either by providing default values for input parameters or by overloading the subprogram declaration for all counts of parameters. When the number of parameters is unbounded, as is the case with MAX0, the subprogram could be overloaded with an array argument. For example,

```
function MAX0( M: INTEGER_ARRAY_1 )
  return INTEGER_FORT;
```

would permit the translator to convert the input argument list into an aggregate, as illustrated below.

```
MAX0( 24, I+5, J )
```

would translate to

```
MAX0( INTEGER_ARRAY_1(24, I+5, J) )
```

Alternate returns. FORTRAN-77 allows subroutine subprograms to return to a statement other than the one that follows the CALL. For example,

```
CALL DECIDE( X, *100, *200 )
```

would allow the subroutine DECIDE to return to statement 100 or 200, or to the statement that follows the CALL as is normally the case. The construct would be translated as follows.

```
CALL_DECIDE:
declare
  RET_POINT: INTEGER_FORT := 0;
begin
  DECIDE( X, RET_POINT );
  case RET_POINT is
    when 1 => goto LABEL_100;
    when 2 => goto LABEL_200;
    when others => null;
  end case;
end CALL_DECIDE;
```

Notice that the interface to the subroutine changes from three parameters to two. The subprogram declaration is translated in a compatible manner to include an OUT parameter (say, RET_LABEL) in place of the first * parameter (* in FORTRAN stands for the label parameter), ignoring all other * parameters. The return statements such as

```
RETURN I
```

are translated to

```
RET_LABEL := I;
return;
```

Subprogram parameters. FORTRAN-77 permits parameters to subprograms to be subprogram names themselves. It is common, for example, to use this feature to build numerical subroutines which work with a user-specified function. Although Ada does not provide the same facility, the "generics" feature of Ada can be employed to obtain a similar effect provided all actual parameter subprograms have the same type signature. When this condition does not hold (FORTRAN-77 does not require it), there is no simple translation.

Functions with side-effect. FORTRAN functions that produce side-effect by manipulating parameters pose yet another problem since Ada permits only IN-mode parameters to functions. If the use of such functions is present, the translation is effected as illustrated below.

```
FUNCTION SIDE( A )
A = A+1.0
SIDE = A
END
:
Z = SIDE( A )+2.0
:
X = SIDE( A )+SIDE( B )
```

translates to

```
procedure PROC_SIDE( SIDE_VAL: out REAL_FORT;
  -- SIDE_VAL is a new parm for return value
  A: in out REAL_FORT ) is
begin
  A := A+1.0;
  SIDE_VAL := A;
end;
:
declare
  SIDE_VAL: REAL_FORT;
begin
  PROC_SIDE( SIDE_VAL, A );
  Z := SIDE_VAL+2.0;
end;
:
declare
```

```
    SIDE_VAL_1, SIDE_VAL_2: REAL_FORT;
  begin
    PROC_SIDE( SIDE_VAL_1, A );
    PROC_SIDE( SIDE_VAL_2, B );
    X := SIDE_VAL_1+SIDE_VAL_2;
  end;
```

COMMON and EQUIVALENCE

COMMON blocks of FORTRAN define globally accessible data, and as such they readily map into library packages in Ada to be 'WITH'ed by the units needing access to the data. The translation of COMMON statements, however, is made more difficult by the freedom provided by FORTRAN to alias the common data area. Each program unit referencing a common block may define its own set of variables. Each such definition is assumed to remap the same storage region. An additional aliasing capability is available through the EQUIVALENCE statement, which may be used to remap common blocks as well as local data areas.

Ada supports a limited form of aliasing with the RENAMES clause. Slape and Wallis [1] discuss how to use this clause to translate limited forms of COMMON and EQUIVALENCE. A more general solution would have to be implementation-dependent. Two methods have been developed toward this end. The first method consists of using a variant record declaration for each COMMON area and invoking pragma SUPPRESS(DISCRIMINANT_CHECK) to be able to access storage without regard to the discriminant value. This method requires that the various components of the record be mapped to storage exactly the way the FORTRAN environment does. Hence the implementation-dependency of this method, which is illustrated by the following example.

```
  COMMON /AREA1/ DIST, RADIUS, NTRI
  :
  COMMON /AREA1/ DIST, RADIUS, FLAGS
  CHARACTER*1 FLAGS(4)
```

would translate in Ada to

```
  with FORTRAN_ENVIRONMENT;
  use FORTRAN_ENVIRONMENT;
  package COMMON_AREA1 is
    type AREA1_RECORD(TAG: POSITIVE := 1) is
      record
        case TAG is
          when 1 =>
            DIST: REAL_FORT;
            RADIUS: REAL_FORT;
            NTRI: INTEGER_FORT;
          when 2 =>
            UNNAMED_1: REAL_FORT;
            UNNAMED_2: REAL_FORT;
            FLAGS: CHAR_ARRAY_1(1,1,4);
          when others =>
            null;
        end case;
      end record;

    pragma SUPPRESS(DISCRIMINANT_CHECK);

    AREA1_VARIABLE: AREA1_RECORD(1);
    -- The following renames permit the retention
    -- of the original COMMON variable names.
    DIST: REAL_FORT renames AREA1_VARIABLE.DIST;
    RADIUS: REAL_FORT renames AREA1_VARIABLE.RADIUS;
    -- etc.
  end COMMON_AREA1;
```

The second method is useful if pragma SUPPRESS is not implemented in the target Ada system. In this method, each COMMON area layout is defined as an independent record type along with an access type to that record. Using UNCHECKED_CONVERSION and the ADDRESS attribute, each record is forced to map to the same storage address. For example, the COMMON declarations used in the previous example are translated as follows:

```
  with SYSTEM, UNCHECKED_CONVERSION;
  with FORTRAN_ENVIRONMENT;
  use FORTRAN_ENVIRONMENT;
  package COMMON_AREA1 is
    type AREA1_RECORD_1 is record
      DIST: REAL_FORT;
      RADIUS: REAL_FORT;
      NTRI: INTEGER_FORT;
    end record;
    type AREA1_ACCESS_1 is access AREA1_RECORD_1;
    type AREA1_RECORD_2 is record
      DIST: REAL_FORT;
      RADIUS: REAL_FORT;
      FLAGS: CHAR_ARRAY_1(1,1,4);
    end record;
    type AREA1_ACCESS_2 is access AREA1_RECORD_2;

    function ADDR_TO_ACC1 is new
      UNCHECKED_CONVERSION
      (SYSTEM.ADDRESS, AREA1_ACCESS_1);
    function ADDR_TO_ACC2 is new
      UNCHECKED_CONVERSION
      (SYSTEM.ADDRESS, AREA1_ACCESS_2);

    AREA1_VARIABLE: AREA1_RECORD_1;
    AREA1_ACCVAR_1: AREA1_ACCESS_1 :=
      ADDR_TO_ACC1( AREA1_VARIABLE'ADDRESS );
    AREA2_ACCVAR_2: AREA1_ACCESS_2 :=
      ADDR_TO_ACC2( AREA1_VARIABLE'ADDRESS );

    -- The following renames enable the retention
    -- of the original COMMON variable names
    DIST: REAL_FORT renames AREA1_ACCVAR_1.DIST;
    RADIUS: REAL_FORT renames AREA1_ACCVAR_1.RADIUS;
    -- etc.
  end COMMON_AREA1;
```

The use of access variables to get at the COMMON data may result in a performance penalty on some implementations.

Style Issues

If validity and run-time performance were the

only criteria, a table-driven transliteration system could convert most FORTRAN programs to Ada. The limited abstraction capability of such a system would tend to carry the FORTRAN style over to the Ada output making the translation unsuitable for subsequent maintenance. This section abstraction techniques which capture the *intent* of the input statements and reconstruct it in Ada.

The following list of output refinement techniques represents a modest step in the direction of abstraction and reconstruction. Most of the refinements apply to single statements or simple sequences of statements. The techniques that apply are by no means limited to those presented in this paper, but the list is intended to illustrate the potential of the approach.

Naming in Ada

Names in the translation should be meaningful. Maintaining the original FORTRAN names whenever possible is important, but does not always yield acceptable Ada code. Due to a limitation of the language, FORTRAN names tend to be cryptic. The translator should provide for renaming of variables. Additionally, the translation is likely to generate new names, such as type names, which do not correspond to any names in the FORTRAN program. Such names should be derived from other related entities in FORTRAN and should be renamable by the user. For example,

```
SUBROUTINE SORT(A,N)
REAL A(100)
```

should generate a reasonable type name for A:

```
type ARRAY_A_IN_SORT is REAL_ARRAY_1(1..100);
procedure SORT(A: in out ARRAY_A_IN_SORT;
               N: INTEGER_FORT);
```

Some FORTRAN labels will translate to GOTO labels in Ada. Again, the user should be allowed to rename the labels with descriptive names.

Another form of renaming that can greatly improve the style of output code is grouping FORTRAN variables into a record structure in Ada. For example, it is not uncommon to have

```
INTEGER DOBDAY, DOBMON, DOBYR
```

in FORTRAN to represent the day, month and year of someone's date of birth. In Ada, a record structure would be a more natural choice here:

```
type DATE_OF_BIRTH_RECORD is
  record
    DAY, MONTH, YEAR: INTEGER_FORT;
  end record;
BIRTH_DATE: DATE_OF_BIRTH_RECORD;
```

The translator should permit such restructuring of data items at the user's option.

Control Flow Structures

The FORTRAN-77 range constructs for flow control, though richer than that of its precessors, is limited in comparison with Ada. For example, there are no loop constructs other than the DO-loop. It would, therefore, be worthwhile to attempt to improve the structure of flow control in the translation. Freak [3] describes several techniques for improving the structure when translating from FORTRAN to Pascal, all of which are applicable in the Ada context, with two main differences: (a) Ada provides the EXIT statement whereas Pascal does not; (b) Pascal provides the REPEAT..UNTIL statement whereas Ada does not.

Another simple analysis can provide a significant improvement to the output code. In FORTRAN-77, labels are permitted on any statement regardless of the need. Translating each executable statement label to a GOTO label in Ada could result in poor output style. Labels that are not referenced should be eliminated in the translation. It should be noted that a label (e.g., the target of a DO) may have a reference in FORTRAN, but the same may not be true in Ada. Thus, the criterion should be to eliminate labels that have no significance in the translation.

Inline Input/Output

Most implementations of FORTRAN handle formatted input/output via run-time procedures that match up the format codes and the data stream interpretively. A literal translation would result in numerous calls to similar interpretive procedures in Ada. This not only is likely to obscure the the translation, but also could lead to performance degradation. Torsun and Robinson [2] describe techniques to implement FORTRAN i/o using noninterpretive procedures under suitable conditions. The same techniques can be used to invoke inline translation to Ada. A simple of example of this approach is given below.

```
      WRITE(6,100) N, (A(I),I=1,N)
  100 FORMAT(I4/(10I8))
```

can be translated to

```
with TEXT_IO; use TEXT_IO;
:
PUT(N, WIDTH=>4);
NEW_LINE;
for I in 1..N loop
  if I /= 1 and then I mod 10 = 1 then
    NEW_LINE;
  end if;
  PUT(A(I), WIDTH=>8);
end loop;
NEW_LINE;
```

DO loops

While simple forms of the FORTRAN DO-loop map readily into Ada's for-loop structure, the translation of the general case is more complicated. A number of reasons exist for the complexity: (a) The FORTRAN loop variable may be of the type INTEGER, REAL or DOUBLE-PRECISION whereas the Ada for-loop index must be a discrete scalar which excludes the real and double-precision types. (b) In Ada, the scope of the loop index is the same as the body of the loop, whereas in FORTRAN the loop variable is either a local, COMMON or parameter variable. (c) The Ada for-loop index must be incremented (or decremented) in unit steps, whereas FORTRAN DO-loop index can be stepped by any non-zero value.

The general case of the FORTRAN DO-loop is translated as illustrated below.

```
     DO 10 I = init, fini, step
       :
  10 CONTINUE
```

translates to:

```
  declare
    STEP_I: typeof(I) := step;
  begin
    I := init;
    for LOOP_I in
      1..INT((fini-I+STEP_I)/STEP_I) loop
      :
      I := I+STEP_I;
    end loop;
  end;
```

The above translation scheme is too cumbersome for the simple cases which are far more frequent. If output style is important, the translator must work toward generating a more readable translation whenever possible. The following considerations will help achieve this goal.

1. Eliminating the block statement.
   The declare..begin..end sequence can be eliminated if the step expression is a simple variable or constant, or is omitted. The temporary variable STEP_I then is replaced by *step* itself.

2. Simplifying the iteration count expression.
   When *init*, *fini* and *step* are all static (which is most often the case) the iteration count expression can be simplified to a constant. Simplification is possible even when one or two of the parameters are nonstatic expressions.

3. Eliminating duplicate loop indexes.
   The FORTRAN loop variable (I) and the Ada loop index (LOOP_I) may be identified as one, under certain conditions: (a) both indexes are integers, (b) the step is +1 or -1, (c) the FORTRAN index is not a COMMON variable, and (d) there are no transfers out of the loop. Although the conditions seem quite restrictive, they are frequently met in practice.

The following examples illustrate the output style improvement that can result from an application of the above refinements:

```
C     EXAMPLE 1: THE GENERAL CASE.
      DO 10 I=N,M,K+MYFUN(I)
   10 CONTINUE
C     EXAMPLE 2: DECLARE BLOCK ELIMINATED.
      DO 20 X=10.1,0.0,-0.05
   20 CONTINUE
C     EXAMPLE 3: SINGLE LOOP INDEX.
      DO 30 J=N,2,-1
   30 A(J) = A(J-1)
```

is translated to

```
-- Example 1: The general case.
   DO_10_I:
   declare
     STEP_I: INTEGER_FORT := K+MYFUN(I);
   begin
     I := N;
     for LOOP_I in
         1..INT((M-N+STEP_I)/STEP_I) loop
       I := I+STEP_I;
     end loop;
   end DO_10_I;
-- Example 2: Declare block eliminated.
   X := 10.1;
   for LOOP_X in 1..203 loop
     X := X-0.05;
   end loop;
-- Example 3: Single loop index.
   for LOOP_J in reverse 2..N loop
     A(LOOP_J) := A(LOOP_J-1);
   end loop;
   J := 1;  -- FORTRAN-77 requires this.
```

Parameter Modes

FORTRAN parameters are passed by reference. When an actual parameter expression is not a simple variable, the reference applies to a temporary variable containing the value of the expression; otherwise the reference is to the actual parameter variable. One simple way to translate a subroutine with parameters is to declare each parameter as an IN OUT parameter; in translating the calls, the actual parameters that are expressions are passed via temporary variables. This technique, which is employed by the Slape-Wallis converter [1], has two problems: (a) it is not applicable to functions, and (b) it constitutes merely an acceptable solution rather than an exploitation of the Ada feature which requires an explicit declaration of the intended mode of parameter passing. Instead, the translator should attempt to determine the exact mode from the information available to it. It is always possible to determine the precise mode -- IN, OUT or IN OUT -- for a given formal parameter by analyzing the body of the subprogram,

provided the same knowledge is available for all subprograms to which the formal parameter is being passed as an actual parameter. The mode IN is applicable if and only if the formal parameter is never assigned a value by an assignment statement nor passed as an actual parameter against an OUT or IN OUT formal parameter. The mode OUT is applicable if and only if the formal parameter is never referenced in an expression. The mode IN OUT is applicable if the modes IN and OUT are not applicable.

The only difficulty then is having the required knowledge regarding the other subprograms being invoked from the one under translation. The present approach gets around this by requiring any subprogram invoked by the current subprogram to: (a) have been translated earlier in the same execution of the translator, or (b) be described by an annotation declaring the correct parameter modes. The user needs to provide the information via an annotation only in the rare cases involving subprograms which are not to be translated for some reason or which invoke each other.

Identifying Constants

FORTRAN-77 provides for the declaration of constants using the PARAMETER statement. However, the PARAMETER statement cannot be used to declare constant arrays. It is common, therefore, to declare constant arrays as variables and initialize them via DATA statements, as illustrated below.

```
INTEGER MAXTMP(12)
DATA MAXTMP/45,50,68,79,2*100,98,3*75,60,53/
```

In Ada, there may be a significant improvement in run-time performance of the program if such arrays are identified as constants. In general, identifying constant entities as such improves both readability and performance. The nature of analysis required to make this identification is similar to that required for determining parameter modes. A "variable" is determined to be a constant if and only if it never is modified by assignment and is not passed as an actual parameter against an OUT or IN OUT formal parameter. Such a variable naturally would be initialized by a suitable DATA statement. Thus, the above declaration of MAXTMP would be translated to:

```
MAXTMP: constant INTEGER_ARRAY_1(1..12) :=
  (45,50,68,79,100,100,98,75,75,75,60,53);
```

The techniques listed above, while helping to improve the style considerably, cannot exploit the entire range of Ada constructs in the translation. For example, the use of range constraints and associated exception handlers to deal with out-of-range conditions would require considerably more analysis and user-input than represented in the techniques presented in this paper.

Conclusion

Automatic translation of FORTRAN to Ada is feasible for a large subset of FORTRAN, especially the popular dialect FORTRAN-77. While a valid translation can be constructed readily from a table-driven transliteration system, the task of producing maintainable code at the output presents a greater challenge. In the effort to improve the style of output code, this paper has presented an approach that is based on abstraction and reconstruction rather than on simple transliteration. Even with the conventional analysis techniques employed, the improvement is shown to be significant. Knowledge-based translators capable of global abstraction and expert reconstruction are just a step ahead.

References

1. Slape, J. K., Wallis, P. J. L. "Conversion of FORTRAN to Ada using an Intermediate Tree Representation," *The Computer Journal*, vol. 26, no. 4, pp. 344-353, 1983.

2. Torsun, I. S., Robinson, S. K. "Non-'Interpretive' FORTRAN input/output," *Software --Practice and Experience*, vol. 7, pp. 205-213, 1977.

3. Freak, R. A. "A Fortran to Pascal Translator," *Software--Practice and Experience*, vol. 11, pp. 717-732, 1981.

4. Wallis, P. J. L. "Automatic Language Conversion and its Place in the Transition to Ada," *Proceeding of Ada International Conference*, Cambridge University Press, pp. 275-284, 1985.

5. Wichmann, B. A., Meijerink, J. G. J. "Converting to Ada Packages," *Proc. Third Joint Ada-Europe/AdaTEC Conference*, Cambridge University Press, pp. 131-139, 1984.

6. *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, American National Standards Institute, 1978.

Dr. V. Santhanam is an associate professor of Computer Science at Wichita State University. He joined WSU after earning his Ph.D. in Computer and Information Science from Ohio State University in 1975 and his B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, Kanpur, India in 1971. He is a member of the ACM and the IEEE Computer Society.

VERIFICATION OF DIANA PRODUCERS AND DIANA CONSUMERS

Carl F. Schaefer

Intermetrics, Inc.

A Diana verification system presupposes a complete specification of Diana, which does not now exist. Further requirements on a Diana verification system are that it be independent of the particular function and independent of the particular implementation of the Diana producer or consumer being verified. A verification system is justified only if it furthers the goals of the Diana program. In this light, the Diana standard should be made more concrete by specifying a user interface to a Diana environment, including a particular set of Ada access routines for pre-defined Diana nodes and attributes as well as a definition capability for user-defined nodes and attributes.

1 Introduction

Once a standard is widely recognized and accepted, and once there is an accepted means of verifying conformance with the standard, it is a safe investment for a developer to commit resources to build a tool that conforms to this standard. But if the standard is not widely recognized and accepted, or if there is no accepted means of verifying conformance with the standard, the investment carries considerably more risk. In the case of Diana, the latter unfortunately applies. Most Ada compiler projects have not used Diana, and those that have used Diana (AIE, ALS, DG/Rolm, Rational, to name several) have used divergent versions of Diana. Furthermore, there is no accepted means of verifying conformance to a standard Diana. An important part of any plan to achieve wider acceptance of Diana would be developing the capability to verify that a tool that advertises itself as a Diana producer or consumer is indeed a producer or consumer of the Diana defined in the reference manual.

There are three important issues in a Diana verification program.

- Completeness of the Diana specification

  Unless there is somewhere a document specifying completely the structure and semantics of Diana, it is not possible to embark on a verification program.

- Tool-independence of the verification system

  The purpose of a Diana verification system would be to verify that in one of its functions a tool behaves in a certain way. The verification system should be neutral with respect to any other function of the tools.

- Implementation-independence of the verification system

  The Diana specification imposes no specific implementation. Therefore, a verification system must not presuppose any particular range of implementations. This is a very challenging requirement on the verification system.

2 Completeness of the Diana Specification

The current version of the Diana Reference Manual [3] uses several means to specify Diana: an IDL [1, 2] structure description, comments in the IDL structure description, diagrams, and running English text.

The IDL structure description is analogous to Appendix E of the Ada Language Reference Manual, the syntax summary of Ada. Just as the BNF of Appendix E specifies the syntactic structure of all legal Ada programs, so the IDL of the Diana Reference Manual specifies the graph structure of intermediate forms of legal Ada programs. But just as there are more Ada restrictions than can be captured in BNF, so there are more Diana restrictions than can be captured in IDL. To take a very simple example, the IDL

```
type =>
    as_id        : ID,
    as_type_spec : TYPE_SPEC;

ID ::= type_id;

type_id =>
    sm_type_spec : TYPE_SPEC;
```

says:

- that a type node has the attributes as_id and as_type_spec
- that type_id is a kind of ID
- that a type_id node has the attribute sm_type_spec
- that the type of the attribute as_id is ID
- that the type of the attributes as_type_spec and sm_type_spec is TYPE_SPEC

What the IDL does not say is that the as_type_spec of a type node must have the same value as the sm_type_spec of the as_id of the type node. This is analogous to static semantic restrictions of Ada such as the restriction that in an assignment statement the named variable on the left-hand side of the statement and the expression on the right-hand side of the statement must be of the same type.

The Diana Reference Manual does specify some of the static semantics of Diana, but it does not do so systematically. The means chosen to specify the semantics of a particular attribute depends in part on the kind of attribute. Diana attributes are divided into four classes, each class having its special prefix:

- lexical (lx_) attributes record lexical information such as source position (lx_srcpos : source_position; lx_symrep : symbol_rep)
- structural (as_) attributes represent the parse tree
- semantic (sm_) attributes represent miscellaneous semantic information
- code attributes (cd_), of which there is only one (cd_impl_size : integer), provide target-machine-specific information

The values of the lexical attributes and of the single code attribute, cd_impl_size, are specified informally in Section 3.10.1 "Summary of Attributes". The following extract, for example, specifies the value of lx_prefix:

> lx_prefix is of type Boolean, indicates whether a function call was written using prefix (True) or infix (False) notation, see 3.3.4.

Section 3.3.4, referred to in this extract, gives justification for the inclusion of this attribute in Diana:

> Diana records whether a function call was made using infix of prefix notation through the lx_prefix attribute. This information is necessary for subprogram specification conformance rules (Section 6.3.1 of the Ada LRM).

For lexical and code attributes, this means of specification is adequate.

The structural attributes are not included in the "Summary of Attributes". However, the order of specification of the Diana nodes and attributes, together with the Ada syntax included as comments in the IDL specification, makes clear the intended static semantics of these attributes. Thus, to resume the earlier example, the following fuller extract from the manual makes it clear what the intended values of the as_ or structural attributes are.

```
-- Syntax 3.3.1.A
-- full_type_declaration ::=
--     type identifier [discriminant_part] is
--     type_definition;
--

type => as_id        : ID   -- a "type_id",
                            -- "l_private_id" or
                            -- "private_type_id"
        as_dscrmt_var_s : DSCRMT_VAR_S,
                -- discriminant list, see 3.7.1
        as_type_spec    : TYPE_SPEC;
```

Like the lexical attributes, the semantic attributes are included in the "Summary of Attributes". The entry for sm_type_spec is not untypical:

> sm_type_spec denotes the specification which belongs to a type identifier; for private and incomplete types, see Section 3.5.1, for tasks and task body identifier, see Section 3.5.5.

The summary specification of semantic attributes is often clarified, narrowed, or reinforced by diagrams and running English commentary in Section 3 "Rationale". Thus, the diagram reproduced in Figure 1 illustrating an incomplete type makes clear the intended value of sm_type_spec.

While values of many of the semantic attributes are adequately specified by this mix of means, there are numerous gaps. The designers of Diana were fully aware of the shortcomings in the specification of static semantics and devoted Section 1.1.4 "Specification of Diana" to a discussion on how to remedy the problem (see also [4]).

For a few examples of incompleteness in the current Diana Reference Manual, we turn to aggregates. The representation of aggregates in Diana is complicated, and it is not surprising that the Diana Reference Manual is less than complete in specifying the static semantics of Diana in this area. We will look at two examples: the attribute sm_normalized_comp_s and the representation of subaggregates.

Aggregates have the following structural description in IDL:

```
EXP ::= aggregate;
aggregate =>
    as_list                 : Seq Of COMP_ASSOC,
    sm_exp_type             : TYPE_SPEC,
    sm_constraint           : CONSTRAINT,
    sm_normalized_comp_s    : EXP_S;
EXP_S ::= exp_s;
exp_s =>
    as_list                 : Seq Of Exp;
```

The "Summary of Attributes" tells us that sm_normalized_comp_s

> ... denotes the normalized list of values for a record aggregate or for a discriminant constraint, including default values.

But what does the sm_normalized_comp_s attribute in an aggregate node really look like? One might presume that this is list of the component expressions in the proper order; however, such a list is not always desirable or possible to construct. Consider the array aggregate (1..N => 0). The value of N is not statically determinable, therefore one cannot make a list of N zeros. A list containing a single zero without the corresponding range would be of no use. Consider the array aggregate (2..1000 => N, 1 => 0). It would not be very efficient to construct a list of 1001 elements, particularly when only two would suffice if the corresponding ranges were included. It seems that a normalized components list using named associations would be more appropriate; but then why does the IDL explicitly specify EXP_S rather than, say, named_s as the type of sm_normalized_comp_s?

The Diana Reference Manual is also unclear on the representation of a subaggregate of a multi-dimensional aggregate. In the following example, the Ada code will raise CONSTRAINT_ERROR during elaboration, but the compiler must be able to produce the code that will raise this exception.

```
type A is array ( integer range <>,
                  integer range <> ) of boolean;
c: constant A := ( 1 =>
                       ( 4..5 => false ),
                   ( 2 =>
                       ( 6..7 => false )
                   );
-- will raise CONSTRAINT_ERROR during elaboration
```

Presumably a subaggregate ("4..5=>false" or "6..7=>false" in the above example) is represented by a separate aggregate node, but it is not clear what value the sm_exp_type attribute of a subaggregate of a node should have; there is no explicitly declared type for the subaggregate. Should a new TYPE_SPEC node be created? We might also ask how the sm_constraint attribute of the subaggregate is related to the sm_constraint of the enclosing aggregate; is the second index constraint of the enclosing aggregate taken from the constraint of the first subaggregate (4..5) or from the second subaggregate (5..6). In this case it does not matter whether the first or second subaggregate contributes the constraint, but one must be chosen in order to detect the CONSTRAINT_ERROR.

This last example illustrates an interesting point about completeness of specifications: a specification must also make it clear when something is deliberately undefined. There are several ways for a complete Diana specification to accommodate undefined values:

- There could be a general rule that if something is not explicitly specified in the Diana Reference Manual, then its value is undefined. This is dangerous, since it would obligate the specification to be complete even for the values of obvious attributes (the structural attributes, for example).
- The manual could say that for particular attributes (in particular contexts) the value of the attribute is undefined.
- The manual could say that for particular attributes (in particular contexts) the value must be one of several things, the

choice among possibilities being undefined (e.g. in the above example, could say the index constraint of the enclosing aggregate must be a node shared by one of the enclosed subaggregates, the particular choice being undefined).

This is not the place to propose a scheme for formal specification of the static semantics of Diana (see [8] for one proposal). We note, however, that it simply makes no sense to speak of verifying that a purported Diana producer really produces Diana unless there is agreement on the static semantics of Diana. This is not to say that a completely formal specification of static semantics is required; carefully worded English prose may be sufficient. But something more than we now have is required. In the absence of this, it is is premature to speak of establishing a Diana verification system.

3 Tool-independence of the Verification System

The Diana Reference Manual wisely distinguishes between Diana producers and Diana consumers, with the following definitions (Section 1.1.3):

> In order for a program to be considered a Diana producer, it must produce as output a structure that includes all of the information contained in Diana as defined in this document. Every attribute defined herein must be present, and each attribute must have the value defined for correct Diana and may not have any other value. ... There is an additional requirement on a Diana producer: The Diana structure must have the property that it could have been produced from a legal Ada program.
>
> In order for a program to be considered a Diana consumer, it must depend on no more than Diana as defined herein. This restriction does not prevent a consumer from being able to take advantage of additional attributes that may be defined in an implementation; however, the consumer must also be able to accept input that does not have these additional attributes. It is also incorrect for a program to expect attributes defined herein to have values that are not here specified.

A Diana verification system should be tool-independent; that is, it should focus on the tool's use of Diana -- whether the tool was a Diana consumer or a Diana producer -- ignoring other functions of the tool. Furthermore, it should distinguish between a tool whose sole function is to translate Ada source into Diana (the front end of a compiler) and other Diana producers. In the case of the front end of a compiler, one would try to verify not only that the output is Diana (hence the representation of some legal Ada program) but also that the output is the correct mapping of the given input. In the case of other Diana producers, one would try only to verify that the output is Diana, without attempting to verify that the mapping from input to output is correct. In the case of a Diana consumer, one would try to verify that the tool accepts (i.e. does not issue any error diagnostics against) any Diana representation, but one would not attempt to verify that the tool produces the correct, non-Diana output for a given input. In the case of Diana consumers, there is no predefined unique product, and Diana verification should not become enmeshed in verification of some other function of a tool.

As a first approximation we might suppose that a Diana verification system would involve developing the following components:

- Input test cases for Diana producers.
- A means of verifying that the Diana output of Ada compilers is the correct mapping of the Ada source to Diana.
- Diana input test cases for Diana consumers.
- A means of verifying the Diana output of general Diana producers.

Several considerations lead to a simpler picture. First, for a Diana producer we do not know, in general, what the input to the tool is. The input could be, among other possibilities, Fortran source, Ada source, some other intermediate language, or Diana itself. Accordingly, it is probably simplest to declare the input test cases of the tool's own validation suite to be the input test cases for verifying that the tool is a Diana producer. That is, input test cases for Diana producers are not part of the Diana verification system. Second, if a Diana producer is also the front end of a compiler, it will have undergone the ACVC and it is therefore not necessary for the Diana verification system to check that the mapping from Ada source to Diana is correct.

From the foregoing, it appears only two components are required:

- A set of Diana inputs that systematically sample the set of all Diana representations.
- A Diana assertion checker -- a program that can check the truth of assertions about the structure and semantics of Diana representations output by a Diana producer.

The construction of the set of Diana input test cases is not particularly difficult provided there is a reliable Ada-source-to-Diana translator available; many of the test cases could be derived from those ACVC tests consisting of legal Ada. The construction of an assertion checker with any pretensions to completeness would be arduous.

4 Implementation-independence of the Verification System

The specification of Diana in the Diana Reference Manual is intended to be implementation-independent. The specification does not say how the nodes and attributes are to be encoded at the level of a programming language, and certainly says nothing about how they are to be encoded at the level of bits in memory or on a disk. Any program producing or consuming Diana will have a particular encoding of this information. How, then, can there be a single verification system to verify that any alleged Diana producer is really a Diana producer?

One solution is use a standard external representation of Diana in the verification system. The Diana Reference Manual (Section 5) provides an example of an external representation. In the verification system, we then verify not that a program is a Diana producer but that a pair consisting of the program and a compatible Diana writer (where compatible means understanding the same encoding) is a Diana producer. Likewise for the Diana consumer, we verify that the pair (program, compatible Diana reader) is a Diana consumer.

Figure 2 shows the resulting configuration for Diana producers, Figure 3 for Diana consumers.

5 Implications for the Definition of Diana

We have sketched a fairly simple-minded Diana verification capability. We might now ask whether this capability would promote the goals of the Diana program the way the ACVC promotes the goals of the Ada program.

Suppose there were a Diana producer DP and a Diana writer DWDP that translates the output of DP to the the standard external representation of Diana. Suppose further that DP has been verified in the following way (assume that DP is a Fortran-to-Diana translator; DP is therefore a Diana producer without being the front end of an Ada compiler):

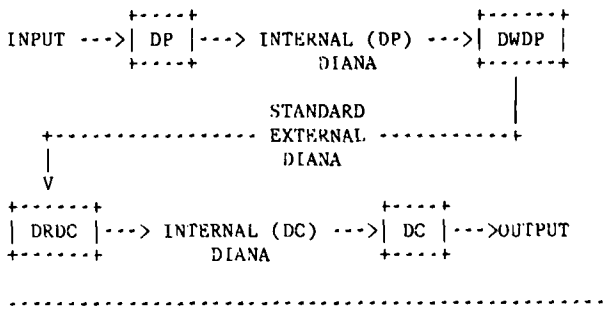
1. The original validation suite for DP is run through DP.
2. The resulting Diana output is run through DWDP.
3. The resulting external Diana is run through the Diana assertion checker.

Since DP is not the front end of an Ada compiler we do not need to verify the accuracy of the mapping of input to output.

Suppose now there is a Diana consumer DC (perhaps a target-independent optimizer) and a Diana reader DRDC. Suppose further that DRDC has been verified in the following way:

1. The implementation-independent test cases (in standard external Diana) are converted to internal Diana via DRDC.
2. The test cases (in internal form) are run through DC.

We now have two tools, DP and DC, the first of which is a verified Diana producer and the second of which is a verified Diana consumer. Furthermore, the tools can be run in conjunction provided that the output of DP is run first through DWDP and the result of that run through DRDC.



But this translation to and from standard external Diana is costly, and there is no reason to believe that DC can run directly on the output of DP. In this case it is clear that the tools actually do conform to the Diana standard; but it is not clear that the tools are more widely usable because they conform to the Diana standard. The verification program we have outlined above is in danger of becoming verification for its own sake while loosing sight of the original goal of Diana: to promote the development of inter-operable Ada-related tools by defining a standard representation for Ada programs.

For different programs to operate on common data, one of two conditions must hold:

- the programs must incorporate knowledge of how the data is represented; or
- the programs must call on a common set of access routines.

If the goal of Diana is to enable a given program to operate on (produce and/or consume) Diana representations across N different Diana environments, then either

- o there must be N different versions of the program, each incorporating the knowledge of how Diana is represented in one environment; or
- o all N environments must provide the same set of Diana access routines.

The first alternative is wasteful of human resources; furthermore, market considerations (the small number of potential users per version) might inhibit the development of tools. The second alternative, by standardizing at a greater level of detail, can easily inhibit technological progress; furthermore, it may be wasteful of machine resources by precluding certain efficient implementations. But if we are to be serious about the goals of Diana, the choice is clear: a Diana standardized to the level of access routines (presumably in Ada) is required. The Diana Reference Manual currently specifies Diana in an abstract manner: using IDL and associated text for specification of the static semantics. An example of a possible set of routines is given in Section 4 of the Diana Reference Manual, but the example is not proposed as a standard Diana interface. Serious consideration should be given to augmenting the standard with a more concrete level of specification: a set of Ada routines providing the operations of creating and deleting nodes, reading and setting attributes, walking sequences, and so forth.

If Diana is defined to be a unique set Ada of access routines with clearly specified semantics, we can speak of a Diana environment existing independently of Diana producers and consumers. It would then be possible to separate the verification of Diana environments from the verification of Diana producers and consumers and to dispense with the requirement for Diana writers and Diana readers.

This was the approach taken in the VHSIC Hardware Description Language Program [6, 7]. The intermediate form of VHDL is called IVAN, for Intermediate VHDL Attributed Notation. In addition to an IDL structure description and an informal English specification of semantics, IVAN is defined by a set of Ada access routines for creating and deleting nodes, reading and setting the values of attributes, determining the kind of a node, walking sequences, and so forth [5]. The Ada package IVAN provides a standard environment; any IVAN-producing tool that uses this standard environment is guaranteed to be inter-operable with any IVAN-consuming tool that also uses the standard environment. There are currently four VHDL tools (Analyzer, Reverse Analyzer, Simplifier, Simulator) under development that use the standard IVAN environment.

Taking seriously the goals of Diana entails considering another change to the Diana standard -- provision for user-defined extensions. The definitions of Diana producer and Diana consumer were carefully phrased to allow a Diana producer to output a structure containing non-Diana nodes and attributes and for a Diana consumer to operate on a structure containing non-Diana nodes and attributes. Experience has indicated that real Diana tools do need additional nodes and attributes. To take one example, the front end of the AIE defines some 532 node kinds and some 638 attributes, in contrast to the 170 node kinds and 131 attributes defined in the Diana Reference Manual. If the environment does not provide for additional nodes and attributes, each tool will have to provide its own, separate implementation. But if Diana is defined to be a set of access routines, and if Diana producers and consumers are to be portable across Diana environments, how are the additional nodes and attributes to be accommodated? One possibility is to require that a Diana environment provide a definition capability allowing a using tool to declare additional nodes and attributes that can be accessed in exactly the same way as the predefined Diana nodes and attributes. It is premature to propose this addition to the Diana standard; but a study of how to integrate a data definition capability with the pre-defined Diana is in order.

6 References

[1] J.R. Nestor, W.A. Wulf, D.A. Lamb, IDL - Interface Description Language: Formal Description, Carnegie-Mellon University, Computer Science Department, CMU-CS-82-002, June 1982.

[2] D.A. Lamb, Sharing Intermediate Representation: An Interface Description Language, Carnegie-Mellon University, Computer Science Department, CMU-CS-83-129, 1983.

[3] K.J. Butler and A. Evans, Jr. DIANA (Descriptive Intermediate Attributed Notation for Ada) Reference Manual, Revision 3, Tartan Labs, Inc., Report No. TL-83-4, Pittsburgh, Pa., February 22, 1983.

[4] K.J. Butler, "Diana Past, Present, and Future," Ada Software Tools Interfaces, ed. P.J.L. Wallis, Springer-Verlag, 1984.

[5] Intermetrics, Inc., VHDL Design Library Specification, IR-MD-019-3, 30 March 1985.

[6] Intermetrics, Inc., VHDL Language Reference Manual, Version 7.2, IR-MD-045-2, 1 August 1985.

[7] Intermetrics, Inc., VHDL Support Environment System Specification, IR-MD-024-3, 15 March 1985.

[8] G. Persch and M. Dausmann, "The Intermediate Language Diana", Ada Software Tools Interfaces, ed. P.J.L. Wallis, Springer-Verlag, 1984.

[9] J. Uhl, "A Formal Definition of Diana", Ada Software Tools Interfaces, ed. P.J.L. Wallis, Springer-Verlag, 1984.

Carl Schaefer is program manager for the Diana/IDL configuration management project at Intermetrics, Inc. He was also responsible for the implementation of the VHDL Analyzer and is currently working on the design and implementation of a VHDL simulator. He has a PhD in Linguistics from Cornell University and an MS in Computer Science from American University. Mailing Address: Intermetrics, Inc., 4733 Bethesda Ave., Bethesda MD 20814.

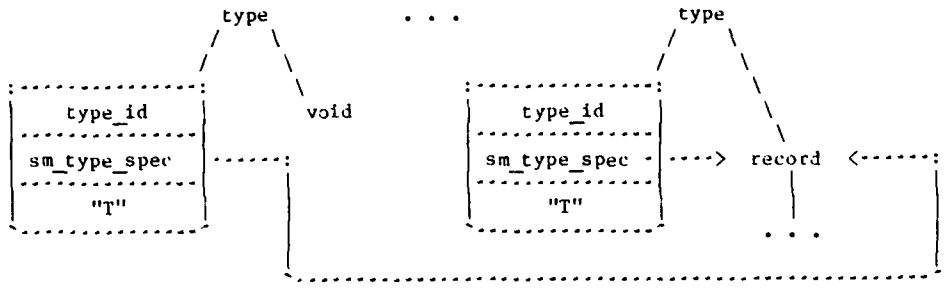


Figure 1: Incomplete Type

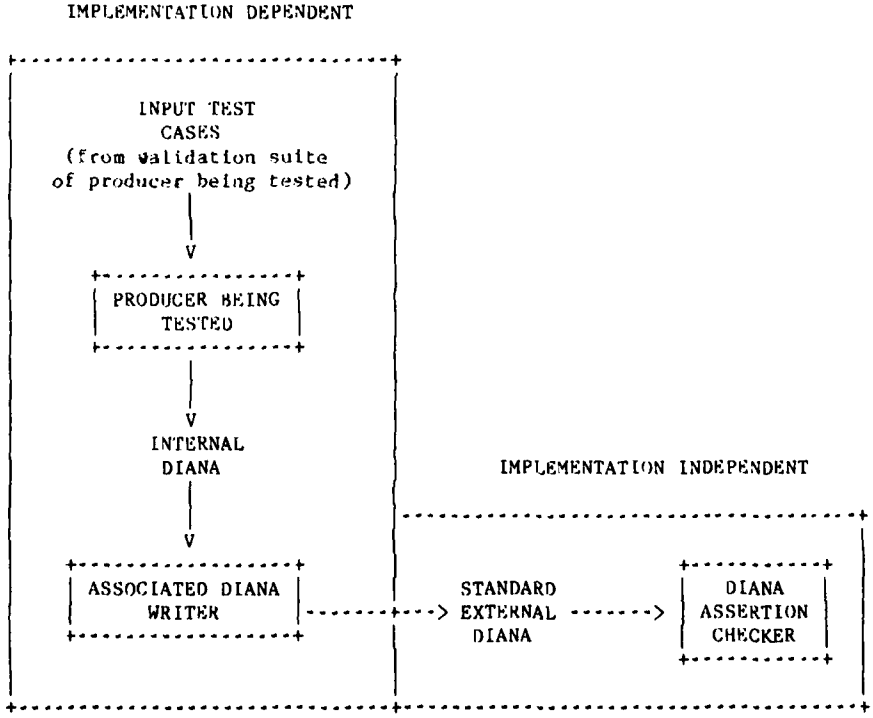


Figure 2: Configuration for Diana Producers

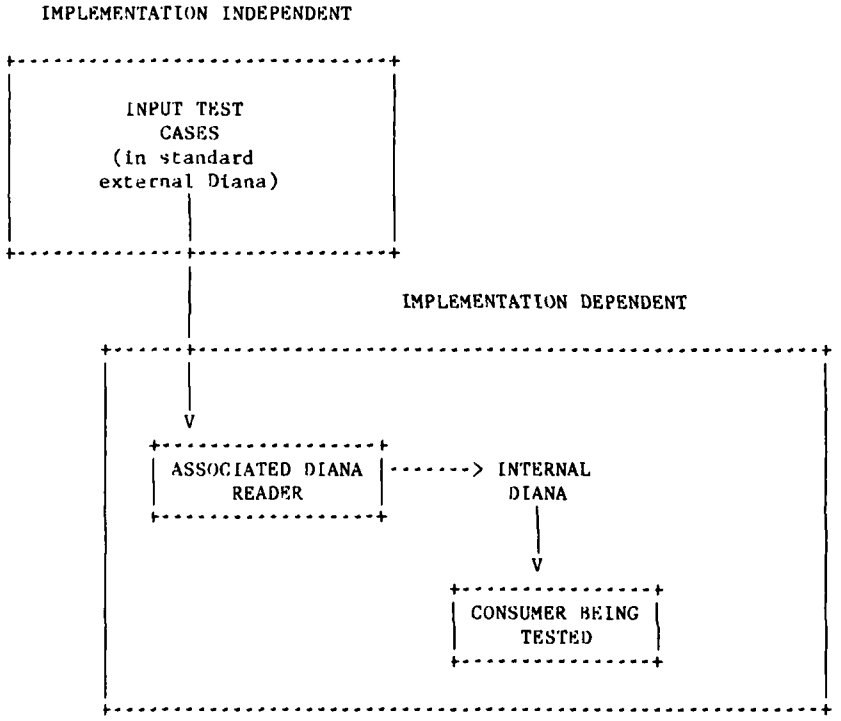


Figure 3: Configuration for Diana Producers

THE BACK-END OF A MULTI-TARGET COMPILER

Gil De Bartolo and Ron Richards

Intermetrics, Inc., Cambridge, Massachusetts

Abstract

This paper discusses the implementation of the back-end of an optimizing Ada compiler which is able to defer the binding of its target machine until compiler execution time. There are several advantages to this deferral. Any compiler enhancements are immediately available to all targets. A consistent user interface is guaranteed across all targets. The cost of adding a new target is minimized. The apparant disadvantages of this approach involve performance as such a compiler might be slower and bigger. We present an implementation in Ada which minimizes these disadvantages yet retains the advantages of implementing in Ada (specifically, strong type checking and range checking). Further our approach allows the mechanical construction of a smaller, faster compiler supporting only one target from the multi-targeted compiler.

Introduction

This paper describes the implementation of a multi-target compiler. That is, a compiler able to defer the binding of its target until compiler execution time. The first two sections put the back-end into perspective first by briefly describing the front and middle-ends of the compiler. The third section is concluded with a discussion of BILL, the intermediate representation which the back-end translates to machine code. The next section describes what we had hoped to achieve by constructing a multi-targeted compiler. The fifth section is the heart of the paper. In that section we describe the four different approaches that were employed to implement the multi-targeted compiler. Which of the four approaches was employed depends of the characteristics of the phase. The material on the implementation approaches assumes a knowledge of Ada. The paper is concluded with a brief discussion of the success of this endeavor.

Overview of the Compiler

The Intermetrics Ada compiler has five phases which process a program prior to the six back-end subphases. In this section we will present a brief overview of the first five phases: the front and middle ends of the compiler.

*The authors are indebted to Mark Davis, Tucker Taft, and Dennis Struble for substantial contributions to both the design of the implementation presented and commenting on earlier drafts of this paper

The front-end of the compiler is comprised of the first two phases, tree build and semantics. The tree builder builds an abstract syntax tree for the Ada input program which the semantics phase translates into a DIANA [1] representation of the program. There are no significant target dependencies in these phases.

The middle of the compiler comprises three phases: storage, expand and flow. We will briefly discuss each of these phases.

The storage phase adorns the DIANA representation of the program with information about how objects in the input program should be allocated storage.

The expand phase reads the DIANA representation of the program and translates this into a interface description language (IDL) [2] called BILL. In this low-level representation all calculations are made explicit.

The flow phase massages the BILL tree performing target independent optimizations. For example, redundant constraint checks and inaccessible code are deleted. Opportunities to reuse previously calculated values (constant sub-expressions) are recognized by this phase. Flow also decorates the BILL tree with access mode sets. An access mode is an indication of how a node (and possibly some its descendants) may be addressed.

All three of the middle phases do contain some target dependencies. For example, the layout of a record may well be affected by what the target architecture supports. Similarly, the BILL produced by expand will depend significantly on the target-specific run-time model chosen. Since the focus of this paper is not on target dependencies in the middle-end, we won't elaborate here on how these dependencies are dealt with. However, it is accurate to say that the solutions employed in the middle phases are not unlike some of those employed in the back-end which will be discussed in detail.

Overview of the Back-End

The back-end of the compiler consists of six phases. One could view the major task of the back-end phases of the compiler as translating the program from the low-level intermediate representation produced by the middle-end to either an assembly language like representation of the program or an object module representation of the program. The representation produced by the middle phases is in BILL. Before we discuss the roles of each of the back-end phases we will present an overview of BILL.

Introduction to BILL

BILL is a tree structured language designed to meet three apparently conflicting goals. First, BILL makes explicit all calculations (including address calculations). Second, BILL abstracts away from the target specific detail which is inessential in producing high quality code. Third, BILL maintains a "structured" representation of the program.

Since BILL is a tree-structured language, the elements of the language are nodes and these nodes have attributes (which are often other nodes). Examples of typical BILL nodes for arithmetic operations are bl_int_plus, bl_int_minus, bl_flt_plus, and bl_flt_times. Examples of BILL nodes pertaining to address calculations are bl_offset and bl_known_frame_offset. Some BILL nodes related to flow of control would be bl_case, bl_handlers (for exception handlers), bl_if, bl_loop and

bl exit. For calling subprograms we have bl proc call and bl func call as example nodes.

Two kinds of calculations are commonly required in a program. The program text itself specifies that certainly calculations be performed. For example, a variable may be assigned the sum of two other variables or a variable might be incremented. Of course, the list is endless and not very interesting. Less obvious (but possibly more prevalent), are calculations required to address objects. For example, a local object may be a certain offset from the frame pointer or a record component in an array of records may require a very involved expression for addressing. In the BILL representation of the program both of these types of calculations are explicit.

There are many operations which require several machine instructions but which are best viewed as atomic operations even in a low-level language like BILL. For example, raising a user defined exception or calling another subprogram may require several target machine instructions. Yet each of these operations is a single node in BILL (bl raise and bl proc call, respectively). In this way, BILL abstracts away from inessential target dependencies.

BILL maintains a structured view of the program by using nodes for structured flow of control constructs like "if then else", "loop", and "case". By keeping these constructs and not translating immediately into conditional branches, analysis of the program for optimization and register allocation is made easier.

Back-End Subphases

The six subphases of the back-end are in the order in which they execute, vcode (virtual code generation), tnbind (bind temporary names to machine resources), codegen (code generation), jump (jump optimization), branch (branch resolution), and objgen (generate an object module). We will discuss each phase briefly.

Vcode creates temporary names (Tns) for compiler created temporaries. When two Tns are known to have the same value (at least initially), vcode sets up a preference relation between the two Tns. If two Tns are preferenced, tnbind will attempt to allocate them to the same machine register. The output of this phase is the BILL tree supplied by the middle phases of the compiler adorned with Tns representing intermediate results. Each Tn has an attribute which indicates what type of register would be best for storing this value as well a list of preferences to other Tns.

Tnbind allocates the Tns to machine registers. Before the allocation can be made tnbind must determine which Tns conflict (two Tns which hold values at the same point in a program are said to conflict). After this phase has run, each Tn has been allocated to a machine register or spilled.

Codegen generates code given the allocation of machine registers to Tns. A table describing the code to emit for a particular sequence of BILL nodes and the allocation of the Tns used by those nodes is consulted to determine what code to generate. While the contents of the table are obviously target dependent the manner in which information is gleaned from the table is not. The implementation of this table will be discussed in section 5.1. The output of this phase is the IDL CODE, which is very similar to assembly language.

Jump processes the CODE representation of the program attempting to optimize jumps to jumps, unlabeled code following an unconditional branch, and jumps around jumps. The output of this phase is also CODE.

Branch processes the CODE representation to build the exception map, to pool literals, and to choose between long and short jumps. After this phase has run every CODE node has a known location with in the module.

Objgen processes the CODE representation and builds an object module.

Goals of multi-targeting

Intermetrics had previously developed an easily retargetable compiler. As a separate effort we took advantage of our retargetable approach to implement the multi-target compiler. This compiler was designed to enhance our compiler development environment.

Intermetrics is simultaneously developing ada compilers targeted to many different architectures, to be hosted on several different operating systems. Each of the target compilers is developed in two stages. First, we build a cross compiler which is hosted on our development machine. After testing the cross compiler, we rehost it to the specific host operating system, normally using the cross compiler to rehost itself. The multi-target backend was proposed not only to reduce the costs of the the first stage, when many targets are developed and maintained on the same host machine; but also to reduce the whole life-cycle cost of a target. Rather than being forced to maintain several different compilers, we will essentially be maintaining just one compiler as bug fixes are immediately available to all targets.

Prior to this project, we had developed a retargetable backend. Periodically, separate teams would take a release of this backend and retarget it. Target dependencies, although well identified, existed at all levels of the backend sources. The existence of target dependencies in low level package specifications ruled out any sharing of the program libraries, or object modules accross the different targets. Even though most packages in the backend were target independent, the retarget teams found it very difficult to share fixes and enhancements. If the interface to packages changed, as often happens in a developing project, copying a single updated package often required copying several other changes. Keeping the different target backends up to date became very costly, both in manpower and computer resources.

To reduce the costs of simultaneously developing several compilers, we identified several goals for the multi-target backend. First, sources for all targets must be able to reside in the same program library. Second, we want to maximize the number of lines of shared code. Third, target dependent code should be easily changed without requiring recompilation of other sources.

Each compiler contract requires the delivery of a backend for a specific target. Each delivered backend should not have the runtime overhead of selecting a target. Thus, another goal of the multi-target backend, is that a single target backend be easily built from the multi-target sources.

Implementation of multi-targeting

As we discussed in the section presenting an over view of the back-end, many of the algorithms employed in the back-end are inherently target dependent. In this section we discuss how the "essential" target dependencies were dealt with. We ended up employing four different approaches to this problem. The first of these approaches could be summarized as reading in the target dependent information in at compiler execution time. This technique accounts for most of the target dependencies. Given the decision to read in information, there is no extra performance cost associated with supporting more than one target.

Tables initialized by reading a file

Both the vcode and codegen phases of the back-end process the BILL representation of the program with an aim toward pseudo code generation. The BILL tree supplied by the middle is processed by beginning at the root and matching as much of the tree as possible with a pattern chosen from the table of all matchable patterns. After the actions appropriate for the pattern matched are completed, the matching process is repeated on the unmatched children of the matched nodes. The children are processed in reverse execution order.

The algorithm used to decide whether a pattern matches a portion of the BILL input is target independent. We will discuss this algorithm as an excellent example of how essential target dependencies are deferred until compiler run-time.

Patterns are comprised of four different types of tokens: bill, operand, restriction tokens, and end_mark. The bill tokens are used to match the BILL node of the same name. The operand tokens indicate the value of an "Operand Class" which describes a set of sub-trees which could match this token. An operand class is actually a set of access modes. An access mode is an enumeration corresponding to the different ways data may be accessed on the target machine. The restriction tokens are used to describe a condition which must be true about the values of the nodes (or attributes of the nodes) if the restriction is to be true. All of the restriction tokens for a pattern must follow all of the bill and operand tokens. The end_mark token designates the end of a pattern.

Using this terminology, a pattern matches a sequence of BILL nodes if every node in the sequence matches. The conditions to determine whether a node matches a pattern token depend on the type of pattern token. If the pattern token is a bill token, then the token matches the corresponding node in the sequence of BILL nodes if and only if the node and token are of the same kind. If the pattern token is a restriction token, then the restriction token matches if and only if the restriction holds given the values of the attributes of the nodes in the sequence. If the pattern token is an operand token, then the token matches if either of the following conditions hold:

- the operand class includes an access mode which is included in the access mode set of the BILL node corresponding to the operand token.
- the operand class includes an access mode which represents a *direct* access to a register

Before we can present example patterns for different architectures we must define some access modes and operand classes. For this example we will use four different access modes, listed below

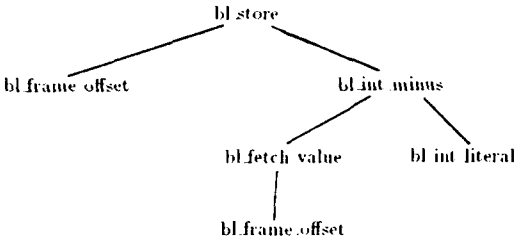
- r_a: address of a register (this is a direct access to a register)
- r_v: contents of a register (this is a direct access to a register)
- x_a: address of memory (using a base register and a displacement)
- x_v: contents of memory location (addressable through x_a)
- lit: an integer literal

We will also have four operand classes as shown below

- OCA: { x_a }
- OCB: { r_v }
- OCD: { r_a }
- OCD: { x_v, lit }
- LIT: { lit }

For our examples we will use the same access modes and operand classes for different target architectures. It is important to understand that this is for convenience in this exposition. In the compiler no such restrictions apply. In fact we will discuss below how different access mode and operand class enumerations are supported for different targets.

Consider the BILL tree below which represents decrementing a local variable:



We will present some example patterns for the 1750A and IBM 370 architectures. Recall that a pattern is a sequence of tokens. A bill token is represented by the name of the BILL node kind it matches. An operand token is represented by the a parameter number and operand class separated by a ':'. Restriction tokens are represented by the name of the restriction followed by the list of parameter values surrounded by parentheses. The end token is not explicitly represented as it is implicit in the terminating ';'.

For the 1750A architecture we would have a pattern like

```
bl_store $0:OCA bl_int_minus $1:OCB $2:LIT same(0,1)
   int_val(2,1);
```

This pattern matches the whole example tree. Lets see why. The first token matches since it is a bl_store and the root node is also a bl_store. The second token is an operand token which matches since the operand class OCA includes the access mode x_a (which transforms frame_offset). The third token is a bill token for bl_int_minus which matches since the corresponding BILL node is also a bl_int_minus. The fourth token is an operand token which corresponds to the first child of the minus node. This token matches since the operand class includes the access mode x_v (which transforms a fetch from a frame_offset). The next token is also an operand token. This time corresponding to the other child of the minus node. The token matches since the operand class contains lit (which transforms the int_literal). The last two tokens are restriction tokens. The first test whether the operand zero represents the same location as operand one. The second restriction tests whether the value of the literal for operand two has value one. Since all the tokens match, the whole pattern matches. Hence for 1750A we can process this whole tree during one pattern match.

On an architecture like the IBM 370 this tree does not correspond to a single pattern. For the 370 we would have patterns like the following:

```
bl_store $0:OCC bl_int_minus $1:OCB $2:OCD ;
bl_store $0:OCC bl_int_minus $1:OCB $2:OCB ;
bl_store $0:OCC $1:OCD ;
bl_store $0:OCA $1:OCB ;
```

For the 370 matching this tree requires first matching the store to memory then matching the subtract one, and finally the fetch from memory (recall that we match BILL trees in *reverse* execution order).

The single target approach

We will ignore the multi-target issue for a moment and sketch how we would implement some of the important data structures and algorithms. For any particular target we might first set up the access mode and operand class enumerations. For example we could have:

```
type access mode is ( r_a, r_v, x_a, x_v, lit);
type operand class is ( oca, ocb, occ, ocd, lit);
```

We might have the information on access modes and operand classes in arrays of records. The declarations and objects might be:

```
type access_mode_record is record
    is_direct : boolean;
end record;

type access_mode_set is array ( access_mode ) of boolean;

type access_mode_array is array( access_mode ) of
    access_mode_record;


type operand_class_record is record
    contains : access_mode_set;
end record;

operand_info_table : array ( operand_class ) of
    operand_class_record := (
    oca => ((x_a=>true, others=>false)),
    ocb => ((r_v=>true, others=>false)),
    occ => ((r_a=>true, others=>false)),
    ocd => ((x_v=>true, lit=>true, others =>false)),
    lit => ((lit=>true, others=>false)));

access_mode_info_table : access_mode_array := (
    r_a => (true),
    r_v => (true),
    x_a => (false),
    x_v => (false),
    lit => (false));
```

In the actual implementation, the tables are in bodies and functions are defined to access them.

Before defining the data structure for patterns, we define some preliminary types which will be necessary for dealing with pattern records.

```
type token_type is (bill, restriction, operand, end_mark);

type restriction_type is (same, int_val);

type formal_digit is new integer range 0 .. 2;
```

The patterns would be set up as an array of records.

```
type pattern_range is new integer range 0 .. number_pat_nodes;

type pattern_record (token : token_type) is record

    case token is
        when bill =>
            kind : bl_bill.Kind;
        when restriction =>
            restriction_name : restriction_type;
            param_1 : integer;
            param_2 : integer;
            param_3 : integer;
        when operand =>
            operand_name : operand_class;
            formal : formal_digit;
        when end_mark =>
            null;
        end case;

    type pattern_array is array ( pattern_range ) of
        pattern_record;

    pattern : pattern_array;
```

Now lets consider what some code might look like to test whether a pattern matched a BILL tree. We will present a function which will take two parameters the tree to match and the start of the pattern. The function will return true if the tree matches a pattern. This function does not check whether the restrictions at the end of a pattern are satisfied (which is required for the whole pattern to match). It should be emphasized that the code below just sketches what the function might look like. The function relys on several more primitive functions. We assume Last_Child returns the number of children given the BILL node kind. similarly, child(i, tree) returns the i^{th} child of the tree. The function intersect_ams returns true if its two arguments (both access mode sets) have a non-null intersection. Contains_direct returns true if its argument contains an access mode which has is direct equal to true. The variable pattern_index is a global variable representing the next pattern token to check. It is advanced by the function advance_pattern_index when a pattern token is matched.

```
function match(
    tree : bl_bill.locator
    ) return boolean
is
begin
    case node(pattern_index).token is
        when bill =>
            if bl_bill."="(node(pattern_index).kind,
                bl_bill.kind(tree)) then
                advance_pattern_index;
                for i 1 .. bl_bill.Last_Child(bl_bill.Kind(tree))
                    loop
                    if not match(bl_bill.Child(i,tree)) then
                        return false;
                    end if;
                end loop;
                return true;
            else
                return false;
            end if;
        when operand =>
            if intersect_ams( bl_bill.fl_amset(tree),
                operand_info_table(node(pattern_range).
                operand_name).contains) then
                    advance_pattern_index;
                    return true;
            elsif contains_direct(operand_info(node(
            pattern_range).operand_name).contains) then
                advance_pattern_index;
                return true;
            else
                return false;
            end if;
        when restriction =>
            return false;
        when end_mark =>
            return false;
    end case;
end match;
```

The match function mimics closely our definition of matching presented above. If the pattern token is a bill token, the check if the BILL node kind corresponds. If they match, recursively check if the children match. The recursion ends when we reach an operand token. The test for matching an operand involves first checking if the BILL node and the operand class have an access mode in common. If there is no common access mode, a check is made for an access mode which directly accesses a register. The match function as presented does not check any restriction tokens which might follow the bill and operand tokens. These would be checked by the subprogram which called match if match returned true.

This section has sketched out the matching algorithm employing a target dependent approach.

The multi-target approach

The development of the matching algorithm as presented above was target dependent. In this section we will show how it would have to be modified to become target independent.

What is target dependent about our development in the previous section? A close look at the match function will indicate that the target dependencies are not there. Further a look at the declarations of the basic record types (operand class record, access mode record, and pattern record) also don't reveal any target dependencies. The target dependencies are in the most basic enumerations, the upper bounds of ranges (pattern range and formal digit), the contents of the objects (access mode info table, operand class info table, and pattern). Even if one were not planning to retarget the compiler, some of these target dependencies are likely to be a problem during development. We found that it was very difficult to get the operand class and access_mode enumerations just right. Each time the enumerations were changed substantial recompilations were required.

It is possible to maintain the spirit of the development shown above, yet defer (or hide) the target dependencies. We desire to keep the compile time strong type checking and execution time range checking of Ada. We are able to accomplish these goals by replacing our earlier type declarations with the variants shown below.

```
type access_mode_base is new integer;
subtype access_mode is access_mode_base range (
    0 .. access_mode_base(TGT.num_access_mode));

type operand_class_base is new integer;

subtype operand_class is operand_class_base range (
    0 .. operand_class_base(TGT.num_operand_class));

type formal_digit_base is new integer;
subtype formal_digit is formal_digit_base range
    0 .. formal_digit_base(TGT.num_formal_digit));

type unconstrained_access_mode_set is array (
    access_mode_base range <> ) of boolean;

subtype access_mode_set is unconstrained_access_mode_set(
    access_mode'first .. access_mode'last);

type unconstrained_operand_class_array is array (
    operand_class_base range <> ) of operand_class_record;

subtype operand_class_array is
    unconstrained_operand_class_array(
    operand_class'first .. operand_class'last);

operand_info_table : operand_class_array;
```

We have introduced a new package TGT which contains a function for each target specific constant (num access mode, num operand class, num pat nodes, and num formal digits). If there were only one target, the body of TGT might just contain values for each of these constants. This would minimize the recompilation required to change a basic enumeration. We did something a little different (but much better). The body of TGT reads an external file to get the values of the constants. In this way the ranges on the enumerations are not bound until compiler execution time. Hence a switch to the compiler triggers which input file is read and hence which set of bounds are selected.

To support many targets it is not sufficient to just change the bounds on enumerations. The contents of the objects like operand - info table, access mode info table, and pattern may also be initialized by reading a file. Hence at compiler execution time the target dependent ranges and objects are initialized.

We will briefly describe how the objects are initialized. Each object is initialized by reading its binary image. For example we use a statment like

```
Image_IO.Read(file => "patterns.ini",
    address => pattern'address,
    size => pattern'size);
```

to read the file "patterns.ini" and to place the indicated number of bits (size) at the indicated location (address). To insure that the binary image that is read is compatible with the binary image being written by the target specific code, we have only one declaration of the defining types. That is, there is only one package which defines types like access_mode_set, operand_class_array, and pattern array. There are several packages which have versions of the objects (access mode info table, operand class info table, and pattern). First there are the copies which are initialized at run-time in the compiler. In addition there is a copy in the file writer for each target. These target-dependent file writers are run once to produce an image of the object which is appropriate for that target. Each of these file writers "withs" the same set of packages that the compiler "withs" to define the objects.

Selecting code thru a case statement

Some phases of the back-end must contain some code which is target specific, even if the algorithm is largely not specific to any one target. The following approach deals with these phases.

The jump optimization phase is a good example of a phase which must contain some target specific code. The jump phase scans instructions in the CODE intermediate language searching for three sets of target independent patterns:

- chains of jump instructions
- dead code
- a conditional jump instruction immediately followed by an unconditional jump.

To find the patterns, the algorithm must distinguish conditional and unconditional jumps from the other instructions in CODE. This distinction is target dependent, as it involves looking at the opcode and condition code in the instruction.

Jump optimization's transformations of the CODE are both target dependent and independent. The first two optimizations are target independent. To eliminate jump chains, the second jump instruction is deleted and the first jump instruction is changed to point to the ultimate destination of the jump chain. The structure of the CODE language allows the label operand of the jump instruction to be changed by target independent code. Dead code elimination involves only deleting instructions and is also target independent. The third optimization requires some target dependent code to invert the conditional jump instruction. Deleting the unconditional branch is target independent.

Once the target dependent code is identified, it must be isolated into separate procedures. The interface to the procedures will be visible to the shared target independent code, and thus must be wide enough to accommodate all planned planned targets. We placed all of the target dependent procedures for the jump phase into a set of separate packages.

We implemented the target dependent packages in two layers. The first layer is a package called TGT_Jump. It's specification contains all of the target dependent procedures for the jump phase. All references to target dependent routines are made to this package. The procedure bodies in TGT_Jump consist of a single case statement which calls a specific implementation of the target dependent routine based on a global target identifier.

The second layer consists of the target specific implementations of the target dependent routines. Each target's implementation is in a separate package. The name of each target implementation package begins with a prefix that identifies the target, e.g. T_370_Jump or T_1750A_Jump. The specification of these packages is identical to the specification of TGT_Jump. The package bodies contain the actual implementations for the target.

The package prefixs "TGT_", "T_370_", "T_1750A_", ect. are used throughout the compiler to identify the switch package and the various implementations. It is trivial then to locate the target dependent parts of the compiler. We can add new targets to the jump phase by adding a target implementation package to the second layer and modifying the case statements in TGT_Jump to include the new target.

With this structure we can easily build a single target compiler. For example, to build a 370 only version of the jump phase, we substitute the body of T_370_Jump for the TGT_Jump body. Since the procedure specifications are identical, this substitution only requires changing the name of T_370_Jump to TGT_Jump. This substitution also breaks the link between the other target implementations and the rest of the back-end code, so other target implementations will no longer be included in the back-end executable.

Selecting code only when necessary

In both vcode and codegen, some parts of the BILL tree are walked by hand written code instead of by the pattern matcher. The hand written tree-walks process BILL nodes that cannot be expressed in patterns. These include BILL nodes whose resulting code is dependent on the runtime model of the target. For example, the processing of a procedure call involves knowledge of how parameters are passed. Targets which have similar runtime models can share the same tree walk procedures, while other targets must have separate tree walk procedures. The technique of selecting code only when necessary deals with situations where a given procedure may be target dependent.

We have defined a standard runtime model for procedure calls in the Ada compilers. In the standard model, a certain number of parameters, and the static back chain (SBC) are passed in registers. Additional parameters are passed in a block called the subprogram communication area (SCA). The address of the SCA is also passed in a register. We have implemented a shared procedure for walking procedure call nodes according to this model. The number of parameter registers and their specific assignments are all parameterized in the BILL. As long as a target shares this runtime model, it can use the shared routine.

One of our targets has a very different runtime model from our default model for procedure calls which takes advantage of a special instruction. As in the standard runtime model, some number of parameters are passed in registers. However, additional parameters are passed via argument pointers which follow the special instruction. This model differs enough from the standard to require a separate implementation.

Optional target dependent procedures are implemented in the same manner as in the case statement approach. We first identify all hand coded tree-walk procedures that might be target dependent. We then implemented the target dependent routines in the two layer approach with a switch routine at the top layer and target dependent implementations at the second layer. Each specification in the switch routine has a boolean variable which indicates if the routine is implemented the specified target. Target dependent specifications for procedure calls would be:

```
package TGT_Tree_Walk is

    dependent_procedure_call : boolean;
    procedure walk_procedure_call ( tree : bl_bill.locator );

    dependent_case_statement : boolean;
    procedure walk_case_statement ( tree : bl_bill.locator );

end TGT_Tree_Walk;
```

The second layer package specifications are identical to TGT_Tree_Walk, except that the boolean variables become boolean constants. If the constant is false, then the corresponding procedure body is null and will never be called. If the constant is true, then the procedure body must be implemented. The following tree_walk package specifies that only walk procedure call is implemented:

```
package T_SPC_TRG_Tree_Walk is

    dependent_procedure_call : constant boolean := true;
    procedure walk_procedure_call ( tree : bl_bill.locator );

    dependent_case_statement : constant boolean := false;
    procedure walk_case_statement ( tree : bl_bill.locator );

end T_SPC_TRG_Tree_Walk;
```

The package body of TGT_Tree_Walk initializes the boolean variables in its specification from the constant in the selected target package. As in the case approach, the procedures in TGT_Tree_Walk are simply case statements that call the selected target package.

The shared procedure for walking procedure call nodes first interrogates the boolean variable in TGT_Tree_Walk and then either calls the target dependent routine or executes the shared code:

```
procedure walk_procedure_call ( tree: bl_bill.locator ) is
begin
    if TGT_Tree_Walk.dependent_procedure_call then
        TGT_Tree_Walk.walk_procedure_call;
    else
        -- code for processing procedure calls
        -- according to standard runtime model.
    end if;
end walk_procedure_calls;
```

When we build a single-target back-end, the unneeded code will be eliminated automatically. To build a single target for the special target, we replace the specification and body of TGT_Tree_Walk by T_SPC_TRG_Tree_Walk. The boolean variable dependent_procedure_call becomes static, causing the unused code in the procedure walk_procedure_call to be removed by constant folding and dead code elimination in the host compiler. Thus, the single target back-end for the special target will contain only the code for walking procedure calls in the special way.

Using target independent tools in a target dependent way

As our final example of the approaches we employed to make a multi-target compiler we will consider the branch resolution phase. Branch resolution process the CODE representation of the program by choosing between short and long jumps as necessary, pooling literals, building the exception map, and breaking the code stream into chunks as necessary. There is a tremendous diversity across targets concerning these tasks. For example some targets don't have to choose between short and long jumps. On some targets the literals are pooled in the code stream while on other targets they are pooled in a read-only data area. Most targets do not require that the code be broken into chunks.

The approach we employed for this situation was to create a package of utilities upon which target dependent drivers for the branch resolution phase could draw. In our utilities package we have routines dealing with literal pooling. These routines are called to create a new literal table, free an existing table, pool a literal, and to output the literal pool (the code stream which holds the pool is a parameter of this procedure). The utilities package also contains procedures to update the exception map and to include the exception map in the code stream. Other frequently used procedures which are included in the utilities package is a procedure to include an instruction (and its operands) in the current chunk and a procedure to put the following instructions in a different code stream. These procedures are implemented in a target independent fashion. Only the drivers which call them are target dependent.

Conclusion

In section 4 we discussed the goals of constructing a multi-targeted compiler. As discussed there, a major goal was to reduce the cost of retargeting our compiler. This goal has been achieved. Retargets are now possible within a fraction of the effort required previously.

This endeavor would not have been considered successful if the performance of the compiler were significantly worse than a single target compiler. In fact, the results have been very encouraging. The multi-targeted compiler seems to be essentially as fast as the single targeted compiler and only about 10 percent larger.

References

[1] G.Goos, W. A. Wulf, A. Evans Jr and K J. Butler (editors). DIANA, An Intermediate Language for Ada (Revised version). Lecture Notes in Computer Science No. 161, Springer-Verlag (Berlin, Heidelberg, New York, Tokyo) 1983.

[2] J.R. Nestor, W. A. Wulf, D. A. Lamb. IDL-Interface Description Language:Formal Definition. Technical Report CMU-CS-81-139. Carnegie-Mellon University, Computer Science Department, June 1982, Revision 3.

Automated Drawing of Data Structure Diagrams

Prabhaker Mateti & Gerald M. Radack
Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106

Abstract

We describe the architecture of a subsystem that helps draw data structure diagrams similar to those found in text books on programming. These diagrams are generated from the variable and type declarations, a few hints about the composition of the diagram, and the memory image that contains the internal values of the variables. Our drawing process consists of three stages: construct an abstract figure that gives the topological constraints among the various figure elements, determine a display representation that specifies in a device-independent way the relative sizes, coordinates, color and fill texture for the figure elements, and finally produce an actual display from this representation using a device-specific driver. The hints are about the abstract relationships between index variables and arrays, between pointer variables and objects pointed to, and whether the graphs are indeed trees, etc.

This drawing subsystem is being presently constructed, and will be incorporated into the Unix debugger dbx.

1 Introduction

For a variety of psychological reasons, many people prefer pictures to textual descriptions. In the context of programs and data structures, figures become all the more important. Indeed, we do not know of any text book on data structures that does not use figures extensively. In a given section, they are usually concentrating on one data structure at a time, and much attention is devoted to the composition of the figure. However, the situation becomes very tedious even for moderate sized (say 1000-line long) programs. It would improve our effectiveness as programmers, if we could automate the production of pictorial views of data structure both in archival documentation about the programs and during the debugging phase.

In this paper, we describe the architecture of a subsystem that helps draw data structure diagrams similar to those found in text books on programming. These diagrams are generated from the variable and type declarations, a few hints about the composition of the diagram, and the memory image that contains the internal values of the variables. The hints are about the abstract relationships between index variables and arrays, between pointer variables and objects pointed to, and whether the graphs are indeed trees, etc. We are currently in the process of implementing this subsystem as part of the Unix debugger known as 'dbx'. This work is part of an effort to build a software design environment named CaseDE [Mateti et al. 1984].

2 Data Structure Diagrams

Figure 1a is typical of data structure diagrams (dsd). The declarations of the relevant variables and types are shown in Figure 1b. This diagram is hand-drawn based on the layouts indicated by the drawing algorithms discussed in the paper. We will be using these figures as running examples.

2.1 Basic Semantic Concepts

Connected Diagrams

Often data structure diagrams are connected; i.e., there is a connecting line from one (part of the) diagram to another (part of the) diagram. Examples of such diagrams (see Figure 1a) include arrays and their indices, pointers and pointed objects, etc. Some of these connections are such that they cannot be deduced from either the declarations, or their values: that i and j are used as indices of array A can only be deduced from the executable statements but not from their declarations.

Composition of Related Diagrams

We often wish to see together the diagrams of several 'related' data structures. While we have defined connectedness among diagrams based on programming language concepts, we leave it entirely to the programmer to specify which diagrams he wishes to see together.

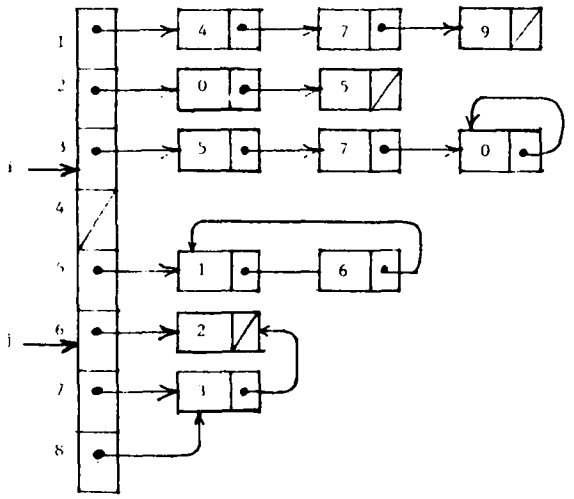


Fig. 1a

```
type  NODE;
type  PNODE  is access NODE;
type  NODE   is
  record
    info  : integer;
    link  : PNODE;
  end record;

var
  A       : array (1 .. 8) of PNODE;
  i, j    : integer;
```

Fig. 1b

Generic Diagrams

The diagram in Figure 1a is a pictorial view of specific values that the variables had: i was 3, j was 6, and the first element of A was pointing to a list of three items, etc. A generic diagram, on the hand, aims to illustrate a typical situation for these variables without being so abstract as to suggest nothing; Figure 2 is an example of this. To produce the most 'general' generic diagram automatically is a futile task for the declarations in a typical language has too little semantic information. For example, the declarations of Figure 1b do not say whether shared list structures are possible in the lists pointed at by the array elements.

Derived Diagrams

We consider certain diagrams, e.g., bar charts and pie charts drawn from numeric data, as derived diagrams. One might also think of the binary tree hidden in the array of heap sort as a derived one. Clearly, this notion of derivedness is a subjective one and depends on how much computation is involved in producing the perceived diagram from one that immediately suggests itself as a result of our familiarity with traditional data structures.

In the rest of the paper, we do not consider generic and derived diagrams.

2.2 Display Considerations

Good Diagrams

An important question that we must discover answers to is "What is a good diagram?" Answers to this highly subjective question depend on the information content, and the ease with which it is communicated as well as symmetry, relative sizes of components, and traditional ways of drawing. Although it will require experimentation to devise rules that capture one's notions of "good" diagrams, we can state some obvious taboos here:

(1) Arrows should not cross figure elements. (2) No two displays of figures should overlap. (3) The appearance of variables of a given type should not be inconsistent.

In addition, the kinds of traditional diagrams we are used to seeing in text books and articles play a role.

Whenever pointers are used, we have, in general, a graph structure. There are many ways to draw a graph data structure. In order to make the graph easier to comprehend, we would like to reduce visual complexity. The number of edge crossings, average edge length (relative to the size of the picture), the

symmetry of the picture with respect to rigid rotations and reflections have all been proposed as measures of visual complexity. Figures 2 and 3 indicate the variety possible in the layouts of graphs.

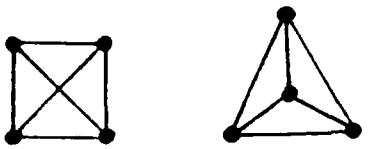


Fig. 2: K_4

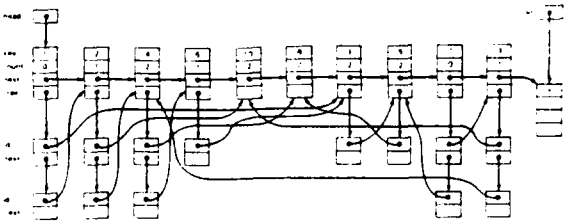


Fig 3: [Wirth 1985]

Binary trees could be handled by the same algorithms used to draw graphs. However, trees occur very frequently in computer science and are important enough to deserve special treatment, and there is a virtual consensus about what constitutes a 'tidy' picture. Wetherell and Shannon[1979], and Reingold and Tilford[1981] list the following requirements of aesthetics for drawing pictures of trees. Almost all well-drawn diagrams of trees that we have seen satisfy these requirements; obviously a layout of a tree drawn to satisfy requirements beyond aesthetics may look quite different (see Figure 4).

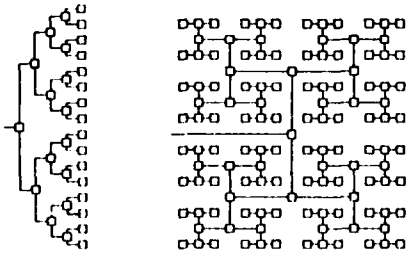


Fig. 4

1. Nodes at the same level of the tree should lie along a straight line, and the straight lines defining the levels should be parallel.
2. A left son should be positioned to the left of its father and a right son to the right.
3. A father should be centered over its sons.
4. A tree and its mirror image should produce drawings that are reflections of one another; moreover, a subtree should be drawn the same way regardless of where it occurs in the tree.

Customization

Though we do not yet have a complete set of rules for producing aesthetically pleasing figures, we are confident that many common situations can be captured by such a set. In spite of this, it is inevitable that users will find occasionally that some fully automated drawings produced by our subsystem are unacceptable. For example, they may wish for certain arrays to be drawn vertically, rather than horizontally. Our solution to these problems is a hint language to influence the layout and composition of the diagram.

2.3 Subsystem Considerations

The drawing of data structures, whether fully automated or not, is not an end in itself. We see it as part of two most time consuming tasks: debugging and documentation. Both these tasks impose special requirements on the drawing subsystem.

Debugging requires that diagrams of acceptable quality be drawn fast in real time as we debug and monitor the execution of the program. It is even necessary to consider animation techniques to display the diagram as it changes values. Documentation requires higher quality diagrams drawn perhaps off-line and incorporated later into the documentation. It also typically requires generic diagrams.

We construct a representation of the display generated from the figure in a machine independent way. From this representation, we can drive most graphics terminals and laser printers. Our representation is a dialect of VDI [Bono 1985]. We expect our interface to the symbol table and memory contents to be quite independent of the host programming language, compiler and operating system.

3 The Process of Drawing

There are three major steps in the process of generating the diagrams, which are sketched out in more detail in later sections. From the given type and variable

declarations, the memory contents, and additional input from the user that we call hints, we produce first an abstract figure, then a device-independent display representation, and finally the actual display as seen on a graphic terminal.

3.1 Figure Elements

A figure element (figel) is an abstract object that captures certain attributes of a dsd. No rigid notions of dimensions, geometric shapes, color or fill-texture are associated with a figel. The attributes are such things as content, label, and descriptions of how the content is to be displayed.

Associated with each variable of the program is a composite or atomic figel, depending on whether the variable can be considered composite (such as an array), or atomic (such as a scalar). The figels associated with a variable are considered independent and their layout (sans scale and absolute coordinates) is determined independently of others.

3.2 Display Representation

Having produced an abstract figure for the data structure, which gives certain topological constraints that must be satisfied, we decide on the placement, size, geometric shape etc. of each of the atomic and composite figels, and the visual representations for the figel composition operations.

Our global strategy for determining the display of an abstract figure is as follows: Determine the display of each independent figel. Enclose each of these in a polygon and begin tiling a potentially infinite sheet with these polygons while satisfying the composition operations given in the abstract figure. The display of the independent figels is similarly determined, except that we expect certain kinds of regularity in the constituent figels (such as elements of an array, nodes of a tree, etc.).

4 Computing the Independent Figels

Figels have at least three attributes: a content, a label, and a description of how the content is to be displayed. The label of an atomic figel is usually the name of the variable (if the variable is atomic), the index (if an array element), or the field name (if a record component). The content of an atomic figel is the internal value contained in the memory location corresponding to (that part of) the variable.

The computation of the independent figel corresponding to each variable is driven by the abstract syntax tree of the type declaration of the variable, and user-given hints. A standard file contains a schematic description of (1) all scalar types, and type constructors along with their atomic figels, (2) the figel composition operations corresponding to composite type constructors (such as array, record, etc.). In this context, we associate with each access type an atomic figel (and separate the associated record type),a nd consider the entire structure reachable via the pointers to constitute a single data structure variable.

So far, we have identified two binary composition operations that are basic: juxtaposition, and pointing. Other operations are combinations of these two. Juxtaposition requires that the two figels be displayed so that they are aligned along a chosen axis separated by a gap that is a parameter to this operation. Pointing is an asymmetric relation, and requires that in the display there will be an arrow from one figel to the pointed figel.

In Figure 1a, the box containing the integer value, and the one containing the tail of the arrow are atomic figels. These two atomic figels are juxtaposed to give the composite box representing the nodes. The arrow from one box pointing to another is an example of yet another composition operation. Indeed, the dsd shown in Figure 1a contains thirty atomic figels – eight due the array elements, twenty due to the nodes, and two due to the indices i and j. There are nine independent figels, one corresponding to each list, and one for the array and two for the index variables. Note also that i and j do have boundaries but were made invisible, and their values were indicated by positioning them appropriately on the elements of A.

5 Layout of Independent Figels

We sketch our layout algorithms assuming that we know what kind of a data structure (an array, binary tree, linked list, etc.) an independent figel represents.

5.1 Well-Known Data Structures

Scalars

The default layout for an atomic figel that corresponds to a scalar variable is a horisontal composition of the label (which is the character string denoting the identifier of the variable), and a rectangular box containing the value of the variable. This value is displayed according to the prescription given in the third attribute. For example, an enumerated value is dis-

played as an identifier denoting that value obtained from the symbol table.

Arrays

We consider the figels corresponding to the N elements of the array as independent, and determine their layouts. Compute the smallest polygon P that can enclose each of the elements. Make N juxtaposed copies of P aligned along an axis, and insert the k-th element's layout in the k-th copy of P. The orientation of this axis, and the scaling of P are determined by how the array diagram fits in with the rest, or is controlled by a user-given hint.

Records

Determine the layouts of the component fields of the record. Arrange these layouts in a 2-D packing (see below) inside a rectangle.

Linear list

Linear list layouts are similar to arrays. Determine the layouts of all, say N, the reachable nodes. Compute the smallest polygon P that can enclose each of the nodes. Make N equi-distant, say d, copies of P aligned along an axis, and draw straight line arrows. Distance d is dependent on the size of P.

Non-linear lists

Do a depth-first search, and arrange the layouts of the nodes of each forward chain along parallel axes as for linear lists, and then draw straight line arrows for these, and spline-curved arrows for the back pointers.

Graphs

Non-linear lists are nothing but graphs; however, circular lists etc. are so common that we treated them separately. The general problem of drawing graphs has received considerable attention over the decades. For example, it is well-known that graphs can be drawn using only straight lines unless self-loops are present. Such layouts however are often ugly and take unacceptably large areas. Lipton, North and Sandberg [1985] described an algorithm that draws graphs emphasising symmetry. They characterize symmetry in terms of automorphism groups. They assume that nodes are very small, and use straight lines for edges without worrying about edges crossing nodes. We are in the process of adapting this algorithm for the usual case of dsds where the nodes are of non-trivial size, and edges should not cross the nodes.

Binary trees

Reingold and Tilford [1981], Wetherell and Shannon[1979] and Supowit and Reingold[1983] present algorithms for drawing aesthetically pleasing trees (see Section 2) while attempting to minimize the width of the picture. We have chosen to adapt Reingold and Tilford's algorithm. This algorithm was intended for positioning nodes on a rectangular grid, where a node could be drawn within one grid cell. We are interested in positioning nodes on a real coordinate plane, so each "grid cell" would be infinitesimally small. However, Reingold and Tilford's algorithm takes as a parameter a variable called minsep, which is the minimum separation between nodes. If we consider the algorithm to be positioning the centers of boxes and set minsep to the box size plus the desired separation between boxes, then the algorithm will produce the correct result.

5.2 Two-Dimensional Packing

Given a set of layouts of figels, we pack them into a rectangle. This problem is related to bin-packing but since we are not interested in minimising the number of bins or some other quantitative optimisation, we have chosen to use the following simple algorithm. This algorithm assigns positions to nodes of the graph searching in the breadth first manner. Since each node has its own local coordinate system, we need simply specify a translation to relate its local coordinate system to the graph's coordinate system.

The following algorithm places a node X so that it is as close as possible to a given node Y without its enclosing rectangle overlapping any other enclosing rectangles. We can represent a position of X as a translation to be applied to its local coordinate system. Let rect(X) denote the enclosing rectangle of X. Consider the case where rect(X) is centered at the origin in its local coordinate system and has width w and height h. Then the space of invalid positions is obtained by expanding the rectangles by w/2 horisontally and h/2 vertically and taking the interior of their union. The valid position of X closest to Y must be on the boundary of the invalid region. Thus, to find this position, we need only check points on this boundary. This can be done efficiently (in N log N time where N is the number of boxes) by sorting the coordinates of the rectangles and using a plane sweep algorithm.

6 User-Given Hints

We have aimed to reduce the need to control the display image by choosing default styles of display after observing numerous diagrams of data structures found in text books. For each programming language handled by our display system, there is a file of descriptions as to how various built-in and user-defined types are to be displayed. A user may choose to edit this file to reflect his tastes, or temporarily override the default descriptions by providing hints along with the declaration of the variable in the programming language. Another occasion where a user may want to provide hints is when the system cannot mechanically recognize a data structure to be, say, a binary tree in spite of suggestive names in its declarations.

Controlling the appearance of the final image of the dsd is possible at three levels by

(1) defining what parts of a data structure are considered as independent figels,

(2) influencing the construction of the display representation of an independent figel,

(3) altering the mappings of abstract attributes of figels to visual representations.

The particular form our hint language takes is an active topic of our research; consequently, we have stated these hints in free style English. However we expect it to be influenced by drawing languages such as PIC [Kernighan 1981], IDEAL [Van Wyk 1981] and MIRA [Magnenat-Thalmann and Thalmann 1981].

6.1 Figel Hints

Connection hints tell how variables are related, or just assert that they are. This introduces figel composition operations among otherwise unrelated figels causing these variables to be connected (usually by arrows) in the figure.

We quite frequently construct structures with multiple pointer fields such that if we consider all the fields they become an arbitrary graph. However, if we restrict ourselves to a chosen set of pointer fields they exhibit more structure such as a certain kind of a tree.

Figel compositional hints are used to express declarative procedures for composing abstract figures whose relationship is not explicit enough in the declarations of the data structure. For example:

```
figure F5( a, x1, b2, c1, d)
caption 'An Example Figure' centered
layout hints about the layout of a, etc., if any
end figure
```

where the a etc are the names of variables visible at the present scope and denote the dsds for them.

6.2 Display-Specific Hints

Diagram composition hints control what will actually appear in the picture, and how parts will be drawn; e.g., we may wish to omit field d from variable m. Spatial layout hints state a preference as to where components of a diagram should be laid placed. Some additional examples are:

```
use bgcolor maroon, fgcolor (RGB 1 .9 .2)
bordercolor (HSV .6 .5 .7)
when drawing m1,m2;

use shape circle when drawing type node
use line pattern pat1 when connecting i

draw A horizontally

place x east of y
place m within .5 of q
layout node as vbox(hbox(a,c),b,hbox(l1,l2,l3))
align arrays a, b, c vertically
```

6.3 Data Abstraction Hints

Data abstraction hints allow the programmer to map record/pointer structures onto certain common abstract data structures. These structures are then drawn in the customary way. This is a stop gap measure – it does not support data structures implemented in a nonstandard way (e.g. trees implemented with arrays). Once a design specification language, such as CaseDL [Mateti 1985], is integrated with the debugger, these hints will become unnecessary.

```
stack p, top(stk), next(p.next)
(p is a dummy variable used in the hint only. stk is a
pointer to the top of the stack.)
linked list p, head(studlist), tail(p.next)
(The programming language implementation of lists
and stacks are identical, yet the abstract data structures are drawn differently.)
binary tree p, root (dict),
left(p.son), right(p.daughter)

circular doubly linked list p, start(list3),
prev(p.back),next(p.fwd)
```

7 Related Work

The development of visual aids to support programming is on the rise. The papers [Griswold 1984], [London and Duisberg 1985], [Reiss 1984, 1985], [Teitelman 1985] are a sample from the academic community, and there are quite a few commercial products that pictorially support software development.

The work we described in this paper is related to typesetting text (cf. "boxes" and "glue" of [Knuth 1982]), drawings ([Kernighan 1981], [Van Wyk 1981]), and also to VLSI layout [Ullman 1982]. In the context of data structures, the work of [Wetherell and Shannon 1979], [Reingold and Tilford 1981], [Lipton et al. 1985], and [Myers 1980] has influenced us. Previous visual debuggers include Incense [Myers 1983], DDS by our student [Ojeda 1985] and VIPS [Isoda 1985].

Myers [1980] surveys early work in debugging, and in particular efforts to display data structures. His system [Myers 1983] displays Mesa data structures. While these displays are in general impressive, often they are unpleasing to us since his layout techniques are ad hoc and too simple. E.g., his display of a tree would have nodes whose size decreases as we move away from the root, and customization (except in very simple cases) requires writing drawing procedures in Mesa.

Wetherell and Shannon [1979] and later Reingold and Tilford [1981] and Supowit and Reingold [1983] have formalized notions of aesthetics of tree drawing, and present algorithms based on these. Supowit and Reingold indeed show that tree drawing minimizing width when the nodes must be positioned on an integral lattice is NP-complete.

Meyers [1983] lays out the records pointed to by a given record in a vertical column to the right of the parent record (except for records which have already been laid out elsewhere). The entire column of children is constrained to fit in the same vertical space as the parent. This space is simply divided into as many cells as there are records, then the records are scaled to fit the cells. Thus records will be drawn progressively smaller towards the right. With an infinite resolution display, the entire data structure could be shown with this method. Meyers simply stops drawing records when they reach a certain (small) size. This method does not meet our requirements because we want all nodes of a given type to be drawn the same size.

Ojeda[1985] uses the same layout algorithm as Meyers, but he does not scale down the records. Instead, when a cell is too small to display a record, an abstract node consisting of a circle is drawn, and no children of this node are displayed. The user can interactively cause such abstract nodes to be displayed fully elsewhere on the screen. This method does not meet our requirements because we would like to be able to display an entire data structure at one time.

8 Conclusion

We described the architecture of a subsystem that helps draw data structure diagrams similar to those found in text books on programming. These diagrams are generated from the variable and type declarations, a few hints about the composition of the diagram, and the memory image that contains the internal values of the variables. This drawing subsystem is being presently constructed, and will be incorporated into the Unix debugger dbx.

References

Bono, Peter R., "A Survey of Graphics Standards and their Role in In- formation Interchange," IEEE Computer, vol. 18, no. 10, pp. 63- 75.

Brown, Mark H. and Robert Sedgewick, "Techniques for Algorithm Anima- tion," IEEE Software, vol. 2, pp. 28-39, 1985.

Griswold, Ralph, "Diagramming Icon Data Structures," TR-84-5, Depart- ment of Computer Science, University of Arizona, Tuscon, 1984.

Isoda, Sadahiro, Takao Shimomura, and Yuji Ono, "Visual Debugger VIPS: Visual Representation of Program Execution," in Proceedings of IEEE Compsac 85, IEEE Computer Society, Chicago.

Kernighan, Brian W., "PIC–A Language for Typesetting Graphics," in Proceedings of the ACM SIGPLAN Symposium on Text Manipulation, pp. 92-98, Portland, Oregon, June 1981.

Knuth, Donald E., The TeX Book, Addison-Wesley, 1982.

Lipton, R. J., S. C. North, and J. S. Sandberg, "A Method for Drawing Graphs," in Proceedings of the ACM Symposium on Computational Geometry, pp. 153-160, Baltimore, 1985.

London, Ralph L. and Robert A. Duisberg, "Animat-

ing Programs Using Smalltalk," Computer, vol. 18, no. 8, pp. 61-71, August 1985.

Magnenat-Thalmann, Nadia and Daniel Thalmann, "A Graphical Pascal Ex- tension Based on Graphical Types," Software–Practice and Experi- ence, vol. 11, pp. 53-62, 1981.

Mateti, Prabhaker, Frances Hunt, George Ernst, Raymond Hookway, and Gerry Radack, "CaseDE: An Environment for Precision Design of Software," internal memorandum, Department of Computer Engineering and Science, Case Western Reserve University, 1984.

Mateti, Prabhaker, "CaseDL: A Design Specification Language," techni- cal report, Department of Computer Engineering and Science, Case Western Reserve University, to appear in January 1986.

Meyer, Bertrand and Brad A. Myers, "Displaying Data Structures for In- teractive Debugging," CSL-80-7, Xerox PARC, June 1980.

Myers, Brad A., "Incense: A System for Displaying Data Structures," Computer Graphics, vol. 17, no. 3, pp. 115-125, July, 1983. (Proceedings of SIGGRAPH 83)

Ojeda, Francisco J., DDS: A Subsystem for Displaying Data Structures for Interactive Debugging, Department of Computer Engineering and Science, Case Western Reserve University, August 1985. M.S. thesis

Reingold, Edward M. and John S. Tilford, "Tidier Drawings of Trees," IEEE Transactions on Software Engineering, vol. SE-7, pp. 223- 228, 1981.

Reiss, Steven P., "Graphical Program Development with PECAN Program Development Systems," in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 30-48, Pittsburgh, April 1984.

Reiss, Steven P., "PECAN: Program Development Systems that Support Multiple Views," IEEE Transactions on Software Engineering, vol. SE-11, no. 3, pp. 276-285, March 1985.

Schneiderman, Ben, Philip Shafer, Roland Simon, and Linda Weldon, "Display Strategies for Program Browsing," in Conference on Software Maintainenance, IEEE Computer Society, Washington, DC, 1985.

Shamos, Michael Ian and Dan Hoey, "Geometric Intersection Problems," in Proceedings of the 17th Annual Symposium on Foundations of Computer Science, pp. 208-215, Houston, 1976.

Supowit, Kenneth J. and Edward M. Reingold, "The Complexity of Drawing Trees Nicely," Acta Informatica, vol. 18, pp. 377-392, 1983.

Van Wyk, Christopher J., "A Graphics Typesetting Language," in Proceedings of the ACM SIGPLAN Symposium on Text Manipulation, pp. 99-107, Portland, Oregon, June 1981.

Wetherell, Charles and Alfred Shannon, "Tidy Drawings of Trees," IEEE Transactions on Software Engineering, vol. SE-5, pp. 514-520, 1979.

Yarwood, Edward, "Toward Program Illustration," CSRG-84, University of Toronto Computer Systems Research Group, October 1977.

Ada* and the PC, It's Time Has Come

Freeman L. Moore

Texas Instruments, Inc.
McKinney, Texas 75069

ABSTRACT

Until recently, Ada compilers have been found on minicomputers and mainframes. With the acceptance of the personal computer as an individual programmer workstation, there is the need for usable Ada compilers which operate in that domain. This paper identifies some the Ada-like compilers for personal computers as well as the most recent developments indicating that validated compilers are available for personal computer (PC) users.

INTRODUCTION

The Ada programming language has been in existence in one form or another for the past seven years. Over that time period, it has matured from the preliminary definition of 1979 to its current version, approved in 1983 as ANSI/MIL-STD 1815A. Concurrent with the language development process, efforts were also taking place towards the development of compilers and the necessary runtime support systems. The first Ada compilers were developed and implemented on mainframes and superminicomputers. The current trend in software development environments is towards distributed environments, moving away from the mainframe system to individual engineering workstations. While it may be that the power of some workstations rivals that of some of the minicomputers, the real concern is whether it is practical to develop Ada applications using a personal computer as the workstation.

In an education oriented environment, as well as a software development environment, the need exists to make computer resources available to all concerned. If Ada compilers can be made usable on a personal computer of reasonable cost, then some of the access burden can be taken off of the mainframe, alleviating its workload for other purposes.

Most business software for small systems is written in compiled Basic, interpreted Basic, Cobol, Pascal, or assembly language. Most software for home systems is in interpreted Basic or assembly language. In both of these areas, the principles of software engineering may be unknown and not practiced. The approach taken may be one of code first, and solve second. Pascal programmers are generally better than some because of the structure imposed by their language. Since the Ada language allows for more organization that Pascal, the introduction of the Ada language to the personal computer market could have a favorable impact on the quality of software developed for personal computer users.

The search for and status of current implementations of Ada compilers for personal computers is the emphasis of this paper. It must be realized that this area has the potential for rapid changes, with new product announcements occurring frequently. The information presented here is accurate as of the writing of the paper, and will be supplemented with additional material during the presentation as appropriate.

SCOPE

The first validated Ada compiler was developed concurrently during the language specification phase by New York University, and delivered on a DEC-VAX* computer system. The second validated compiler was for the Data General computer System. Neither of these

* Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

computer systems is likely to be found in the home of the average software engineer. For purposes of this paper, a distinction is made between personal computers and workstations. Machines such as the SUN and APOLLO shall be considered as advanced workstations, whereas machines such as the IBM PC*, TI-Professional, KAYPRO* are considered in the personal computer category. In particular, attention will be focused on personal computers which are capable of supporting their own compiler. That is, the compiler runs on and produces code for that machine. Cross compilers are programs which execute on one machine while producing machine code for a different processor. There has been a substantial amount of work in this area, but this is beyond the realm of personal computers and is not addressed here.

KINDS OF IMPLEMENTATIONS

When developing an Ada compiler for a personal computer, one is confronted by several choices. The first is to develop either an interpreter or a compiler. Interpreters do not produce any machine code which can be kept from one execution to the next. Most implementations of Basic make use of the interpreter approach. If a compiler approach is taken, the compiler can be make to generate machine for the target machine, or else generate a hypothetical machine code. This is the approach taken with some Pascal compilers and their use of p-code. The p-code represents the hypothetical machine code. While the use of p-code is attractive from the developer's point of view for portability reasons, it has the distraction that the final product will execute slower than if actual machine code had been generated. It is commonly recognized that the writing of interpreters is easier than true compilers, but again, there is the execution performance price that must be paid. In the case of the Ada language, an interpreter would not be adequate because of the language features allowing for separate compilation of packages and subunits.

As part of the development of an Ada compiler, the compiler must pass the validation process as controlled by the Ada Validation Office. This process will ensure compilance with the Ada Reference Manual. Thus subset compilers are not permitted to be called Ada, nor are superset compilers allowed. However, this does not permit the development of compilers for "Ada like" languages as long as they do not use the name "Ada".

The Ada Validation Office maintains a collection of programs (currently over 2000), which must be properly processed by an Ada compiler and corresponding runtime environment. This collection of programs is referred to as the Ada Compiler Validation Capability (ACVC), and is subject to change every six months. If a compiler does not pass the ACVC, it can not be called an Ada compiler.

NON-ADA IMPLEMENTATIONS

In this section, implementations of "Ada-like" languages and subsets are considered. Because of the ACVC test requirements, these products can not be called Ada translators. Even though they do not implement the entire language, some are worthy of consideration.

An inexpensive possibility is an implementation called AUGUSTUS by Edward Michael. AUGUSTUS is not a pure subset implementation of the Ada language and does not purport to be an Ada compiler either now or in the future. Minor modifications to the Ada syntax were incorporated along with the restriction of not being able to recognize the entire syntax. The translator and supporting interpreting system were originally developed on an Osborne I computer system in Basic. As such, the compiler will run on other microcomputers with little or no modification. The source code has been published in Dr. Dobb's Journal (1983).

The advantage of considering AUGUSTUS is really not using it for learning the Ada language, but rather as a case study of compiler writing. It's use for teaching and learning the Ada language is not recommended, except for other than the simplest of home/hobby users. It would not be practical to list the limitations of AUGUSTUS because of its great divergence from the Ada standard.

AUGUSTUS is just one example of a product which may be more appropriate for compiler study rather than language learning. Other examples from academic institutions can be found in the publications of the special interest groups on programming languages and computer science education, groups within the Association for Computing Machinery.

JANUS/ADA, from RR software, is clearly considered as the most widely available and supported system approaching the full Ada language. RR software has been marketing and improving

JANUS/ADA for some time, and has indicated their willingness to continue their development. According to literature from RR software, several U.S. government agencies, including the Department of Defense, have begun programming with JANUS/ADA. It is also stated that JANUS/ADA is being used by the governments of Canada, France, and Australia. The claims of the company are documented by the number of published references to their product. While JANUS/ADA is not a complete implementation, it does support a rich portion of the language. Some of the features of Ada which are not implemented include:

fixed point numbers
slices/aggregates
boolean array operations
tasking
separate compilation with subunits
generics
representation specification

From this list, it is apparent that only the more advanced features of the language are not available. One would not experience Ada tasking, programming in the large concepts, and other subtle points of the language but still, JANUS/ADA is a practical alternative to learning a major portion of this exciting language. A major benefit of JANUS/ADA is its error handling, both at compile time and execution time.

The JANUS/ADA compiler is a multi-pass compiler which produces relocatable files. Compilation speed varies based upon the machine and resources available. All code generated is ROM-able and re-entrant.

TeleSoft* has the TeleSoft-ADK (Ada Development Kit) including the TeleSoft-Ada compiler and various tools and utilities, available for the IBM PC. The system is based on a p-code interpreter. Floating point operations require the use of an 8087 math coprocessor. The utilization of p-code, rather than native code, saves memory space during development and execution, but limits execution speed. TeleSoft also has versions available for various 68000 processors and operating systems, typically using the UNIX* operating system. These 68000 systems are not as generally accepted in the personal computer area and will not be discussed further in this paper. Hardware requirements specify either a hard disk along with 320K bytes of RAM or two DSDD floppy disk drives, and 576K bytes of RAM. In addition to its microcomputer compilers, TeleSoft has been developing compilers on the DEC-VAX systems since 1981, some with cross compiler support for other processors.

VALIDATED COMPILERS

This paper has been written upon the assumption that the following vendors will have validated their products by March 1986; the time of the conference.

New York University
General Transformation Corp.
Alsys, Corp.

New York University announced their personal computer version of the AdaEd system in July of 1985, with expectations of validation by November of 1985. General Transformation Corporation plans to validate their compiler in the first quarter of 1986, and Alsys has a pre-validated version for the IBM PC, using validation suite 1.6. Further information about each of these is presented in the following sections.

New York University (NYU) AdaEd

The first version of the New York University (NYU) AdaEd compiler was written in SETL, a very high level interpretive language. Being interpreter based, the performance of the compiler is less than outstanding. Because of the computer resources required, it is generally not acceptable for any degree of high volume development on a time shared system. NYU is currently rewriting their compiler using the C language, with the intention of porting the system to personal computers. It was stated in the Ada Information Clearinghouse News Events on July 30, that NYU had announced the availability of the compiler. No further information is available, although the compiler was reportedly demonstrated at a conference in the fall of 1985. It is unknown at this time if the compiler will still generate its internal machine code or be a native code compiler. Speculation is that it will continue to use its own machine code, and that the translation from SETL to C will show a 10X improvement in compilation speed.

General Transformation Corporation (GTC)

Again, at the time of this writing, the product is not yet available but is expected by conference time, March 1986. An earlier schedule had planned for the compiler to be available in late 1985. Plans now indicate an early 1986 delivery date, with internal validation being

expected by December 1985, and formal validation in first quarter of 1986. GTC is a relatively new company, founded in 1983 with the goal of producing Ada products of exceptional quality and providing excellent technical support for its products.

This compiler appears quite promising. An earlier brochure had stated full implementation of representation clauses, along with Pragma INTERFACE. A revised brochure indicated that most of the features will be implemented, along with the inclusion of package MACHINE_CODE.

It is worthwhile to note that GTC claims to be developing their system on an IBM PC, so as to have their development engineers fully aware of the capabilities of the system it is intended to run on. Also noteworthy is the company's decision not to release a subset compiler. "In the spirit of this valuable effort toward standardization, and despite the long time to product release, the company chose to release only full Ada compilers and not subsets of the language, believing this to be in the best interests of its customers." Compilation speeds of 1000 lines per minute are expected. A 8087 math coprocessor, 512K RAM and 10 megabyte hard disk are required. While this hardware configuration is not a minimal system, it does fall within the range of what a software engineer might have, although the cost of the compiler ($1000) may be too much for the average home personal computer user.

ALSYS

Alsys announced in late 1985 their Alsys Ada compiler for the IBM PC-AT*. This is a full Ada compiler that will be unofficially pre-validated under ACVC version 1.6 prior to shipment. It is claimed that the compiler generates very efficient machine code for the 8086 microprocessor family. The compiler can be directed to take advantage of the 80286, in which case the generated code will include 80286 instructions where appropriate. The host for the compiler is an IBM PC-AT with at least 512K RAM and a 20 megabyte hard disk, running DOS 3.0. In addition, one full slot must be available for a memory board delivered along with the compiler. The board will have approximately 3 megabytes of extended memory. The target can be the same as the host machine, or any IBM PC (or compatible) running DOS 2.1 or higher. Floating point in an application is implemented using the 8087/80287 math coprocessor; however the compiler itself does not require the coprocessor.

Although the introduction of this compiler is for the IBM PC-AT, its capabilities include generating code for the IBM PC. One might speculate on how long it might be before the compiler is hosted on the IBM PC. Alsys has produced various other compilers, including compilers for the SUN and APOLLO workstations, in addition to a DEC-VAX compiler. Alsys also has the distinction of having the first cross-compiler ever validated, using the DEC-VAX as the host with the ALTOS 68000 as the target.

SPECULATIONS

Will the future find Ada compilers for the small machines with only 64K of RAM? Probably not hosted on those machines, but we may see more work in the area of cross compilers. Ada compilers will have substantial memory requirements, both in RAM and disk storage. The development of Ada compilers for other machines will depend heavily upon the demands of the market and what software engineers what and need. Pascal is widely accepted in the academic and home markets, an area in which the Ada language has not yet penetrated. If an compiler existed for your favorite eight-bit processor, who knows how the markets would respond.

ACKNOWLEDGMENTS

The information presented in this article was obtained from various vendors. Their support in responding to information request is greatly appreciated. The support given to me by Guy Dame during the research for this effort is gratefully appreciated.

* TRADEMARK INFORMATION

DEC and VAX are trademarks of Digital Equipment Corp.
IBM, IBM PC-AT are trademarks of International Business Machines Corp.
Unix is a registered trademark of AT&T Bell Laboratories.
Telesoft is a trademark of Telesoft.
Kaypro is a trademark of Kaypro Corp.

BIOGRAPHY

Freeman Moore is a Software Engineer in the Human Resources Development Department of Texas Instruments. Primary responsibilities include development and delivery of courses in the Ada curriculum. He has also participated in the redesign and reimplementation of an existing realtime software system using the Ada language. Freeman Moore is a candidate for the Doctoral degree in Computer Science at North Texas State University. Freeman Moore has also held the position of Assistant Professor of Computer Science at Western Kentucky University and at Central Michigan University.

MAILING ADDRESS

Freeman L. Moore
P.O. Box 801, M/S 8006
Texas Instruments, Inc.
McKinney, Texas 75069

AUTHORS INDEX

NAME	PAGE
Anderson, E. R.	85
Ausnit, C.	28
Baker, P. L.	51
Bamberger, J.	132
Belz, F. C.	57
Blackmon, L. F.	88
Boebert, W. E.	86
Bolger, J. E.	22
Broido, M.	42
Buoni, J. J.	38
Burden, R. L.	38
Burton, B.	42
Burton, B. A.	114
Canavan, R.	112
Carrio, M. A. Jr.	75
Caverly, P.	112
De Bartolo, G.	158
Dousette, P. J.	13
Goldstein, P.	112
Griffin, L.	1
Hart, S. R.	87
Keller, S. E.	67
Koppes, M. R.	114
Laird, J. D.	114
Lease, D. M.	67
Martin, B. J.	100
Mateti, P.	165
Mers, R. C.	104
Moore, F. L.	173
Pastuzyn, K.	112
Pepper, W. S. IV	8
Perkins, J. A.	67
Perkins, M. T.	22
Platek, R. A.	84
Radack, G. M.	165
Richards, R.	158
Ritter, P.	132
Rodriguez, T.	1
Rudolph, R. S.	96
Santhanam, V.	142
Schaefer, C. F.	149
Selwood, M.	125
Setzer, B.	100
Walker, J. E.	109
Walker, R.	100
Wilson, J.	132
Zuk, M. M.	83

NOTES

NOTES

END

DTIC

6-86