Software Technology for Adaptable Reliable Systems

(STARS) Workshop

April 9-12 1985

AD-A163 463

DTIC FILE COPY

567 259

DTIC
ELECTE
JAN 3 0 1986
S D
E

Naval Research Laboratory
Washington, DC 20375-5000

Approved for public release; distribution unlimited.

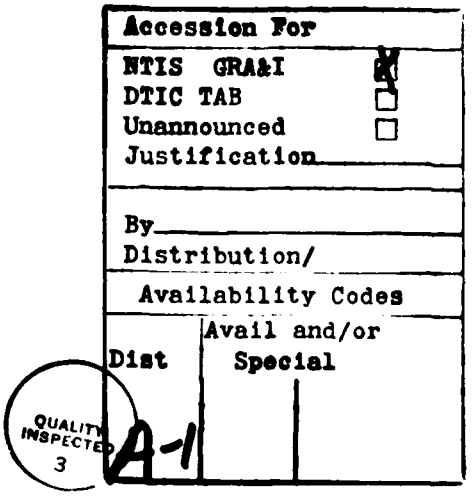
86 1 29 066

PII Redacted

Software Technology for Adaptable Reliable Systems

(STARS) Workshop

April 9-12 1985



Naval Research Laboratory
Washington, DC 20375-5000

PREFACE

Commerce Business Daily

WORKSHOP ON REUSABLE COMPONENTS OF APPLICATION SOFTWARE

The Software Technology for Adaptable Reliable Systems (STARS) Project of the Office of the Secretary of Defense seeks sources of information and expertise in the building of mission critical applications software. Component specification, building, testing, maintaining and reutilization are of interest. With the promulgation of Ada as a single High Order Language (HOL) to build future applications within the three services, there exist new opportunities for reuse of software. Reuse can reduce software system development time and maintenance costs, and improve reliability. The Applications Tri-Service Working Group with the STARS project is sponsoring a four-day, workshop at the Naval Research Laboratory in March 1985 to discuss and present summarized material on the following issues and questions:

(1) Specification/Design: Evidence of usefulness of methodologies and standards for writing reusable software components. Attach user manual of the methodology or standard discussed.

(2) Reusable Component Definition: What constitutes a component? Are documentation, source text, regression test and data a sufficient set? How is related information bound and kept in synchronization to components? How are security problems resolved? (Specific questions include: How does one avoid contamination of a component by classified data when passed from one environment to another? How can we avoid/detect trojan horses or planted time triggered events?

(3) Validation of Software Components: What manual and automated tools are now effective? At what level must tools be run to be effective? What is reasonable for a user to expect and what can be provided for a user to be confident in a reusable component? Are documentation and test results efficient to validate the components?

(4) Library experience: Discuss item storage, cataloging, configuration management (personnel security, cost, logistics, remote and local access, change and errata notification). Identify application supported, e.g. Weapon System, C3I, space and the length of experience.

(5) Automated part composition: State issues and solutions for prototyping and delivery of operational systems, manual and automated techniques.

(6) Logistics of organizational reuse of software and as government furnished material, and problems encountered with data rights and license arrangements, and user liability claims.

(7) Encouraging deposits: Discuss company and employee incentives to encourage the submission of or to provide access to reusable components.

(8) Ada experience: Discuss special problems encountered in the use of Ada. Relate to the use of Ada, or recast existing tools and experiences in the Ada language. Interested firms, and individuals are invited to respond with name and resume of candidates to participate in the workshop.

STARS Applications Workshop
April 9-11 1985

Issue Panels and Initial Issues

I. Parts Taxonomy/Requirements

II. Incentives

III. Library

IV. System/Design/Integration with Reusable Parts

V. Metrics

The issues are broken down by the subject area for each of the panels. The intent of these issues is to provide guidance to the panel: the panel is free to add, delete or modify these in the work sessions.

I. Parts Taxonomy and Requirements

In the opening session, terms were given in several perspectives. In order for future communication to be productive, we need to decide upon a common terminology, particularly the terms package, component, part, and piece (and similar decompositions). Once determined, what guide to the production of that thing should be used (if any)? Deliberations must include definitions and specifications that can allow knowledge-based access of pieces and ultimate pull for prototyping and automated parts composition. Consideration would be given to possible function migration between SW/FW/HW categories.

A. Define terms and provide rationale for the selection:

a. the collection of source code, design, requirement, etc. to perform a function

b. Individual items such as the source code or a document

B. What is to be included in each definition and what information is needed for each of these items to reuse software at each level of the definition?

C. The SDS DID's are soon to be mandated. Can parts of them be used as guides for the production/description of the lowest level items? At higher levels?

(These will be available in the meeting area)

D. Are there guides in industry that should be considered as replacement or modifications to the SDS DID's?

E. If you were to acquire a product from another company (reusable piece), do you need information not covered by an SDS DID?

F. A primary focus is Ada but it is possible that other languages must be uncovered for repair and communicated for elimination of rediscovery costs. What are effective means to perform these functions?

II. Reuse Incentives

We need to reuse existing software to allow DOD and industry to focus their resources on developing new extensions and more advanced systems, tools and concepts. What motivators can DOD and Industry management provide to the performers to encourage reuse?

A. What contractual actions can assist this process? Are there examples of clauses that have been inserted that have been successful? Needing to learn from our errors, are there any that demotivated the performers?

B. What internal incentives have been motivators? Can they be extended?

C. Are there funding initiatives that could be established that might serve as incentives? For example bonuses at the end similar to schedule/ quality bonuses used today? Have they met with success? If of use, what contract language is appropriate?

III. Library

This group must address the repository issues of holding the reusable items, worrying about CM and maintenance issues, and serving the users of it.

The list of issues are a starting point for discussion. We expect that internal panel deliberations may modify this set significantly.

A. What should be the acceptable criteria for a new item to enter a repository? Who should do it?

B. Should we consider anything but system-high security? If we do so, how much burden is being placed on the user in finding needed items? Of special interest is how to avoid data pollution, trojan horses and time bombs in code?

C. What information must be available to the potential user? (we expect some interaction with the definition panel here)

D. How can software licenses and proprietary information be addressed in the context of reuse and life-cycle maintenance constraints?

E. Some believe that a library should be centralized and some that it should be distributed. List the pros and cons of each approach and for the distributed approach, suggest the partitioning approach for distribution, i.e. by ownership, application, etc.

F. Of the cataloging methods used today, what parts are successful and what parts have proven to be problems (if problems were resolved, how?)

G. What the cataloging method should STARS approach? Is there a starting point that can be adopted or modeled after?

H. What are recommended means for library access, distribution of holdings?

I. Suggest alternative approaches to the maintenance of items in the library? Consideration is needed on handling of problem reports, fixing priorities, financing, etc.

J. What are strengths and weaknesses of DOD and Industry products (at least descriptions thereof) in the same library/catalog or access path?

K. What part can propriatory software play in application system building? What factors guard the propriatary nature of these items? Is there a point at which they can be released for general use? Is there a motivator that would allow this to occur earlier (eg. a high fee paid in the initial years to seed new innovations)?

IV. System Design/Integration with Reusable Items

To provide information to the Methodologies Area and to assist us in the guide to reusable items, we would like to assess the current status and determine future approach.

A. What approaches are being used today and what are the good and bad "lessons" learned"?

B. For those who are addressing reuse, have there been any changes in methodology to help promote or hinder reuse?

C. Are there special attributes required to be accessed to assist in the integration of reusable items with new items? Are these different depending on application type, e.g. missiles, command and control, etc.

D. Are there unique requirements to the development environment for a specific application that differ across applications? Take into account a basic environment, coupling to the target environment, external simulators and stimulators, etc. Take a couple applications and work out the requirements.

E. What applications areas are believed to be the greatest opportunities for cost savings thru resuability (consider both software items and environment items and tools)?

F. What part composition systems are finding success? Where do the problems lie? Is there any sufficiently advanced to be adopted as a baseline for evaluation in the STARS Project?

G. What activity is being focused on knowledge based access of parts? Is it applicable to only selected applications or of general use? What are basic rules to apply?

H. What part composition systems exist and are there common denominators that lead to cause for guidance for partitioning rules in building future systems and thus affect resuable parts? (be sure to identify existing system so that the scope of the result is understood)

V. Metrics

We know that we must measure items and communicate the goodness to potential users. Known factors are how fast it is, how much space it takes, how tested is it, etc. What should be known and what would you as a user have to be assured of in order to look closer?

A. What characteristics need to be measured?

B. For each characteristic, can a good, better, best criteria be assigned (or some other consumer index?)

C. What functional and historical criteria is relevant - what could make it believable?

D. If DOD were to give incentives to contractors to develop reusable parts, how could we measure success? Are there measures that can be applied directly or must it be a function of historical usage?

E. What metric values would be required for a part to be acceptable from the standpoint of inclusion of regression tests/data, etc? For example, for systems require high reliability, must the testing cover 100% of the paths?

CONTENTS

PREFACE iii

PROJECT REQUIREMENTS FOR REUSABLE SOFTWARE 1
Lyle A. Anderson and Sivey S. Hudson

MACRO ISSUES IN REUSE FROM A REAL PROJECT 55
Thomas D. Arkwright

WORKSHOP ON REUSABLE COMPONENTS OF APPLICATION SOFTWARE 75
Tom Bowen

A PHASED APPROACH TO ADA PACKAGE REUSE 83
Dr. Bruce A. Burton and Mr. Michael D. Broido

REUSABLE COMPONENT DEFINITION (A TUTORIAL) 99
Rodney M. Bond

AUTOMATED PARTS COMPOSITION 107
R.M. Bieniak, L.M. Griffin, and L.R. Tripp

A DISCUSSION OF ADA EXPERIENCE AT GENERAL DYNAMICS DATA SYSTEMS DIVISION WESTERN CENTER 129
John J. DaGraca

RAPID PROTOTYPING WITH REUSABLE SOURCE CODE 145
Elaine Frankowski, Mark Spinrad, and Paul Stachour

MODELING A REAL-TIME EMBEDDED COMPUTER SYSTEM USING ADA: SOME PRELIMINARY RESULTS 155
Frank L. Friedman and Paul A.T. Wolfgang

A PROCESS VIEW FOR REAL-TIME SYSTEMS 177
Nancy Giddings

VARIATIONS OF A REUSABLE SOFTWARE COMPONENT 195
Dr. J. Kaye Grau

SOFTWARE VALIDATION OF SIGNAL PROCESSING SYSTEMS AND ITS IMPACT ON REUSABILITY 209
Michael R. Miller, Hans L. Habereder, and L.O. Keeler

REUSABLE SOFTWARE IN SIMULATION APPLICATIONS 227
Frederic D. Heilbronner

REUSABLE SOFTWARE—A CONCEPT FOR COST REDUCTION 243
Christine M. Anderson and Marlow Henne

A UNIFIED SYSTEMS ENGINEERING APPROACH TO SOFTWARE REUSABILITY 271
Ted Hobson

ISSUES IN REUSING SOFTWARE 297
Richard A. Howey and Lynn M. Meredith

SEARCHING AND RETRIEVAL FOR AUTOMATED PARTS LIBRARIES 321
John D. Litke

REUSABLE SOFTWARE IMPLEMENTATION PROGRAM:
RESPONSE TO REQUEST FOR INFORMATION 335
John G. McBride

A SOFTWARE DEVELOPMENT METHODOLOGY FOR REUSABLE COMPONENT 361
Ron McCain

PRELIMINARY TECHNICAL REPORT—STUDY RESULTS 385
Dr. Daniel G. McNicholl and Christine M. Anderson

ENCOURAGEMENT OF SOFTWARE REUSABILITY 413
George W. Mebus

COMPOSITION OF REUSABLE SOFTWARE 429
John R. Mellby

MICRO ISSUES IN REUSE FROM A REAL PROJECT 443
Goeffrey O. Mendal

ACHIEVING REUSABILITY OF ADA PACKAGES 521
Susan Mickel

SOFTWARE REUSABILITY 537
J.E. Mortison

ADA* TECHNOLOGY OBJECTIVES AND PLANS (ATOP) 599
Norman S. Nise

WORKSHOP ON REUSABLE COMPONENTS OF APPLICATION PROGRAMS 637
A. Frederick Rosene

SPECIFICATION-BASED SOFTWARE ENGINEERING WITH TAGSOTM 663
G.E. Sievert

SOFTWARE QUALITY 683
Raghu Singh

FUNDAMENTAL TECHNICAL ISSUES OF REUSING MISSION CRITICAL
APPLICATION SOFTWARE 707
J.G. Snodgrass

RESPONSE TO THE TRI-SERVICE WORKING GROUP WORKSHOP
REUSABLE COMPONENTS OF APPLICATION SOFTWARE 721
J.S. Squire

BOEING MILITARY AIRPLANE COMPNAY (BMAX) 733
Earl T Startzman

A DISCUSSION OF PROTOTYPING IN THE SOFTWARE DEVELOPMENT CYCLE 757
M.K. Thomson

A DISCUSSION OF METHODOLOGIES FOR THE DEVELOPMENT OF REUSABLE SOFTWARE 773
M.K. Thomson

INFORMATION PACKAGE FOR WORKSHOP ON REUSABLE COMPONENTS OF APPLICATIONS SOFTWARE 799
Dr. Gregg Van Volkenburgh

REUSABLE SOFTWARE IMPLEMENTATION TECHNOLOGY REVIEWS 823
P. Grabow, W. Noble, C. Huang, and J. Winchester

WORKSHOP PANELS 855

PROJECT REQUIREMENTS FOR REUSABLE SOFTWARE

Lyle A. Anderson and Sivey ... Hudson

Presented at the STARS
Workshop on Reusable Software

April 9, 1985

Abstract

Developing software either for future reuse or using existing software establishes definite project management requirements. These requirements are quite different from those imposed by "from scratch/on time" software development. A software development tool, AQDS, was developed to meet those requirements as they were revealed during the automated development of TOMAHAWK Missile Launch Control Group software.

This paper is based on the work performed for the Naval Surface Weapons Center, Dahlgren Laboratory under contract N60921-82-C-A078 from December 1981 through March 1985. We were responsible for developing major portions of the Program Performance Specification (PPS), Program Design Specification (PDS), and computer program (code) for the TOMAHAWK AN/SWG-3 Launch Control Group.

Introduction

Slides used during the STARS Workshop presentation have been included in our submission to the proceedings and may be referred to for specific illustrations.

Based on our AN/SWG-3 project experience, we believe that it is important for reusable software to provide benefits to its users at every stage of development. Users will resist any system that only promises gains in future maintenance or reuse.

The Naval Surface Weapons Center, Dahlgren Laboratory supported the development of AQDS because they believed that the proposed features of the tool would enable them to meet a nearly impossible project schedule by eliminating one of the phases of software development. AQDS was designed to produce design and code from the same source, thus eliminating the code phase of development.

Benefits from using AQDS were realised from the requirements phase through design, code, testing and finally maintenance. AQDS demonstrated its ability to increase software quality and achieve heighted productivity in all phases of the project.

AQDS Used for AN/SWG-3 Launch Control Group

For the TOMAHAWK Weapon Control System AN/SWG-3 Launch Control Group (LCG) Program Performance Specification (PPS), AQDS was used, primarily, to improve the quality of the document. In addition to automated production of "pretty print" text, it guaranteed that the Input/Output tables of the Detailed Requirements sections matched the signals mentioned in the text of the processing paragraphs. It also promoted consistency of signal naming between processing sections, and provided a global data dictionary and cross-reference list which accurately reflected the text.

For the Program Design Specification (PDS), and code, AQDS provided improved quality and increased productivity. The same AQDS source was used to produce both the detailed requirements sections of the PDS

and the code. Three distinct versions of the code were produced: VAX FORTRAN 77, Rolm FORTRAN 66, and Rolm RATFOR. The latter was the target language for the project. The other two were used in debugging the design and unit testing the code.

After the design was approved at the Critical Design Review (CDR), AQDS produced from the Program Design Language (PDL) approximately 132,000 software lines of code (SLOC) in about a 72 hours of CPU time on a VAX 11-780.

The completed system cycled the first time. There were, however, design problems in about 10% of the software modules. When these wee corrected in the AQDS source database, both the design and the code were updated simultaneously.

The Initial Operational Capability (IOC) version of the AN/SWG-3 completed the acquisition milestone, DTIIe testing, in January 1985 and is scheduled for OPEVAL this summer. The "Block 1" upgrade, which is currently underway, will be constructed from the IOC version database. Only those parts actually involved in a different function in Block 1 will be changed. All other parts will remain the same. This means that changes for approved ECPs common to both baselines will only have to be made once to the common database.

AQDS Development Philosophy

The essence of AQDS development philosophy is that quality and productivity gains come from adhering to the principles of reusability. Since the design is supposed to flow from the performance requirements. they are, in a real sense, reused in the design. The same is true between the design and the code. Information that could be used again within a software project was isolated into a central unit.

AQDS was developed to be one software development tool that would be able to support reusability across all phases of software development. In order to do this it had to have the capability for the selective substitution of information from a centralized source of information. It also had to provide flexibility in output formatting to represent the information in the myriad formats required.

The tool was also used to develop itself using the bootstrap method. Each version produced its successor. This means that it benefitted from all the things to be gained from automated software development. Doing it this way also allowed the developer to be the first user of the tool and eliminated a historically significant source of conflict between applications users and support personnel.

Basic Concepts and Their Origins

AQDS was created to support a real project in "real time." This allowed us to discover real project needs and gave us the opportunity to monitor the performance of enhancements. Many features were added to the tool "just-in-time" to meet the project's needs.

Because we needed to invent as little "new" code as possible, we wanted to use existing programs or hardware configurations whenever possible. Although many manufacturers offer programs that incorporate these characteristics to some degree, Digital Equipment Corporation (DEC) was preferred for this development because they offered an integrated system of computers and word processors.

The DEC WPS-8 series of word processors includes the concept of a "paragraph library." The operator saves commonly used phrases by assigning them a name in a library document. When he or she wants to include the paragraph in a document, the operator touches two function keys and enters the name of the desired paragraph. The word processor then substitutes the contents of the paragraph for the name.

AQDS implements the substitution concept by allowing this keystroke sequence to be embedded in a document so that the substitution can take place automatically during a later expansion.

The DEC WPS-8 also includes a List Processing package, which was originally developed for producing customized "form letters." We adopted this as the means for providing selectivity in substitution.

Unlike most word processing manufacturers, DEC chose to use named fields in the list (variable part) and named references in the form (fixed part), rather than requiring list position to match form position. The main advantage to the user is that a field can be used more than once. In doing this DEC opened the door for a system of considerable power.

Phased Implementation of the Tool

Since we were supporting a real project, one of the guiding principles was to made each function work first; make it fast second. In the beginning, AQDS, on the VAX, could only assemble paragraphs into finished documents. All List Processing was done using the WPS-8. This was slow, but it work. By the time the PPS got big enough for this to be too slow, we had AQDS doing list processing. Since we had implemented the WPS-8 LP package in AQDS, we did not have to rewrite the source for the PPS.

Over the last year we have concentrated on finding the routines in which AQDS spends a lot of time and improving their efficiency. As a result, the AN/SWG-3 code can now be regenerated in only 24 CPU hours, instead of the 72 that it took a year ago.

Substitution With the Reusable Part — AN AQDS Object

A hardware part has both a physical manifestation and a label by which it is referenced. The same must be true for a software part. In AQDS, "object" with a body and a name.

Following the format of a WPS-8 Paragraph Library, the name is delimited using double angle brackets. The body is all the characters from the last right angle bracket after the name to the next double left angle bracket. An AQDS object may contain nothing, a single letter or number, a phrase, sentence, paragraph or several pages of text. There is no restriction on the size or style of the body of the object (other than it cannot contain double angle brackets). The user determines the unit of reusability represented by the object. An example of an AQDS object is on Slide 4/9/85-7.

Text Stream Orientation

AQDS is text (character string) stream oriented in input, output and processing. A database is updated from a text file or document. AQDS reads the input until it finds an AQDS object. The object is stored in the database in very much the same way that is appears in the input.

When the user wants to output the contents of the object, he simply writes a reference to the object name. The reference tells AQDS to substitute the body of the object where the reference occurs. An example of the most commonly used reference, the lexical (^) reference is on Slide 4/9/85-8. When AQDS encounters the begin lexical reference delimiters (*^), it begins a character-by-character substitution of the object body for the reference. The end lexical reference delimiters (^*) tells AQDS to continue outputting the text that follows just as it is until it encounters another reference to be processed.

Slide 4/9/85-9 shows three objects and a representation of the process of expanding the reference *^CONSEQUENCES^*. Note that none of these objects has a "hard" carriage return in them. All end of line marks are "soft." AQDS provides these "soft wraps" as part of its text stream oriented output processing.

The positioning of the margins, tabs, and centering are controlled by "Rulers" embedded in the text. This is the key to the flexibility of output formatting and is another concept taken from the DEC WPS-8 series. (Slide 4/9/85-23) We have extended the ruler concept to include the ability to generate characters within the margins. This capability allows AQDS to produce output suitable for every programming language or design language processor that we have

encountered. (Slides 4/9/85-27 through 32)

Centralization of Information

The objects discussed so far are "simple" objects. The object body is always included in its entirety whenever it is referenced. It is generally acknowledged that a reusable software part will have various representation. These might include a title, descriptions of purpose, performance and implementation, representation of the design, and representation of the code.

AQDS handles this subdivision of information by implementing the List Processing (LP) function found in DEC's WPS-8 series of word processors. A "list" object corresponds to the List Document on the WPS-8. The "form" and "spec" objects correspond to the Form and Selection Specification documents respectively.

An object can be divided into one or more records. Most list objects contain a single record. Each record may, in turn, be divided into one of more named fields. Slide 4/9/85-11 shows a list object containing one record. Slide 4/9/85-12 shows how three such list objects interact with a single form object to produce three different result documents. Slide 4/9/85-17 gives an example from the AN/SWG-3.

Flexible Database Structures

List processing gives AQDS many of the features of a very flexible relational database. Unlike most standard relational database management systems, there is no data declaration language for specifying the field and record structure. AQDS stores the list objects just as they are defined (see Slide 4/9/85-11). In effect the data language is also the data definition language. This means that not all records need to have the same fields. AQDS can also tell the difference between a field being named, but empty, and a missing field name.

With its five different reference types (` # > < =), AQDS provides explicitly typed connections between objects. (Slide 4/9/85-16) These are very flexible connections because their meaning can vary based on the form and selection specification in force for each type of reference.

Object references can be used to construct different network or hierarchical structures in addition to the natural relations. This means that the system is flexible enough to represent any data structure in its most natural form. Slide 4/9/85-25 shows the natural decomposition of "SOFTWARE-TRACKING" using lexical (^) references.

Flexible Processing (Database Queries)

Many database management systems can provide a central repository and query functions. However the format of the query results is often quite restricted.

AQDS provides all the format transformations that are needed to produce the full range of development products. It is this flexibility that sets AQDS apart from other systems. This flexibility means that a piece of code can be encoded so that is will come out in Ada, Fortran, PL/ONE or BASIC. It means that the input to a PSA database can come directly from the source for a requirements or design document. Slide 4/9/85-27 and 28 show five different reformatting of the "Main Object."

Because the database contains no fixed schema, one can change one's mind about how things are to be related and processed any time during a software development. This is very important in real projects, because all the requirements and constraints can't be known in advance.

Just as AQDS has no separate data definition language, it also has no separate database query language. This purpose of a database query is to produce some output which contains the answer. With AQDS one constructs a reference structure of list, form and selection specifications objects which produce the desired output.

List processing is activated when a list object is referenced. AQDS begins reading the form object which as been declared for the reference type. When it encounters a

reference to a field, it extracts the field value from the referenced list object. Much of the power of AQDS comes from the fact that the form to be used on subsequent references can be changed inside the current form. In this way the form which formats the main sections of a document can declare a form to do main paragraphs before referencing the subparts field of the list representing the current section.

Macro-like compound references are supported which allow one object to be passed the names of other objects and constants as parameters. These provide the closest thing to a query language. Using them and the operator interactive features in AQDS, one can build a fairly sophisticated query system.

The flexibility of processing is enhanced because, in addition to selecting fields from a record using a form object, AQDS provides the means of selecting records based on relational expressions. This is done using the Selection Specification feature of DEC List Processing. Slides 4/9/85-14 and 4/9/85-15 present examples of selection specification objects, and the command which defines them.

AQDS also includes features which allow output to go to an internal string stack. Information can be taken from the string stack and placed in one of ten (10) string variables. Additional selection features are provided through the use of a series of conditional expansion (e.g., IF, ELSE-IF, ELSE) and loop control commands (e.g., DO-WHILE, REPEAT-UNTIL) that operate on these internal string variables.

Flexible Input and Output

In creating AQDS, we made a conscious decision to reinvent as little of the word processor or the text editor as possible. There seemed to be enough of them around to satisfy just about everyone. Therefore, AQDS will accept input in the form of sequential files or DX/VMS documents. The latter can be edited using a DEC WPS-8 series word processor or one of the VAX-based word processing programs such as CT*OS or Word-11.

The word processor programs allow the inclusion of highlighting information such as super-scripting, sub-scripting, boldface, underlining, etc. Highlighting information can be entered using a text editor if the internal AQDS format is used. This format is the same as DEC's DX/VMS format. In our opinion, it is less complicated to learn than the control characters in word processing programs such as Word-Star.

Input and output can be in the form of VAX ASCII Sequential Files or DX/VMS formatted documents. The latter can be transferred to any of the DEC WPS-8 series of word processors or converted into several other VAX word processing program formats using Word-to-Word. (Word-to-Word is available from Redwood Technology Group, Inc. 170 Aquidneck Avenue, Middletown, RI 02840 Phone (401) 849-8440).

Providing Built-in CM and QA

It is simply a fact of life that for manual software development projects Configuration Management and Quality Assurance are "add-on" functions. They are not part of the main stream of a development project. There is no alternative to this unless all products of the development process are automated. In that case QA and CM can be built-in to the development process.

Some tool developers have taken this to the extreme that no delivery can be made until the document or program is perfect.

In an academic or other non-commercial setting this probably is desirable. In the world of DoD it is unacceptable. Deliveries have to be made, sometimes regardless of the state of the program. The IOC of a major system is not delayed because some improbable error path may exist in the software.

Built-in QS with AQDS

AQDS provides features that support built-in QA, but they are not mandatory. It provides the flexibility to make a delivery.

Using AQDS, documentation is "compiled" much like computer programs. A significant feature is that the word processor or text filed output from AQDS can be edited, i.e., patched, in order to make a delivery. This feature is very important when one discovers a missing equal sign and there is no time to regenerate the entire program or document from source.

Slides 4/9/85-35 and 36 list some of the functions that AQDS provides to support Quality Assurance during production and review.

The signal cross reference shown on slide 4/9/85-37 was produced by processing the PPS database with forms which eliminated all text except for signals. The signals were processed into a format shown, and sorted by title. The sorted file was processed to eliminate and annotate duplicates.

Built-in CM with AQDS

AQDS provides several CM support functions listed on slide 4/9/85-39. Perhaps the most important of these is the use of multiple databases to segregate changes. Slide 4/9/85-40 shows this schematically.

When AQDS tries to resolve a reference, it first looks in Area 0 for the object. If it is not there, AQDS scans each successive area until it either finds it or runs out of areas. This means that approved changes and baselined objects may remain untouched while trial changes are tested. It also means that several users may try changes against the baseline at the same time.

Summary

During its three years of development, AQDS has shown that a tool which supports the flexible reuse of software parts can make a significant contribution to improved productivity and quality in software development.

We believe that the tool set which supports a reusable software library must include a tool with the characteristics we have identified above.

RESUME

LYLE A. ANDERSON

Senior Programmer/Analyst

EDUCATION

M.S., Computer Science, University of Rhode Island, 1981
U.S. Naval Nuclear Power Training, Bainbridge, MD/West Milton, NY, 1971
Graduated work in Solid State Physics, Iowa State University, 1968

CURRENT EMPLOYER/SECURITY STATUS

Aquidneck Data Corporation (ADC) Dahlgren Office
01 June 1979 – Present
Full-time Employee
Available for work: Start of Contract
Industrial Secret Security Clearance
Manager, Dahlgren Operating Center

EXPERIENCE WITH AQUIDNECK DATA CORPORATION

General Experience:

Manage the TOMAHAWK Weapon Control System (TWCS) Ground Launched Cruise Missile (GLCM) and Surface Launched Cruise Missile (SLCM) project contract (N60921-82-C-A078). Responsible for assuring product quality, overall Dahlgren office operation, and customer development within the Dahlgren/Washington community. Provided technical and consulting services for the acquisition, evaluation, and management of Combat and Weapond System software. Developed and coordinated the company's efforts to apply automated structured methods to software engineering projects.

Specialized Experience:

Computer Program Design Specification Support. Worked on the design of the FLIT Program (1970). Acted as Technical Software Director for the Fire Control System (FCS) MK 113 MOD 10 (1974-1976) and for the FCS MK 117 (1976-1979). Performed the final Technical Review (TR) for both the Program Design Specification (PDS) and the Data Base Design Document (DBDD) for the FCS MK 117 and for the FCS MK 113 MOD 10.

Provided major input to the automated development of the TOMAHAWK Weapon Control System (TWCS) EX 3 MOD 2 Launch Control Group (LCG) Program Design Specification (PDS), including selection of technical approach and development of standard objects. Performed internal QA review of AQUIDNECK DATA CORPORATION (ADC)'s portion of the TWCS EX 3 MOD 2 LCG PDS prior to delivery to the NSWC TOMAHAWK project under contract N60921-82-C-A078.

Data Base Design Document Support. Performed the TR for the MK 117 and MK 113 MOD 10 DBDD. Participated in the design of the data interface for the FCS AN BQQ Sonar Set (1974-1975). Will perform internal QA review of ADC's portion of the TWCS EX 3 MOD 2 LCG Data Base Design Document (DBDD) prior to delivery to the NSWC TOMAHAWK project under contract N60921-82-C-A078.

Computer Program Code Support. Wrote and debugged code for FLIT in Assembly Language on the Honeywell DDP-24 (1970). Programmed the Honeywell H316 computer onboard the USS TREPANG SSN 674 to perform information management functions, sound velocity profile tracking, and navigational satellite alert calculations. Coded the AN/UYK-7 in CMS-2Y and Ultra-32 for FCS MK 113 MOD 10 and FCS MK 117 (1974-1979). Designed and programmed A Quality Software Development System (AQDS), the ADC Job Cost system, and WBS Cost Estimate system (1979-Present).

Informal Test Specification and Module Level Testing Support. Wrote informal test specifications and procedures and performed module level testing for all code mentioned above. Significant module testing was required for corrections made to the FCS MK 113 MOD 10 Kalman Automatic Sequential TMA (KAST) and Manual TMA (MATE) modules which allowed them to handle close-in targets. These tests included the writing of a simulation program to produce expected results which are compared to the actual program performance.

LCG Integration Test Support. Was the final technical approval authority for FCS module integration for the FCS MK 113 MOD 10 and FCS MK 117. Served as the senior NSWC representative supporting integration of the Inertial Navigation program into the SSN 688 and SSN 700 Central Computer Complex (CCC) programs, respectively. This effort included support of land-based certification testing as well as ship-board testing.

Response to Problems During Testing Support. Was the technical point of contact for all Program Trouble Reports (PTRs) against the FCS MK 133 MOD 10 and FCS MK 117. Provided initial impact assessment and assigned the PTR to the appropriate government program or contractor. Evaluated the corrective action recommended for PTRs and presented these solutions to the CCC Software Change Control Board (SCCB). Was the NUSC voting representative on the SCCB.

Review Support. Prepared and presented results of FLIT at-sea testing to OPNAV personnel (1970). Participated in the preparation and presentation of patrol reports while aboard USS TREPAND SSN 674 (1972-1974). Presented technical portion of progress reviews for FCS MK 113 MOD 10 and FCS MK 117 (1974-1979). Prepared an presented various technical and management review while at ADC (1979-Present).

LCG Module Support. Performing internal QA review of ADC's portion of the TWCS EX 3 MOD 2 LCG code prior to delivery to the NSWC TOMAHAWK project under contract N60921-82-C-A078.

Development and Maintenance and Support Software. Conceived the idea of the automated software development tool (AQDS) in January 1981. Designed and implemented the initial version by March 1981. Presented the concept of automated software development to the NSWC TOMAHAWK Project during the summer of 1981. This presentation resulted in sole source contract N60921-82-C-A078 to support the automated development of TWCS EX 3 MOD 2 LCG. Provided both actual design and coding support as well as supervision of other programmers working on needed AQDS improvements.

WCS Diagnostic Program Support. Provided technical supervision of government and contractor personnel developing and maintaining the on-line diagnostic software for the FCS MK 113 MOD 10 and FCS MK 117 (1975-1979).

Internal Software Baseline Support. Provided technical supervision of government and contractor personnel who maintained the internal software baseline for the FCS MK 113 MOD 10 and FCS MK 117 (1975-1979).

Release Management Support. Provided technical supervision of government and contractor personnel who released the software baseline for the FCS MK 113 MOD 10 and FCS MK 117 (1975-1979).

PRIOR EXPERIENCE

1974-1979 Naval Underwater Systems Center (NUSC), Newport

PHYSICIST (GS-12) and MATHEMATICIAN (GS-13), managed the development and delivery of Fire Control System (FCS) software for FCS MK 113 MOD 10 and FCS MK 117. Wrote key sections of both programs including the AN/BQQ-5 Interface Module (FDN) and Console Module Executive (FSA). Designed and implemented the warm restart function including modifications to the Data Management module (FDM). Provided technical supervision to government programmers and technical direction to contractor personnel working on these projects. Served as Assistant Test Director on the first Operational Functional Check-out of the FCS MK 113 MOD 10 aboard USS LOS ANGELES (SSN 688). Cited by NUSC and NAVSEA for contributing the solutions to several problems regarding the operational performance of both FCS. Designed and implemented the system programming changes which allowed the easy integration of the TOMAHAWK WCS functions into the FCS MK 117. Developed the fundamental design philosophy behind Combat Control System MK 1 which is the successor to the FCS MK 117.

1968-1974 U.S. NAVY
Discharged with Rank of Lieutenant, Junior Grade.

SONAR OFFICER aboard USS TREPANG (SSN 674), tested and evaluated experimental sonar systems and provided liaison between the ship and Navy Laboratories and Contractors. Developed and implemented the algorithm used in the first successful sea test of the FLIT concept in 1970.

PUBLICATIONS

The Journal of Chemical Physics:

"*Quadrupole Coupling in Dirhenium Decacarbonyl*," with S. Segel, August 1968

"Zeeman Quadrupole Resonance in Powders," with S. Segel and R. Creel, June 1969.

As Commander, Submarine Development Group TWO:

"Technical Research Contribution 3-71," 1971

For the University of Rhode Island:

"The Performance of Algorithms: A Research Plan," with E. A. Lamagna and L. J. Bass, July 1980.

"Systematic Analysis of Algorithms," August 1981.

CERTIFICATION OF RESUME

I hereby certify that the previous information regarding my experience and qualifications is true and accurate to the best of my knowledge.

Individual's Signature: _____ Date: _____

Facility Manager's Signature: _____ Date: _____

RESUME

SIVEY S. HUDSON

Junior Programmer/Analyst

EDUCATION

B.A., Mary Washington College, Art History, 1963
12 hours Computer Science (Assembly Language, FORTRAN) Charles County Community College, 1983
19 hours Math (through Calculus II) Charles County Community College and Rappahannock Community College, 1983

CURRENT EMPLOYER/SECURITY STATUS

Aquidneck Data Corporation (ADC) Dahlgren Office
03 February 1982-Present
Full-time Employee
Available for work: Start of Contract
Industrial Secret Security Clearance
Currently managing the Configuration Management (CM) documentation production project.

EXPERIENCE WITH AQUIDNECK DATA CORPORATION

General Experience:

Analyzed data to Computer Program Performance Specifications (CPPSs) and Computer Program Design Specifications (CPDSs) to produce performance and design specifications and computer programs for Common Weapon Control System (CWCS) Vertical Launching System (VLS). Directed the maintenance of the TOMAHAWK XWS 21717 EX 3 MOD 2 Program Performance Specification (PPS) computer program baseline and prepared the data base objects for its establishment. Designed the automated procedures for the Request For Action (RFA) and Software Trouble Report (STR) Resolution Efforts. Wrote the RFA and STR Procedures Manuals. Have over 2 years experience using the VAX 11/780 to produce documentation and to develop analysis procedures for this system. Experimenced in the operation of the Digital WS-200 and WS-78. Experienced in the application of techniques and procedures outlined in Military Standards (MIL-STDs) 483 and 1679.
Current Supervisor: L. Anderson

Specialized Experience:

Analyzed data to Computer Program Performance Specifications (CPPSs) and Computer Program Design Specifications (CPDSs) to produce performance and design specifications and computer programs for Common Weapon Control System (CWCS) Vertical Launching System (VLS).

Review Support: Attend TOMAHAWK Software Change Control Board (SCCB) meetings. At request of NSWC personnel, provide AQDS documentation to be presented at project meetings. Responsible for internal tracking of ADC-developed Software Trouble Reports (STRs) and follow-up on Action Items (AIs) assigned to ADC. Direct the efforts of personnel assigned to reviewing proposed changes and preparing configuration change forms. Provide consultation on the editing of Program Design Review (PDR) and Critical Design Review (CDR) information.

Development and Maintenance of Support Software: Responsible for identifying bugs in A Quality Software Development System (AQDS) and designing new features to alleviate the problems and/or

enhance the program.

Internal Software Baseline Support. Supported the software (development and product) baselines by directing the efforts of the maintenance of the TOMAHAWK XWS 21717 EX 3 MOD 2 PPS computer baseline. Prepared the data base objects for establishing the EX 3 MOD 2 PPS baseline. Implemented system changes, maintained updated printouts, analyzed baselines, and corrected errors. Supervised the editing, updating, and testing of PPS data bases throughout development and changes generated by the Change Control Board (CCB). Interfaced with NSWC personnel to discuss the status of prepared reports and change packages, updated master documents by integrating approved change packages, verified updated documents, and delivered updated copies and verification reports. Supervised conversion of CCS MK 1 Program C4T PPS (AN/UYK-44 and AN/UYK-7) Input/Output (I/O) tables into Problem Statement Language (PSL) data base; documented the conversion and data entry procedures. Coordinated the PPS EX 3 MOD 2 Request For Action (RFA) Resolution Effort system with NSWC Configuration Management/Quality Assurance (CM/QA) personnel. Designed the automated procedures for the RFA Resolution Effort. Wrote the RFA Procedures Manual and trained NSWC personnel in the use of the VAX 11/780 and the PPS computer baseline. Archived files to establish the post-Program Design Review (PDR) baselines. Established the design for the ADC CM Library which includes an efficient manual/automated storage and retrieval system and which makes it possible to produce a Library inventory, as well as many other outputs. Supervise the efforts of personnel assigned to the development of a retrieval system for the TOMAHAWK Launch Control PPS SCCB-approved changes.

Release Management Support. Directed the efforts of personnel involved in the release of the EX 3 MOD 2 PPS, PDR, and CDR Revision A magnetic and paper baselines. These efforts included the establishment of procedures which: enabled NSWC to confirm a complete and accurate delivery; informed NSWC about the form of the delivery (revision pages, computer program, etc.); provided NSWC an avenue for acknowledging receipt of deliveries and for reporting problems; defined standards for assuring the accurate duplication of software products; and produced a thorough and efficient labeling system.

PRIOR EXPERIENCE

1963-1980 King George County Public Schools

TEACHER, held various permanent and temporary teaching positions.

PUBLICATIONS

U.S. Patent 3,388,708 issued June 18, 1968

CERTIFICATION OF RESUME

I hereby certify that the previous information regarding my experience and qualifications is true and accurate to the base of my knowledge.

Individual's Signature: _____ Date: _____

Facility Manager's Signature: _____ Date: _____

PROJECT REQUIREMENTS

FOR

REUSABLE SOFTWARE

Aquidneck Data Corporation

Slide 4/9/85-1

BASED ON OUR EXPERIENCE

WITH TOMAHAWK AN/SWG-3

LAUNCH CONTROL GROUP

SOFTWARE DEVELOPMENT

Aquidneck Data Corporation

Slide 4/9/85-2

EXPERIENCE AS IT RELATES TO:

- TOOLS REQUIREMENTS
- PRACTICAL PROJECT REALITIES

Aquidneck Data Corporation

Slide 4/9/85-3

AQDS WAS DEVELOPED FOR THE

NAVAL SURFACE WEAPONS CENTER

AT DAHLGREN, VA.

UNDER CONTRACT N60921-82-C-A078.

IT WAS USED TO PRODUCE

THE PPS,THE PDS, AND THE CODE.

Aquidneck Data Corporation

Slide 4/9/85-4

Automated Quality Development System

AQDS is a Trademark of Aquidneck Data Corporation

- SUBSTITUTION
- SELECTIVITY
- CENTRALIZATION

Aquidneck Data Corporation

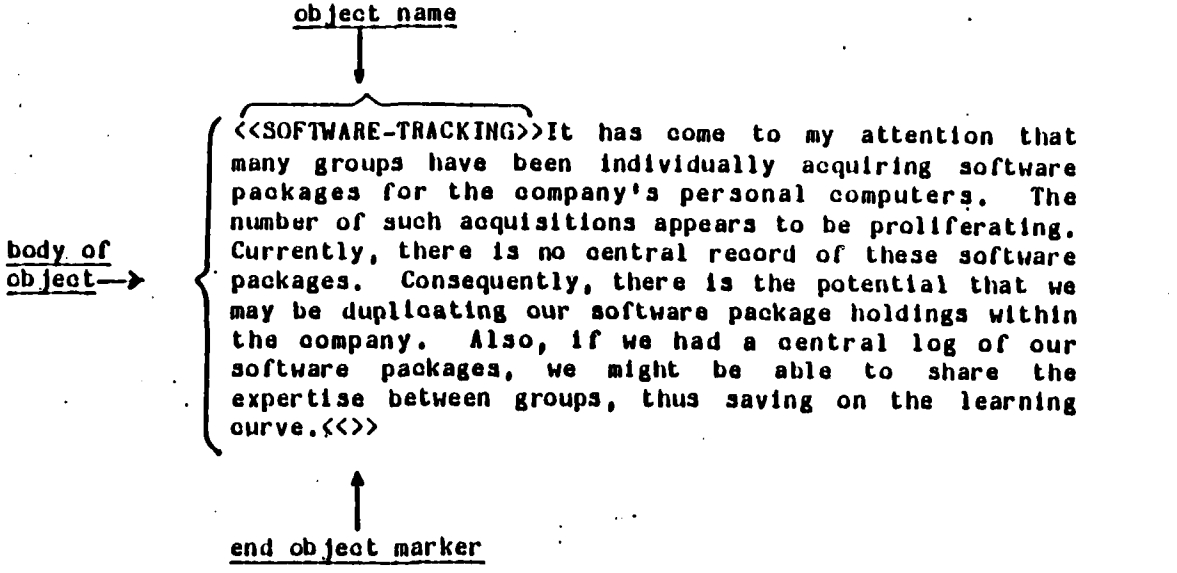
Slide 4/9/85-5

SUBSTITUTION

Aquidneck Data Corporation

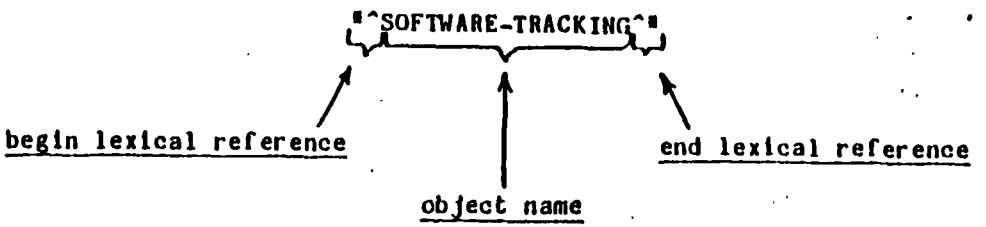
Slide 4/9/95-6

AN AQDS OBJECT



Aquidneck Data Corporation

LEXICAL REFERENCES



Aquidneck Data Corporation

USING OBJECTS AND LEXICAL REFERENCES

```
<<CONSEQUENCES>>Consequently, ^^POSSIBLE-DUP^^  Also, ^^EXPERTISE^^<<>>

<<EXPERTISE>>if we had a central log of our software packages, we might be able to
share the expertise between groups, thus saving on the learning curve.<<>>

<<POSSIBLE-DUP>>there is the potential that we may be duplicating our software
package holdings within the company.<<>>
```

Consequently, ^^POSSIBLE-DUP^^ Also, ^^EXPERTISE^^

Consequently, there is the potential that we may be duplicating our software package holdings within the company. Also, ^^EXPERTISE^^

Consequently, there is the potential that we may be duplicating our software package holdings within the company. Also, if we had a central log of our software packages, we might be able to share the expertise between groups, thus saving on the learning curve.

Aquidneck Data Corporation

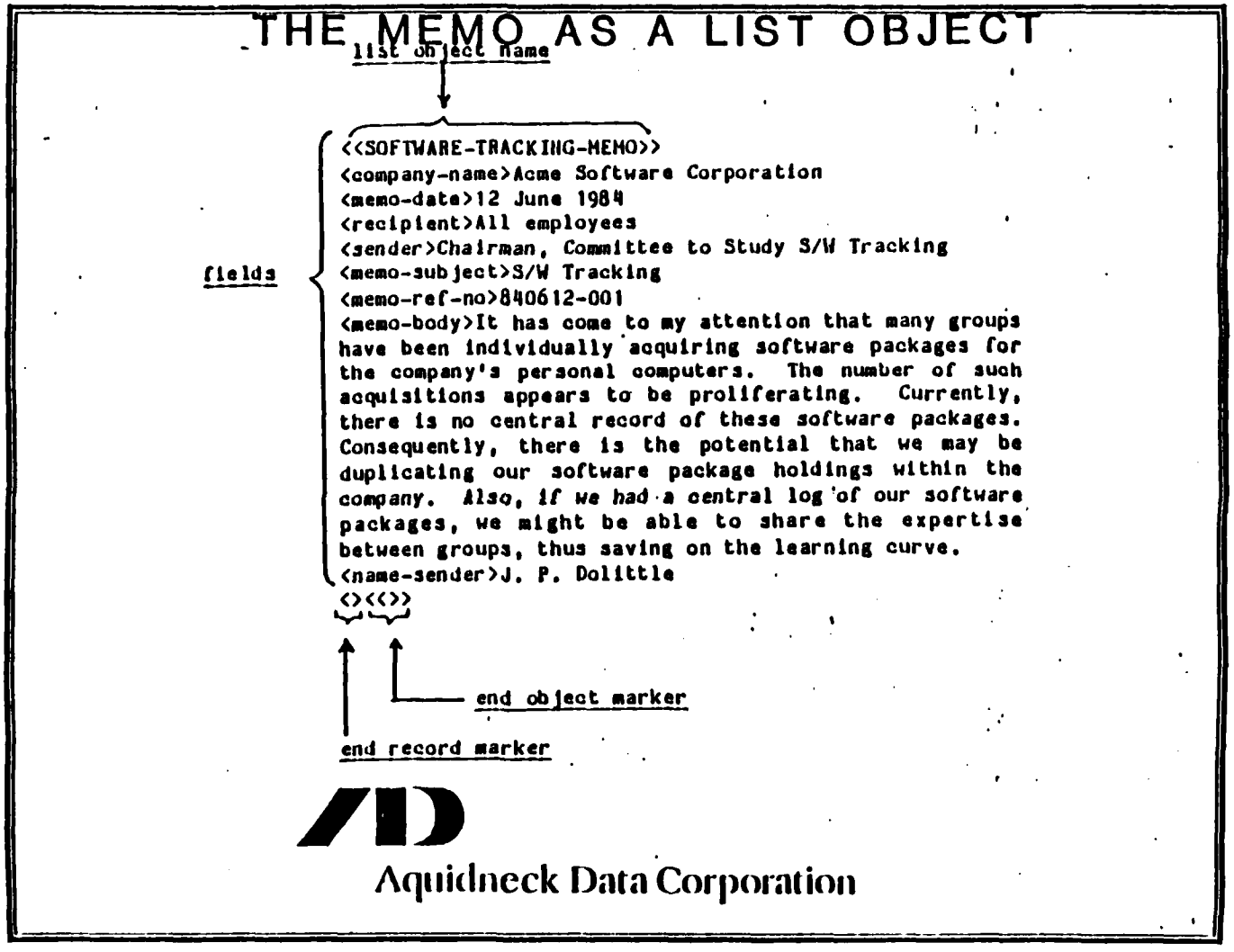
Slide 4/9/85-9

CENTRALIZATION

Aquidneck Data Corporation

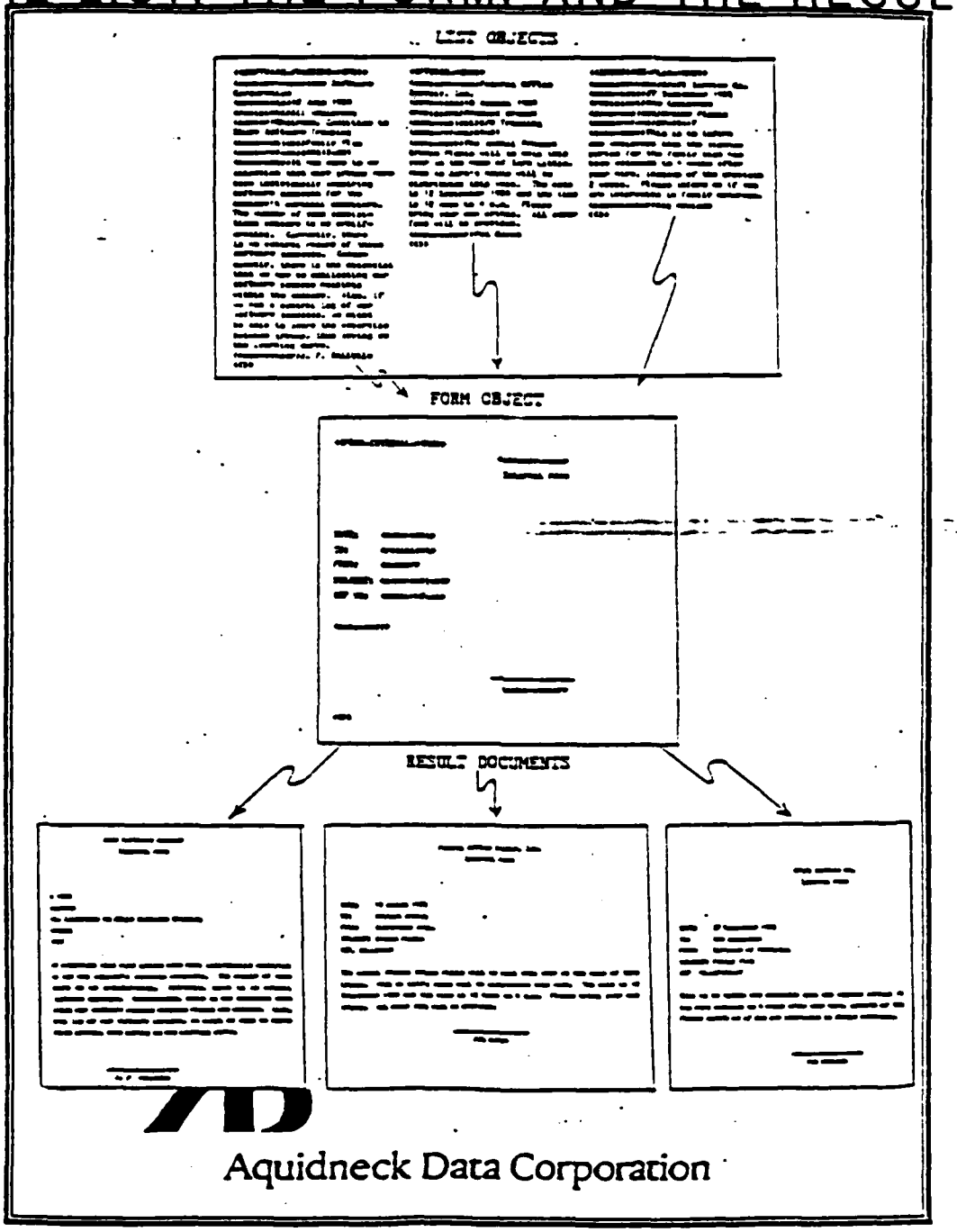
Slide 4/9/85-10

THE MEMO AS A LIST OBJECT



Slide 4/9/85-11

THE LIST, THE FORM, AND THE RESULT



Slide 4/9/85-12

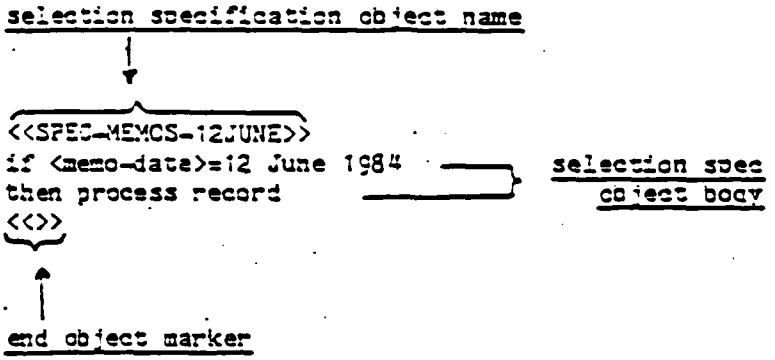
SELECTIVITY

Aquidneck Data Corporation

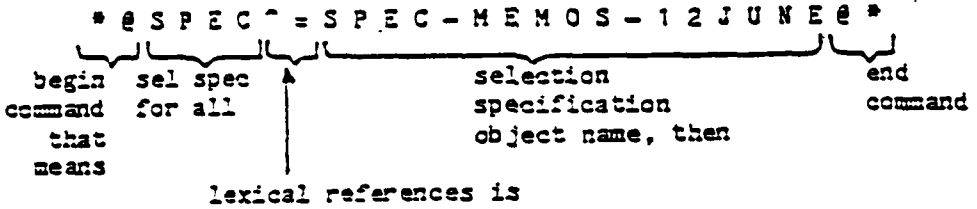
Slide 4/9/85-13

SELECTION SPECIFICATIONS

THE SELECTION SPECIFICATION OBJECT



THE SELECTION SPECIFICATION COMMAND



Aquidneck Data Corporation

Slide 4/9/85-14

SELECTION SPECIFICATIONS (Cont'd)

MORE SELECTION SPECIFICATIONS

```
<<ACCEPTABLE-VALUES-SPEC>>
if <last>=Davis
   or =Anderson
   or =King
then process record
<<>>
```

```
<<SEVERAL-FIELDS-SPEC>>
if <state>=Alaska
   or =Hawaii
and <credit>=good
but not if <occupation>=Policeman
then process record
<<>>
```

```
<<COMBINATION-SPEC>>
if <married>=Y
then process record

or if <state>=Alaska
   or =Hawaii
and <credit>=good
but not if <occupation>=Policeman
then process record
or if <state>=California
and <occupation>=Beach Bum
then process record
<<>>
```

Aquidneck Data Corporation

Slide 4/9/85-15

REFERENCE TYPES

There are five reference types:

name	symbol	example
lexical	^	"^OBJECT-NAME^"
value	#	"#OBJECT-NAME#"
input	>	">OBJECT-NAME>"
output	<	"<OBJECT-NAME<"
update	=	"=OBJECT-NAME="

Aquidneck Data Corporation

THE MAIN OBJECT

```
<<e-PROC-VALID-RESTART-EPO>>
<name>e-PROC-VALID-RESTART-EPO
<typ>PROC
<DESC>If the #``LAUNCH-SEQ:IIP``# is
#^ALL-MISSILES-SAFED^#, the
#~~LAUNCH-SEQ:IIP~~# shall be reset to
the prior to pre-armed state,
#^MISSION-DATA-LOADED^#.  After
performing the above,
#^TERM-SUBFUNCTION-PROC^#.
<c-pdl>#^IF:e-LSI=ALL-MISSILES-SAFED^#
#^SET-ARRAY:~~g-LAUNCH-SEQ~~:=e-PLN-ID=:
#g-MISSION-DATA-LOADED#^#
#^CALL-SUBR:W-EXTRACT:e-EPC-LSI-EXTRACT-
ARGS^#
#^ENDIF^#
<PPS>#^PROC-VALID-RESTART-EPO^#,#^DPC-EP
O-RESTART^#
<><<>>
```

Aquidneck Data Corporation

AQDS PROVIDES

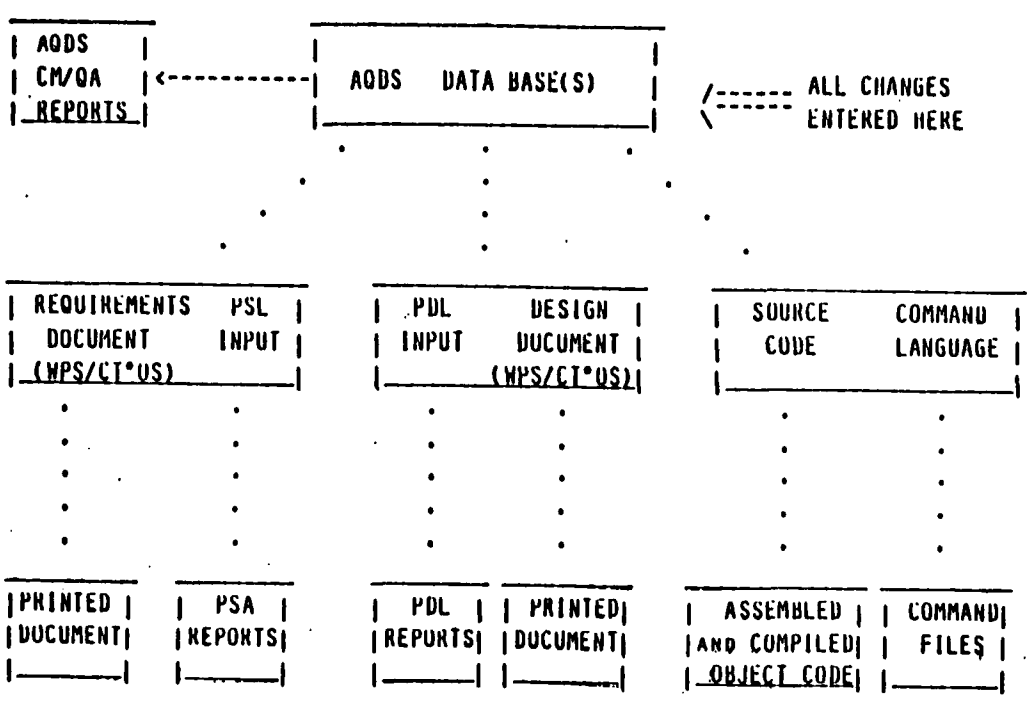
FLEXIBILITY

THAT PERMITS

REUSABILITY

Aquidneck Data Corporation

RELATIONSHIP BETWEEN DEVELOPMENT PRODUCTS



Aquidneck Data Corporation

FLEXIBILITY

Aquidneck Data Corporation

FLEXIBLE

- INPUT
- PROCESSING
- OUTPUT FORMAT

Aquidneck Data Corporation

Slide 4/9/85-21

INPUT: USER CHOICE

- ANY VAX TEXT FILE
- DX/VMS DOCUMENT
- INTERNAL FORMAT

Aquidneck Data Corporation

EASE OF DATA ENTRY

¶···SECTION 1 - CONGREGATING ALL DOCUMENTATION¶

¶

L---T---W----------R----------

¶···1. In the "marked-up" version of the Revision B PPS, make ·→ SOFT WRAPS

·····copies of all the functions affected that need editing. ·

·····Place each section in its own folder labeling the folder ·

·····with the module's name.¶

¶···2. Gather any other necessary information needed for the ·

·····editing process such as the NAMEMOSIG provided from TNCS ·

·····EX3 MOD2 XNS 21717, Revision A of the PPS. This will help ·

·····in most instances knowing what object to pull for that ·

Aquidneck Data Corporation

PROCESSING

- RELATIONAL DATABASE
  (free format)
- SCHEMA FREE
  (alphabetical order only)
- EXPLICITLY TYPED CONNECTIONS
- MULTIPLE RELATIONS/
  NETWORKS/HIERARCHIES

Aquidneck Data Corporation

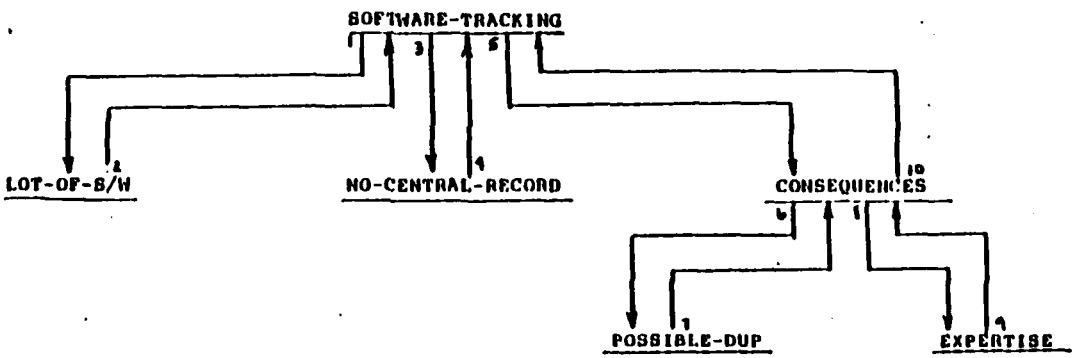
THE REFERENCE TREE

An organizing object contains lexical references to other objects in the order in which they are to processed. The processing is called expansion.

<<SOFTWARE-TRACKING>>^LOT-OF-S/W^ ^NO-CENTRAL-RECORD^ ^CONSEQUENCES^<<>>

<<CONSEQUENCES>>Consequently, ^POSSIBLE-DUPS^ Also, ^EXPERTISE^<<>>

If the expansion begins with the object, SOFTWARE-TRACKING, the objects that it references will be expanded in the order depicted below.



Aquidneck Data Corporation

OUTPUT

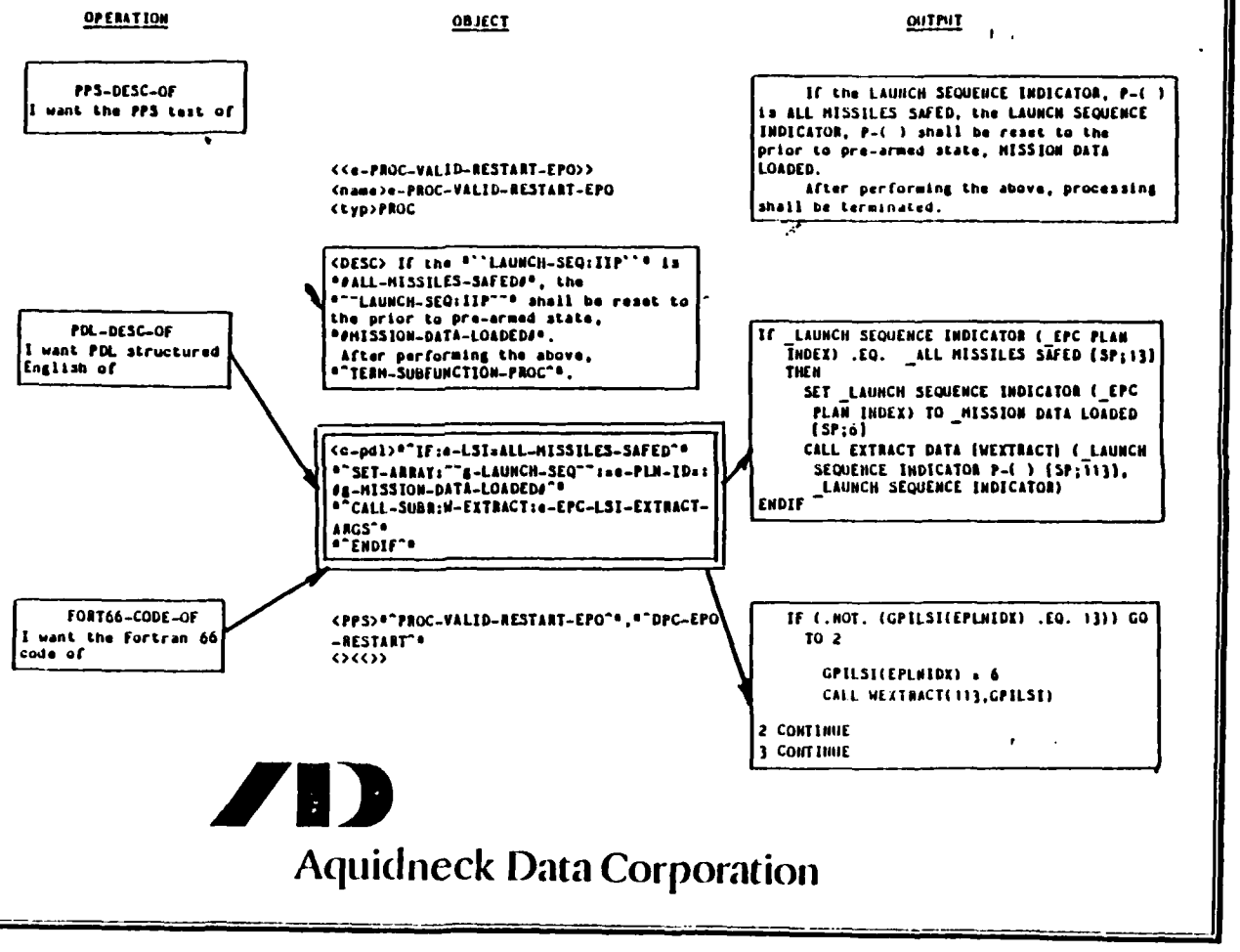
OUTPUT FOR ALL PHASES OF

SOFTWARE DEVELOPMENT

- REFORMATTING
- RULERS

Aquidneck Data Corporation

REFORMATTING THE OUTPUT



REFORMATTING (Cont'd)

OTHER OUTPUT PRODUCED FROM THE <c-pdl> FIELD OF THE REQUIREMENT OBJECT

```
<c-pdl>@^IF:e-LSI=ALL-MISSILES-SAFED^@
      @^SET-ARRAY:^^g-LAUNCH-SEQ^^]=e-PLN-ID=1#g-MISSION-DATA-LOADED#^@
      @^CALL-SUBR:W-EXTRACT:e-EPC-LSI-EXTRACT-ARGS^@
@^ENDIF^@
```

OPERATION

```
CODE-OF
I want the RATFOR code of
```

OUTPUT

```
IF (GPILSI(EPLNIDX) == 1)
  $(
   GPILSI(EPLNIDX) = 6

  CALL WEXTRACT(11),GPILSI)
  $)
```

```
e-PDL-OF
I want the PDL preamble of
```

```
#**********************************************************************#
#*                                                                    *#
#*                    If the _LAUNCH SEQUENCE INDICATOR _,            *#
#*                P-() is _ALL MISSILES SAFED (SP;) , the             *#
#*                _LAUNCH SEQUENCE INDICATOR _, P-() shall be         *#
#*                reset to the prior to pre-armed state, _MISSION     *#
#*                DATA LOADED (SP;) .                                 *#
#*                    After performing the above, processing          *#
#*                shall be terminated.                                *#
#*                                                                    *#
#**********************************************************************#
```

Aquidneck Data Corporation

THE EXTENDED RULER

- SPECIFY CHARACTERS TO APPEAR OUTSIDE MARGINS
- SPECIFY CHARACTERS TO MARK CONTINUATION LINES

Aquidneck Data Corporation

RULER EXAMPLE

```
<<RULER-EXAMPLE>>
This is a test object for showing the many possibilities of the
ADC extended ruler. It has some soft wraps and center marks, as
well as hard returns.
There is some text, and then some "code":
ALPHA = BETA
IF (ALPHA .GT. DELTA) THEN
   PROCESS BETA INTO GAMMA
                          DO FOR ALL BETA
                        SLICE UP CUCUMBER
                              END DO
   DETERMINE WHICH VAR IS THE CURRENT VALUE OF CUCUMBER
END IF
<<>>
```

Ruler Object →

```
<<R=TYPE-JUSTIFY-EXP>>
----------L-------------------------C----------J
     TYPE *, '                                  '
<<>>
```

Expanded Output

```
TYPE *, ' This is a test object  for  showing  the '
TYPE *, ' many possibilities of the  ADC  extended '
TYPE *, ' ruler. It has some soft wraps and center '
TYPE *, ' marks, as well as hard returns.          '
TYPE *, ' There  is  some  text,  and  then   some '
TYPE *, ' "code":                                  '
TYPE *, ' ALPHA = BETA                             '
TYPE *, ' IF (ALPHA .GT. DELTA) THEN               '
TYPE *, '                    PROCESS BETA INTO GAMMA '
TYPE *, '                       DO FOR ALL BETA    '
TYPE *, '                     SLICE UP CUCUMBER    '
TYPE *, '                           END DO         '
TYPE *, '  DETERMINE  WHICH  VAR  IS  THE  CURRENT '
TYPE *, ' VALUE OF CUCUMBER                        '
TYPE *, ' END IF                                   '
```

Aquidneck Data Corporation

COMMENT RULERS

```
<<R=FORTRAN>>
L-------W--T--T--T--T----------------------R
     *
<<>>

<<R=FORTRANC>>
---L----------------------------------------R
C
<<>>
```

Regular Ruler

Comment Ruler

Object to be Expanded

```
<<MONSTER-MANIA>>*@R=FORTRAN@*
   APPLES = ORANGES  /* This comment ends a line*/
   IF (APPLES .OR. /*  This one is mid-line  */ ORANGES) THEN
      /* This has its own line */
/* This one starts the line */        END IF
<<>>
```

Expanded Output

```
      APPLES = ORANGES
C     This comment ends a line
      IF (APPLES .OR.
C      This one is mid-line
     *    ORANGES) THEN
C     This has its own line
C     This one starts the line
     *     END IF
```

Aquidneck Data Corporation

RULER FAMILIES

```
<<THINGS TO DO>>
The following chores should be performed daily:
*@R=+@*
1).   Vacuum
2).   Dust
*@R=+@*
(including the baseboards)
*@R=-@*
3).   Laundry
*@R=-@*

These chores should be performed weekly:
*@R=+@*
1).   Scrub kitchen floor
*@R=+@*
(and wax)
*@R=-@*
2).   Wash windows
3).   Shampoo carpets
*@R=-@*
<<>>
```

Object to be Expanded

Ruler Objects

```
<<R=IND1>>
L---T---T---T---T---T---T----------------------------------R
<<>>

<<R=IND2>>
----L---T---T---T---T---T----------------------------------R
<<>>

<<R=IND3>>
--------L---T---T---T---T----------------------------------R
<<>>
```

Expanded Output

```
The following chores should be performed daily:
    1). Vacuum
    2). Dust
        (including the baseboards)
    3). Laundry

These chores should be performed weekly:
    1). Scrub kitchen floor
        (and wax)
    2). Wash windows
    3). Shampoo carpets
```

Aquidneck Data Corporation

FLEXIBILITY TO:

- MEET CHANGING PROJECT DEMANDS
- MEET DELIVERY SCHEDULES
- PROVIDE BUILT-IN CM & QA
- ASSURE EASY EVOLUTION OF USER EXPERTISE

Aquidneck Data Corporation

QUALITY ASSURANCE

Aquidneck Data Corporation

QA DURING PRODUCTION

- AUTOMATED GENERATION OF SIGNAL TABLES FROM ACTUAL USE IN TEXT
- CONSISTENT USE OF DATA AND SIGNAL NAMES OR TITLES
- MAINTENANCE OF A UNIFORM STYLE AND FORMAT
- IDENTIFICATION AND EXACT REPRODUCTION OF FREQUENTLY USED WORDS AND PHRASES
- RECORD OF ALL CHANGES MADE TO THE DOCUMENT AS A RESULT OF SOFTWARE TROUBLE REPORTS (STRs)

Aquidneck Data Corporation

QA DURING REVIEW

- FINDS ALL OCCURRENCES OF A PARTICULAR SIGNAL, VALUE, OR SUB-PROCESS
- REPORTS THE HIERARCHICAL STRUCTURE OF DOCUMENTS
- CREATES KEYWORD-IN-CONTEXT LISTS OF ALL DATA BASE NAMES
- CONSTRUCTS A GLOBAL CROSS-REFERENCE OF SIGNALS
- ISOLATES ALL ENTRIES THAT CONTAIN A COMMON CHARACTER STRING IN A SIGNAL CROSS-REFERENCE

Aquidneck Data Corporation

CROSS-REFERENCE AND STRING SEARCHING CAPABILITIES

AQDS can produce a cross-reference of signals or other referenced data used in a document. The cross-reference gives an alphabetical listing by the signal title, data type, paragraph in which it was referenced, the AQDS object name of the signal, its reference type encoding, the name of the object in which it was used, and the number of occurrences in the paragraph. This information is very useful for analyzing signal usage and selecting objects for editing.

Central DPS [VALUE] in 3.4.9 (#LDPS:#" in PPS-LDPS-INITIALIZATION) (3 times)
Central DPS [VALUE] in 3.4.9.2 (#LDPS:#" in PPS-LDPS-INIT-PROC)
Central DPS [VALUE] in 3.4.9.2.2 (#LDPS:#" in LDPS-INS-INTERFACE-INIT)
Central DPS [VALUE] in 3.4.9.2.2 (#LDPS:#" in LDPS-TDPS-INTERFACE-INIT)
Central DPS [VALUE] in 3.4.9.2.3 (#LDPS:#" in PPS-LDPS-RESTART)
Dedicated DPS [VALUE] in 3.4.9.2.2 (#DDPS:#" in LDPS-DDPS-INTERFACE-INIT) (2 times)
FLOATING POINT ACCUMULATOR 0 of the System [USED] in 3.4.9.2.3 (>FAC0:IISYS>" in PPS-LDPS-RESTART) (2 times)
FLOATING POINT ACCUMULATOR 1 of the System [USED] in 3.4.9.2.1 (>FAC1:IISYS>" in PPS-LAUNCH-CRIT-FAULT-CHKS) (3 times)
GMT FROM INS of the System [USED] in 3.4.9.2.2 (>GMT-FROM-INS:IISYS>" in LDPS-INS-INTERFACE-INIT)
NOE FIELD VALUE of the System [SET] in 3.4.9.2.2 (<NOE-FIELD-VALUE:IISYS<" in LDPS-AFTER-INITIAL-DISPLAY)
NON-VOTS [VALUE] in 3.4.9.2.2 (#NON-VOTS:#" in LDPS-AFTER-INITIAL-DISPLAY)
NUCLEAR EVENT DETECTED ALERT INDICATOR [SET] in 3.4.9.2.3 (<NUCLEAR-EVENT-DETECTED-ALRT:COALR<" in PPS-LDPS-RESTART
OIDT() ACTIVE of the System [USED] in 3.4.9.2.2 (>OIDT()-ACTIVE:IISYS>" in LDPS-OIDT-INTERFACE-INIT)
ORDNANCE [VALUE] in 3.4.9.2.2 (#ORDNANCE:#" in LDPS-AFTER-INITIAL-DISPLAY)
SYSTEM TIME of the System [SET] in 3.4.9.2.2 (<SYSTEM-TIME:IISYS<" in LDPS-INS-INTERFACE-INIT)

AQDS can also select all entries from the cross-reference that contain a common element. The search string, which is not word-oriented, may be any string of characters. For example, the search string, IISYS, was found in the following entries from the previous cross-reference.

FLOATING POINT ACCUMULATOR 0 of the System [USED] in 3.4.9.2.3 (>FAC0:IISYS>" in PPS-LDPS-RESTART) (2 times)
FLOATING POINT ACCUMULATOR 1 of the System [USED] in 3.4.9.2.1 (>FAC1:IISYS>" in PPS-LAUNCH-CRIT-FAULT-CHKS) (3 times)
GMT FROM INS of the System [USED] in 3.4.9.2.2 (>GMT-FROM-INS:IISYS>" in LDPS-INS-INTERFACE-INIT)
NOE FIELD VALUE of the System [SET] in 3.4.9.2.2 (<NOE-FIELD-VALUE:IISYS<" in LDPS-AFTER-INITIAL-DISPLAY)
OIDT() ACTIVE of the System [USED] in 3.4.9.2.2 (>OIDT()-ACTIVE:IISYS>" in LDPS-OIDT-INTERFACE-INIT)
SYSTEM TIME of the System [SET] in 3.4.9.2.2 (<SYSTEM-TIME:IISYS<" in LDPS-INS-INTERFACE-INIT)

CONFIGURATION MANAGEMENT

Aquidneck Data Corporation

CM FEATURES

- LOCK DATA BASE OBJECTS TO PREVENT UPDATES
- USE MULTIPLE DATA BASES - ALLOWING SEGREGATION OF CHANGES
- IDENTIFY VERSION OF OBJECT
- IDENTIFY ALL OBJECTS MODIFIED DURING A SPECIFIED TIME PERIOD
- PRODUCE BASELINE AND FINAL VERSION OF OBJECT FOR COMPARISON

Aquidneck Data Corporation

AQDS

ALLOCATION OF MULTIPLE DATA BASES FOR CHANGE SEGREGATION

A LOGICAL AQDS DATA BASE

AREA:		TYPICAL USE:
0	FIRST PHYSICAL AQDS DATA BASE	TRIAL CHANGES
1	SECOND PHYSICAL AQDS DATA BASE	APPROVED CHANGES
2	THIRD PHYSICAL AQDS DATA BASE	BASELINE CONFIGURATION ITEM
3	FOURTH PHYSICAL AQDS DATA BASE	CM OBJECTS AND STANDARDS

Aquidneck Data Corporation

OPERATING AT THE APPROPRIATE LEVEL OF USER EXPERTISE

MENU	STORE FILE	OPERATOR ENTERED COMMANDS
S - Select an Area	(Contents of SELCOP.DAT)	AQDS (SEL/AREA0)
C - Copy Object from Database	SEL/AREA0	ENTER Area 0 Data Base File Name (EXAMPLE)
U - Update Objects in Database	EXAMPLE	AQDS (COPY)
E - Expand Database Object	COPY	Extract Objects from the Database: COPY
A - Archive Database Object	EXAMPLE.DAT	name of OUTPUT file or document (EXAMPLE.DAT)
H - Help	FIRST-EXAMPLE	File Name SYS$SYSDEVICE:[EXAMPLE]EXAMPLE.DAT;1
M - More Menu Selections	^Z	Data Base Object Name (FIRST-EXAMPLE)
	(Recalling the Store File)	Data Base Object Name (^Z)
	AQDS:@SELCOP	

Aquidneck Data Corporation

OBJECTS WHICH PRODUCE NAMENOSIG

```
/*    1  12-19-84 18:49 NAMENOSIG (UNLOCKED)*/
<?<F$-DESC>>{ *@NAME@* =<DESC>}<<>>

/*    2  12-19-84 18:49 NAMENOSIG (UNLOCKED)*/
<?<F$-MAIN-PARAGRAPH>>*@END C@*
*@FORM#=F$-TITLE@*
*@FORM<=F$-TITLE@*
*@FORM>=F$-TITLE@*
*@FORM==F$-TITLE@*
*@R=4@*
<hdr>
                                        *@NAME@*
*@FORM#=F$-UTITLE@*
<pr#>  <utitle>
*@FORM#=F$-TITLE@*
*@FORM^=F$-DESC@*
*@R=1@*
     <DESC>
*@FORM^=F$-MAIN-PARAGRAPH@*
<tbl-figs>*@BEGIN C@*
<SUBP>*@END C@*
*@R=1@*
*@BEGIN C@*
<<>>

/*    3  03-25-85 09:01 NAMENOSIG (UNLOCKED)*/
<?<PARA-OF>>*@FORM@=F$-MAIN-PARAGRAPH@*
*@BEGIN C@*
*@P1@*
*@END C@*
<<>>
```

Aquidneck Data Corporation

EXPAND WITH NAMENOSIG

DECLASS

3.4.6 Declassification

The purpose of the Declassification function is to declassify the *#LDPS#* and *#DDPS#* during an orderly Launch Control Group shutdown, and to declassify *#S-TLAM#*s as required during normal operations.

DECLASS-INPUTS

3.4.6.1 *#DECLASS#* Inputs

A list of the input signals required by the Declassification function is provided in *#TBL#-OF:DECLASS-INPUT-TABLE#*. A detailed listing of each signal is found in Appendix C, and signal cross reference is found in Appendix B.

DECLASS-PROCESS

3.4.6.2 *#DECLASS#* Processing

The processing requirements of the Declassification function are presented in terms of the following subfunctions:

a. Orderly Shutdown

b. *#S-TLAM#* Declassification

The following subparagraphs specify the detailed performance requirements of the above subfunctions.

DECLASS-SHUTDOWN

3.4.6.2.1 Orderly Shutdown

The purpose of this subfunction is to remove all classified data from the *#LDPS#* and *#DDPS#*, and to remove all classified mission data from the System Disk.

Aquidneck Data Corporation

MACRO ISSUES IN REUSE FROM A REAL PROJECT

Thomas D. Arkwright

Lockheed Missiles and Space Company
Sunnyvale, CA

1.0 OVERVIEW*

The emerging Ada* technology has already surfaced in Sunnyvale, California, at Lockheed Missiles and Space Company (LMSC), where Ada is now in use on a multi-billion dollar project.

A business-like, and intellectually honest, flavor characterized the voluntary Ada acceptance process at LMSC for this project. However, we believe that the acceptance of Ada hinged on the underlying structure of our acceptance procedures, which followed two concurrent tracks, the macro and the micro.

When we speak of macro issues we reference themes that affect the bottom line. Macro issues agitate or comfort decisionmakers. When we speack of micro issues, we reference those themes that affect the implementation. Micro issues motivate the technical judgements of the implementors.

Chart 1 enumerates some selected macro issues in reuse. The purpose of this paper is to report that, in our experiences, the ability to project for decisionmakers the expected benefits of our maintenance cost reduction program was perhaps the biggest factor in the acceptance of Ada for one major project. In our experience, LMSC and military decisionmakers have had strong concern for reuse's impact on maintenance costs, to the point that LMSC's maintenance cost reduction program is now perceived as the most influential macro issue in reuse. Accordingly, we have selected that topic as the basis for illustrating this article.

*Our thanks to Jim Kaplan, LMSC for his contribution to this study. *Ada is a registered trademark of the U.S. Government (AJPO)

Chart 1. Selected Macro Issues in Reuse

Achieving Universal Buy-In on Ada Software Reuse

Publishing a Schedule of Benefits

Implementing a Development Cost Reduction Program

Implementing a Maintenance Cost Reduction Program

Preparing the Implementors

A shared feature of the issues in Chart 1 is that they are intelligible to the class of higher decisionmakers that no one would expect to discuss the details of Ada and software engineering. For example, a software engineer should be able to assert whether or not the preparation of implementors is being accomplished. A second shared property of each macro issue is the availability of an observable impact that a decisionmaker may attempt to control.

2.0 DYNAMICS OF MACRO ISSUES

Macro issues are important because a successful review of these features is a precondition for the managers who occupy the most responsible positions to give their approval to implement the micro issues. In general, a decisionmaker will not fund a technically elegant new solution if an adequate old solution is less costly. On the other hand, a businessman is virtually compelled to follow a cheaper elegant solution if the macro and micro issues appear manageable.

We shall see in this paper that reuse appears to be of immense importance on the macro level

because of its effects on cost; whatever the individual decisionmaker's personal opinion, if all other macro and micro issues are equal, cost will generally carry the day.

3.0 CASE STUDY OVERVIEW: QUANTIFYING THE IMPACT OF REUSE ON MAINTENANCE COST REDUCTION

A project at LMSC recently decided to use Ada instead of its original baseline, JOVIAL, to implement its software. In retrospect, after all the other issues feeding the decision had been addressed, we were able to reach a decision only after quantifying the maintenance phase costs with a projection. Building on LMSC's earlier work in reuse we built a model which reflects our perception of the role of reuse on the project. By our calculation, the largest contributor to maintenance phase savings would be software reuse.

Our case study projected the maintenance cost of coding in JOVIAL and that of coding in Ada. The projections were performed using two sets of assumptions which we call the nominal case and the worst-for-Ada case. The actual projected values conform in the aggregate to generally accepted notions of the cost of maintenance (see section 6.1, Calibration). Some values are not based on the literature; for example, the software reuse experience lies largely in the future rather than in the past. However, our assumptions are clearly laid out in section 3, and alternative values can be readily submitted to the model, on demand, in order to assess the sensitivity or impact of alternate values for selected parameters. To reduce the need for multiple alternative analyses, we have worked through a second set of assumptions significantly less optimistic for Ada.

A key feature of this model is its realistic accomodation of de factor practive, in that error removal and new requirements are likely to continue to be somewhat difficult to distinguish in the maintenance timeframe of the project being discussed. This feature reflects the fact that while maintenance budgets are programmed in advance of need, the maintenance effort has two legitimate components from the budget officer's point of view. The uncontrollable component associated with errors, and the controllable component which responds to requirements.

We have formulated a uniform model suitable for projecting maintenance costs of various languages. We have included terms in the model which take into account special features of Ada such as the ability to accommodate a significant amount of software reuse. An instance of this model is given for JOVIAL in Table I, and an instance for Ada is given in Table II. The possibility of extracting a delta is based on the idea of applying the rules of the model to a baseline set of data (JOVIAL) and to a treatment set of Data (Ada). In reality, every output of the model is a virtual delta. For example, the total lines of JOVIAL code on hand in 2005 A.D. could be compared with the total lines of Ada code on hand in that year. This richness facilitates the review of assumptions, and enable a fuller interpretation of the ultimate cost impact of language selection during the maintenance phase.

The data mentioned above represent the best available information. For example, the divisor (2000 lines of code) used to compute the JOVIAL unadjusted yearly maintenance manyears (see column L) is a DoD historical statistic for the average number of lines of code (LOC) maintained (serviced) per year. This agrees roughly with LMSC experience. Other data are more solid; for example, the timespan (1900 - 2008) is the current planned-for maintenance phase. Still other data fall unequivocally into the realm of assumptions. For example, the staff attrition curves for each language are presented as inversely proportional. This and all other assumptions are equitable and logically defensible, but clearly are not subject to strict empirical confirmations: the future lies ahead of us, and projections interpret the future.

3.1 Nominal Case Assumptions.

The following subparagraphs describe the assumptions used in the nominal case for each language. The exposition proceeds from left to right for each term (column) of the model. For each datum subject to and/or resulting from computation, the formula for its term is shown beneath the column heading. When no formula is given, the values of the column are strictly expository, and may (see column N) enter into other formulas. Please examine Tables I (JOVIAL Projected Maintenance Costs: Nominal Case) and II (Ada Projected Maintenance Costs: Nominal Case) while reading sections 3.1.1 through 3.1.11.

3.1.1 Year (Column A).

Each year is listed consecutively, starting with 1990, the start of the maintenance phase, and

continuing through 2008, the last year of maintenance.

3.1.2 Total Lines of Code (LOC) On Hand (Column B).

Our working assumption is that in 1990 there will be 500,000 lines of code on hand, regardless of implementation language. Thereafter, the total LOC on hand will be the LOC on hand from the previous year (Column B) minus the LOC supplanted from the previous year (Column F) plus the new LOC actually composed from the previous year (Column K). Total LOC on hand is at the start of the year. A LOC is defined as an executable high order language (HOL) instruction, exclusive of reusable off-the-shelf modules. Earlier work at LMSC by our group showed that the comparability of LOC across languages can be increased by making a distinction between declarative and executable lines of code, in the sense of the RCA Price S cost estimation tool.

3.1.3 Predicted Yearly Errors (Column D).

The predicted yearly errors (LOC that will have to be serviced due to errors) is defined as being two or less of the total LOC on hand. The formula for determining where, in this range, the predicted yearly errors will fall is as follows: Two percent of the total LOC is determined, and then multiplied by a factor. Those factors are listed at the bottom of Tables I and II as variable V1. They are figured as follows:

3.1.3.1 Factor for Jovial.

Errors decay smoothly over a nineteen year period down to ten percent (of two percent).

3.1.3.2 Factor For Ada.

Debugging is assumed to be less than the effort of JOVIAL due to Ada's ability to avoid global scopes, to reduce module interactions, and to deliver other benefits discussed in the literature. With Ada, errors are assumed to be removed over a four year period. Subsequently, errors level off at a nominal five percent (of two percent).

3.1.4 Supplanted Code Removed (Column F).

For both JOVIAL and Ada, the supplanted code removed as new lines are added (we assume that errors removed is a tit for tat replacement) is assumed to be twenty-five percent of the new LOC required. This percentage has been deemed rational by experienced hands at LMSC, but it could not be grounded in any known data. The assumption favors JOVIAL, in our opinion. JOVIAL programmers are sometimes reluctant to jettison code that just might be needed elsewhere. In Ada, this can be determined more readily. We are not convinced that all supplanted JOVIAL code will actually be removed.

3.1.5 New LOC Required (Column H).

The new LOC required values for JOVIAL and Ada are assumed to be ten percent of the total LOC on hand minus the predicted yearly errors. Thus, ten percent of the total LOC on hand will be serviced annually. That ten percent will be split between LOC serviced due to errors (uncontrollable maintenance) and LOC necessary due to new requirements (controllable maintenance). Note the explicit claim that the total level of effort is mostly a reflection of the budget available, rather than the number of errors or the number of new requirements in need of service.

3.1.6 Reusable LOC For New Requirements (Column I).

This term quantifies the LOC not composed, due to reusable code that already exists.

3.1.6.1 For Jovial.

Jovial does not constructively support reusability, so this entry is zero.

3.1.6.2 For Ada.

For Ada, the number of reusable LOC is determined as a percentage of the new LOC required. This is 40 percent, starting in 1991, with a straight line growth curve to 60 percent in 2008.

An earlier LMSC study gathered estimates in a survey of LMSC Ada programmers. Their consensus was that eventually the maintenance reuse factor would be eighty percent. The sixty percent eventual reuse factor used here is a more conservative assumption. To further reduce any potential overestimation of software reuse in Ada, we have started with a forty percent factor in 1990, increasing in a fairly straight line through 2008, as described by the vector, V2, at the bottom of Table II (See also Ramachendra, P (1984) Software Development Evolves into Software Engineering, Computer Design 23, 10, pp. 165-176, for a more optimistic scenario.). The reusa-

Table I. JOVIAL PROJECTED MAINTENANCE COSTS: NOMINAL CASE

A	B	D	F	H	I	K	L	N	P	Q
YEAR	TOTAL LOC ON HAND	PREDCTD. YEARLY ERRORS	SUPPLNTD CODE REMOVED	NEW LOC REQUIRED	REUSABLE LOC FOR NEW REQT	NEW LOC ACTUALLY COMPOSED	UNADJSTD MAINT. MANYEARS	PROJECT ATTRITN. PERCENT	ADJUSTED YRLY MAINT. MANYEARS	CUMULATIVE ADJUSTED MANYEARS
	PREV(B) – PREV(F) + PREV(K)	B*.02*VI	H*.25	(B*.1)-D	0	H-I	K/2000 + D/2000		(L*N/6) + L	PREV(Q) + P
1990	500,000	10,000	10,000	40,000	0	40,000	25	.33	26	26
1991	530,000	10,070	10,733	42,930	0	42,930	27	.33	28	54
1992	562,198	10,120	11,525	46,100	0	46,100	28	.35	30	84
1993	596,773	10,145	12,383	49,532	0	49,532	30	.37	32	116
1994	633,922	10,143	13,312	53,249	0	53,249	32	.39	34	150
1995	673,859	10,108	14,319	57,278	0	57,278	34	.41	36	186
1996	716,817	10,035	15,412	61,646	0	61,646	36	.42	38	224
1997	763,052	9,920	16,596	66,386	0	66,386	38	.44	41	265
1998	812,841	9,754	17,883	71,530	0	71,530	41	.47	44	309
1999	866,489	9,531	19,279	77,117	0	71,117	43	.49	47	356
2000	924,327	9,243	20,797	83,189	0	83,189	46	.5	50	406
2001	986,719	8,880	22,448	89,791	0	89,791	49	.52	54	459
2002	1,054,062	8,432	24,243	96,974	0	96,974	53	.54	57	517
2003	1,126,793	7,888	26,198	104,792	0	104,792	56	.57	62	578
2004	1,205,387	7,232	28,327	113,306	0	113,306	60	.59	66	645
2005	1,290,366	6,452	30,646	122,585	0	122,585	65	.6	71	715
2006	1,382,305	5,529	33,175	132,701	0	132,701	69	.62	76	792
2007	1,481,831	4,445	35,934	143,738	0	143,738	74	.65	82	874
2008	1,589,634	3,179	38,946	155,784	0	155,784	79	.67	88	962
SUMS:		161,108	402,157	1,608,630	0	1,608,630	885		962	

VI TAKES ON THE FOLLOWING VALUES: 1, .95, .9, .85, .8, .75, .7, .65, .6, .55, .5, .45, .4, .35, .3, .25, .2, .15, .1
TOTAL LINES OF CODE ON HAND IN 1990 IS 500,000

Table II. ADA PROJECTED MAINTENANCE COSTS: NOMINAL CASE

A	B	D	F	H	I	K	L	N	P	Q
YEAR	TOTAL LOC ON HAND	PREDCTD. YEARLY ERRORS	SUPPLNTD CODE REMOVED	NEW LOC REUIRED	REUSABLE LOC FOR NEW REQT	NEW LOC ACTUALLY COMPOSED	UNADJSTD MAINT. MANYEARS	PROJECT ATTRITN. PERCENT	ADJUSTED YRLY MAINT. MANYEARS	CUMULATIVE ADJUSTED MANYEARS
	PREV(B) + PREV(E) + PREV(K)	B*.02*V1	H*.25	(B*.1)-D	H*V2	H-I	(K/3000) + (D/3000) + (I*.5/3000)		(L*N/6) + L	PREV(Q) + P
1990	500,000	10,000	10,000	40,000	0	40,000	17	.67	19	19
1991	530,000	4,240	12,190	48,760	19,504	29,256	14	.67	16	35
1992	547,066	875	13,458	53,831	22,071	31,760	15	.65	16	51
1993	565,369	565	13,993	55,971	23,508	32,463	15	.62	16	67
1994	583,839	584	14,450	57,800	24,854	32,946	15	.6	17	84
1995	602,335	602	14,908	59,631	26,238	33,393	16	.59	17	101
1996	620,821	621	15,365	61,461	27,658	33,804	16	.57	18	119
1997	639,259	639	15,822	63,287	29,112	34,175	16	.54	18	137
1998	657,612	658	16,276	65,104	30,599	34,505	17	.52	18	155
1999	675,841	676	16,727	66,908	32,785	34,123	17	.5	18	174
2000	693,238	693	17,158	68,631	35,002	33,629	17	.49	19	192
2001	709,709	710	17,565	70,261	37,238	33,023	17	.47	19	211
2002	725,166	725	17,948	71,791	38,767	33,024	18	.44	19	230
2003	740,243	740	18,321	73,284	40,306	32,978	18	.42	19	249
2004	754,899	755	18,684	74,735	41,852	32,883	18	.41	19	269
2005	769,099	769	19,035	76,141	43,400	32,741	18	.39	20	288
2006	782,804	783	19,374	77,498	44,949	32,549	19	.37	20	308
2007	795,979	796	19,700	78,802	46,493	32,309	19	.35	20	328
2008	808,587	809	20,013	80,050	48,030	32,020	19	.33	20	348
SUMS:		26,240	310,987	1,243,947	612,365	631,582	231		348	

V1 TAKES ON THE FOLLOWING VALUES: 1, .4, .08, .05, AND REMAINS CONSTANT AT .05.
V2 TAKES ON THE FOLLOWING VALUES: 0, .4, .41, .42, .43, .44, .45, .46, .47, .49, .51, .53, .54, .55, .56, .57, .58, .59, .6

bility level in the first year of maintenance (1990) is assumed conservatively to be zero, due to overhead costs. Such overhead may include establishing a data base for cataloging modules or establishing procedures for reuse.

3.1.7 New LOC Actually Composed (Column K).

This term expresses the LOC that will acdtually have to be composed for each language.

3.1.7.1 Jovial.

The new LOC actually composed is the new LOC required minus the LOC that are reusable. Since JOVIAL has no reusability, this figure is the new LOC required.

3.1.7.2 Ada.

The new LOC actually composed is the new LOC required minus the LOC that are reusable.

3.1.8 Unadjusted Yearly Maintenance Manyears (Column L).

This is the LOC for new requirements divided by the productivity rate for new development plus the predicted yearly errors divided by the productivity rate for error correction. In the case of Ada, a burden is included that amounts to half of the cost of actually compassing the code reused.

3.1.8.1 Jovial.

The productivity rate for new development of JOVIAL is assumed to be 2000 lines per year per person (see 3.0). The productivity rate for JOVIAL error correction is assumed to be 2000 lines per year per person, also.

3.1.8.2 Ada.

The productivity rate for new development of Ada is assumed to be 3000 lines per year per person, and that of Ada correction is assumed to be 3000 lines per person per year.

The reason for adding in the burden is that reuse is not free. We have to include the cose of locating, evaluating, purchasing, integrating, testing, and documenting the reusable code. As expressed here, the burden is almost certainly too high, but does reflect our desire to avoid any unduly liberal conclusions.

3.1.8.3 Note.

The Ada/JOVIAL differentials are conservative with respect to the recent literature (e.g., Boehm, B. and T Standish (1983) Software Technology in the 1990's, IEEE Computer, 16, 11, pp. 30-37.), which uniformly suggests productivity improvements might be up to an order of magnitude greater, based on standardization, program generators and so on. These are likely to more available to Ada. Recently, usable data have started to emerge from the aerospace firms' experience with Ada. While encouraging, the interpretation of interlanguage coding rates is beyond the scope of this paper.

3.19 Project Attrition Percentage (Column N).

This represents the fact that people will leave the project.

3.1.9.1 Jovial.

We assume that a person will stay on the project for three years at the start of maintenance, i.e., 33 percent of the maintenance staff will turn over annually; at the other extreme, 2008, we assume that the maintainer will endure 1.5 years, i.e., a 67 percent turnover rate. Attrition percentages are straightlined between these endpoint values during the maintenance phase.

3.1.9.2 Ada.

We assume the inverse of JOVIAL, starting with a 67 percent turnover rate in 1990 straightlined to 33 percent in 2008. This reflects increasing availability of Ada programmers over time, and the converse for JOVIAL. It would be plausible to argue that Ada programmers will be readily available in 1990; this would be a less conservative assumption, but more favorable to Ada.

3.1.10 Adjusted Yearly Maintenance Manyears (Column P).

We assume equal costs for training a new staff member for JOVIAL and Ada. Thus the adjusted yearly maintenance manyears is the unadjusted maintenance manyears plus the cost of training the new staff members. Training cost is determined by multiplying the unadjusted maintenance manyears by the project attrition percentage and dividing by six. (Training costs are assumed to be 300 hours or 1/6 manyear for classroom and on the job development.)

3.1.11 Cumulative Adjusted Manyears.

This value represents the accumulating sum of individual yearly projections.

3.2 Worst-Case-For-Ada Assumptions.

The following subparagraphs describe those assumptions which differ from the nominal case in that they are considerably more pessimistic for Ada than the already conservative nominal case assumptions.

We believe this set of assumptions represents the lower bound on reasonable Ada assumptions, in that stricter assumptions begin to appear improbable, and would simply represent a future failure to follow all of the Ada methodology guidance developed for our project. In that case Ada would have been applied as if it were a traditional language; this would lead toward a traditional maintenance cost profile.

3.2.1 Changes To The Ada Nominal Case.

We have introducted two changes to the Ada nominal case set of assumptions to derive this still more conservative projection.

First of all, the error removal rate for Ada is assumed to be less than that of the Ada nominal case, leveling off at .05 (of two percent) after six years. This change would be expressed in the vector variable, V1 (1,

Second, the productivity differential of Ada programmers is moved to 2500 from 3000, nearer parity with JOVIAL (2000 lines per year).

3.2.2 Changes To The Jovial Nominal Case.

The single change to the set of nominal case JOVIAL assumptions is that the yearly predicted errors will reach .1 (of two percent) in nine years (via intervals of .1 (of two percent) in all nine years) rather than in nineteen years, and stabilize at a lower .05 (of two percent) in the tenth year.

4.0 METHOD

We made an a prior decision to base our findings on two deltas, the cumulative delta of the year 2008, and the cumulative delta of the year 1999. The latter delta is of considerable interest because historically, large systems are frequently abandoned after ten years of deployment.

On the other hand, because of the presumed higher maintainability of Ada code we might expect that total system replacement would be forestalled, or that the step function replacement concept could give way, because of Ada and the associated methodology, to a continuous refresh concept which would obsolete the idea of totally replacing a system. Be that as it may, by evaluating both the 1999 and 2008 deltas we are adopting the most conservative possible posture based on a plausible set of assumptions which are clearly laid out and available for alternative simulations.

5.0 RESULTS

Key findings of the model are given in Table III, and described in sections 5.1.5.4. Section 5.5 presents the dollar impact in present dollars. Finally, section 5.6 describes the relative contribution to savings of software reuse, as shown in Table IV.

5.1 The Nominal Case In The Year 2008.

The model predicts 962 manyears of maintenance phase effort for JOVIAL, versus a lesser 348 manyears for Ada.

5.2 The Nominal Case In The Year 1999.

The model predicts 356 cumulative manyears of effort in the maintenance phase through 1999 for JOVIAL, versus a lesser 174 manyears for Ada.

5.3 The Worst-For-Ada Case In The Year 2008.

The model predicts 998 manyears of effort for JOVIAL in the maintenance phase through 2008, versus a lesser 416 manyears for Ada.

5.4 The Worst-For-Ada Case In The Year 1999.

The model favors Ada through 1999 timeframe with 208 manyears of maintenance phase effort versus a greater 359 manyears for JOVIAL.

The reader will note that even though the rate of removal of JOVIAL errors speeds up in comparison with the JOVIAL nominal case, the manyears projected increase somewhat. The accelerated reduction in errors speeds up the servicing of new requirements, which leads to a larger JOVIAL code body after 1991 under assumptions intended to favor JOVIAL!

Table III. CUMULATIVE MAINTENANCE PHASE COSTS JOVIAL/ADA IN 1999 AND 2008 (IN MANYEARS)

	YEAR	JOVIAL	ADA
NOMINAL CASE			
	2008	*962*	*348*
	1999	*356*	*174*
WORST-FOR-ADA-CASE			
	2008	*998*	*416*
	1999	*359*	*208*

Table IV. CUMULATIVE MAINTENANCE PHASE COSTS, JOVIAL/ADA, WITH AND WITHOUT ADA REUSE (IN MANYEARS)

	YEAR	JOVIAL	ADA	ADA, NO REUSE	SAVINGS DUE TO REUSE
NOMINAL CASE					
	2008	*962*	*348*	*689*	*56%*
	1999	*356*	*174*	*253*	*43%*
WORST-FOR-ADA CASE					
	2008	*998*	*416*	*817*	*69%*
	1999	*359*	*208*	*301*	*62%*

5.5 Dollar Impact Of Ada Versus Jovial.

In the most conservative plausible scenario, (the worst-for-Ada case in 1999) a savings of over eighteen million dollars would arise from selecting Ada. In the most probable scenario (Nominal case in 2008) the dollar differential would grow to seventy-four million dollars. These dollar estimates are based on constant present manyear costs of $120,000. A $150,000 figure might have been more appropriate, but the lesser figure gives a more conservative result.

5.6 Contribution Of Reuse.

We can isolate the contribution of reuse to the maiontenance phase savings of Ada. In the nominal case, through the year 2008, a fifty-six percent of the savings are due to reuse. Through the year 1999, forty-three percent of the savings are due to reuse. To develop these results we simply altered column I and column L in Table II. The coefficients for V2 (see 3.1.6.2) were set to zero in column I; the cost of reuse (see 3.1.8 and 3.1.8.2) was removed from column L.

In the worst-for-Ada case, sixty-nine percent of the savings is due to reuse by 2008, while sixty-two percent is attributable to reuse through 1999. These findings are reflected in Table IV.

6.0 DISCUSSION

6.1 Calibration.

A conventional rule-of-thumb is that seventy percent of a system's costs (assume a ten year life cycle) is maintenance. If we assume 500,000 lines of JOVIAL code were produced at 2000 lines per manyear then 250 manyears would have gone into development. A total of 356 years is predicted by the model in Table III to be the maintenance cost. Adding the development and maintenance figures gives 606 manyears of which the maintenance portion is nearly sixty percent. This a posteriori calibration suggests that the assumptions applied to the model are both realistic and conservative.

A second a posteriori calibration comes from a nearby firm which was able to demonstrate 40%

reuse during development, on a 4,000 line (est.) project. The vector, V2 described in 3.1.6.2, appears to be unduly conservative in light of this experience, related to us as this paper reached its final form.

6.2 Cost Impact Of Management's Reuse Policies In The Maintenance Phase. A recent LMSC study expended significant effort to spell out development procedures that would promote maintainability. Discussions there of safe structures, cost impacts, reusable generics, coding techniques, maintenance, and methodology were designed to communicate software development strategies that can yield highly modular code suitable for a future maintenance technique of swapping in reusable modules. We assume that management can control to some extent the exploitation of software reuse technology. Clearly, managers could suppress the practice entirely; presumably, they could also encourage reuse; this is what we see happening now at LMSC.

6.3 Conclusion Of Case Study.

Based on the assessment presented above, we conclude that the maintenance phase cost of using Ada on the project being examined here will be considerably less than the maintenance cost of using JOVIAL. We expect reuse to be the main contributor to the lesser cost.

7.0 GENERAL CONCLUSION

Our review of maintenance costs' sensitivity to reuse is telling. While our projections must be treated as highly tentative, macro issues in reuse can have (and have had at LMSC) a decisive influence on the acceptance of Ada (and therefore the acceptance of reuse).

However, many times we in the technical Ada community have not been meeting the needs of decisionmakers for understandable impact projections associated with macro issues in reuse. As noted at the outset of Section 2.0 above, micro issues can not gain funding unless the macro issues have been sold to the people in the most responsible positions. We at LMSC have sold the maintenance cost reduction macro issue, and the result is that use of Ada has been funded for a major program.

RESUME

Thomas D. Arkwright

RECENT LEADERSHIP EXPERIENCE (LMSC 1982-1985):

Designed/funded/managed 63 miniprojects (over 100 people) in Software Engineering to assess and manage impacts of Ada on LMSC projects.

Funded/established/headed the new Ada Technology Support Laboratory.

Leading the AIDER project, an editor-resident Ada expert system for Ada programmers.

Administered a series of LMSC training development projects to evolve a second generation of training materials for Ada.

Administering LMSC methodology development for architecting Ada systems (tied to DoD life cycle reviewers).

Designed/funded/leading the GIT-Ada project, an artificial intelligence system to teach Ada PDL to CDR reviewers.

Designed/leading evaluations of Ada compilers, advanced architectures, and other Ada tools.

EARLIER LEADERSHIP (1973-1982):

DLIFLC (a joint DoD installation) 1975-1982.

- Designed/funded/initiated the Instructional Technology Department (computer/video R & D for training.)

- Responsible for all technical software development and deployment (non-administrative software) 1975-1982).

UNIVERSITY OF QUEBEC

- Led research projects and research classes as Assistant Professor (1973-1975).

RECENT EXPERIENCE (LMSC 1982-1984):

Developed $2,500,000 in new business in 1984 (LMSC's first Ada revenues), in three programs.

Set forth the LMSC Ada Training Plan.

Delivered first Ada training to managers and lead programmers at new LMSC Austin Division.

Converted the first DoD project to use Ada (effected change from baseline (JOVIAL) to Ada for a multibillion dollar USAF distributed communications project); concluded December 20, 1984, by customer, project management, and LMSC management.

Participated in proposal work.

ACADEMIC WORK (1962-1974):

McGill University (Ph.D) computer simulation of a natural language system, numerous awards, 1969-1974).

Notre Dame University (Master's) merit scholarship, 1969-1971.

Notre Dame University (Bachelor's) competitive awards, 1962-1966.

RECENT CONSULTING EXPERIENCE WORK (1983-1985):

IBM (Santa Teresa Labs) -

Designed/directed development of the user manual for a new main-frame fourth generation database product: The Information Facility (TIF END USER Manual: IBM Publications Number TBD).

Digital Research Inc. (DRI) -

Coded the QA software for a distributed network and file server.

USAF (CINCPAC/Hawaii) -

Delivered a course: Ada for Managers

Commercial (Northern California) -

Contract development of custom turnkey multiuser business applications.

Macro Issues in Reuse from a Real Project

Thomas D. Arkwright
Lockheed Missiles and Space Company
Sunnyvale, CA

REQUIREMENTS FOR A DESIGN TO PROJECT THE COST OF REUSE DURING MAINTENANCE

- Compare JOVIAL and Ada under two sets of assumptions
- Project the maintenance costs for a real project
- Reflect the nature of maintenance budgets
- Permit changes to the assumptions and the model

DEFINITIONS

Macro issues

Micro issues

Attributes

SELECTED MACRO ISSUES IN REUSE

Achieving Universal Buy-In on Ada Software Reuse

Publishing a Schedule of Benefits

Implementing a Development Cost Reduction Program

Implementing a Maintenance Cost Reduction Program

Preparing the Implementors

ROLE OF KEY DECISION MAKERS

Obligation to avoid risk

Need to react to cost data

RESULTS I

CUMULATIVE MAINTENANCE PHASE COSTS JOVIAL/ADA IN 1999 AND 2008 (IN MANYEARS)

	YEAR	JOVIAL	ADA
NOMINAL CASE			
	2008	962	348
	1999	356	174
WORST-FOR-ADA-CASE			
	2008	998	416
	1999	359	208

RESULTS II

CUMULATIVE MAINTENANCE PHASE COSTS, JOVIAL/ADA, WITH AND WITHOUT ADA REUSE (IN MANYEARS)

	YEAR	JOVIAL	ADA	ADA, NO REUSE	SAVINGS DUE TO REUSE
NOMINAL CASE					
	2008	962	348	689	56%
	1999	356	174	253	43%
WORST-FOR-ADA CASE					
	2008	998	416	817	69%
	1999	359	208	301	62%

DISCUSSION

Tentativeness of the model and findings

Calibration of the model

Impact of management's policies

Lesser cost of Ada maintenance

Reuse as the biggest money saver

WORKSHOP ON REUSABLE COMPONENTS OF APPLICATION SOFTWARE

Tom Bowen
Boeing Aerospace Company
Software Technology
P.O. Box 3999
Seattle, WA 98124

The following section contains a position paper on reusable software. Section 1 describes BAC capabilities and experiences related to developing and reusing embedded software. Section 2 summarizes the objectives of the Information Science Technology organization and contains the resume of the principal investigator for reusable software.

SECTION 1. POSITION PAPER: Technology Considerations for Reusable Software

Industry capacity to produce reliable, maintainable software for embedded systems is not keeping pace with demand. One accepted approach to reducing costs is to reuse software that has been developed for similar applications. The expected benefits include (1) increased productivity through avoiding duplication of effort and (2) increased quality and reduced risk through use of proven products.

Successful reuse of embedded software depends on (1) techniques for developing software that is inherently reusable, (2) methodologies for reusing software in the life cycle, and (3) tools and a library system that promote the reuse of software. Initial reusability thrusts should emphasize building on current technologies to enable economic reuse of software in the near future. For example: reusability levels-of-abstraction should be compatible with phases and products identified in DoD-STD-SDS; techniques should take advantage of features provided by Ada; and reusable library systems should be compatible with APSE tools. Long-term efforts should include automation of software development processes and software generation and should not be constrained by current technology. Other specific areas need to be addressed in evolving a reusable software technology and should include:

(a) Criteria for selecting software application areas, functions, and components as economically reusable (e.g., degree of commonality between applications).

(b) Characteristics that promote reuse of software (e.g., generality, modularity, independence, self-descriptiveness, and simplicity). These characteristics should be the basis for development standards and techniques and for measurements indicating the degree of reusability.

(c) Criteria for accepting and retaining a software entity for reusable library (e.g., frequency of use and degree of reusability).

(d) Validation and verification responsibilities for reusable library entries.

(e) Access and security considerations for a reusable software library.

(f) The impact of reusability methodologies and techniques on existing DoD policies and standards.

(g) Product liability should DoD supply reusable software to a developer.

(h) Use of knowledge-based engineering and rapid prototyping capability.

(i) Types of incentives that can be provided to encourage development and use of reusable software and the impact of reusability on developer inventiveness.

(j) Technology transfer approach and mechanisms for integrating with system acquisition practices, including training.

SECTION 2. PERSONNEL

The following paragraphs contain a brief description of the goals and objectives of the Information Science Technology organization and the resume of our principal investigator for reusable software.

Information Science Technology

The Information Science Technology organization is responsible for developing and applying a high level of competence in all pertinent technology areas, for leading in development and implementation of concepts and systems relative to the overall acquisition, collection, processing, reduction, display, control, encrypting/decrypting; for control of security; and for leadership and focal point direction of information science technology and applications for BAC. In coordination with other technology organizations, Information Science Technology has joint responsibility for all systems incorporating computers, displays and controls, and has primary responsibility for computer and display related subsystems.

RESUME
THOMAS P. BOWEN

EDUCATION
BS, Mathematics
University of Washington, 1970

EXPERIENCE

Mr. Bowen has 20 years experience in computer-related fields and 15 years experience with Boeing Aerospace Company. He has held project lead engineer positions in software systems, software design, and software test organizations and has supported proposals, conceptual designs, and analyses and evaluation efforts. Major projects and proposals supported include: B-1 Avionics, Morgantown Personal Rapid Transit (MPRT), Digital Avionics Integration System (DAIS), Precision Emitter Location Strike System (PELSS), SATIN IV, ASW/Standoff Weapon, and SPADOC. Research and development contracts and proposals supported include: Software Interoperability and reusability, Quality Measurement for Distributed Systems, SPADOC Interim Communications and Data Base Manager, Specification of Software Quality Attributes, and Common Ada Missile Packages (CAMP).

Mr. Bowen is responsible for the 1985 Reusable Software IR&D effort. The overall, multiyear objective for this effort is to implement a computerized, reusable software library system to support company-wide software development for embedded systems.

For the past four years Mr. Bowen was working on software quality metrics R&D contracts with the Rome Air Development Center (RADC). He was the principal investigator for the Specification of Software Quality Attributes contract (RADC) contract F30602-82-C-0137). The prime tasks for this contract were to develop the methodologies and procedures for specifying and measuring software quality, to refine the quality framework, and to prepare guidebooks for use in specification and evaluation of quality for Air Force command and control software. He was also involved in the efforts for enhancing the RADC quality framework for distributed systems (RADC F30602-80-C-0330) and for interoperability and reusability (RADC contract F30602-80-C-0265).

Prior to these assignments, Mr. Bowen developed a Software Development/Management Methodology tutorial and a Software Development Methodology document. These presented a development model and methodology recommendations in order to encourage a more unified company approach to software development. He supported a project to install a message handling and retrieval system for SPADOC-3 in the Cheyenne Mountain complex-performing an industry survey of 14 data base management systems (DBMS) and an indepth analysis of 4 DBMS's to support recommendations. He contributed to an AIA study of principles and policies for computer software acquisition (TMC118-7). He also has taught a graduate-level, company-sponsored course in computer architecture.

RELEVANT PUBLICATIONS

Specification of Software Quality Attributes, Thomas P. Bowen, et. al., Boeing Documents D182-11678-1, -2, -3, November 1984 (final report, prepared for RADC under contract F30602-82-C-0137)

Software Quality Measurement for Distributed Systems, Thomas P. Bowen, et. al., RADC-TR-83-175 (3 volumes), July 1983

Software Interoperability and Reusability, P. Edward Presson et. al., RADC-TR-83-174 (2 volumes), July 1983

Software Development Methodology, Thomas P. Bowen, Boeing Document D180-26176-1, October 1981

Conceptual Framework for Reusable Software, Robert W. Lawler and Thomas P. Bowen, Boeing Document D180-25964-1, February 1981

RELEVANT TECHNICAL PAPERS AND PRESENTATIONS

"Quality Metrics in Software Development", Spring COMPCON 85, San Francisco, CA, February 1985, IEEE 85CH2135-2, P. 308

"Software Quality Metric Data Collection", SESAW III, San Francisco, CA, October 1984

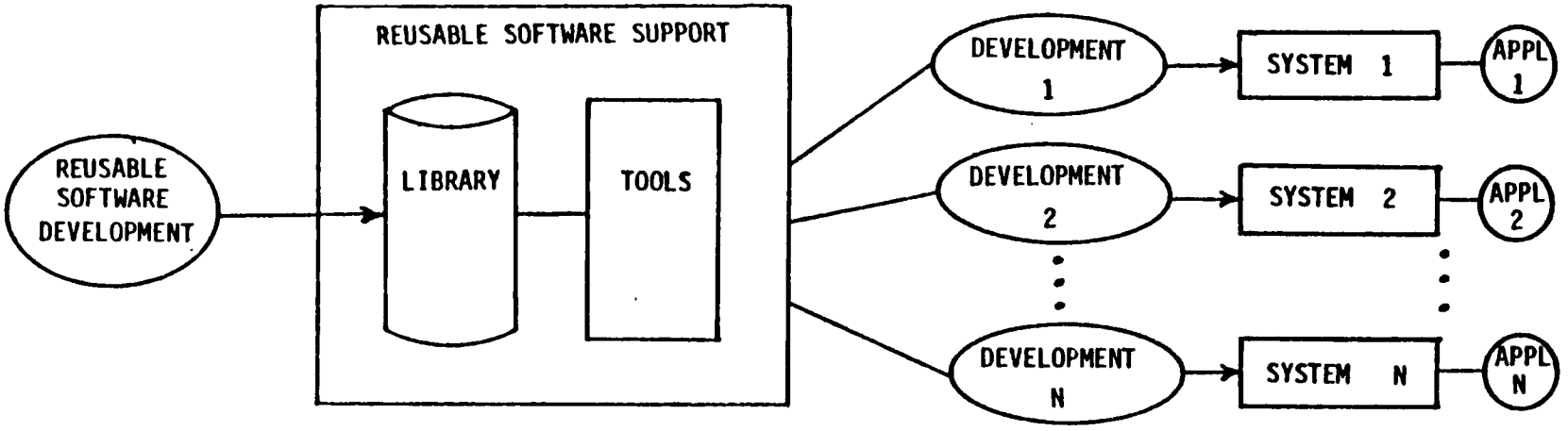
"Metrics for Evaluating Software Quality", 1983 Pacific Northwest Software Quality Conference, Corvallis, OR, September 1983

"Software Quality Measurement for Distributed Systems", RADC Distributed System Technology Exchange Meeting, Griffiss Air Force Base, Rome, NY, June 1983

TECHNOLOGY CONSIDERATIONS FOR REUSABLE SOFTWARE

TOM BOWEN
BOEING AEROSPACE CO.
9 APRIL 1985

CONCEPTUAL STARTING POINTS



TECHNIQUES	PRODUCT REPRESENTATION	AUTOMATED TECHNIQUES	METHODOLOGY & TECHNIQUES	COMMONALITY
ADA REUSABILITY ATTRIBUTES	DIDS FROM DOD-STD-2167	ADA SUPPORT ENVIRONMENTS	DOD-STD-2167	FUNCTIONAL ANALYSIS

SOME QUESTIONS

- MEASUREABLE CHARACTERISTICS PROMOTING REUSE
- KEY PRODUCT DESCRIPTORS
- SUPPORT ENVIRONMENT INTERFACES
- METHODOLOGY IMPACT ON DOD STANDARDS AND POLICIES
- CRITERIA FOR SELECTION (APPLICATION AREAS, FUNCTIONS, COMPONENTS)
- CRITERIA FOR RETAINING
- V&V RESPONSIBILITIES, PRODUCT LIABILITY
- ACCESS AND SECURITY
- TECHNOLOGY TRANSFER
- INCENTIVES
- KNOWLEDGE-BASED ENGINEERING
- RAPID PROTOTYPING AND GENERATION

A PHASED APPROACH TO ADA PACKAGE REUSE

Dr. Bruce A. Burton and Mr. Michael D. Broido

Intermetrics

29 January, 1985

Abstract

This paper discusses some of the significant issues associated with the reuse of Ada(EM) packages. It presents the authors' strategy for developing a library to support Ada package reuse. The paper presents the functional description of a phased implementation for a software library. It also delineates the different capabilities required to expand the package library from one serving a single project to one serving multiple companies at many different sites.

Introduction:

A key objective of the Department of Defense, in the program which led to the creation of Ada, is the reduction of life-cycle costs for the development and maintenance of software. An important component of that cost reduction will necessarily involve the elimination of the repeated practice of reinventing a software function for each new target system or application. If an existing piece of software can be reused to suit a new situation, considerable savings can accrue in the specification, development and verification of the software which includes that component.

Software reuse is not without its disadvantages, however. Any system designed to promote reuse, such as the one presented in this paper, must recognize and address these drawbacks.

Problem:

Intermetrics is actively involved in a program to identify a method of promoting the reuse of Ada packages. The system we are creating is designed to recognize and reduce the impact of past problems which have inhibited software reuse, while allowing the gradual build-up of functionality. This way, partial products have utility while new features are added.

Approach:

Ada packages form a natural unit around which to build a software reuse capability. To fully exploit the work that has gone into previously created packages requires an extensive list of functions. We present here a practical approach for gradually building up these functions in a way that makes partial results useful, with later releases increasing the utility of the total package. Since the production and support of Ada packages is intimately related to a software development environment's configuration management system (CMS), this support function forms a natural basis upon which to build an expandable Ada package reuse capability. To the CMS and the software library controlled by it, we intend to add an Ada Software Catalog (ASCAT) for on-line user inquiry and report generation. In the final phases, additional features will be added to support multiple sites and sharing of information among the users of each package.We are pursuing a strategy for providing an initial reuse capability within a narrow group of users. By gradually adding new features, a complete system capable of supporting reuse across diverse projects, target machines, and companies is built up. Our ultimate system will include the combined features of a comprehensive configuration management system, a catalog of available software, a library for the access and distribution of software, and a bulletin board system for communication among all users interested in a particular package or topic.

Reuse System Motivation and Description:

Background:

A software designer or implementor, in seeking to minimize his development time, costs and risks, needs the answers to several questions in preparing his plan to create a software solution:

(1) What software is available for consideration?

(2) Is it suitable to my needs (functionability, interfaces, size, speed, costs, availability on my target system, test and debugging aids, limitations, etc.)? What adaption features are available?

(3) In what forms can I get it (specification, design, source, object, executable, documentation, tutorials)?

(4) What changes are pending or under consideration?

(5) How do I get it? From whom? From where? At what cost? What restrictions apply to my use? What support is available if I have problems or need changes? Can I make changes to it myself, and if so, what help do I get?

(6) How do I get notified of problems and changes?

A large scale software library/CMS system, if used as part of a development environment or network, can provide at least partial answers to all these questions. A limited software catalog could certainly provide answers to questions 1, 2, 3, and 5. Additional information may be needed to answer the others.

Significance of Ada Package Reuse:

The solution to any problem, in order to gain acceptance of the decision makers, must satisfy three sets of criteria: it must address the technical, economic and political requirements. The bulk of literature on requirements and systems for the reuse of software has focused on the technical issues.

The principal advantages of software reuse are economic: reduced life-cycle costs through increased productivity, shorter schedules and reduced risk. These in turn allow developers to produce more general packages and put greater flexibility into the hands of the end user. Knowing that software is being produced for reuse over the long term can encourage better designs with more emphasis on modern development techniques and coding practices. Creation of software for general applicability leads to the development and use of standards, which further reduce life cycle costs. More thorough design, documentation and testing are encouraged. Much of the power of UNIX(TM) comes from its standardized, simple interfaces and the composition of software through the assembly of well-encapsulated functions.

Yet there is a dark side to software reuse. Companies which have spend considerable resources creating a software capability are reluctant to provide their tools, through a common library, to their competitors. Programming by the composition of existing pieces is not as much "fun" as writing programs from scratch. In-house "experts" lose their special status if open, well-documented code is available to anyone who needs it.

The very economic incentives which reduce life cycle costs can work to the disadvantage of the developers as well. Potential drawbacks include: higher initial package development costs, performance degradation due to excessive generality, the lack of component composition paradigms (Standish 83), closeness of fit issues, module update issues, the wider impact of errors, the first package available (which by its very existence tends to set the standard) may not be a good one, the lack of standardized specifications for the sharing of packages across projects, and the lack of standards for the documentation of packages suitable for inclusion in the common library.

Phased Implementation Supporting Software Reuse:

We are constructing an Ada package library as a multi-phased project that initially offers a restricted set of software catalog functions. As Ada packages become more prevalent, the Ada package library will be expanded to include higher levels of formality and automation, a higher degree of interaction with the Configuration Management System, and extensions to support library interaction across companies and machines. Our strategy involves the development of seven distinct phases. Some of these may be built in parallel as external needs and funding evolve. A functional description of each of the phases follows.

Phase 1: Analysis and Requirements Definition

During this phase the requirements definition of the package library is being formulated. Baseline capabilities as defined by the Ada Language Reference Manual (DoD83) are being examined for reuse implications. Previous software libraries, such as COSMIC (NASA 84), IMSL (IMSL 76), and the Statistical Package for the Social Sciences (SPSS), will be analyzed so

that the requisite data items that facilitate the description of generic software components can be identified. The sponsoring organizations will be contacted to illuminate features which help and hinder reuse. Specific information unique to Ada packages, which should be included in the software catalog, will be identified through an analysis of our present collection of Ada packages. Alternative sources for related information (such as design documentation, package specification data, user's guide, etc.) need to be identified, and mechanisms for readily extracting relevant data need to be designed.

Phase 2: Initial Software Catalog

Based upon the information collected in Phase 1, we will design a kernal Ada Software Catalog (ASCAT) for the storage and retrieval of Ada package reuse information. This initial system will simply support the storage of data base records that specify the purpose, specification, algorithm, keywords, author, and other relevant package identification data. Data collection and entry will be automated; a processing tool will be used to extract necessary data and to input and format the data into the software catalog. Since the catalog will be developed around an extensible data base management system, support for interactive query and report generation will be automated. This system represents a limited beginning; software authorization control and distribution will remain unsupported in this phase. The kernel system will also lack support for automated error notification, change notification, and standards enforcement. The initial software catalog will be a passive entity, accepting package description data when provided by the processing tools or manual updates. It will respond to user queries and report requests. Interaction between the software catalog and the Configuration Management System (CMS) will be entirely under manual control.

Phase 3: Automated ASCAT/CMS Interface

During this phase, an interface will be developed to facilitate communication between the ASCAT and the existing Configuration Management System. This interface will allow automatic update by the CMS of several important ASCAT data items. Through the use of the CMS, information on package version data, authorization control, error identification and status, and related documentation (such as design specifications, user manuals, test data and sample output, object size, etc.) can be automatically sent to the ASCAT.

Phase 4: Integration of Standardization Support Tools

The early versions of the ASCAT will employ a standards policy for submissions; the policy will be manually enforced. Initially, submitted Ada packages will be screened for adherence to coding, reuse and minimal documentation standards by visual examination (code walkthroughs). During Phase 4, standardization will be ensured by the integration of standardization support tools, such as Intermetrics' Byron (TM), into the package library system. These tools will aid package standardization by supporting the scanning of newly submitted packages for adherence to the reuse conventions (e.g., checking that all required reuse data items are present). Also during this phase, the packages which had been previously entered into the library will be evaluated to determine discrepancies between the library contents and the enforced standards.

Phase 5: Expansion of the User Community

The initial versions of the ASCAT will represent fairly passive entities. That is, information flow (except for explicit inquiries and report generation requests) is primarily directed from the outside world in. During this phase the package library system that contains the ASCAT will become a more dynamic entity. A user community will be added to the system. The inclusion of electronic mail will allow an increase in the communications traffic between the Ada package library and the user community. Now, based upon library and CMS events (such as new entries, error notifications, standards changes, etc.), the Ada package library can automatically notify the affected user community through the electronic mail system.

The expansion of the user community will provide additional problems for the Ada package library system. With the development of the user community a mechanism for both system/user and user/user interactions needs to be established. The electronic mail system might be used for one or both of these functions. The precise approach taken will address the problems of separate users in different groups. Preliminary steps to address security concerns will be incorporated.

Phase 6: Automated Catalog and Library Interaction

This phase will be characterized by a fully automated system of interaction between the program library and the CMS. Results of the mechanized standards enforcement, as developed in Phase 4, are automatically recorded in the library and catalog. Manual overrides (via documented deviations and waivers) are permitted, but a flag is set in the library and catalog entries.

This phase will also include an automated system for distribution and authorization of software and related products. This system will include a software order entry capability, with the ability of the owners (controlling organization) of library entries to restrict the release of specifications, source code, etc., by individual item or by item class (source, object, specifications, test information, etc.). The ordering and distribution system will include the ability to defer the release/distribution to a new user until the owner has given manual approval.

The order system will maintain lists of users who might not otherwise receive update information. For example, this can be off-site users who are not immediately accessible to the CMS. It can include users who have supplemented the set of test cases and would need to know about changes to the base set of tests. It can include "casual" users who are trying out a package, but have not yet fully or formally incorporated the library package into their new programs (e.g., they are developing a prototype).

The ordering system will allow the user to specify the desired action in case changes are made or serious bugs are found. The user could specify such actions as performing automatic updating, sending the updated files but not integrating them, simple notification that a change has occurred with a brief description of what it does, or do nothing. The system will aggressively notify any users who are not on-line (or have a permanent mailbox) by creating hard-copy notification suitable for sending through the regular mail.

Phase 7: Multi-Site and Multi-Company Extensions

During this phase, additional restrictions can be placed on distribution of software items, such as licensing, purchase agreements (implicit agreement to be billed for products delivered), and restrictions on the number of machines on which something may be used without additional actions occurring. Enhanced security features will be added. Methods for sharing libraries in distinct systems will be added, and a method for charging remote users will be included.

The library and distribution system will include positive feedback of the incorporation of changes released by the package owner, even in a distributed network or multiple system configuration. This will allow the owner of the package to archive obsolete versions which are no longer in use anywhere in the user community.

In addition to the (Phase 6) ability of the users to communicate with the library and CMS, the library/CMS to send unsolicited data to the users, a "bulletin board" will be added. This bulletin board is a multiple-message service to which anyone with access to the system may send messages. In particular, this gives the equivalent of an electronic users' group so that the diverse users of a package may share information among themselves, independently of the package owner. The bulletin board differs from the electronic mail system added in Phase 5 in that the mail system requires an explicit list of addressees, whereas the bulletin board facilitates communication among parties who are initially unaware of each other (and their mailbox ID's). The bulletin board facility will have the ability to be partitioned, so that a separate "board" can be made available for each topic or package. The bulletin boards can serve as a forum for informal position papers, searches for new uses and adaptations of the covered packages, discussions about the desired priority of pending changes, suspected bugs which have not been formally verified and isolated, workarounds for known problems, informal discussions about potential extensions, etc.

With the bulletin board concept, separate bulletin boards may be added on topics for which no packages currently exist. This allows users to search for uncertified capabilities, developed or under development by others, which might be available by the time the new user needs it. This forecasting can serve to lower costs by reducing the amount of parallel development that would be done. It will also allow the new user to suggest changes early in the design and development process. These changes help to generalize the package so that it can be reused in more programs than originally intended.

Summary/Conclusions:

Software reuse is an issue currently receiving a lot of attention. The reuse of Ada packages offers the software developer significant opportunities to achieve high productivity and to lower

life-cycle development costs. Although Ada provides a natural vehicle for encouraging software engineering reuse, the same technical and political obstructions that have limited reuse in the past will once again impede the sharing of software engineering products across the 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.

While a large-scale Ada package library system that is tightly integrated to a configuration management system may provide a reuse mechanism that offers maximum long-term benefit, the construction of such a system would necessitate the expenditure of large amounts of up-front money. A phased implementation that starts with a kernel Ada package library that is steadily extended to a large-scale package library system can be used to reduce this initial outlay. In addition, the phased implementation approach offers the opportunity of incremental success which might provide the necessary economic catalysis for continued development.

This is precisely the approach that we are employing within the California Division of Intermetrics. The California Division is heavily involved in the development of applications software for the aerospace industry. In order to improve our software development productivity, we are currently building a facility that promotes the automatic extraction of package reuse information from on-line design documentation and source code. The information will be stored in a data base system called ASCAT (Ada Software Catalog) that supports the storage and retrieval of package reuse information. Distribution of Ada packages will initially be informal and no direct interaction will occur between the ASCAT and our configuration management system. As we iron out the difficulties with the ASCAT and refine its capabilities, we intend to continue our phased implementation plan and to increase its use within our company.

References:

(1) DoD 83- Reference Manual for the Ada Programming Language, MIL-STD 1815 and an ANSI Standard Document. ADA Joint Program Office, Washington, D.C. Reprinted by Intermetrics, Inc., March, 1983.

(2) IMSL 76- Reference Manual, International Mathematical and Statistical Libraries, Fall, 1976.

(3) NASA 84- NASA's Computer Software Management and Information Center, "COSMIC Software Catalog, 1984 Edition." University of Georgia, 1984.

(4) STANDISH 83- Standish, Thomas A., "Software Reuse", presented at the Workshop on Reusability in Programming, Newport, Rhode Island, September 7-9, 1983.

Trademarks:

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

Byron is a trademark of Intermetrics, Inc.

UNIX is a trademark of AT&T Bell Laboratories.

Biographies:

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. His research interests include configuration management, software quality assurance, and performance improvement.

RESUME

MICHAEL D. BROIDO

EDUCATION

MS, Computer Science
USC 1973

BS, Mathematics
CalTech, 1970

2.1 Continuing Education Units

Holder of California Jr. College Teaching Credential in Mathematics and Computer Science; Completed accredited training course in Ada

EMPLOYMENT HISTORY SUMMARY

Name of Employer	Period of Employment	Title
Intermetrics, Inc.	May 1983 - Present	Sr. Analyst
Data Card Corporation	April 1982 - April 1983	Manager
Computer Automation, Inc.	May 1979 - March 1982	Manager
TRW Comm. System & Services	March 1978 - May 1979	Project Leader
Burroughs Corporation	April 1970 - March 1978	Systems Analyst

HISTORY OF PROFESSIONAL EXPERIENCE

Relevant Area
Company Description
Duration

Systems Analysis, November 1984 to Present

Under IR&D funding, conducted a study entitled "Software Configuration Management for a Large Distributed Environment". Described mandatory and optional requirements to implement such as system. Included substudies on "Software Rollbacks as a Fault Recovery Technique" and "Configuration Management and Software Reusability".

Systems Engineering, Intermetrics, Inc., - 6 Months

Defined the tasks and system requirements for the Integrated Support Facility for the User Equipment Segment of the Global Positioning System. Included hardware/software maintenance tools, configuration management, test requirements, reliability/availability statistics, and various support functions.Ada Training Course - 1-84 - 3-84

Completed a 10 week course on the Ada programming language. The course surveyed typical programming language constructs e.g., looping and constructs, conditional execution constructs, data structure constructs, subprogram constructs, etc. The course also covered the use of an Ada-oriented program design methodology (object-oriented design). In addition, the course examined the role of Ada concurrent programming features in the development of embedded computer systems. A class project was included; my team developed a project management system.

Software Analysis and Development, Intermetrics, Inc., - 5 months

Maintained and upgraded database software for the collection, analysis, and display of contractor

performance for the Global Positioning System. Used a VAX/750 with Datatrieve, FMS, RGL and FORTRAN '77. Specific duties include developing database access routines, extending data collect software, extending and optimizing the database display program using both text and graphics, writing utility programs, and upgrading user documentation.

Software Analysis and Planning, Intermetrics, Inc. - 3 months

Prepared the Program Management, Computer Program Development, Software Quality Assurance and Software Configuration Management Plans for Intermetrics roles on the Post Mission Data Analysis for an experimental infrared satellite. Outlined and developed evaluation criteria for a survey of available software tools and conducted part of the survey. Served as deputy program manager.

Software Quality Assurance, Intermetrics, Inc. - 7 months

Developed division-wide tools, plans, policies and standards for software quality assurance and configuration management according to MIL-STD-483, MIL-STD-490, MIL-S-52779A, etc. Tailored to the needs and audited implementation of those items in various programs.

Software Management, Data Card Corporation - 12 months

Manager of Software - Planned, directed and coordinated six programmers, an aide, and a secretary/librarian in producing standard and customized software for financial printing systems using MICR encoding. Applications included disbursements, checkbooks, loan coupons and other cash management functions. Worked closely with Hardware and Marketing departments on product planning, including hardware/software evaluation of several microprocessor-based systems for use in our next generation products. Principally used Data General NOVA and Eclipse computers with AOS, RDOS, RTOS and stand-alone operating systems. Responsibilities included staffing, budgets, contractor selection, and design/implementation reviews.

Software System, Software Development Computer Auto. - 10 months

Manager of Software Development - Planned and directed the activities of eleven programmers, plus a secretary, in the design, development and release of operating systems, language compilers, and data communications subsystems, including a real-time multi-user mapped OS, a hard-core generating PASCAL compiler, and a synchronous communications protocol handler. Participated in long-term strategic planning and negotiations with outside vendors. Prepared budgets, salary plans, and performance appraisals. Selected and monitored outside consultants and contractors. Outlined and directed a competitive analysis (features and performance) among our new operating system (CARTOS), DEC's RSX-11M, Data General's RDOS, and Intel's RMX-86.

Software Quality Assurance, Computer Auto. - 24 months

Manager of Software Quality Assurance - Led a team of programmers in projects to ensure and enhance quality of software products for CA, including acceptance testing, documentation review, development and use of regression tests, and performance testing. Created and implemented standards for the definition, development and release of software. Personally designed and implemented a test bed for qualification of a real-time multi-user operating system and acceptance/regression tests for a single user COBOL compiler and runtime interpreter. Active member of the Software Change Control Board, validating both form and content of new releases.

Performance Measuring and Analysis, TRW Communications Systems and Services - 14 months -

Scheduled, designed, programmed, executed and documented performance analyses on and for real-time data communications systems on Data General NOVA and Eclipse minicomputers. Created and used external/internal hardware/software measurement tools, analytic and event simulation models,

and remote terminal emulators to define, validate and predict such performance criteria as response time vs. load, capacity planning, location and removal of bottlenecks, etc. As a senior staff member, I also reviewed functional specifications, designs, schedules, documentation methods, quality assurance activities, programming methodologies, etc.

Performance Measuring and Analysis, Burroughs Corporation - 10 months -

Led a group chartered with the development and world-wide support of tools and techniques for measuring and improving performance on Burroughs large systems (B6000/B7000 series). The tools comprised a series of programs which allowed trained personnel, including customers, to do their own performance analyses. Specific duties included: (a) planning and scheduling all phases of program development; (b) scheduling and teaching classes in use of the tools and on advanced performance issues; (c) consulting at various customer sites, including corporate headquarters to audit such areas as performance, operations, programming standards, data base and data communications designs, program development methods, etc.

Performance Measuring and Analysis, Burroughs Corporation - 12 months -

Member of Systems Performance Improvement group described above. Did the design, programming, documentation and maintenance of a program which uses the system log to produce almost 200 tabular and graphic reports on various system resources, elapsed times, turnaround, mix depth, etc. The program has the ability to analyze subsets of the log based on time intervals, selection criteria, etc. Also consulted at customer sites and taught classes.

This work required detailed knowledge of the hardware, operating system, use and output of the compilers, tools, data communications systems and data base management system.

Sales Technical Support, Burroughs Corporation - 24 months -

Member of the Marketing Support Activity, responsible for customer consulting and pre-sales activities. The latter included presentations, demonstrations, and developing benchmarks that involved heavy data communication and/or data base applications, plus simulations, on Burroughs large systems.

Data Processing, Burroughs Corporation - 48 months -

Member of internal data processing organization, using a B5500. Specifically responsible for design and programming of data base and applications to support engineering, industrial engineering and accounting activities in such areas as product structure (BOM) with revision control, product costing, product cost forecasting, eliminating obsolete information, labor distribution and inventory pricing. Interfaced with payroll, stockroom control, factory schedule and general ledger systems.

Publications:

(1) Broido, Michael D., "Exception Handling Improves Real-Time System Performance," Computer Design, November, 1982.

MANAGEMENT ISSUES IN SOFTWARE REUSE: AVOIDING MISUSE OF EXISTING SOFTWARE

A PRESENTATION TO THE
STARS WORKSHOP
APRIL 9-12, 1985

MICHAEL D. BROIDO
INTERMETRICS, INC.
HUNTINGTON BEACH, CA.

INTERMETRICS

REUSE IS NOT FREE OR AUTOMATIC

- REUSE OF SOFTWARE HAS SIGNIFICANT METHODOLOGY IMPACTS
  - NEED METHOD TO PARTITION PROGRAMS FOR REUSE
  - NEED UNIFIED COMPONENT COMPOSITION PARADIGMS
  - NEED TO SEARCH EXISTING SOFTWARE AS PART OF DESIGN PROCESS
  - REUSE IS LARGELY A BOTTOM-UP BUILDING METHOD
- GENERALIZED REUSE OF SOFTWARE RAISES SEVERAL ISSUES:

  TECHNICAL, ECONOMIC/LEGAL, AND SOCIAL/POLITICAL
- SIMPLY WRITING PROGRAMS IN THE ADA™ LANGUAGE DOES NOT GUARANTEE REUSABILITY. (WE HAVE PROOF.)

ADA IS A TRADEMARK OF THE U.S. DEPT. OF DEFENSE (AJPO).

INTERMETRICS

KEY TECHNICAL ISSUES

- LACK OF UNIVERSAL STANDARDS FOR COMPONENT COMPOSITION, LEVEL OF DOCUMENTATION, CODING TECHNIQUES, TESTING,...
- FIRST ONE BUILT BECOMES DE FACTO STANDARD, OFTEN WITHOUT ADEQUATE PRIOR REVIEW.
- IT IS DIFFICULT TO DEBUG, TUNE AND SHRINK PROGRAMS WITH MANY EXISTING "BLACK BOXES".
- LACK OF CATALOGING STANDARDS.
- CLOSENESS OF FIT AMONG COMPONENTS, BETWEEN EXISTING COMPONENTS AND NEW APPLICATIONS.
- GREATER IMPACT OF ERRORS.
- MECHANICS OF PROVIDING UPDATES TO OFF-LINE USERS.

INTERMETRICS

KEY ECONOMIC/LEGAL ISSUES

- HIGHER INITIAL DEVELOPMENT COSTS AND LONGER SCHEDULES.
- WARRANTY, LIABILITY AND ACCOUNTABILITY ISSUES.
- ABILITY TO PROPERLY CHARGE FOR PROPRIETARY SOFTWARE.
- AVAILABILITY OF TRAINING AND SUPPORT.
- "COST-PLUS" CONTRACTS DISCOURAGE REUSE, BUT FIXED PRICE CONTRACTS CAN MAKE CATALOG SEARCH TIME RISKY.
- CONTRIBUTED SOFTWARE MAKES COMPANY INVESTMENTS AVAILABLE TO COMPETITORS.

INTERMETRICS

KEY SOCIAL/POLITICAL ISSUES

- "NIH" SYNDROME; REUSE IS LESS ENJOYABLE; LOSS OF PRESTIGE BY INTERNAL "EXPERTS".
- "EXCESSIVE" DOCUMENTATION NEEDED.
- CONTROL OF CHANGES - CHOICE, PRIORITY, TIMING, ...
- ADMINISTRATIVE IMPEDIMENTS TO OBTAINING OR MODIFYING EXISTING SOFTWARE.
- REDUCED MANAGEMENT CONTROL OF RISKS.

INTERMETRICS IS TAKING A CAUTIONS APPROACH

- SEVEN PHASE PLAN FOR Ada PACKAGE REUSE

    1. INITIAL REQUIREMENTS ANALYSIS
    2. INITIAL CATALOG - ENTRIES, FORMAT, SEARCH
    3. AUTOMATED CONFIGURATION MANAGEMENT INTERFACE
    4. STANDARDS ENFORCEMENT
    5. EXPANSION OF USER COMMUNITY
    6. AUTOMATED CATALOG AND LIBRARY INTERACTION
    7. MULTI-SITE AND MULTI-COMPANY EXTENSIONS

- USABLE PARTIAL RESULTS AT THE END OF EACH PHASE

- REASSESS AFTER EACH PHASE

- INCLUDE "BULLETIN BOARD" FOR ELECTRONIC USERS GROUP

INTERMETRICS

RECOMMENDATIONS TO THE STARS OFFICE

CONDUCT OR FUND STUDIES TO INVESTIGATE:

1. CATALOG ENTRY FIELDS NEEDED TO PERMIT COMPANY- AND APPLICATION-INDEPENDENT REUSE.

2. SOFTWARE PACKAGE CLASSIFICATION SCHEMES AND SEARCH TECHNIQUES.

3. WARRANTEE AND LIABILITY FOR GFE REUSE LIBRARIES.

4. TECHNIQUES TO FACILITATE COMPLETE, CURRENT AND ACCURATE DOCUMENTATION FOR CATALOG ENTRY DATA.

5. IMPACT OF ALTERNATE CONTRACT STRUCTURES ON SOFTWARE REUSE ISSUES.

INTERMETRICS

REUSABLE COMPONENT DEFINITION (A TUTORIAL)

Rodney M. Bond
General Dynamics/Data Systems Division

"Man is a tool-using animal.... Without tools he is nothing, with tools he is all" (Carlyle 1834).

In this paper several paradigms are discussed in the context of the software product lifecycle. The intent is to survey various methodologies of software development and their associated tools, and to pose the question of what components support reusability. To identify reusable components, the discussion was written, and then scanned for potential reusable products, which were highlighted by underlining. Finally, a summary section was written. The methodologies to be discussed are the classic software engineering lifecycle (alternately - the waterfall method), symbolic programming as practiced by the artificial intelligence community, executable requirements specifications which are formalized abstractions of current language systems, and a knowledge based approach.

Figure 1 is a generic instantiation of the classic software engineering lifecycle. The output of each phase acts as the input for the following phase. The first activity is to identify a system and its needs that perform some functional requirement. There are no tools currently available to help specifically with this task; however, some tools under development do address support of this effort. These tools typically are called rapid prototyping tools and are used to perform feasibility analysis of required system concepts and designs. This might be something as complex as asynchronous communication between programmed tasks or as simple as the layout of data on a CRT screen. Other tools supporting this activity help with the analysis of the effort required for implementation. Cost estimating systems provide support through parametric modelling, or through analogy with similar efforts in size and complexity. Project management tools help identify and allocate required resources and range in complexity from simple chart generators to integrated spreadsheet/database systems such as are currently being marketed on personal computers.

The first software development phase is used to formalize the requirements of the proposed computer program. Several tools have been implemented to support this phase. These requirements tools provide a formal system of specification which can be analyzed for consistency and completeness. A specification for a large system can be hundreds of pages long and be assembled from multiple diverse organizations. Many times this will lead to contradictions or omissions in the requirements specification for the system. If inputs are expressed in the formal language of a requirements specification tool, less ambiguity is likely in the interpretation of the descriptions, and the system can be automatically analyzed as well as automatically documented.

The design phase specifies "how" the requirements are to be accomplished. Through the process of "abstraction" design decisions are identified without detailing the specific implementation choices. The primary tools to support this phase are structurizers and pseudocode. Structurizers are tools which help the user specify the control strategy of the program. Through one of several methods the modules of the system are described as processes, data objects, or interfaces; and a structurizer will document and analyze the design specification. Program description languages (PDL's) are used to implement pseudocode. Pseudocode provides a language which is between the free form, ambiguous English language and the structured, formalized programming language. A PDL tool also will implement, for a specific project, a formal pseudocode syntax which will have the same benefits as the formal requirements language: enchanced understanding among users, automated analysis of decisions (usually of a limited nature), and automated documentation.

The implementation phase specifies the "with" decisions of the design phase. Each design decision is implemented "with" the best available constructs in the programming language. A multi-way branch may be implemented as a CASE construct or as a series of IF-THEN-ELSE constructs or any of a number of other ways, primarily dependent upon the definition of the language being used, and the available extensions to the language which might be implemented on a particular host computer. Tools to support these types of choices are not available except as systems which check for compliance with standardization. Even though most of the tools developed in support of the development of software have been in the implementation phase, they have primarily been built to increase the quantitive productivity of the programmer, not to support his decision processes. These tools are various compilers, language constructs, smart editors, code libraries, optimizers, and many others. The Ada?TMO programming support environment is the first real attempt to provide the programmer with a set of implementation tools which will support qualitative as well as quantitative productivity improvements.

The test phase as described here represents the unit test of each code module, the test of module integration, and the system test. There are multiple tools to help with the test process, but by nature are tied closely to the implementation language. Examples of these tools are the Fortran Automated Verification System, the Cobol Automated Verification System, and the Fortran '77 Analyzer. These tools perform testing coverage to determine if all of the code has been exercised by the test set developed, static analysis to generate statistics about the implementation details such as number of branches and number of variables, and dynamic analysis to evaluate performance characteristics.

The final phase, maintenance, is a recursion through the previous phases to correct deficiencies. The phase to which the recursion returns is dependent on the nature of the deficiency. An error in the specification of the system requirements will require a total recursion of the lifecycle for some segment of the system. The most important tools for this phase are documentation control systems. These tools include a database in which all development documentation, source files, and object files reside; and a control mechanism which can regenerate any historical version of the database. Without this capability, modifications to the generated system are complex and sometimes impossible.

"Give us the tools, and we will finish the job" (Churchill 1941).

The software product life cycle presented in Figure 2 has been in existence almost as long as the classic life cycle.

LISP was invented to solve the symbolic manipulation problems that FORTRAN did not seem to be able to handle. The attempt to instill "intelligence" into computers was approached by attempting to model the human thinking processes, which did not appear to work in numerics but instead symbolics. Because LISP was interpreted rather than compiled, there was also a tendency towards interactive programming versus batch programming. Do develop a system in LISP, the total system needs to not need to be identified prior to addressing design, code and test. If a specific requirement can be identified, then a function is developed to satisfy the "functional validity" of the requirement. This is then followed by a similar effort for the next identifiable requirement until the total system has been implemented as a set of functions. System integration and testing is performed in increments and a final validity check accomplished. Maintenance then becomes a task of modifying an existing function, or in the case of an error of omission in the requirements, simply the generation of another function. Since LISP has been in use primarily in the university research environment, tools to support the symbolic programming lifecycle are not extensive. The decomposition of the requirements does not lend itself to analysis, nor does the incremental design of functions provide for design analysis. Coding is supported through syntax editors which generally are not considered very powerful, but necessary; and testing is supported through run-time symbolic debuggers. Current application oriented tools are emerging with greatly enhanced features to support design, coding and test. However, these are tools which define a new programming lifecycle approach embodying concepts such as object-oriented programming, top-down decomposition, code libraries, and others; with LISP as the underlying language of implementation.

"The tools to him that can handle them" (Napoleon 1817).

Not everyone wants the flexibility of the previously described programming lifecycles.

Ideally once the requirements for the system have been established, the rest of the process could be automated. This would mean that, given that the automation methodology was correct, there would be no errors except for those described in the requirements or caused by operator error. The system would only be modified when there was an identified requirement error or a need for an update in requirements. No documentation, design decisions, programming, or incremental testing need ever occur beyond the requirements specification phase. Figure 3 represents the tools which attempt to implement this executable requirements specification lifecycle. These tools require the use of a formal requirements specification language which can be analyzed for consistency and completeness. Once accepted the specification is then used to automatically generate source code in one of multiple languages that can be compiled and executed. System performance tuning is limited with this approach, but even when this necessary it might be accomplished through manual modification of the source code. Even when this is not possible, the execution of the requirements prior to being used in the classic lifecycle would provide insight into requirements errors which would be extremely expensive to correct in later phases.

"Intelligence...is the faculty of making artificial objects, especially tools to make tools" (Bergson 1907).

The Development Arts for Real Time Systems (DARTS?TMO) lifecycle presented in Figure 4 is one of several called "knowledge-based" approaches, a subfield of artificial intelligence. The intent is to use knowledge of previously developed systems and knowledge of the targeted application to generate a new system. In this and other similar systems, which were alluded to in the symbolic programming discussion, tools exist as primitive functions to be used to build other tools, which eventually bootstrap the desired system. A unique feature of the DARTS lifecycle is the use of previously developed systems, decomposed into semantic units, to baseline the new system. As much code as can be identified as useful by its semantic description will be reused in producing the new system. The AXE feature of the DARTS technology is a powerful programming language which is simply a set of tools to help build other tools (possibly as functions) such as graphics interfaces, domain-specific natural language interfaces, or specialized pattern recognition systems. When a system is built, its unique features are incorporated (archetyped) into the DARTS database as new semantic units for future use. Many versions of the same system with small variations can easily be produced with this method. The primary thrust of the knowledge based approach is to give the user a good set of primitive yet flexible tools with which to implement a new system.

"Every tool carries with it the spirit by which it was created" (Heisenberg 1958).

SUMMARY: **the following can be identified as**

REUSABLE COMPONENTS

LIFECYCLES

DEVELOPED SYSTEMS
- SPECIFICATIONS
  - REQUIREMENTS
  - DESIGNS
- CONCEPTS
- CONTROL STRATEGIES
- MODULES
- PSEUDOCODE
- DOCUMENTATION
- SOURCE CODE
- EXECUTABLE REQUIREMENTS

TOOLS
- STRUCTURIZERS
- PROJECT MANAGERS
- PDL'S
- TESTERS
- DATA BASES
- MODELS

HISTORICAL DATA
- DESIGN DECISIONS
- RESOURCE ALLOCATIONS
- COST ESTIMATES
- PERFORMANCE CHARACTERISTICS
- DEVELOPMENTAL VERSIONS

PROJECT RESOURCES
- COMPUTERS
- COMPILERS
- PROGRAMMERS
- PROGRAMMING ENVIRONMENT

TECHNIQUES
- ABSTRACTION

TOP DOWN DECOMPOSITION LIBRARIES

This exercise demonstrates the potential for reuse of almost all products, processes, concepts and other resources. The keys to reusing these resources would be knowledge of their existance, the determination of their applicability to the problem domain, and an ability and willingness to use the resource. The tools are obviously the most reusable by design, however may not be reused due to one or more of the key factors cited. The least reusable resource is probably a total system implementation, unless redundancy is applicable or the system is easily modifiable. In either case, the current state of computing requires one other resource to be reused, and that is the person with the knowledge which gets the resource reused. The process of looking for reusable resources versus starting from scratch is not a common practice, but could become one with the proper incentives applied. The earlier a system component is developed in the lifecycle, the more readily it can be reused. The later, the higher the incentive for reuse. Those products "designed" for reuse generally are reused.

'TMOAda is a registered trademark of the U.S. Government (Ada Joint Program Office) 'TMODARTS is a trademark of General Dynamics Corporation

AUTHOR Rodney M. Bond, General Dynamics, Data Systems Division, 1745 Jefferson Davis Highway, Suite 1000, Arlington, VA 22202 H - 301-731-0213 B - 703-553-1320

RESUME

RODNEY BOND

Chief, Software Technologies

KEY EXPERIENCE

Program Management, 2 years
Project Management, 2 years
Project Support, 1.5 years

Mr. Bond is responsible for DSD research activities in Washington, DC. The research is primarily oriented around artificial intelligence (AI) and the Ada programming language. Mr. Bond has been responsible for direction and administration of software technology programs at two General Dynamics engineering divisions. Primary areas of concern have been modern programming environments, AI, military standards, advanced computer languages, computer architectures and software cost estimating.

Prior to assuming a management role, Mr. Bond was the lead project engineer for a research effort to define a modern progreamming environment to meet the needs of the Defence Mapping Agency (DMA) through 1990. This effort included comprehensive research into the policies, procedures, plans, and daily operations of the DMA as well as into the trechnologies requiring support. Mr. Bond also participated in research efforts to define new tools to support distributed processing systems, and to provide a modern programming environment for the Air Force avionics laboratories. Other research included the evaluation of productivity tools to support software project development in general with specific interest in compiler development.

In addition to his management role, Mr. Bond also serves as an internal consultant to General Dynamics engineering divisions in support of their software development activities.

Mr. Bond is a member of ACM, IEEE Computer Society, and AAAI.

EDUCATION

500+ hours of technical courses/seminars 1980-1984.

M.S. - Computer Science, University of Texas at Arlington 1980

B.S. - Aerospace Engineering, Mississippi State University 1973

SECURITY CLEARANCE

Secret

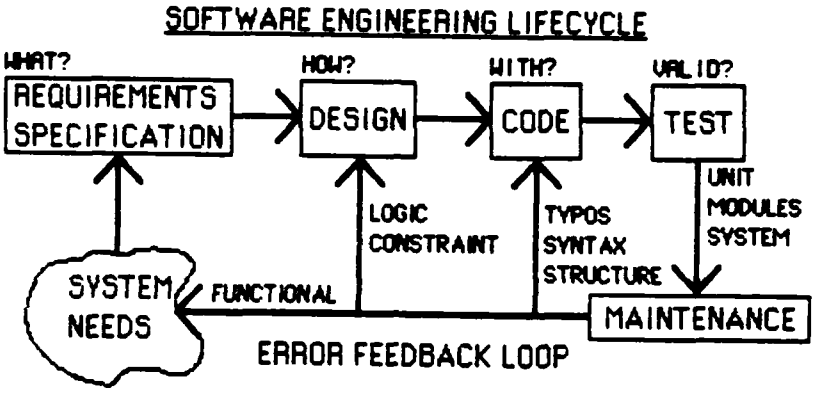


FIGURE 1

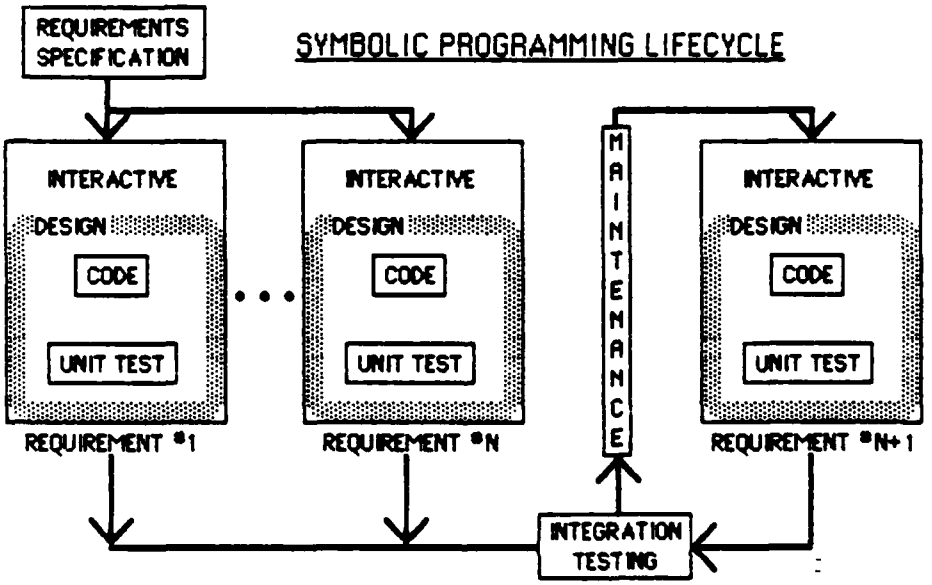


FIGURE 2

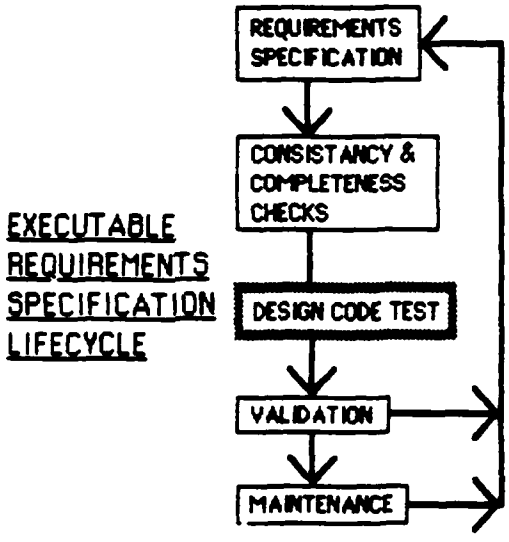


FIGURE 3

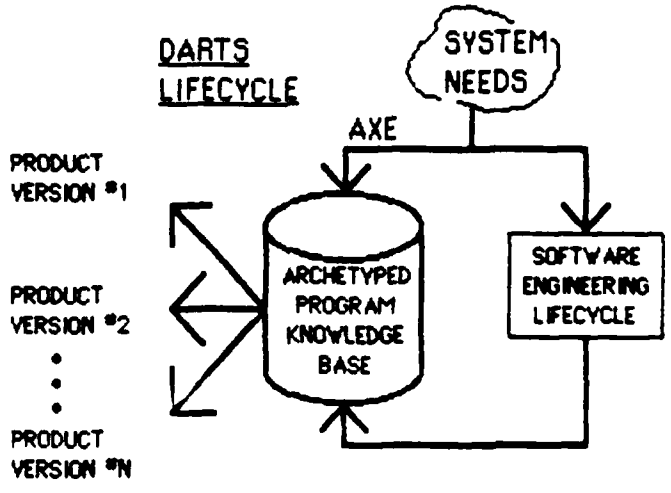


FIGURE 4

POSITION PAPER
AUTOMATED PARTS COMPOSITION
JANUARY 28, 1985

FORD AEROSPACE & COMMUNICATION CORPORATION
AERONUTRONIC DIVISION

R.M. Bieniak
L.M. Griffin
L.R. Tripp

DIGITAL SYSTEMS DEPARTMENT

Introduction

This paper presents a summary of issues, ideas and experiences relating to the area of automated parts composition based on hardware and software experience from members of the Digital Systems Department of Ford Aerospace and Communications Corporation, Aeronutronic Division, Newport Beach.

The methods used in the development of automated hardware design systems may have applicability in the creation of similar tools for developing large software systems from reusable software components. Although several universities are currently sponsoring research projects in the area of automated software parts composition, no readily available systems exist for the construction of large software systems.

The value of an automated software design system is especially evident when one considers the base of reusable software components that will evolve as a consequence of recent Department of Defense standards requiring the use of Ada as a high order language.

This paper will cover the following:

- o Major issues of reusable parts composition
- o Suggested approaches
- o Ada experience at Ford Aerospace

Major Issues of Reusable Parts Composition

The identification of the major issues of reusable parts composition will serve to help focus our attention so that no major issues are overlooked in our suggested approaches. There are many issues associated with reusable software components all of which have to be addressed in a working system. This section will identify only the issues of reusable parts composition.

Interface Between Components

The interface between components must have several properties which will simplify both the design of new components and the use of existing components in larger programs. It is necessary to identify the properties of the interface since it will strongly influence the design and function of the entire automated part composition system.

The interface should be as flexible as possible so as to allow many components in the library to be connected together. At the same time, the interface should prevent the connection of components that are clearly mismatched in both number and type of parameters being passed.

A weak interface that does not do any syntactic or semantic checking of the incoming data gives the user the most flexibility when interconnecting components.

Flexibility is important since it permits and encourages the interconnection of components without regard to whether the correct numbers and types of parameters a being passed. The Unix system of reusable components is a very successful system based on a weak interface between components. Only uniform character-stream data is passed between components. The one outstanding problem with this type of interface is that it allows the connection of components that produce no meaningful results. It places the burden of semantic checking on the user.

Strong interfaces shift the burden of semantic checking from the user to the computer. This type of interface increases the change of connecting components that will produce meaningful results. Strongly typed languages like Ada enforce strong interfaces.

A uniform and unambiguous interface that incorporates some of the properties of both weak and strong interfaces needs to be developed. Once developed, a specification should be set forth early so that programmers can design new components for addition to a component library with the specification in mind. Also, once these specified, there will be no question about which components can be connected.

Also at issue is the hidden interface between components that access common data structures. This pathological connection between components needs to be controlled since it can cause anomalous behavior in tested and verified components. Since common data structures are an efficient means of passing large data structures between components, they should not be eliminated, only controlled.

Component Library

There are several issues related to the component library that are of particular concern to automated part composition. One issue is how the components are to be created and inserted into the library. Creating new components may be needed for at least two reasons. The first reason is that several smaller components could be connected together into a new component of higher complexity. In this case, automated tools could simplify the creation of the new component since no new code would have to be added to the library. The second reason for creating a new component is that no component or group have to be written and verified. Inserting new components into a global library would have to go through some sort of review process. This process is necessary to limit the library components to those components that are generally applicable to the problem domain of the library.

Loading, linking, and invoking components in the library is also an important issue. Ideally, loading and linking should be done on-the-fly to provide immediate feedback as each component is added to the program being constructed. This dynamic loading and linking also provides a means for automatically linking in components that are invoked by the component being added. Invoking functions in a data flow type of program made of components is complicated by the fact that functions are often invoked by name. Automated techniques that modify component code to invoke the following component may have to be developed unless an alternative to function invocation by name similar to Unix pipes is used. This affects the strengths and weaknesses of the component interfaces.

The final issue related to component libraries deals with the kind of information that is needed to aid the programmer in building a composite program. For large libraries of components to which new components are added on a regular basis, some automated means to determine which components may be useful in performing a particular function would be helpful. With such a system, the user could be spared the task of reading and memorizing the function of each component in the library.

User Interface

Considerable attention has to be paid to the issue of the user interface. The easier a system is to use, the more often it will be used. The user interface should be highly interactive providing feedback to the user along each step of the composite program construction. Interfaces that provide both graphical and textual information similar to those used for computer aided design of electronic circuits should be explored. Some

simple programming systems that are primarily icon driven have been developed. These systems show the usefulness of graphics in a programming environment.

Testing and Debugging of Components and Composite Programs

In any programming environment, tools for testing and debugging modules and programs are very important. This is especially true for an automated part composition system since the user has less knowledge and control of the quality of the code in the components. A means for testing individual components similar to source code debuggers found on the VMS and Unix operating systems is necessary. This type of debugger provides the control to closely inspect the function of components at the source code level. Given that the components have been verified to some level of confidence, the user must also be provided with tools that allow the inspection of data as it is being passed from one component to the next. Once a large library of components has been established, this will be the primary means of debugging the composite program since few new components will have to be written.

Suggested Approaches

The rising costs of software development has stimulated interest in the concept of reusable software components. The inclusion of these reusable components into a system design can reduce design and implementation costs. The four areas to be discussed under approach to reusable components include:

- o Experience with Reusability
- o Manual Approaches
- o Automated Approaches
- o Proposed Automated System Scenario

Experience with Reusability

Ford Aerospace's experience with reusability during the last few years has included some traditional and innovative ways to reuse software design and coding efforts.

The traditional means of software reuse has been to generate general purpose modules which may be used in various systems. These were typically done in high order languages and some assembly language modules. The system designer could use these modules and save time implementing similar support facilities. This traditional mechanism for software reuse has been enhanced with the advent of the Ada language and the Package program unit. During the previous year reusable Ada Packages were generated as a baseline for reusable Ada software.

A significant expense in software development takes place in the design phase. To recapture the effort the engineer spent designing the unique system software would save many engineering hours. During the past several years, Ford Aerospace has been developing software for our specialized, real-time image processing hardware. In the development process, numerous software components were found to be common in image processing applications. To minimize costs of designing software, these common pieces were designed to be reusable diagnostic and tracker frameworks.

The diagnostics generated for our hardware had many common pieces. These common pieces were isolated from the diagnostic specific pieces. The common pieces were linked together to form a reusable diagnostic framework. This framework consists of control parameter, error tables, hardware configuration tables, graphics output, verification mechanisms, etc. Upon completion of the diagnostic framework, the diagnostic dependent section could be inserted into the framework.

The tracker applications proposed for our real-time processor possessed the same characteristics as the diagnostics. They had many common pieces and some application dependent pieces. The common pieces were grouped together to form a reusable tracker framework. This framework consists of loop control mechanisms, timing framework, algorithm tailoring functions, memory manager, track file data base, etc. With this generalized framework, new tracking algorithms could be inserted into the framework. As an ongoing research project, this framework is

being enhanced with better interfaces, easier algorithm insertion, and provide reusable frameworks for new application code.

Manual Approaches

During the next several years the reusability of software will become a major issue in the software community. As a contractor with the DoD, our emphasis will revolve around Ada and reusability. The Package construct will help generate components which can be used for system generation. The packages will consist of functions which perform on specific objects, for example, set operators or trigonometric functions. In order to make them as general purpose as possible, Ada generics will be used, so the type of values used by a function is more flexible. After generating these reusable components, they should be grouped by problem domain and cataloged for easy user access.

In the early design phases of a new system the reusability issue needs to be addressed for two reasons: (1) there may be existing software components which could be used to solve the problem, and (2) as a product of our design, new reusable packages may be generated which will benefit future products. The new reusable components should isolate machine dependencies which limit their portability. Also, all the major issues of reusable components should be addressed to extract maximum benefit of the reuse effort.

As systems designers use the reuse philosophy in their design practice, a large selection of reusable components will be available to future projects. These components will only be beneficial to designers if they are made aware of what is currently available. This awareness can be implemented manually through a company wide component catalog which might be separated by problem domain.

Automated Approaches

The manual approach initiates the concept of the reusability issue. The first step in automation consists of creating a database of reusable components and their associated documentation. This database should group the components by the problem domain with which they interact. An automated search and retrieval mechanism will allow quick and easy access of the reusable components.

The next step is to automate the creation of new components, reusable or not. This automation will provide a similar capability to the software engineer as a Computer Aided Engineering (CAE) system provides to a design engineer. There will be basic, low-level building blocks to create new components. Upon completion of a new component, it can be entered into the reusable component library. Systems can be created using this Computer Aided Programming (CAP) system by linking together components to solve the system problem. The CAP system will validate interfaces and verify that all the components mesh correctly together. For real-time, mission critical systems, characteristics about each component could be stored in the CAP database, allowing automated size and speed analysis for the proposed solution. The software engineer, from his workstation, could evaluate various system configurations until the desired space and performance is achieved.

Proposed Automated System Scenario

The development of libraries of reusable software components is a means for taking advantage of automated techniques for software development. The smaller the amount of new code that has to be developed, the greater the productivity gains that can be realized through automated techniques. This section outlines our approach for an automated system for developing composite programs from software components which can be thought of as a prototype for future CAP systems.

Our approach is based on two central concepts: (1) an interactive graphics environment similar to that used for Computer Aided Design (CAD) of electronic circuits, and (2) automated tools that support the checking and building of the program under construction.

The first concept is based on an attempt to duplicate the productivity gains realized with CAD of electronic circuits in the development of composite programs. In such

a system, a unique graphic symbol or icon is used to represent each component in the library. The shape of the icon can be used as a visual cue by which the user can determine what component icons can be connected together. This type of icon driven programming system requires a fairly sophisticated graphics editor.

The graphics editor allows the user to fully control the interconnection of icons in an interactive environment. This means that each step of building a graphic diagram of a composite program from component icons is done using the graphic editor. The user should be able to recall icons from the component library, place them on the screen, and connect them with data paths. Removing icons and data paths should be just as easy. As evidenced by the current CAD tools for electronic circuit design, graphics editors can provide a user-friendly high information channel between user and computer. This can translate to improved programmer productivity as long as the programs one is trying to graphically compose can indeed be represented in such a form. It is therefore necessary to initially restrict the domain of the icon driven system to programs with simple control and single-stream flow of data. While this restriction eliminates multi-tasking and interrupt driven programs, it does provide for a wide range of programs that can be broken down into modules that perform specific data transformations. One problem domain that falls neatly into this category is image processing. Because of our experience in image processing, our initial reusable software component library will be restricted to the image processing domain.

Restricting the component library to a specific domain has two major advantages. First, the number of components in the library will be easier to manage. Second, it will be easier to develop reusable components that cover most of the anticipated processing needs.

Another way of implementing the CAP system would be to host it on a popular personal computer with a hard disk to store the necessary database. There will be two creation levels on the CAP system. The component and system. In the component mode, the user will be able to create reusable or library units from a set of basic or elementary components. This might consist of basic Ada language constructs to form a building mechanism. Upon completion of the component, it may be entered into the component database along with its corresponding statistical data. This data might include number of source lines created and execution speed. To make the task more manageable, a single source language will be selected, Ada. In the system mode, components may be linked together and if necessary, discrete basic components may be used in the interconnection. The component library may include common filters, transforms, etc., commonly used in image processing.

The user interface will be very similar to a CAE system, it will have graphics which represent program units in which the user interfaces to through a mouse or keypad. The component or system being built will be displayed on the graphics monitor for viewing. Upon completion of a component or system build, the CAP system will output the graphic representation of the system, perform interface verification, source code output, size and speed information, and source code documentation.

The other concept of using automated tools to support the checking and building of the program under construction is necessary to provide for those tools that are not directly related to the graphic composition of the program. Some tools that fall into this category would be interface checking tools, testing and debugging tools, and component identification and location tools.

Interface checking tools would verify that the connection between components is syntactically and semantically correct. These tools are necessary to catch errors that are typically caught during the compilation and loading phases in a regular programming environment.

Testing and debugging tools will aid the programmer in catching errors that can only be found during the execution of the program. These tools should have features similar to debuggers found in the VAX/VMS or Unix operating systems. The more graphically oriented these tools are, the easier they will be to learn and use.

Because of the abstract nature of software components, some automated method of identifying existing components that may be used in the program under construction is needed. This technique may cut down on the creation of components that are only slightly different from existing components or groups of components. A simple automated component identification method could be based on keyword matching. Using such a method, the programmer would enter some keywords that describe the program or part of the program under construction. The automated system would then search for matching keywords in the descriptions of the existing library components, and display the names of the matched components.

ADA Capabilities

Ford Aerospace and Communications Corporation, Newport Beach has closely tracked Ada development since 1980. Ada research projects are underway as well as an Ada contract. Ada training began in 1982 with courses after work. Ford has also played an active role in Ada Users groups. A member of our staff is the national AdaJUG Education Committee chairperson. Ford also has a corporate Software Engineering Steering Committee (SESC). One of the tasks of the SESC is the sharing of Ada knowledge across the corporation. In addition, the SESC has set up a reusable software library. The following sections summarize Ada capabilities.

ADA Research Project

The Ada Research Project has been concentrating on the usage of Ada for Embedded Computer Systems. Part of the project has produced reusable Ada packages. In 1985 an existing system in Ada will be redesigned and produce reusable packages identified from the redesign.

The Ada packages produced last year were:

Fixed Point Math Library
Includes all Cody Waite algorithms

Generic Fixed Point Math Library
Includes Sin, Cosine, Tangent, Arc tangent, Arc sine, Arc cosine, Square root, Cube root, Logarith, Logarithm 10

Generic Transformations Package
Includes Rectangular To Spherical
Spherical To Rectangle
Coordinate Transformation

Table Lookup Math Library
Includes Sin, Cos, Asin, Acos

The Research Project is also producing a guidelines document for the Usage of Ada in Embedded Computer Systems. These guidelines can then be used by the systems designer to utilize Ada constructs and packages to produce an optimal Ada system. Therefore, it is the start of our knowledge base for producing Ada systems from reusable components.

ADA Contract

The FLIR Mission Payload Subsystem (FMPS) contract was awarded in June of 1983. It requires Ada for design and implementation. Ada is to be used in an embedded Motorola 68000. The FMPS contract is currently in the design phase.

ADA Training

In order to assure that the Ada language is properly utilized and to aid in meeting our research objectives, Ford Aerospace has an Ada training plan in effect for the Aeronutronic Division which has resulted in over sixty engineers taking Ada courses. The training plan identifies Ada for Managers, Introduction to Ada, an Ada design course, and Ada for Real-Time Programmers. Courses have been given during the day as well as in-plant University of California at Irvine (UCI) extension Ada courses after hours.

ADA Facilities

For Aeronutronic has an Ada Software Development Center for the research projects and contract work. This consists of a Data General MV10000 computer with 12 remote terminals, a nine track tape drive, a 940 megabyte disk, and the Ada Development

Environment (ADE). The ADE has a validated Ada compiler targetted only to the MV10000. Also in the Ada center is an Intellimac IN/7000M Motorola 68000 based computer with a 50 megabyte disk, a 1.6 megabyte floppy, a 330 megabyte Winchester disk, a nine track tape drive, two printers and six remote terminals. The Ada compiler currently on the Intellimac is the Telesoft Ada subset compiler under the ROS operating system. The Intellimac has a phone line to another lab where the Tektronix 8561 development station is located. One of the terminals on the Tektronix can be used in a virtual terminal mode, using the phone line, to download Motorola S-record formatted code from the Intellimac to the Tektronix.

Ford also has three VAX 11/780's available for engineering work as well as a classified VAX. One VAX has the validated NYU Ada Translator/Interpreter. It has been used for Ada training. It has also been used to verify or refute results from the other Ada compilers.

Our Ford facility in Colorado Springs is one of the field test sites for the validated DEC Ada compiler. Our Newport Beach facility will become a spawned beta site as soon as one of our VAX 11/780's has installed the latest version of the operating system, VMS 4.0. Many of our Ada programs will be recompiled using the DEC Ada compiler, in order to evaluate it during the field testing period.

Ford also has access to Arcturus, an experimental Ada environment being built by UCI, on a VAX 11/780. Arcturus contains a subset Ada compiler with other Ada-specific tools.

RESUME

RICHARD M. BIENIAK
R&D Engineer

Education

BS, Information and Computer Science University of California, Irvine

Supplemental Education

Master's degree in Computer Science in process. Introduction to Ada Programming Real-Time Systems with Ada

Clearance

Secret

Years of Related Experience

6

Professional Experience

As a graduate student, Mr. Bieniak provided teaching assistant support to the UCI professors in the areas of software systems design and software engineering.

As a member of the Digital Systems Department of FACC, Mr. Bieniak has been responsible for the design of several real-time, embedded software systems targetted for weapon systems applications. This work addressed the software reuse issue as it applies to unique, custom real-time embedded computer system software.

Significant Assignments

- o Principle designer of the tracker and diagnostic software frameworks which facilitated the recovery of software engineering time by grouping reusable components into frameworks for specialized, real-time, computer hardware.
- o Developed an Assembler/Translator which transforms serial assembly language instructions into parallel bit slice microcode.
- o Generated guidelines for using Ada with embedded computer systems, this document also discussed software engineering practices to be used with the Ada language.

Activities and Honors

Spoke at the UCI Computer Symposium on the issues of reusability of software as it relates to real-time software systems.

RESUME

LORRAINE M. GRIFFIN
R&D Engineer

Education

BS, Mathematics Youngstown University

Supplemental Education

Taught in-plant University of California at Irvine (UCI) on the Ada Programming Language

Clearance

Secret

Years of Related Experience

17

Professional Experience

As a member of the Digital Systems Department, Ms. Griffin is currently project leader of the Ada Technology Research project. She also supports proposal efforts where Ada is being considered and assists in technical questions regarding Ada. She has given Ada programming training as well as Ada Management Seminars. She has participated extensively in Ada tools evaluations.

Prior experience includes the programming of a circuit card test station in ATLAS on the HP1000. She has written many engineering applications programs for radar systems in FORTRAN utilizing graphics. She has experience in Ada, FORTRAN, ATLAS, COBOL, and various assembly languages on several computer systems such as: Honeywell, CDC, VAX, Intellimac, and Data General.

Significant Assignments

- o Ada Technology IR&D project leader
- o Ada tools evaluation team member
- o Taught five UCI extension courses in Ada
- o Programmed circuit card test station

Professional Memberships

SIGAda, AdaJUG, AIAA TC on Computer Systems

Activities and Honors

Chairperson of AdaJUG Education Committee

RESUME

LLOYD R. TRIPP
R&D Engineer

Education

BSEE, BSME—Duke University 1979

Supplemental Education

Graduate Certificate in Signal Processing - University of California, Santa Barbara 1981

Graduate work in Computer Science - University of California, Irvine

Clearance

Secret

Years of Related Experience

3

Professional Experience

As a member of the Digital Systems Engineering Section, Mr. Tripp is currently responsible for the development and enhancement of algorithms for the segmentation of images.

Mr. Tripp is also a responsible engineer for the insertion of VHSIC technology into digital systems that are under development in the Advanced Development Operation.

As a member of the Software Design Section, he participated in the design, code, and testing of tracking software and a real-time multitasking operating system for the FPAD image processing and tracking system.

Professional Memberships

Member of the IEEE Computer Society

Publications

"Autonomous Target Detection by Change Detection in a High Clutter Environment", SPIE April 1983.

Ford Aerospace &
Communications Corporation

STARS
WORKSHOP

AUTOMATED PARTS COMPOSITION FROM REUSABLE SOFTWARE COMPONENTS

RICHARD M. BIENIAK

LORRAINE M. GRIFFIN

STARSWG2.DAT

Ford Aerospace &
Communications Corporation

STARS
WORKSHOP

AUTOMATED PARTS COMPOSITION

MAJOR ISSUES

- AWARENESS
- CREATION

STARSVGS.DAT

Ford Aerospace & Communications Corporation

AWARENESS

ON-LINE CATALOG

- PROBLEM DOMAIN ORIENTED
- KEYWORDS
- ADA BASED REUSE
- LEVELS OF INFORMATION
  - SPECIFICATION
  - ALGORITHMIC
  - ATTRIBUTES

STARSVOS.DAT

Ford Aerospace & Communications Corporation

AWARENESS

CATALOG EXAMPLE SCENARIO

```
$ LIST/DOMAIN
    IMAGE PROC    TRIG    SORT

$ DOMAIN SORT

$ LIST/ALL
    BUBBLE      SHELL    QUICK

$ LIST/SPEC BUBBLE
    PROCEDURE BUBBLE
     (IN VECT : in out VECTOR);
```

Ford Aerospace &
Communications Corporation

CREATION

AUTOMATED PROGRAMMING SYSTEM (APS)

- ICON DRIVEN
- SIMILAR TO CAE SYSTEMS
- RETRIEVAL MECHANISM
- S/W DELIVERABLES CREATED
- BUILDING BLOCK LEVELS
- PICTORAL REPRESENTATION OF SYSTEM

Ford Aerospace & Communications Corporation

CREATION

BUILDING BLOCK LEVELS

- ADA LANGUAGE FEATURES (SSI)
- AMINO ACIDS OF DOMAIN SET (MSI)
- LOW LEVEL COMPONENT (LSI)
- HIGH LEVEL COMPONENT (LSI)
- SYSTEM (VLSI)

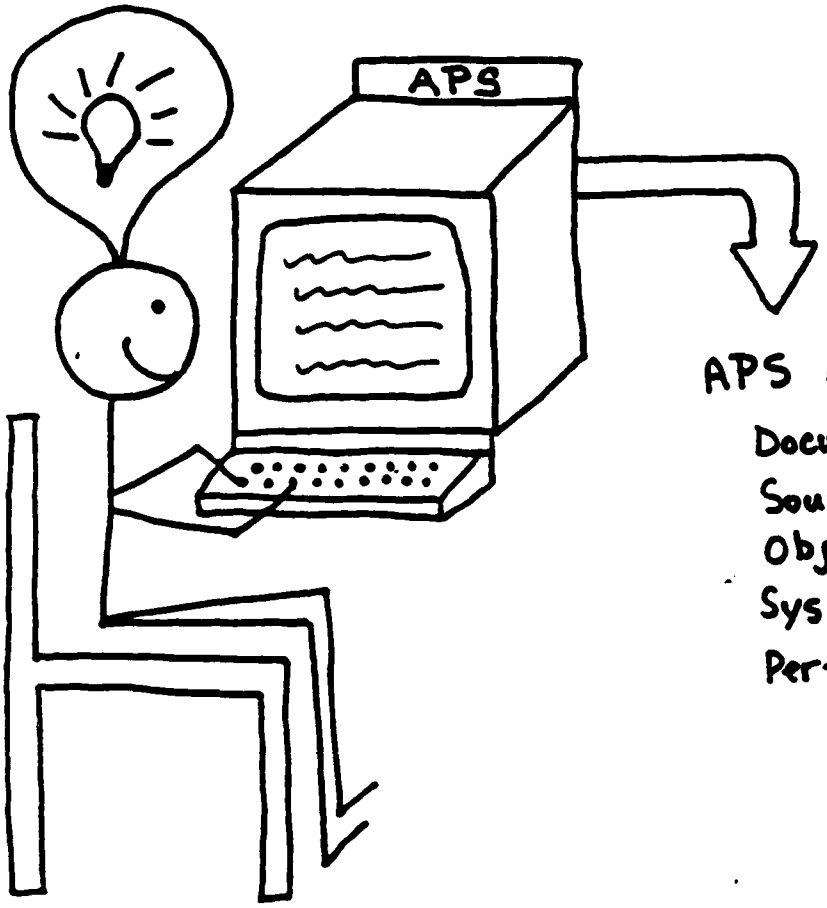
Ford Aerospace &
Communications Corporation

STARS
WORKSHOP

2001 --

A SOFTWARE ODYSSEY

STARSVC1 DAY



APS Generates:

- Documentation
- Source Code
- Object Code
- Sys. Pictorial
- Performance Anal.
- ⋮

Ford Aerospace &
Communications Corporation

STARS
WORKSHOP

1985 --

BACK TO REALITY

Ford Aerospace & Communications Corporation

ISSUES

- DEFINITION
- INTERFACES
- CREATION AND INSERTION
- LINKING AND LOADING
- INFORMATION
- TESTING AND DEBUGGING

Ford Aerospace &
Communications Corporation

REUSABLE QUOTATIONS

TO CODE IS HUMAN. TO REUSE IS PROFITABLE.

WHERE'S THE COMPONENT ?

REUSE IS A BETTER ILEA.

A GENERIC PACKAGE CATALOGED/PUBLICIZED IS WORTH 222 BY AN ADA GURU

STARSVGS.DAT

A Discussion of Ada Experience At General Dynamics Data Systems Division Western Center

John J. DaGraca

General Dynamics

February 1, 1985

Abstract

This paper details certain Ada experience at General Dynamics Data Systems Division's Western Center in San Diego, California. A general description of Ada activities at the Western Center is followed by a project description and details of "special problems" encountered in using Ada on two specific projects performed at the Western Center. The first details experience in using Ada in a missile embedded system under an Air Force contract and the second describes developing resuable software tools and components for an internally funded program.

Introduction

General Dynamics is committed to using Ada as the implementation language of future military systems, and recognizes that the real payoff will be when Ada is used through a project life cycle (i.e., from system definition through software maintenance). Data Systems Division Western Center, has supported the Department of Defense standard language since Ironman and has internally funded review teams that provided inputs to Pebbleman, Stoneman, and the resulting KAPSE and MAPSE. Western Center personnel attend most Ada workshops and have presented a paper at the Ada Test and Evaluation Workshop in Boston expressing specific areas of concern regarding Ada's potential impact on embedded systems.

We are currently using and evaluating several Ada compilers and Ada based Program Design Languages (PDL) as well as continuing our investigation of other available Ada tools to determine those which best suit our needs. Several Ada compilers are currently installed on computers located at our San Diego facility. We are using a Florida State University (FSU) developed Ada compiler, hosted on a CDC CYBER computer, in performance of our Tactical Ada Guidance contract with the Air Force. Two versions of the New York University (NYU) Ada/Ed interpreter, the TeleSoft VAX compiler, and two TeleSoft M68000 based Ada workstations are available to development personnel. We have recently become a field test site for the Digital Equipment Company (DEC) Ada compiler and have installed the DANSK Ada compiler in one of our facilities.

General Dynamics has funded several Ada related Internal Research and Development (IRAD) projects in order to develop Ada expertise. These projects include the Distributed Processing System Architecture IRAD, the Ada - 1750A Standardization Study, the Strike Planning Package, and the Vehicle Elect which Ada is being applied include embedded missile systems, fire control and ballistics, weapons simulation, limited graphics, weapons trainers, command and control, and Man-Machine Interface (MMI). In conjunction with our IRAD projects, we are designing several Ada benchmark programs to allow us to determine not only the speed and accuracy of Ada compilers as they become available, but also to evaluate the feasibility of using full Ada in real-time environments by using the tasking functions. We are active participants in the Ada community in order to remain aware of current and future Ada activity and to contribute to the exchange of Ada technology. We regularly participate in the Ada/JOVIAL User's Group (AdaJUG), and SIGADA conferences and have attended the IEEE "Ada as a PDL Workshop", the AFSC Standarization Conference and the AJPO Evaluation and Validation Team

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

workshops. We participated in the DoD sponsored Software Initiative Workshop in Raleigh which has evolved into the STARS program. We are an active corporate sponsor of the local chapter of SIGADA and co-hosted a joint meeting of the AdaJUG and AdaTEC in San Diego last February.

Experience in Using Ada in a Missile Embedded System

The Tactical Ada Guidance (TAG) program provides an opportunity to demonstrate the full capability of Ada as required in a tactical missile guidance system under contract to the Air Force Armament Laboratory (AFATL) at Eglin Air Force Base. General Dynamics Data Systems Division's Western Center has redesigned and is currently implementing the software in a MRASM Cruise Missile using Ada. Originally implemented in JOVIAL and Zilog Assembly Language, the MRASM Test Instrument Controller (TIC) software provided a unique case to implement a redesign which encouraged extensive use of Ada language features.

The TIC is one of three Digital Integreting Subsystem computers that support MRASM avionics. It was selected for TAG because its functions are more logic intensive than computational, it requires high I/O through-put and interrupt rates and requires asynchronus task. The TIC computer provides three major functions for MRASM. A telemetry formatiing function samples bus data, analog channels and discrete signals and generates a telemetry stream based on a changable, but pre-flight specified, format. A second major function provides the range safety missile remote command and control (RCC) link between a chase aircraft and the missile's autopilot. The third major function provides a continuous built-in-test capability on a non interference basis.

Performance on the TAG contract has provided software designers and developers the opportunity to utilize a number of different Ada compilers in addition to the Air Force AFATL/Florida State University compiler. These included compilers produced by NYU, Rohm Corporation, TeleSoft, DEC and DANSK. The nature of the TAG application encouraged extensive use of Ada language features including:

- o Packages
- o Overloading
- o Subtypes
- o Allocators
- o Aggregates
- o Named parameters
- o Private Types
- o Renaming
- o Exceptions
- o Tasking
- o Entry families
- o Priorities
- o Attributes
- o Abort
- o Address clauses
- o I/O
- o Separate Compilation
- o Unchecked type conversions

The use of Ada in this (MRASM) real-time embedded computer system provided a variety of application types including multitasking, time critical I/O handlers and low level system applications such as hardware memory testing. As might be expected, the development team has and is continuing to encounter problems in using Ada in this environment. A complete "lesson learned" report will be developed as part of the TAG contract. The following paragraphs address three specific areas of general interest that become evident during the TAG project:

- o Desirable language modifications
- o Needed pragmas
- o Time critical interrupt processing

A desirable language modification would be to permit the inclusion of anonymous record types within a record (as does PASCAL) to support the concept of subrecords. At present, a subrecord cannot be declared in this context of the main record. A new type must be declared which has no meaning by itself. The current implementation results in a proliferation of types and forces the reader to understand the record from the bottom up in violation of the principles of top down design and top down understanding.

Three areas have been identified where additional pragmas are desirable:

- o Initialization time control
- o Static object placement
- o Shared variable problem

Ada provides initialization of objects by assignment statements in unit code bodies, in declarations, or in package code bodies. The programmer has no explicit control over whether the

compiler generates run-time instructions or a load-time instructions (pseudo-ops); however, there are several situations where only one method is correct.

- o Objects in ROM are impossible to initialize at run-time.
- o Values in RAM must be initialized at run-time to preclude loss due to power off conditions prior to use.
- o EEPROM variables will be extremely slow to initialize at run-time and should generally be initialized at load-time.

A pragma enabling the programmer to specify objects to be initialized at load-time is desirable. The default should be run-time because initialized values may contain a function or allocator.

Unlike JOVIAL and many PASCALs, Ada does not provide an option to specify that an object is static, i.e., is not placed on the stack. The capability of specifying static is desirable because the program logic may require that the objects value be retained between calls to the program (as in the case first pass through the program logic). At present Ada provides two unsatisfactory methods of implementing a static variable:

(1) the address clause and

(2) declaring variables in library level packages.

The first method is awkward because a static requirement does not imply that a programmer should know where an object be allocated in memory. Attempting to coordinate this information, as with a utility program, is not practical. Moving a static to a library level package mitigates the initialization problem but requires moving the object and its required type statement out of their natural scope.

The Ada method of sharing objects among tasks is to encapsulate them in a "monitor" task which provides the only access to the objects. The Ada run-time system guarantees that all callers of the task will wait until the current one is through, thus providing mutual exclusivity to shared variables. This method is highly reliable but requires many more tasks in the system and entails the overhead of going through the scheduler and other task switching activities. A proposed solution to these problems is an additional pragma which assures special treatment for monitor tasks. Such pragma would directly connect the monitor task to the calling task by issuing a service call, run the monitor task in a non-interruptable mode and eliminate non-essential task switching activities.

Another Ada problem which confronted the TAG program is that response to some interrupt handlers would be too slow if normal Ada constructs were employed. TAG provided two specific instances of this problem: first, when data can "burst in" and be lost if it is not processed in a timely manner and second, then the interrupt represents an emergency situation. Both situations require timely response but for different reasons. The first situation requires insuring that the data is saved before it is destroyed by new data. It can be processed later. The second situation simply requires minimal delay in processing the interrupt.

The TAG approach to this problem is a user provided, quick-reaction handler which performs minimal processing prior to enabling subsequent interrupts. This handler cannot afford the time delays associated with a normal Ada "rendezuous". Instead, TAG enqueued the interrupt data for the run-time system to process later when interrupts are enabled. Of course, both the handler and the Ada run-time system will have to disable interrupts while accessing this shared queue in order to assure data integrity. It should be noted that this approach is not an Ada solution but it minimizes the time during which interrupts are disabled and is necessary until more sophisticated compilers are available.

Developing Reusable Ada Software on an IRAD Project

The Integrated Strike Planning (ISP) project provides an example of DSD's experience attempting to develop reusable software tools and components in Ada. ISP is being developed for the Operations Research Department of General Dynamics Convair Division. Strike planning is the process of decomposing military objectives into discreate weapon-target pairs so as to ensure a high probability of achieving a specified level of damage and of aiding in the deconfliction of tactical air power and cruise missiles. It begins in the target area with the weaponeering of individual target/weapon pairs, then coordinates multiple vehicle paths to maximize damage and minimize interference. General Dynamics anticipates interest in ISP from a number of potential DoD customers.

Software development on ISP is being performed by software engineers who have an average of 1-1/4 years of Ada language experience. Development Work is being accomplished using the October 1983 release of the TeleSoft Ada compiler for VAX computer executing under the VMS operating system. One development effort was also attempted using the DANSK Ada compiler also operating under VMS.

An objective of the ISP software effort was to develop reusable software systems in Ada. Candidate software elements were selected on the premise that stand-alone software meant reusable software. The following reusable software "systems" have been developed in Ada:

(1) Rosette - A screen formatting package designed to help a project maintain consistent menus and terminal display outputs.

(2) Statistics-Operations - A portion of the statistics-operations software in Q-SIM (a Fortran based discrete event simulation package developed by Convair based on the concepts of SLAM) was recoded in Ada. The package permits collecting relative and absolute time statistics, observation statistics and operations such as merging statistics and computing and outputing histograms.

(3) SIMULADA - A discrete Monte-Carlo simulation package.

(4) Joint Munitions Effectiveness Methodologies (JMEM) - The JMEM Air-to-Surface Weaponeering Methodology calculates probability of kill (Pk) against specified targets using various weapon combinations.

(5) WANDA - A graphics primative package designed to drive a Megatek 1650 color-graphics display terminal.

(6) Target Evaluations Module - This module in ISP assists the strike-planning tasks of decomposing a mission objective into a set of weighted targets. An interactive fuzzy-decision algorithm obtains a hierarchy of valuations from any numbers of users and outputs a list of cardinal target values for the selected targets.

The use of Ada to generate reusable software on the ISP project raised certain ADA specific issues. Several of these are discussed in the following paragraphs.

The most apparent issues are the problems caused by attempting a develop of tools. Specifically, ISP was impacted by the lack of:

- o Symbolic debrs
- o Pretty pointers
- o Generic libraries
- o Data base management systems.

The ISP project demonstrated that a validated Ada compiler does not necessarily generate valid Ada code; that is, the validation suite and the LRM are two different beasts. For example, an error was discovered in the TEXT_IO package that reversed the order of statement terminal the text string being disAdditionaMavalidation supragma INnt compiled sondefinition of self documenting featctthfact that the stands and type/object declarations were not possible/normal.

Finally, the ISP project raised several configuration management and quality control issues resulting from the ability to perform separate compilation of program body stubs. One of the questions raised was what is the relationship between a file name and a procedure and a different file name with that procedure's body stub. The potential magnitude of this issue was demonstrated by the fact that there were over 200 VAX files associated with three ISP modules.

Summary

A variety of sample problems encountered in the use of Ada on two General Dynamics projects have been presented. It is important to note the different nature of the two projects and the corresponding difference is the types of problems reported. The earlier IRAD project raised issues that were frequently directly attributable to the immaturity of the specific compiler employed. It also raised serious configuration management issues which will impact all large, full scale development projects. The TAG program raised more specific technical issues directly related to real-time missile software. Some of these issues required solutions outside of a pure Ada context. Many others, however, suggest a need for a better way to solve specific issues demonstrating an increasing understanding of the Ada language by the user. The types of issues raised on TAG further reflects the increasing maturity of the Ada compilers available today.

RESUME

JOHN J. DA GRACA

EDUCATION

MS, Computer Science
Brown University, Providence, RI, 1978

BSEE, Electrical Engineering
North Eastern University, Boston, MA, 1974

ASC, Math-Physics
North Eastern University, Boston MA, 1970

TECHNICAL QUALIFICATIONS AND ACCOMPLISHMENTS

Mr. Da Graca has over 15 years experience in the design and development of real-time mission critical systems, process control systems, compiler developments (PASCAL, JOVIAL), graphics systems, man-machine interface, peacekeeper C?3O software development and software development methodology for peacekeeper. He has taught several Ada classes and has developed a complete set of math-library packages in Ada.

Mr. Da Graca has published several technical papers on Ada.

(1) Design and develop a real-time hardware/software system for field configuration of process control plants. The purpose was to design a small field configurable system using microprocessor hardware, using modern control theory with as many as eight general purpose control algorithms.

(2) Design and develop software system for Fast Error correction program. The purpose was to study the suitability of Ada in communication systems. This involves message text processing, signal processing technique and mathematic library packages.

(3) Design and develop a real-time executive for mission critical system. The purpose was to investigate run-time overhead associated with Ada when using tasking mechanisms and Ada constructs to investigate the run-time dependent features of the various Ada compilers.

(4) Design and develop a complete set of mathematic library packages to be used in real-time Ada application environments. The purpose was to design and develop high speed calculations of math functions such as:

- Square
- Square Root
- Sine
- Cosine
- Secant
- Cosecant
- Exponential
- Logrithmic
- Tangent
- Cotangent
- Integer Random
- Floating Random
- Digital Filter 3rd Order Chebyshev

(5) Design and develop C?3O executive. The purpose was to study kinds of software tools required to support a methodology during the life cycle of software development with Ada. During this effort three methodologies were used:

- Object Oriented Design Approach
- Process Abstraction Methods
- Data Flow Diagram Techniques

(6) Design and develop cross-assemblers, cross-linkers and emulators for several high speed missile processors.

(7) Retarget JOVIAL for missile processor.

(8) Design and develop PASCAL compiler for missile processor.

(9) Design and develop tactical missile system control algorithm.

(10) Design and develop test plan and procedure for peacekeeper C?3O software.

RESUME

EDWARD J. ANDERSON

EDUCATION

MS, Business Administration
University of Northern Colorado, 1977

BA, English
North Carolina State University, 1966

TECHNICAL QUALIFICATIONS AND ACCOMPLISHMENTS

Mr. Anderson has over 17 years experience in systems requirements analysis and design for C?3OI, Communications Systems and Signal Processing applications for Navy, Army, Air Force and NASA customers. He has experience in CAD/CAM data base management applications and is currently serving as technical coordinator for the development of General Dynamics Corporate with Ada training program.

(1) Currently coordinating requirements definition to develop a corporate-wide Ada training program for all General-Dynamics divisions. Reviewing and monitoring ciriculum development activities of an independent Ada training consultant.

(2) Responsible for defining software design requirements to implement a distributed data base processing system in a mini-computer environment for an Army C?3OI system.

(3) Developed requirements for implementing a corporate-wide CAD/CAM data base management system.

(4) Project manager of Test and Evaluation for several advanced acoustic processing and communications systems employed in an ASW environment.

(5) Software designer for the Naval Modular Automated Communications System (NAVMACS) and Naval Ocean Surveillance System (OSIS).

(6) Software designer and developer for several Naval Tactical Data System (NTDS) applications including target tracking algorithms, multiplex processors, missile control software, display systems and electronic warfare.

RESUME

JAMES SCHNELKER

EDUCATION

MS, Computer Science
University of Wisconsin, 1968

MS, Math
University of Chicago, 1966

BA, Math
University of Chicago, 1960

TECHNICAL QUALIFICATIONS AND ACCOMPLISHMENTS

Mr. Schnelker has over 15 years experience in the design and development of real-time command and control, avionics, and process control systems. He has developed data base management systems as well as numerous software development tools.

(1) Responsible for a project for the evaluation of the Ada programming language for real-time systems. The purpose was to establish strategies and tactics for the effective use of Ada in space-critical, time-critical embedded systems.

(2) Responsible for the design and implementation of a real- time operating system for a network of Z8000s. This involved Inter-computer communications software, several I/O drivers, a bootstrap loader, two multi-task scheduler and a memory manager.

(3) Responsible for the development of a context dependent language translator and interpreter for automatic generation of specification languages.

(4) Responsible for the design and implementation of an interactive system to define and manipulate schematics on limited-graphics CRTs.

(5) Responsible for the design and implementation.

(6) Designed and developed trajectory and Kalman filter programs for navigation and mission planning.

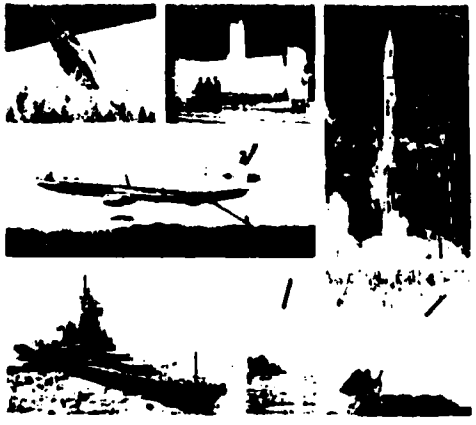
GENERAL DYNAMICS
Data Systems Division
Western center

AGENDA

- INTRODUCTION
- ADA EXPERIENCE
- ADA ACTIVITIES
- ADA PROJECTS
- CONCLUSION

INTRODUCTION

- COMMITMENT
  - — GENERAL DYNAMICS IS COMMITED TO USING ADA AS THE IMPLEMENTATION LANGUAGE OF ALL FUTURE MILITARY SYSTEMS:
    - MANAGEMENT
    - SOFTWARE ENGINEERS



GENERAL DYNAMICS
Data Systems Division
Western center

EXPERIENCE

- DATA SYSTEMS DIVISION INVOLVEMENT WITH ADA
  - IRONMAN (INTERNALLY FUNNDED REVIEW TEAM)
  - STONEMAN (KAPSE, MAPSE)
  - ATTEND MOST OF ADA WORKSHOPS
  - PRESENTED TECHNICAL PAPERS ON
    - TEST AND EVALUATION WORKSHOP
    - ADATECH
- COMPILER EVALUATION
  - NYU
  - FSU
  - TELESOFT
  - DANSK
  - ROLM
  - VAX/VMS ADA

GENERAL DYNAMICS
Data Systems Division
Western center

ADA ACTIVITIES

- PARTICIPATE ON CURRENT STATUS
  - ADATECH
  - TRADE JOURNALS
- ENCOURAGE SOFTWARE ENGINEERS TO ACTIVELY PARTICIPATE ON ADA ACTIVITIES
- SPECIAL INTEREST GROUP
- SPECIAL IRAD PROGRAMS TO STUDY
  - EXPERT SYSTEMS
  - GDADL (USING DARTS)
  - GDACS (1750A)
  - RECONFIGUTABLE ATLAS COMPILER (ADA)
  - DISTRIBUTED PROCESSING ARCHITECTURE
  - SIGNAL PROCESSING ALGORITHMS
  - MATH-LIBRARY PACKAGES (MISSION-CRITICAL SYSTEMS)
  - GRAPHICS PACKAGES
  - COMMUNICATIONS ALGORITHMS

GENERAL DYNAMICS
Data Systems Division
Western center

ADA ACTIVITIES (CONT)

- ADA AS A SOFTWARE DEVELOPMENT TOOLS
  - CROSS-REFERENCE PACKAGES
  - SOURCE LEVEL DEADLOCK DETECTION
- IMPLEMENTATION DEPENDENT FEATURES
- PACKAGE CLASSIFICATIONS
  - REUSABLE SOFTWARE
  - RAPID PROTOTYPING
- METHODOLOGIES
  - PROCESS ABSTRACTION METHODS
  - OBJECT ORIENTED DESIGN
  - STRUCTURED DESIGN TECHNIQUES
- METHODOLOGY TOOLS
  - YORDON
  - ARGUS
  - PROMOD

GENERAL DYNAMICS
Data Systems Division
Western center

PROJECTS

- TACTICAL ADA GUIDANCE (TAG)
  - DEMONSTRATE FULL ADA CAPABILITY IN TACTICAL MISSILE GUIDANCE SOFTWARE BY HARDWARE-IN-THE-LOOP TESTING IN A LABORATORY ENVIRONMENT
  - PROVIDE INFORMATION REGARDING ADA'S PERFORMANCE IN A REAL-TIME EMBEDDED COMPUTER MISSILE SYSTEM
- SCOPE
  - REDESIGN OF THE MEDIUM-RANGE AIR-TO-SURFACE (MRASM)
  - IMPLEMENTATION TEST INSTRUMENTATION CONTROLLER
  - TESTING OPERATIONAL FLIGHT SOFTWARE

GENERAL DYNAMICS
Data Systems Division
Western center

PROJECTS (CONT)

INCLUDES THE PROBLEMS TYPICAL OF AVIONICS AND GUIDANCE APPLICATIONS

- MISSILE RUNTIME ENVIRONMENT
- HARDWARE HANDLERS AND HARDWARE COMPLEXITY
- RUNTIME PERFORMANCE REQUIREMENTS

REPRESENTS AN EXTREME CASE

- MORE LOGIC INTENSIVE THAN COMPUTATIONAL
- HIGH I/O THROUGH-PUT AND INTERRUPT RATES
- ASYNCHRONOUS TASKING

ENCOURAGES EXTENSIVE USE OF ADA LANGUAGE FEATURES

• PACKAGES	• OVERLOADING	• SUBTYPING	• UNCHECKED TYPE CONVERSIONS
• ALLOCATERS	• PRIVATE TYPES	• TASKING	• TASK FAMILIES
• PRIORITIES	• ATTRIBUTES	• TERMINATE	• NAMED PARAMETERS
• EXCEPTIONS	• RENAMING	• AGGREGATES	• SEPARATE COMPILATION

CONCLUSION

- ADA IS NOT LIKE OTHER PROGRAMMING LANGUAGES
- MODERN SOFTWARE ENGINEERING PRINCIPLES
- WIDE SPECTRUM OF PROBLEM SPACE
- HIGH DEGREE UNDERSTANDABILITY OF SOURCE CODE
- TECHNIQUES FOR REUSABLE SOFTWARE
- FUNCTIONAL DECOMPOSITION METHOD

RAPID PROTOTYPING WITH REUSABLE SOURCE CODE

Elaine Frankowski, Mark Spinrad, and Paul Stachour

Software Development Technology
Computer Sciences Center
10701 Lyndale Avenue South
Bloomington, Minnesota 55420

Abstract

Software requirements are negotiated between application area engineers who are not computer experts, and software engineers who are not experts in the application area. Traditional requirement document reviews do not bridge the terminology and concept gaps between the two communities. In addition, document reviews do not exploit application experts' intuitive notions of what a system should do, notions they can express very well once they have some hands-on experience with a software prototype of the system being designed. Using prototypes to get the requirements right before implementation starts eliminates the schedule delays and cost overruns caused by redevelopment to correct incomplete, erroneous, and misunderstood requirements. Correct requirements also result in quality systems.

This paper presents an approach to constructing software prototypes from reusable software components, composed into running prototypes with various forms of software "glue," and describes the tools needed to reuse software in prototypes.

1 BACKGROUND

Software requirements are negotiated between application area engineers who are not computer experts, and software engineers who are not experts in the application area. Traditional requirement reviews ask application area experts to read documents prepared by computer experts. Such reviews do not bridge the terminology and concept gaps between the two communities. This often means delivering software that meets the negotiated requirements but does not meet the user's real needs.

Not even computer manufacturers are immune from this problem. For example, the IBM-developed SYSOUT writer, a spooling utility which met all of IBM's stated functional requirements, was replaced by either HASP or ASP on over 70% of the OS/MVT systems. IBM provided SYSOUT free to its customers; HASP and ASP are both user-developed requirements which had to be purchased or leased. Customers paid for HASP and ASP because they provided functions not available from SYSOUT as well as highly increased performance. This demonstrates vividly that even when stated functional requirements are met, real needs are not.

The STAIRS evaluation [BLAIR] exemplifies how real needs may not be met because the wrong requirements are stated in the first place. The users' real needs from the STAIRS Storage And Information Retrieval System were "the documents relevant to a particular search," that is, all and only the documents pertaining to a particular query. That need was translated into requirements for storing "the full text of all documents in a collection on a computer so that every character of every word in every sentence of every document can be located by the machine." Experiments showed that the system "retriev[ed] less than 20 percent of the documents relevant to a particular search." [BLAIR] argues a system that stores full text without index-terms is theoretically unable to meet the users' real needs, and, adding insult to injury, far costlier than a manually indexed system which meets users' needs better. A prototype full-text retrieval system for almost any large body of text would have shown that the users'

true needs could not be met by a system which stores full text only.

It is clear that written requirements by themselves do not tell the whole story, and that "behavioral feedback may reveal information that is different to discover by analysis of a static system description." [SMITH82] Prototype is one method of getting behavioral feedback, and, using that feedback, getting the requirements "right" by getting at the users' true needs.

2 WHAT IS QUALITY?

"One of the biggest sources of software problems stems from *ambiguity* in the software requirements specifications. A number of different groups—designers, testers, trainers, users—must interpret and operate with the requirements independently. If their interpretations of the requirements are different, may development and operational problems will result." [BOEHM]

To define software quality as "meets requirements" assumes that requirements are clear and unambiguous to all groups. The user may not be able to state requirements clearly: "My office procedures are too *ad hoc*. I need an office information system that will organize my transactions and will allow me to made decisions more effectively." [TAYLOR] Some requirements may be unknown to the customer: how many times have we delivered systems only to find that they can't be used until another function is added? It is clear that quality means meeting users' needs, not their stated requirements.

Current practices for systems development promote quality by using a well-defined lifecycle and verifying consistency between phases; e.g., the implementation is checked against the design. However, requirements cannot be "checked" against a previous phase; they are formulated "out of the air" in response to a perceived problem. A requirements statement is simply a "best guess" at a feasible, timely solution. To verify that this "best guess" indeed solves the problem requires a complete system implementation(s) and user feedback. Therefore uncertainties and inconsistencies in requirements may not become apparent until the system is (nearly) complete. Because this is the longest feedback loop in the entire development process, rapid prototyping of (portions of) a system early in order to "check" requirements has the potential to achieve significant cost savings and insure quality systems.

3 ELICITING USERS' TRUE NEEDS

In *The Mythical Man Month*, Fred Brooks states "Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient [as] to get it right the first time." [BROOKS] Unfortunately, economic reality makes it very hard to bid a two-system development effort on most contracts, or propose a two-system development cycle to most general managers.

Sometimes events allow systems to be developed more than once before they are pushed out the door. The Multics development project [CORBATO], because it was a long-term research and development project, enjoyed the benefits of de facto whole-system prototyping. The Multics project was fully prototyped and the hardware simulated on an existing system. The hardware was build only after the complete run-time environment of the operaing system was built., tested, performance measured, and rebuilt. Four years into the project, Multics became its own development environment, giving its developers the change to validate whether the requirements and designs they were positing really met the needs of software developers. [CORBATO] reports that " ... most areas of the [software] system were redesigned as much as a half a dozen times in as many years." This de facto prototyping is certainly one reason Multics enjoys a reputation as an excellent (modern) development environment.

Software prototyping, that is, building a partial model of critical, new, fuzzy, or otherwise questionable requirements gives software and system developers the ability to check whether a particular set of requirements are intelligible and correctly express users' needs. Because one builds only part of the eventual system, the cost and time spent to determine the correctness of the requirement are both manageable and reasonable [HOOPER].

4 PROTOTYPING OBJECTIVES AND METHODS

In the field of computer science, practitioners often misunderstand each other because their interpretations of popular buzzwords differ. "Rapid prototyping" has gained currency as a popular buzzword largely because it is one of those phrases which means many things to many people.

This section does not offer a single definition of rapid prototyping but categorizes rapid prototyping by motivation and by method, so that the meaning of the term is clear in the variety of contexts in which it is used.

There are 8 (eight) objectives that various rapid prototyping systems seek to address:

1. improved feedback to requirements analysis,
2. selection of design alternatives,
3. one-shot applications,
4. reimplementation for software maintenance [SMITH],
5. rapid response to changed requirements [TAYLOR],
6. feasibility demonstration,
7. incremental system development, and
8. experience acquisition.

The RaPIER system is intended to provide improved feedback to requirements analysis. The objective is to get the requirements *correct* before implementing the production model of the system.

By rapid prototyping methods, we mean primarily the way that prototypes are developed rapidly (prototype building methods). There has been little attention given to how prototypes are used for those prototyping objectives where decisions are made based on "playing with" with prototype. Prototype using methods are a topic of research in the RaPIER project involving human factors and feedback analysis.

There are four major methods for prototype development. These are:

1. scope reduction;
2. software reuse;
3. very high level languages; and
4. reconfigurable test harnesses.

These methods are not mutually exclusive; they are, in fact, mutually supportive, and may, in general, be used in a "mix-and-match" fashion.

Scope reduction isolates critical aspects of the system for prototyping, to reduce the prototype complexity. This method is implicit in virtually all rapid prototyping systems.

Software reuse requires two capabilities: the ability to retrieve and reuse existing software, and the ability to catalog developed software for later reuse.

Very high-level languages allow the prototype developer to implement a system quickly by providing powerful structuring and control tools (e.g., LISP) or by providing powerful application-specific primitives [MCC082].

Reconfigurable test harnesses provide well-defined interfaces for embedded software. Each component of the system may be replaced; e.g., a device simulation may be replaced by the actual device.

RaPIER is based on scope reduction and software reuse. The rest of this paper discusses reusability in the RaPIER context.

5 DEVELOPING RaPIER

5.1 Reusability

Reusing software artifacts—code, designs, algorithms, or basic concepts—promises to raise software productivity by allowing developers "to write fewer total symbols in the development of a system, and ... to spend less time in the process of organizing those symbols [BIGGER]." There are diverse approaches to reusability. [BIGGER] classifies reusable components into building blocks and patters. Building blocks such as modules in application libraries, or filters in a Unix system, are reused by composing them into larger artifacts that do more complex tasks than the individual modules do. Patterns are the underlying "domain analysis" that supports generation systems including application generators such as NOMAD or FOCUS,

problem oriented languages such as ATLAS, and program transformation systems such as PRL [BATES].

Prototyping will be useful for requirements identification if it is quick relative to the time needed for the whole development effort, and cost effective. The same productivity benefits that reusability promises in development make it a natural choice as a prototype implementation technique.

5.2 The RaPIER Reusability Strategy

The RaPIER (Rapid Prototyping to Identify End User Requirements) project has the goal of installing prototyping as a standard operating practice in the user-requirements phase of the (embedded) computer system and (embedded) computer system software development lifecylces. Honeywell develops embedded computer systems primarily under government contract, using a DoD prescribed development life cycle which mandates that requirements be frozen early. Throw-away prototypes are entirely appropriate in this milieu, where the actual system is engineered according to government standards.

Our requirements in a throw-away prototype are that it be a highly modifiable artifact, quickly and cost effectively built. Our requirement for the automated RaPIER tool is that it be ready for practice projects in three years. Reusability supports quick, inexpensive development. Reusing building blocks promotes modifiability. In addition reusing building blocks is a more seasoned technology than reusing patterns. Using seasoned technology makes it probable that we can develop the core of a prototyping system in three years.

A prototype is a program. RaPIER prototype programs will be implemented in Ada. The prototype will be described in a high-level prototype description language. That description will be mapped into an Ada program composed of modules from the software base and the "glue" statements that bind them together. The following sections present details about RaPIER's prototype construction technique and tools.

5.2.1 Reusable Code Modules

Productivity improvements through source code reuse [LANERGAN, HORWITZ] demonstrate that there are functions in many application areas which can be reused and that it pays to analyse and code such functions for reuse. Documented success with code reuse led us to choose that approach as a near term solution to the need for speedy prototype construction. Throw-away prototypes do not have stringent performance or architectural constraints, therefore the inefficiencies and awkward architectures that may result from reusing code will not make the prototypes unusable. In addition, programs constructed of reusable parts will exhibit low coupling [STEVENS], promoting modifiability. Finally, code reuse can be applied in the embedded system area faster than an approach such as program transformation [CHEATHAM] which is still in the research stage. This fact improves our chances of developing RaPIER itself quickly.

Having chosen the reusable source code approach, Ada was the natural choice of source language for several reasons:

- it contains features, such as packages and generics, that facilitate writing reusable code, that promote low coupling, and that support modern programming standards;
- its DoD sponsorship makes it highly likely that many code modules will be available for reuse within the next year or two;
- Honeywell divisions that develop systems for the DoD using Ada will generate application specific code components, providing a stock of reusable parts for the types of prototypes RaPIER is intended to construct.

Our approach, then, is to develop prototypes primarily from Ada packages that have been written according to reusability guidelines. Code developed under the guidelines, or code modified to meet the guidelines, will be easily customizable for each specific prototype. Customization will be accomplished by generic instantiation and selective replacement of bodies and separately compiles units.

There will be guidelines for the general

organization of components and lower-level rules for writing individual components. For example, a general organizational guideline might be that a function which modifies its global environment must restore that environment before terminating to allow other functions in the prototype-program to behave correctly. Another general guideline might be that an interface should be constructed with as many parameters as possible, to provide many options for customization, and all parameters should be given default values to accommodate most users.

At least initially, there will not be a reusable module to realize every behavior a prototype must exhibit. As more prototypes are developed, however, the stock of reusable modules will grow. Barstow's [BARSTOW] experience in developing a rul base for transformational programming showed that "[a]s successively harder programs were attacked during the process of rule development, fewer and fewer rules needed to be added to the knowledge base. And when a new domain ... was tackled, it became clear that much of the necessary knowledge was already covered ... the process of developing rules for a given task was considerably simplified by the fact that much of the necessary knowledge had already been codified for rules in other tasks." This experience will most likely be duplicated for reusable source code.

When a reusable module is not available there are two choices: build the module from scratch or use some function of a complete program such as a text editor, spreadsheet, graphics package to complier. New components may be "hand crafted" according to the reusability guidelines, inserted into the software base and extracted under software base control. However, for a prototype of a state of the art system, there will be many missing functions and hand crafting each one will cause a bottleneck. Therefore we are looking into providing new modules with executed specifications and/or application generators.

Application generation has been applied mainly in the business area. Honeywell's Computer Sciences Center is currently working on an application generation capability for manufacturing systems [BELL]. One of RaPIER's research areas is application generation for embedded computer system prototypes; in our 3-5 year time frame we will most likely develop only an application generation approach.

Executable specification languages and their interpreters are used today in research and advanced development [ZAVE, DAVIS, BALZER]. Every component in a prototype must have a specification, both for the prototype builder's use and for use by the software base browsing facility. In the long term, those specifications could be executable, and interpreted specifications along would be enough to realize some behaviors in a prototype. An executable specification could be incorporated into a prototype under software base operator control, just as reusable source code is incorporated.

Whole programs are another source of behavior not found in the software base. The research problem associated with entire programs is how to "glue" the selected functional behavior into the rest of the prototype. Some glue candidates are: command files, menus, parameters files, editor scripts, and I/O filters.

5.2.2 The Prototype Construction Environment

The prototype construction environment provides builders with a software data base of reusable Ada code units, a prototype construction facility, an Ada component generation capability, and access to large components from other Ada repositories such as the Simtel Library on Arpanet. The prototype construction environment allows the builder to:

- construct, modify, debug and compile individual Ada units,
- construct a prototype by combining (reused and new) units using a set of composition operators,
- exercise and debug the prototype under construction.

The prototype environment is designed for prototype construction using Ada. It offers the option of developing prototypes solely by composing available Ada units, and allows a builder to create or tailor individual units for prototypes of state of the art systems, such as mission

critical embedded computer systems, which cannot be constructed entirely from reusable software. The composition operators are explained below.

The prototype construction environment's platform is the Symbolics 3600(1) developer's workstation; it is designed to be linked in a local area network such as Ethernet. This workstation can be networked to a shared software base management system containing the reusable Ada units and ancillary information about these units, such as behavioral specifications or compiler generated symbol tables.

5.2.3 The Software Base and It Management

The software base in both the repository for RaPIER's inventory of reusable Ada software and the agent for composing that software into prototypes. The software base concepts we plan to use were developed at the University of Maryland [YEH].

The components of a software base system are [YEH]:

(1) A Module Query Language

The query language is used to access the module descriptions. It enables users to browse through the software base, and find the module(s) they need.

(2) A set of Software Base Operators

Very high level software base operators enable users to compose new application programs out of existing modules in the software base.

(3) A Prototype System Description Language (PSDL)

The PSDL is a uniform notation for describing prototypes at all levels of design. It allows a designer to concentrate on the prototype design task without the distraction of transforming the prototype description into another notation such as an Ada program. The operands in a prototype description are names of reusable components. The operators implement data, control and function abstractions. The PSDL description is mapped into a prototype program under software base control.

(4)

The catalogue contains schemas describing all the software modules in the software base. It also maintains each module's history, recording such information as versions of a module and the software base operator used to derive this module.

The software base operators allow users to compose prototype programs out of existing procedures or tasks. Each operator represents a mapping from a set of such procedures or tasks to a new application programs. The mapping is implemented by generating a procedure or task that invokes other modules according to the PSDL specification given.

This example illustrates the software base real-time operator PERIODIC. A digital filter samples an analog signal every 100 ms., computes the frequency and amplitude of the signal and outputs that signal if it is within the filtering range. Here are the steps for using the software base to generate a filtering program:

(1) Query the software base catalogue; find procedure F, a filtering function that performs the required operation.

(2) Use the software base operator PERIOD(F,100ms) to specify the execution constraints. PERIODIC(F,100ms) means that procedure F should be called periodically with period equal to 100ms.

(3) Using its pre-defined code template for PERIODIC, the software base then automatically generates the following code:

```
Procedure F
  — existing code in the software base.
end;
Procedure Main
  Loop          — period loop
    T:=get-time(); — remember the time
                     entering
                   — a period
    Call F;        — Perform the filtering
                     function
    Loop — delay loop
```

(1) trademark

```
    exit (get-time()-T=100ms);— wait until for
                              — the period time over
  end;
 end;
```

The main program generated by the software base fulfills the constraint specified by the operator. Note that the operator specification is at a high level, so that it is very close to the problem requirement. We hop to achieve rapidity of prototype construction both by reusing software and be specifying prototypes in terms of high-level software base operators.

5.2.4 Software Specifications

The key to reusing a software component is knowing its interface to the environment and its external behavior. A software component's interface is the collection of parameters and globals it uses for information exchange with its environment, the operations it offers the environment and the operations its environment provides to it. Its external behavior is the changes it causes that are visible in its environment. The keys to exploiting a software base are: (1) a query or browsing facility that allows a developer to search for a module with some desired external behavior or interface, (2) a way of specifying that behavior and interface, and (3) a specification of a module's characteristics such as execution time and space, re-entrant code, recursive implementation. This third factor may be less important in prototyping than in reusing code for "real" systems.

The software base management system provides a specification-based browsing facility. The specification of the Ada components used for prototype construction has two parts: (1) an Ada language interface specification comprising the task, package, subprogram, or generic context clause (<with-clause>) and declarations [DOD], and (2) a behavioral specification written initially as a structured comment. The software base management facility retrieves components using some keywords appearing in those structured comments. The prototype developer locates a component by comments, and then views its Ada language interface specification for construction information, such as the number and types of the parameters it requires.

There are two candidate methods for structuring comments. One is to define a hierarchical set of categories (analogous to the *Computing Reviews* Classification System [SAMMET]) from which developers choose a classification for each component entered in the software base. For example, a subprogram that sorts integer arrays in ascending order using the quicksort algorithm might be classified "SORT, ASCENDING." This convenient initial convention is, however, too limited for use in a heavily populated software base. Either too many component would be retrieved under the same classification, or the classification scheme would be under constant refinement to discriminate finely enough, necessitating constant software base restructuring.

Commenting standards and indexing techniques offer a finer level of descriptive granularity. Standards would establish that certain characteristics of each component must be presented in a natural language comment at the head of that component (for example, "sorts integer arrays in ascending order using the quicksort algorithm"). Those natural language comments could be indexed by the key work in context ("KWIC") method. The developer could browse the keywords of interest and access components whose comments were promising for the more detailed specification.

Ultimately, because real-time ECS software is complex and subtle, a component's specification should be presented in a formal language such as a first order logic. Developers would then browse the software base by composing formal specifications that the software base management system matched against the specifications of existing components. The problem of searching for equivalent, or sufficiently similar, formal specifications is still a research issue.

6 CONCLUSION

Rapid Prototyping to identify true user needs has the potential for improving the quality of software developed at Honeywell. Current rapid prototyping techniques do not offer a systematic way for developers to construct, execute, and incorporate results of prototype use into a system specification. The RaPIER system, under development at

Honeywell, will provide a rapid prototyping approach based on reusable Ada components is a software base.

This system will allow users who are not "gurus" to develop prototypes quickly and reach an agreement with users what the system really should do, so that true user needs are met.

7 BIBLIOGRAPHY

[BALZER]
Robert Balzer, N. M. Goldman, D. S. Wile. "Operational Specifications as the Basis for Rapid Prototyping," *ACM SIGSOFT Software Engineering Notes*, Vol. 7, No. 5, December 1982.

[BARSTOW]
David Barstow. "On Convergence Toward a Database of Program Transformations," *ACM Trans. on Programming Languages and Systems*, Vol. 7 No. 1, January 1985. 1-9.

[BATES]
Joseph L. Bates, Robert L. Constable. "Proofs as Programs," *ACM Trans. on Programming Languages and Systems*, Vol. 7, No. 1, January 1985, 113-136.

[BELL]
Robert Bell, Hany Atchan, Paul Stachour. "A Comparative Review of Selected Application Generators," *Honeywell Computer Sciences Center Technical Report*, in preparation.

[BIGGER]
Ted J. Biggerstaff, Alan J. Perlis. "Forward: Special Issue on Software Reusability," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 5, September 1984, 474-477.

[BLAIR]
David C. Blair, M. E. Maron. "an Evaluation of Retrieval Effectiveness for a Full-Text Document Retrieval System," Communications of the ACM, March 1985.

[BOEHM]
Barry W. Boehm, John R. Brown, Kaspar, Lipow, MacLeod, Merrit. *Characteristics of Software Quality*, 1978, xxix.

[BROOKS]
P. Brooks, Jr. *The Mythical Man-Month*, 1975.

[CHEATHAM]
Thomas E. Cheatham, Jr. "Reusability through Program Transformations," *IEEE Trans. on Software Engineering*. Vol. SE-10, No. 5, September 1984, 589-594.

[CORBATO]
F. J. Corbato, C. T. Clingen. "A Managerial View of the Multics System Development," *Research Directions in Software Technology*, 1979, 139-158.

[DAVIS]
Alan M. Davis. "Rapid Prototyping Using Executable Requirements Specifications," *ACM SIGSOFT Software Engineering Notes*, Vol. 7, No. 5, December 1982.

[DOD]
United States Department of Defense. *Reference Manual for the Ada Language*, January 1983.

[HOOPER]
James W. Hooper, Pei Hsia. Senerio-Based Prototyping for Requirements Identification, December, 1982, 89-90.

[HOROWITZ]
Ellis Horowitz, John B. Munson. "An Expansive View of Reusable Software," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 5, September 1984, 477-487.

[LANERGAN]
Robert G. Lanergan, Charles A. Grasso. "Software Engineering with Reusable Designs and Code," *IEEE Trans. on Software Engineering*, Vol. SE-10, No. 5, September 1984.

[MCCO]

G. C. McCoyd, J. R. Mitchell. "System Sketching: The Generation of Rapid Prototypes for Transaction Based Systems," *ACM SIGSOFT Software Engineering Notes*, December, 1982, 127-132.

[SAMMET]
Jean A. Sammet. "The new (1982) *Computing Reviews* Classification Scheme — Final Versions," *CACM*, Vol. 25, No. 1, January 1982.

[SMITH]
D. A. Smith. *Rapid Software Prototyping*, Technical Report Number 187, University of California — Irvine, May 1982.

[STEVENS]
W. P. Stevens, G. J. Myers, L. L. Constantine. "Structured Design," IIBM Systems Journal, Vol. 13, No. 2, 1974, 115-139.

[TAYLOR]
T. Taylor, T. A. Standish. "Initial Thoughts on Rapid Prototyping Techniques," *ACM SIGSOFT Software Engineering Notes*, December 1982, 160-166.

[YEH]
Raymond T. Yeh, Nicholas Roussopoulos, B. Chu. "Management of Reusable Software," *IEEE COMPCON*, September 1984.

[ZAVE]
Pamela Zave. "Case Study: The PAISLey Approach Applied to its own Software Tools," *Bell Laboratories Technical Report*, unpublished, undated.

MODELING A REAL-TIME EMBEDDED COMPUTER SYSTEM USING ADA: SOME PRELIMINARY RESULTS

Frank L. Friedman
Paul A.T. Wolfgang

Computer Sciences Corporation
Defense Systems Division
304 W. Route 38, Box N
Moorestown, NJ 08057

January 1985

Abstract

The Ada programming language was designed to address computer programming needs for the real-time, embedded computer system problem domain. Despite this fact, the main issues related to modeling specific real-time, embedded applications in Ada are not well understood. This paper describes some preliminary results of a CSC Internal Research and Development experiment involving the restructuring and recoding in Ada of components of a real-time memory-constrained embedded computer system.*

The system selected for study is a ship-board defensive combat system designed to coordinate radar, weaponry, communications, command, and decision functions distributed on a special-purpose multiprocessor, shared-memory architecture. The system runs on a real-time interrupt-driven, table-directed Executive.

The main thrust of our experiment is to ascertain the extent to which features of the current Executive tasking model can be mapped into Ada using the tasking, preemption, and priority support provided. The extent to which the Ada run-time environment must be augmented to support Executive features such as dynamic priorities, task/entry point priority assignment, and restrictive preemption is also under investigation.

Introduction

The executive program being studied is an extension of an earlier tactical executive program developed at CSC in the early 1970's for use in an embedded ship-board weapons system. The system is written in CMS-2 and supports multiprocessing, memory sharing, and real-time interrupt services on a special-purpose, on-board computer.

Both the weapons system and Ada address the notions of modularization and reusability, controlled data access, and the specification and use of tasks (called modules in the weapons system) as the basic system work units. The reusability concept was—major motivating factor in the design of Ada. The data abstraction facilities support the packaging of data and operations in a manner conducive to ensuring clearly defined interfaces while hiding implementation details. Ada was also designed to handle concurrency, providing an explicit tasking mechanism for representing interprocess communication and control algorithms without forcing the programmer to step outside the high level language environment.

Thus Ada has the potential of supporting much of the behavior of the weapons system and its Executive. The goal of this work is to ascertain the extent to which features of the current Executive tasking model can be mapped into Ada using the tasking support provided by Ada. The extent to which the Ada run-time environment must be augmented to support specialized features of the Executive such as changing priorities, task/entry point priority assignment, and preemption restrictions is also being studied. Our

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

focus is to take a more detailed look at the Executive tasking model, identify the major problem areas in representing the Executive in Ada, and suggest possible solutions, including perhaps, changes to the Executive which would allow it to better fit the Ada model without sacrificing any of the original functional requirements.

In the first phase of this experiment, which is the phase discussed in this paper, we have limited the scope of our study to an analysis of the scheduling and dispatching functions of the ATES Executive, and the relationship between these functions and a small subset of the weapons system applications modules. System functions related to initialization, periodic rescheduling, priority assignment, preemption, common data access, and module communication are currently being studied and recoded in skeletal form in Ada.

An Overview of the Executive

The Executive is divided into two major functional units, an executive service program which provides the nucleus of executive services to control the CPU, input and output channels, and memory; and a dependent executive program, which provides user dependent services for a particular tactical application. These services include system initialization, interrupt handling, scheduling and dispatching, memory management, message processing, input/output and device processing, error processing and recovery, intercomputer communications, common service routines (such as for mathematical functions), and performance measurement and debugging support.

There are currently four tactical applications elements serviced by the Executive. The basic work unit of these elements is the single-function tactical applications module. These modules perform functions related to carrying out the system mission, including the acquisition, processing, evaluation, and display of tactical data.

The Executive Tasking Model

Executive applications modules may be scheduled for processing at any one of at most seven entry points, one for each of the primary processing activities performed by the module. The entry points for all modules are the same, one each for initialization, message processing, error processing, successor processing, buffer complete functions, channel complete functions, and periodic processing. For most modules, however, one or more of the seven entries will be undefined (null) and therefore not schedulable.

The scheduling and eventual execution (dispatching) of all module entry points is managed through a Priority Schedule Queue (PSQ). With but one or two exceptions, entry points to be executed must first be entered into the PSQ ordered by priority. The Executive uses a two-tier priority scheme with a preemption priority (the major priority) governing execution preemption, and a scheduling priority (the minor priority) which is used together with the preemption priority for maintaining the order in the PSQ. A schedulable module entry point may be dispatched (subject to preemption restrictions) when it reaches the front priorities associated with a module entry point to be altered via requests to the Executive.

The scheduling of a module's initialization, error processing, and successor entry points may be done by other application modules. All other entries may be scheduled only via requests to the executive. In addition, module initialization is scheduled directly by the Executive whenever system initialization is required.

Periodic entries are scheduled directly by the Executive each time a countdown clock interrupt occurs. Schedulable periodic entries are stored in a separate periodic queue (PQ) ordered by "next-time-to-go". At each countdown clock interrupt, all ready-to-go periodics are deleted from the periodic queue and entered into the PSQ (subject to conditions concerning duplicate periodic entries in the PSQ). If a periodic entry is "reschedulable", a new time-to-go is determined and the entry is reinserted into the periodic queue. The reschedulable and multischeduling status of a module's periodic entrance may be changed via requests to the executive.

Preemption of an executing module entry point may occur only when an entry point of another module with a higher preemption priority has reached the head of the PSQ, and the currently executing module entry is in a preemptable state. Preemption is not allowed if the preemption entry is from the same module as the currently executing entry, or if it is from a module with a previously activated but preempted entry waiting for completion.

Application Module Interfaces

Application modules communicate with the Executive in basically four ways:

(1) via the use of ESR's;

(2) via the use of common (system data);

(3) via information packets returned by the Executive;

(4) via system temporary storage.

Modules may interface with other modules and with the Executive via system common data. There are various levels of common data, such as the common data base for an applications module, and the more global common system data base. Modules must follow special conventions in accessing an application module also may communicate with the Executive by issuing requests for executive service (ESR's). There are two types of ESR's that can be issued: a request for settypes of ESR's, the require module upon service completion.

Modeling Scheduling, Dispatching, and Communication in Ada

The experiment described in this paper involves the construction in Ada of skeletal versions of several applications modules from the radar system element and 20 or so ESR handlers. These components are being combined into a small mock-up designed to simulate the scheduling and dispatching functions of the Executive, as related to the applications modules and their ESR requests. The project was begun in November, 1984, and is still in the midst of an iterative, step-wise design and implementation process.

An overview of the skeletal system as it is currently envisioned is shown in Figure 1. The main components of the system are described below.

(1) The Executive Task (EXEC TASK) which serves as the parent for all ESR processing modules, and which contains an entry point for each ESR and interrupt that is being supported. This task operates at the next-to-the-highest priority level in the system.

(2) The External Environments Task (EXTENVRN) which provides a very coarse simulation of interrupts that are external to the portion of the Executive that is currently being modeled. This task provides a mechanism for the processing of periodic entry points (removal from the periodic queue and insertion into the schedule queue), and for the random insertion into the PSQ of other schedulable entries. The External Environments Task runs at the highest system priority level.

(3) The Initialization Procedure (EXEC INIT) which implements a part of the Executive initialization, including the initialization of the periodic and priority schedule queues and the activation of the initialization entry points of the applications modules.

(4) The Schedule Package (SCHEDULE) which manages the queing and dispatching of all schedulable module entry points (through the PSQ).

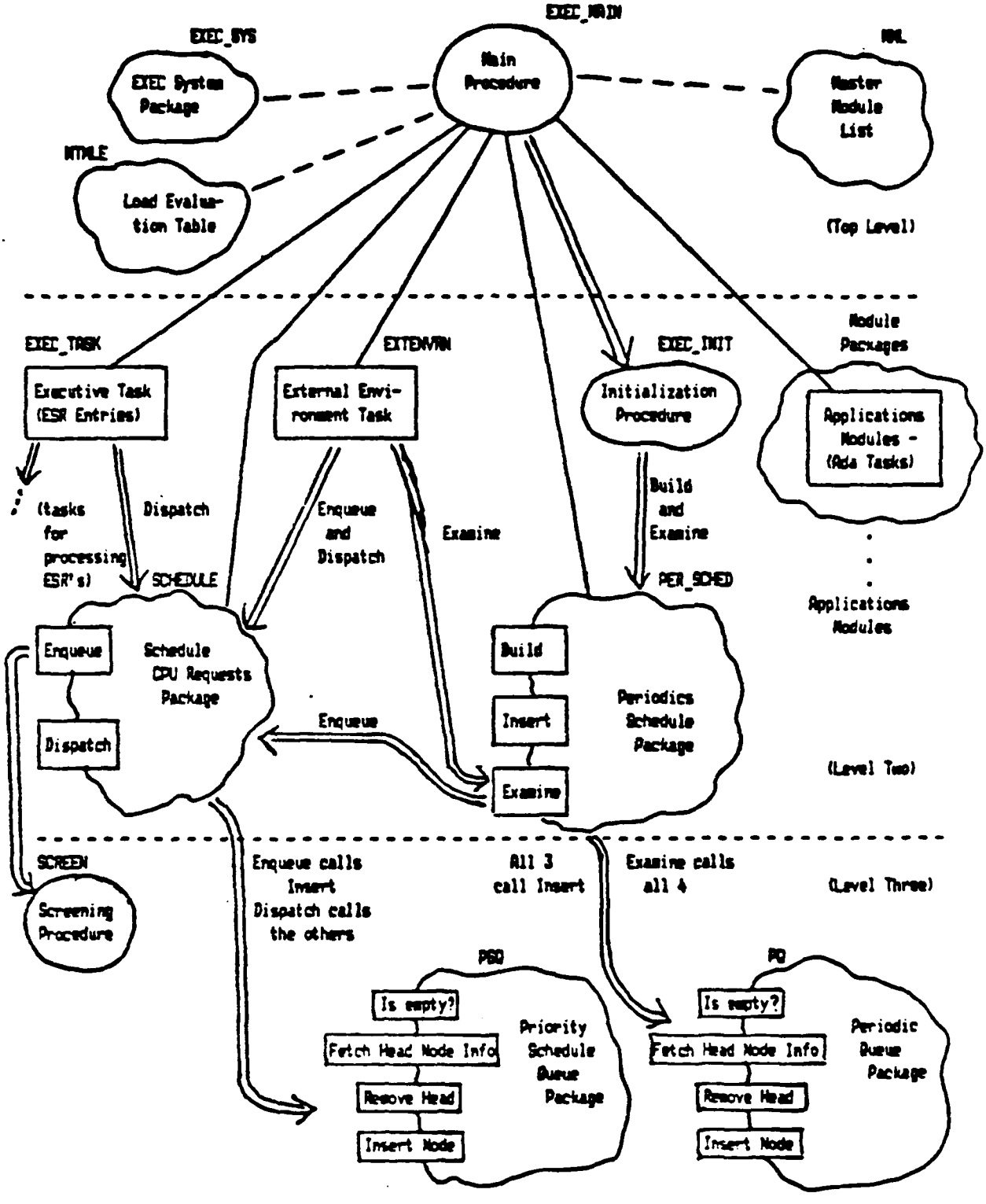
(5) The Periodic Scheduler Package (PER SCHED) which manages the initial build and subsequent insertions and deletions of all periodic entries (through the periodic queue).

(6) The Applications Module Packages each of which models one radar system applications module. Each module is actually modeled using two packages: a data package, containing all of the global types, constants, and data structures processed by the module, and a task package, containing a task with seven entry points, which models the executable portion of a module. These tasks operate at the full range of priorities below the top two system priority levels.

Figure 1 also shows several data packages (EXEC_SYS,MML, and MTMLE) that have been implemented as part of the communications interface for the system skeleton. While we have used packets to communicate information between the Executive and the applications modules, we have retained the table structure of the Executive as the primary source of information for event tracing and system decisions. Each table required in the system mock-up, such as the Master Moreseparate packages to be accessed only as needed through the use of query or update procedures which are also a part of the package. Only the procedure interfaces are visible outside the package, and these are accessible only to selected system components.

Preliminary Findings

The design shown in Figure 1 reflects a very deliberate effort to begin this project by cod-



(EXEC_SYS and MRL are "sithed" in each element (except PSQ and PQ) and are not shown below the top level.)

Figure 1: An Overview of the Executive/Applications Modules System Skeleton

ing in Ada a mock-up of the Executive which mirrors as closely as possible the current features of the system. With the exception of dynamic priorities, which clearly cannot be handled within the Ada tasking model, we have tried not to deviate in any substantial way from the original major design decisions for the weapons system.

Within this framework, we have already encountered a number of difficulties in attempting to model the Executive tasking scheme in Ada. One of the major problems was caused by the fact that Ada does not permit the association of priorities with individual task entries, but only with the task itself. Since Ada allows task invocation via explicit references to task entries it seems inconsistent not to allow these entries to be assigned their own priorities.

One possible approach to solving this problem is to represent each applications module as seven Ada tasks, one for each entry point. While this appears to be a workable solution, it does not provide for a very accurate view of the current Executive model, even if these tasks are packaged in a single unit. This solution also makes it slightly more difficult for our system mock-up to correctly model the preemption restrictions and multiple periodic scheduling rules of the Executive.

We have chosen instead to retain the one-task/seven-entry model in Ada, but we have added an additional level of tasking (see Figure 2). Each schedulable entry in a module task consists of an accept-do block which contains all code required for saving entry parameters and a statement that references yet another task, which actually carries out the work to be done at that entry. This task is assigned a priority, P, that reflects its relative level of importance in the current Executive. This solution is illustrated in Figure 2.

In this model, whenever a task entry point rendezvous occurs, the code in the accept-do block will execute with the priority of the caller, which is always at a higher priority. Since all task entries are referenced from the scheduler, and the scheduler is a procedure in a package that is nested in EXEC MAIN (see Figure 1), the rendezvous, including the reference to the entry point task, will execute at the priority of the Executives. Once the entry point task has copied its parameters, it leaves its accept-do block, and is ready to continue executing at the same relative priority, P, that it was assigned in the original weapons system.

The Ada priority scheme has caused other minor complications as well. The Executive uses a two-tier priority scheme with potentially 66 different levels of urgency. Ada, on the other hand, supports only a single-tier priority scheme. Furthermore, our version of Ada (NYU Ada/Ed) allows only ten levels of urgency. The single-tier priority problem is easily handled through a simple function that can be used to map the Executive two-tier priorities to the single-level of Ada (and vice-versa).

The levels of priority limitation is not a serious problem for our small system mock-up. We are assuming that compilers targeted to embedded computer systems will eventually support some number of priority levels that is consistent with the functional requirements of tactical systems. For the time being, we are simply mapping Executive priorities onto the ten levels supported by Ada/Ed.

The Executive preemption scheme has been another source of difficulty in the design of our Ada mock-up. The Executive allows one module entry to preempt another only if the executing module is in preemptable mode, and then only if:

- the preemption (major) priority of the waiting module entry is higher than the preemption priority of the executing module;
- the module id of the waiting module entry does not match the id of the executing module nor of any currently preempted module.

The latter condition requires that we keep track of the module id for all module entries that have been preempted and not yet completed. This is a definite step in the wrong direction, since one of the primary goals in this project is to use the Ada priority/preemption model as much as possible to keep track of the environment and status of preempted tasks, so that the entire scheme would be transparent to the programmer. The amount of added user code involved in tracking reeempted modules has not yet been completely determined.

Summary and Conclusions

At this point, it is clear that Ada has the potential of supporting much, but not all of the underlying behavior of the Executive program.

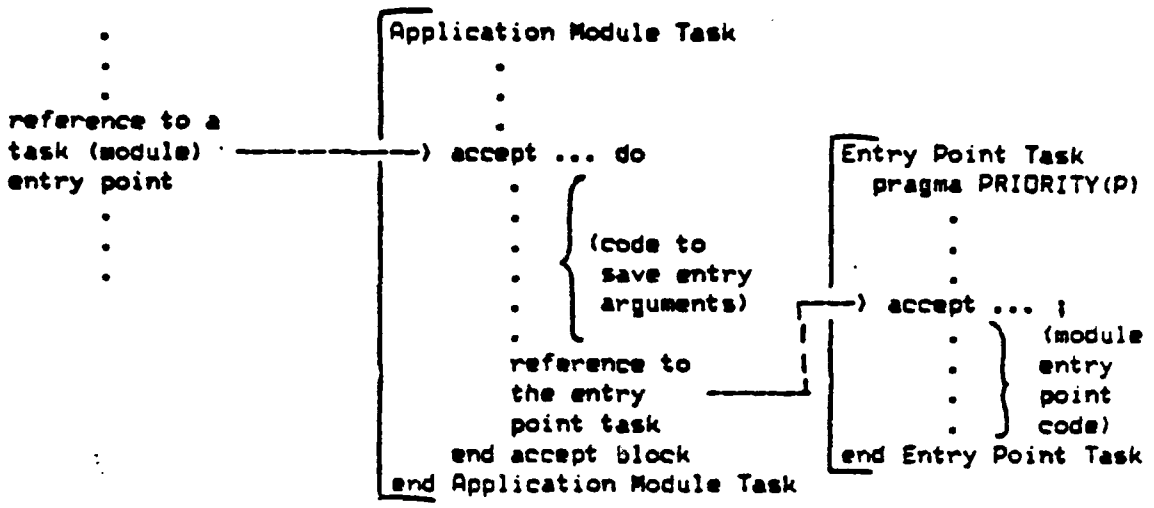


Figure 2: Tasking Schematic for Entry Point References in Application Module Tasks

Since we are currently examining only a small part of this program, it is certain that we have uncovered only a few of the major problems associated with writing real-time systems in Ada. The planned evaluation of additional weapons system components for input/output processing interrupt, error, and message handling, common service routines and other implementation dependent issues will most likely reveal additional problems.

It might be argued that most if not all of these problems are unique to the Executive being studied, specifically, to its original design. Furthermore, it is possible that all of them can be eliminated by redesigning the system for eventual recoding in Ada. At this point, we are not convinced. We believe that the Ada tasking model, including the preemption/priority scheme and the termination mechanism, may not be sufficient to support a typical real-time tactical embedded computer system. We are concerned too, about the efficiency with which Ada run-time systems will support recoded Executive functions and about the portability of such systems. Others (see (4) and (5) seem to have similar reservations. Unfortunately, an Ada environment appropriate to realistic studies in these areas is not likely to be available in the near future.

These issues aside, discussions with systems experts at CSC point to the need for a complete review of the system design, both with respect to Ada and to prospective changes in hardware. If nothing else, such a review might show that during the fifteen year evolution of the system, some features of the tasking model have fallen into disuse, and that what is left can be far more easily modeled in Ada, perhaps with adequate time and space efficiency.

Bibliography

(1) Fisher, Gerald A. et.al., "Developing an Ada to CMS-2 Translator", Draft Copy, Computer Sciences Corporation, San Diego, CA, May 1984.

(2) Gillman, Richard, Crocker, Stephen D. and Taylor, Craig, "Translation of CMS-2 Programs to Ada", Working Paper ISI/WP-19, USC Information Sciences Institute, Marina Del Rey, CA, February 1980.

(3) Intermetrics, "An Analysis of the Problems Associated with the CMS-2 to Ada Transition", Report 0967-LP-598-9720, prepared by Intermetrics Incorporated, Bethesda, MD, for the Naval Sea Systems Command, Washington, D.C, June 1983.

(4) NAVSEA, "A Plan for the Aegis Transition to Ada", Report 0967-LP-598-9730, prepared for the Naval Sea Systems Command, Washington, D.C., June 1983.

(5) Payton, Teri F., and Horton, Michael J., "Study Report on the Transition of RNTDS to Ada", Report FR)A)-3020, prepared by Systems Development Corporation, Paoli, PA, for the Naval Sea Systems Command (Code 06L3), Washington, D.C., 31 July 1983.

(6) SofTech, "CMS-2 to Ada Transition Plan", Report 0967-LP-598-9750, prepared by SofTech Incorporated, Falls Church, VA, for the Naval Sea Systems Command, Washington, D.C., August 1983.

RESUME

FRANK L. FRIEDMAN

Senior Computer Specialist
Computer Sciences Corporation

EDUCATION

BA in Mathematics, Antioch College, Yellow Springs, Ohio
MS Numerical Science, Johns Hopkins University
MA & Ph.D in Computer Science, Purdue University
Completed CSC courses in CMS-2, AEGIS Orientation, and Ada

SUMMARY:

Dr. Friedman taught Computer Science at Temple University from 1974 to June 1984. During this time he supervised numerous independent study and project efforts covering a wide range of topics, including:

(1) Ph.D research for Mr. Daniel T. Joyce, involving a study of abstraction - based design and programming methodologics and their impact upon the software life cycle.

(2) Three Masters Degree projects involving the development of programs for measuring size, control flow, and data accessibility metrics in Pascal programs.

(3) A project involving the design and implementation of a program/data encapsulation facility for Pascal.

(4) The design and implementation of a portable pre-processor for extended FORTRAN. The pre-processor was used on over one dozen mainframe computers at over 100 university and industrial sites in the United States and abroad.

CURRENT INTERESTS

Since coming to CSC in June 1984, Dr. Friedman has been involved in studies of approaches to building system software using Ada, primarily in a real-time embedded systems environment. This work thus far has touched on areas such as:

(1) the use of Ada as a Program Design Language

(2) the applicability of various design methodologies to the real-time Ada environment and to the use of Ada in general

(3) the identification of features of programming support environments critical to the proper use of Ada both at the design and the implementation stages of system development

(4) programming-in-the-large versus programming-in-the-small

(5) system component adaptability and reusability.

The goals of this research are:

(1) to identify and classify common, low-level components that form the kernel of a real-time embedded system;

(2) to represent these components in Ada as parameterized primitives for use as the building blocks for higher level components;

(3) to identify and classify architectural level paradigms common to embedded systems;

(4) to represent these paradigms in Ada as higher level design templates providing a canonical, PDL-like form for both the architectural and detail level components of a system.

Dr. Friedman is currently working on a project to recode a portion of the AEGIS Tactical Executive System and Applications Modules in Ada.

PAST PROFESSIONAL EXPERIENCE

(1) Systems Analyst, Computer Group, USNSRDC, Annapolis, MD, 1965-67

(2) Instructor, Mathematics and Director, Computer Center, Goucher College, Baltimore, MD, 1967-70.

ORIENTATION

GOAL

TO GAIN A BETTER UNDERSTANDING OF MAJOR PROBLEMS ASSOCIATED WITH MODELING A REAL-TIME EMBEDDED COMPUTER SYSTEM IN Ada

SPECIFICALLY, TO ASCERTAIN:

- THE EXTENT TO WHICH THE CURRENT EXECUTIVE TASKING MODEL CAN BE MAPPED INTO Ada USING Ada's SUPPORT FOR:
  - — TASKING
  - — PREEMPTION
  - — PRIORITY
- THE EXTENT TO WHICH THE Ada RUN-TIME ENVIRONMENT MUST BE AUGMENTED TO SUPPORT SUCH EXECUTIVE FEATURES AS:
  - — DYNAMIC PRIORITIES
  - — INDIVIDUAL TASK/ENTRY POINT PRIORITIES
  - — RESTRICTIVE PREEMPTION

5029-2

SELECTED EXAMPLE

SHIPBOARD DEFENSIVE COMBAT SYSTEM

EXECUTIVE AND FOUR TACTICAL APPLICATIONS ELEMENTS

COORDINATION FOR:

RADAR
WEAPONS CONTROL
COMMUNICATIONS
COMMAND AND DECISION

EXECUTIVE:

REAL-TIME
EVENT-DRIVEN
TABLE-DIRECTED

TWO COMPONENTS:

1. NUCLEUS — PROVIDES SERVICES TO CONTROL CPU, I/O, AND MEMORY (HANDLES INTERRUPTS, MESSAGES, INTERCOMPUTER COMMUNICATION, SCHEDULING, AND DISPATCHING)

2. DEPENDENT EXECUTIVE — SERVICES PARTICULAR TACTICAL APPLICATIONS

TACTICAL APPLICATIONS MODULES

1. SEVEN INDEPENDENTLY SCHEDULABLE ENTRY POINTS, EACH WITH ITS OWN DYNAMIC PRIORITY

2. MODULES COMMUNICATE VIA
   - EXECUTIVE SERVICES REQUESTS
   - COMMON SYSTEM DATA
   - INFORMATION PACKETS
   - SYSTEM TEMPORARY STORAGE

SYSTEM IS CODED IN CMS-2 (MOSTLY)

HIGHLY MODULARIZED
SMALL COMPONENTS
RIGOROUS NAMING AND STYLE CONVENTIONS

SPECIAL-PURPOSE MULTIPROCESSOR WITH SHARED MEMORY

5029-3

SCOPE OF STUDY

ANALYZED

- SCHEDULING AND DISPATCHING FUNCTIONS OF THE EXECUTIVE
- RELATIONSHIP BETWEEN THESE FUNCTIONS AND A SMALL SUBSET OF RADAR SYSTEM APPLICATIONS MODULES

CURRENT SYSTEM FEATURES MODELED

- SCHEDULING AND DISPATCHING
  - PRIORITY SCHEDULE QUEUE (PSQ)
  - PSQ BY PRIORITY
  - DISPATCHING FROM FRONT OF PSQ
- TWO-TIER PRIORITY SCHEME
  - MAJOR PRIORITY — GOVERNS PREEMPTION
  - MINOR PRIORITY — GOVERNS SCHEDULING (TOGETHER WITH MAJOR PRIORITY)
- SEPARATE PRIORITIES FOR EACH ENTRY POINT
- PRIORITIES CHANGED VIA ESRs
- RESTRICTED PREEMPTION
  - HIGHER PRIORITY MODULE MUST BE AT THE HEAD OF PSQ
  - CURRENT MODULE MUST BE PREEMPTIBLE
  - PREEMPTION NOT ALLOWED IF ID OF PREEMPTING MODULE
    1. IS THE SAME AS THE ID OF A CURRENTLY EXECUTING MODULE, OR
    2. IS THE SAME AS THE ID OF ANY SUSPENDED MODULE
- PERIODIC ENTRY POINTS
  - SCHEDULED DIRECTLY BY THE EXECUTIVE WHEN THE COUNTDOWN CLOCK INTERRUPT OCCURS
  - SCHEDULABLE PERIODICS STORED IN PERIODIC QUEUE (PQ) ORDERED BY "NEXT-TIME-TO-GO"
  - AT CLOCK INTERRUPT, READY-TO-GO PERIODICS PUT IN PSQ AND DELETED FROM PQ; SOME MAY BE REINSERTED INTO PQ

5029-4

THE Ada MODEL

PURPOSE

- BUILDING A SMALL, SKELETAL SYSTEM MOCKUP TO SIMULATE THE SCHEDULING AND DISPATCHING FUNCTIONS OF THE ORIGINAL SYSTEM

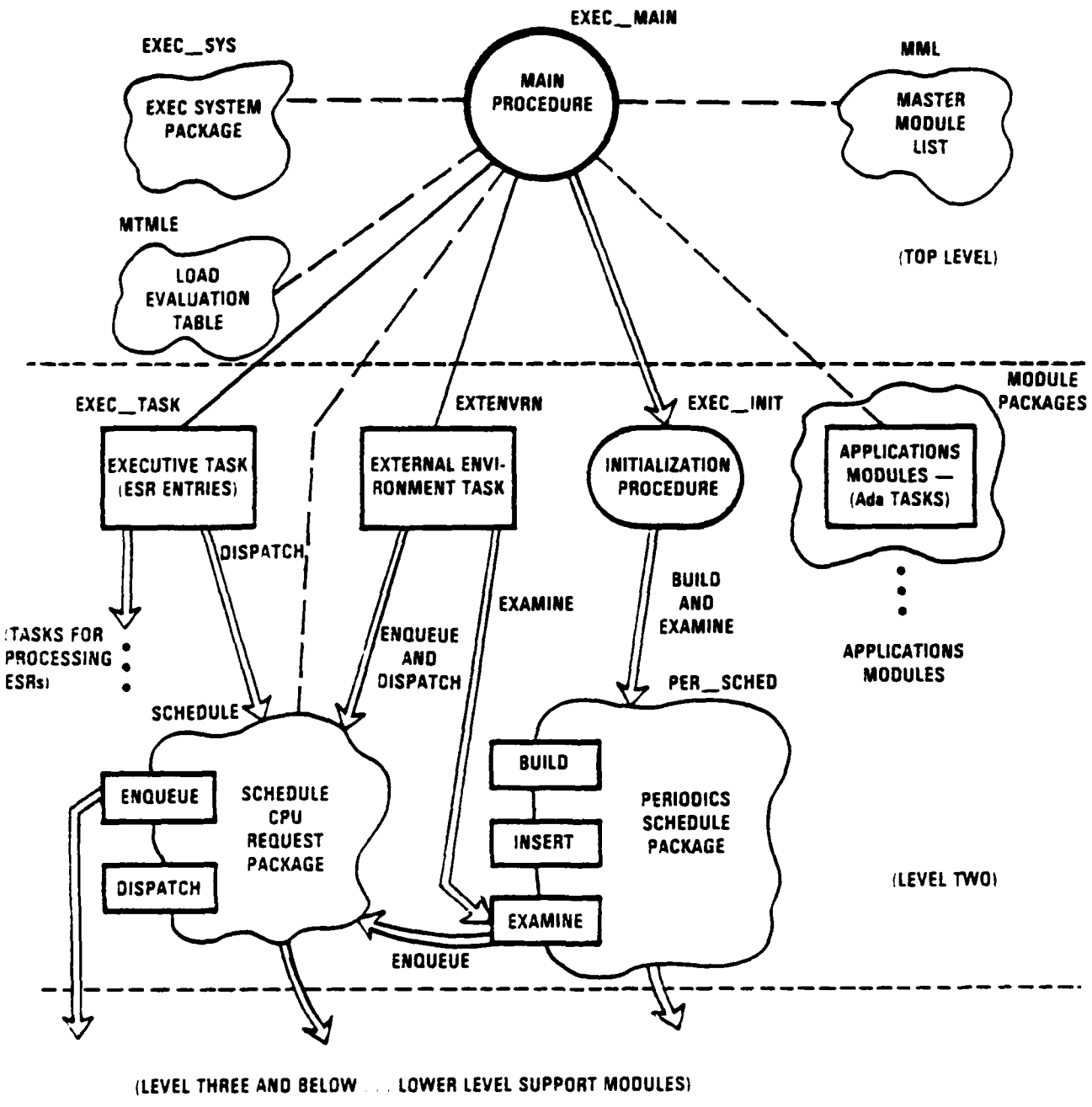
SYSTEM OVERVIEW

- EXTERNAL ENVIRONMENTS TASK — COARSE SIMULATION OF EXTERNAL INTERRUPTS
- INITIALIZATION PROCEDURE — HANDLES INITIALIZATION, INCLUDING THAT OF THE PSQ AND PQ
- EXECUTIVE TASK — PARENT FOR ESR HANDLERS
- SCHEDULE PACKAGE — MANAGES QUEUING AND SCHEDULING OF ALL MODULE ENTRY POINTS
- PERIODIC SCHEDULER PACKAGE — MANAGES PQ
- APPLICATIONS MODULE PACKAGES — EACH PACKAGE PAIR MODELS ONE RADAR SYSTEM APPLICATIONS MODULE
- DATA PACKAGES — TABLE STRUCTURE OF THE EXECUTIVE RETAINED AS THE PRIMARY SOURCE OF INFORMATION FOR EVENT TRACING AND SYSTEM DECISIONS; EACH TABLE IS IN A SEPARATE PACKAGE THAT MANAGES QUERIES AND UPDATES

5029-5

THE Ada MODEL (Cont'd)

SYSTEM OVERVIEW (TOP LEVELS)



5029-6

THE Ada MODEL (Cont'd)

CURRENT STATE

- EXTERNAL ENVIRONMENTS PROCESS
  - A TASK
    - SIMULATES SELECTED SOFTWARE INTERRUPTS
    - CONTAINS A DELAY
    - PERIODICALLY (UPON AWAKENING)
      1. MOVES READY-TO-GO PERIODICS FROM PQ TO PSQ
      2. REINSERTS SELECTED PERIODICS INTO PQ
      3. RANDOMLY INSERTS SELECTED MESSAGE, SUCCESSOR, BUFFER, AND CHANNEL COMPLETE ENTRY POINTS INTO THE PSQ
      4. CALLS DISPATCHER
      5. GOES TO SLEEP
    - RUNS AT HIGHEST PRIORITY: PRIORITY'LAST
- INITIALIZATION PROCESS
  - A PROCEDURE
    - PERFORMS SYSTEM INITIALIZATION (WORK THAT MUST BE DONE BEFORE THE SYSTEM BECOMES OPERATIONAL)
    - ACTIVATES INITIALIZATION ENTRIES OF APPLICATIONS MODULES
    - INITIALIZES PQ AND PSQ
    - RUNS AT HIGHEST PRIORITY: PRIORITY'LAST

5029-7

THE Ada MODEL (Cont'd)

CURRENT STATE (Cont'd)

- EXECUTIVE PROCESS
  - A TASK
    - PROCESS IS CYCLIC
    - INCLUDES LOOP/SELECT CONTAINING ENTRIES (ACCEPTS) FOR 20 ESRs
      1. MODULE TERMINATION/EXIT (WITH/WITHOUT RETURN)
      2. SUCCESSOR RETURN
      3. PERIODIC SCHEDULING
      4. PERIOD AND PHASE CHANGES
      5. CHANGE PREEMPT STATE
      6. CHANGE PRIORITY
    - RUNS AT PRIORITY'LAST - 1
- PACKAGES FOR MANAGING PSQ AND PQ
  - INCLUDES PROCEDURES FOR SCHEDULING MODULE ENTRIES AND DISPATCHING SCHEDULED ENTRIES TO THE CPU (PSQ ENTRY AND REMOVAL)
  - ALSO HANDLES REMOVAL AND INSERTION OF PERIODICS IN PERIODIC QUEUE

5029-8

THE Ada MODEL (Cont'd)

CURRENT STATE (Cont'd)

- APPLICATIONS MODULE PACKAGES
  - — PACKAGED IN PAIRS: EACH PAIR MODELS ONE RADAR SYSTEM APPLICATIONS MODULE
    1. GLOBAL DATA PACKAGE — DATA TYPES, CONSTANTS, AND DATA OBJECTS REFERENCED BY THE MODULE
    2. TASK PACKAGE — SINGLE TASK
       a. MODULES EXECUTABLE CODE
       b. LOOP/SELECT WITH SEVEN ENTRIES (ACCEPT/DO)
       c. FOR EACH DEFINED ENTRY:
          (1) PERFORMS HOUSEKEEPING FOR DATA COMMUNICATION (DEFINITION OF LOCAL DATA OBJECTS)
          (2) CALLS SEPARATE TASK TO PERFORM WORK (TASK HAS ITS OWN PRIORITY)

(ONLY ONE APPLICATIONS MODULE IMPLEMENTED THUS FAR)

5029-9

PRELIMINARY RESULTS

- DELIBERATE EFFORT TO CODE MOCKUP OF EXECUTIVE THAT MIRRORS AS CLOSELY AS POSSIBLE THE CURRENT FEATURES
- STAYED WITHIN THE FRAMEWORK OF Ada TASKING MODEL
  - EXCEPTIONS:
    1. DYNAMIC PRIORITIES (NOT HANDLED)
    2. PREEMPTION RESTRICTIONS (BUILT ON TOP OF Ada TASKING MODEL)
    3. SCHEDULING ENTRIES (BUILT ON TOP OF Ada TASKING MODEL)
- PROBLEMS CAUSED BY SPECIAL FEATURES OF THE EXECUTIVE
  - ASSIGNING DIFFERENT PRIORITIES TO ENTRY POINTS IN THE SAME MODULE

    SOLUTIONS:
    1. EXECUTABLE CODE AT EACH ENTRY POINT WRITTEN AS A SEPARATE TASK
    2. REPRESENT EACH MODULE AS A PACKAGE OF SEVEN TASKS RATHER THAN ONE TASK WITH SEVEN ENTRIES
  - TWO-TIER PRIORITY SCHEME WITH 66 PRIORITY LEVELS

    SOLUTION: TWO-TIER SCHEME MAPPED TO A SINGLE-TIER, 66-PRIORITY PROBLEM NOT SOLVED — AVOIDED
  - PREEMPTION RESTRICTIONS

    SOLUTION: ADDITIONAL HOUSEKEEPING, e.g., GLOBAL NONPREEMPT STATUS FLAG TRACKING OF CURRENT AND PREEMPTED MODULE IDs

5029-10

CONCLUSIONS

- Ada HAS THE POTENTIAL FOR SUPPORTING MOST OF THE UNDERLYING BEHAVIOR OF THE EXECUTIVE
- MANY OTHER PROBLEMS ARE LIKELY TO SURFACE AS THE MOCKUP IS EXPANDED INTO AREAS SUCH AS:
  - I/O PROCESSING
  - INTERRUPT HANDLING
  - ERROR HANDLING
  - MESSAGE HANDLING
  - COMMON SERVICE ROUTINES
  - OTHER IMPLEMENTATION-DEPENDENT CODE
- PROBLEMS ENCOUNTERED ARE NOT UNIQUE TO THE EXECUTIVE WE SELECTED. PROBABLY CANNOT BE TOTALLY ELIMINATED THROUGH REDESIGN TO BETTER FIT THE Ada MODEL:
  - Ada TASKING MODEL TOO RESTRICTIVE
  - Ada RUN-TIME SYSTEM TOO INEFFICIENT

Ada PROBABLY WILL NOT ADEQUATELY SUPPORT ANY TYPICAL REAL-TIME TACTICAL EMBEDDED COMPUTER SYSTEM WITHOUT SOME AUGMENTING

- NONETHELESS, THERE IS A NEED FOR A COMPLETE DESIGN REVIEW OF THE CURRENT EXECUTIVE. REVIEW SHOULD REVEAL:
  - CURRENTLY UNUSED FEATURES OF THE CURRENT TASKING MODEL
  - REMAINING FEATURES MAY BE FAR MORE EASILY MAPPED INTO Ada, PERHAPS EVEN WITH ADEQUATE SPACE AND TIME EFFICIENCY

5029-11

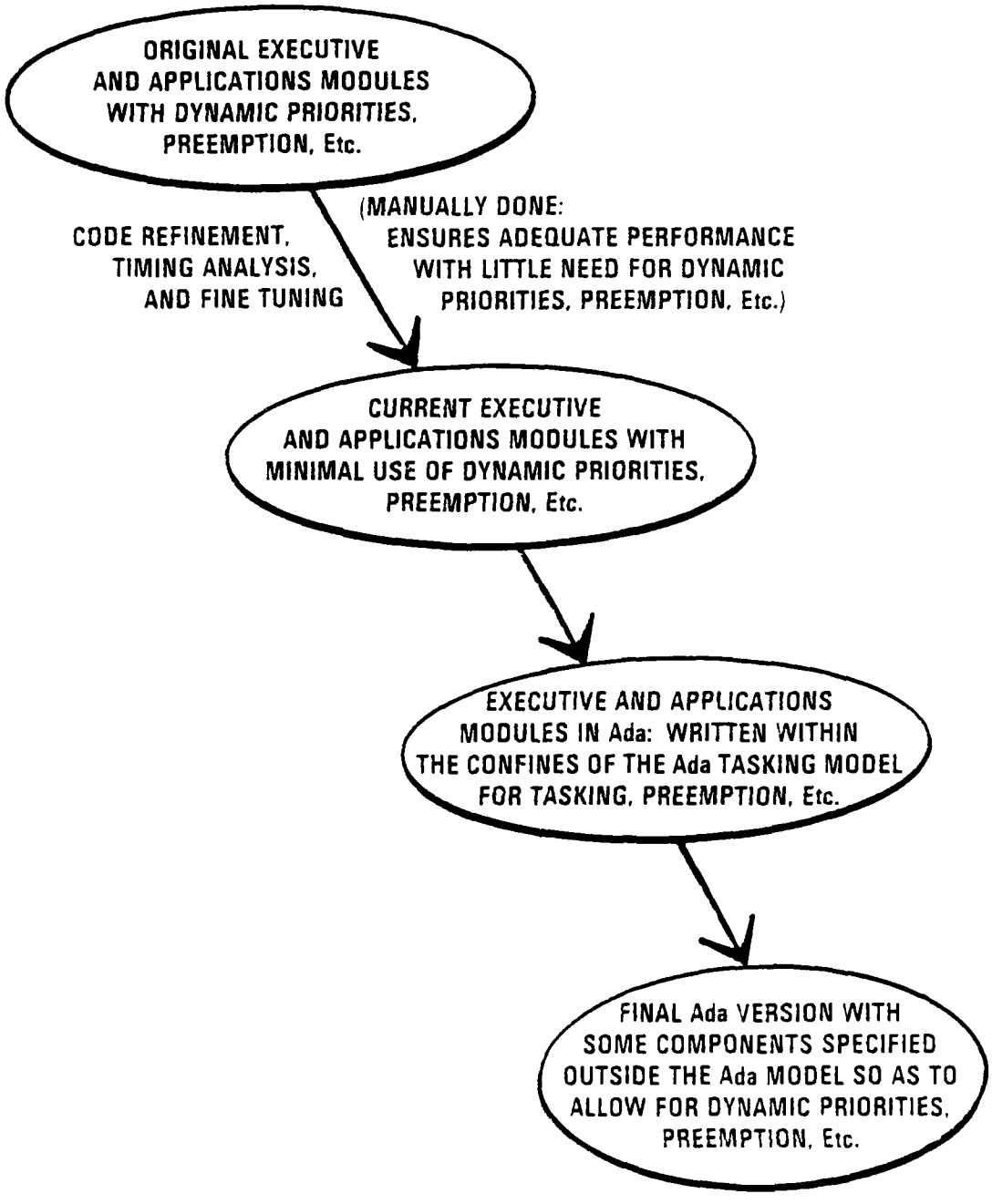
CONCLUSIONS (Cont'd)

CONCEPT VALIDATION AND REUSABILITY

- APPROACH SEEMS IDEAL FOR ANY EVALUATION OF Ada
- Ada (WITH PERHAPS SOME GRAPHIC AIDS) SEEMS IDEAL FOR THE DESIGN AND CODING OF SYSTEM MOCKUPS
  - DESIGN AND BUILD OUTER (HIGHER LEVEL) STRUCTURES IN Ada
  - PRODUCE REUSABLE TEMPLATES WITHIN THE SAME SYSTEM (AS FOR ESR HANDLERS AND APPLICATIONS MODULES) FOR USE IN OTHER SYSTEMS
  - DELAY IMPLEMENTATION OF LOWER LEVEL DETAILS UNTIL OUTER ARCHITECTURAL CONCEPTS HAVE BEEN VALIDATED
  - CHOOSE AN APPROPRIATE IMPLEMENTATION OF ACTUAL PROCESSING PROCEDURES (CAN STAY IN CMS-2 WITH REWRITES IN Ada ON AN AS-NEEDED BASIS)

5029-12

FINAL COMMENTS



5029-13

A PROCESS VIEW FOR REAL-TIME SYSTEMS

Reusable Components Based on Separating Knowledge and Control

Nancy Giddings

Honeywell Systems and Research Center
2600 Ridgway Parkway
Minneapolis, Minnesota 55413

December 11, 1984

ABSTRACT

Real-time, embedded software systems are becoming increasingly complex, and correspondingly more difficult to maintain and validate. One possible software architecture which offers a possibility for reuse of software components is via the separation of knowledge and control as used in some rule-based systems. The issues surrounding this choice, as well as a comparison of this approach to other aggressive approaches such as functional programming and automated program transformation are given.

1.0 Introduction

The embedded software community is challenged with software requirements which:

- Imply rapid and aggressive increases in the complexity of embedded software performance.
- Call for software which is considerably easier to modify than that which results from contemporary software development.

There are several possible approaches to dealing with these challenges. The ones with the greatest payoff potential call for radical changes in the software development process. Most result in shorter turnaround to first implementation and/or automation of the reimplementation process.

This paper advocates "borrowing" a software view from the AI community for general use in embedded systems that supports a level of reusable software.

2.0 Background and Context

The line of research described in this paper was begun in 1981 (SILV81). The initial investigation was in the general "software environments" topic area, and has since branched into three subtopic thrusts--traditional (Ada) software engineering environments, target execution environments, and AI engineering environments.

From the outset, a major tenet of our research is that the "power" of a software environment or particular tools is inversely proportional to its generality of application. For us, this has meant that we have been very careful about defining the domain of application for out methods and tools work, so as (1) to enable tools/methods to be as powerful as is possible without constraining the domain ridiculously small, and (2) to not mislead ourselves and others as to the applicability of our results, conclusions, and observations.

There are two implications to this observation on generality. First, it allows research to be much more directed. Since we do not believe there is a universally applicable solution to the "software problem", we are not tasked with creating a universal, and, by our criteria, weak approach to software engineering.

Second, constraining ourselves to a specific domain allows adoption of powerful domain-specific concepts within the software engineering tools and methods. A simple example is the use of domain-specific vocabulary. A more complex

example is support for a methodology centered on reuse of a domain-specific kernel which may be applicable to only a small domain.

In the spirit of this intent, Section 2.1 defines the domain of interest, Section 2.2 describes the generally accepted software development process used in this domain, and Section 2.3 offers some observations on the appropriateness of the software view in light of experienced maintenance and overall satisfaction with the software product.

2.1 Domain of Interest

The domain under study for the research team is that of real-time, usually embedded, software systems. Those that are not embedded still typically experience severe real-time constraints. Examples of such systems are process controllers, device controllers, autonomous vehicle controllers, autopilots, target recognizers/classifiers, etc.

The term "real-time" as used here precludes systems whose only response time requirement is human tolerance. Human tolerance is a much less stringent constraint then is encountered in embedded controllers. In addition, the portion of the system on which we are concentrating is the processing, not human interface, behavior. An implication of this specialization is that different software engineering approaches may be required for the process and the user interface.

Embedded, real-time controllers tend to be control rather than data driven. The data structures tend to be simple. Custom device interfaces are required. Substantial amounts of the system are devoted to fault isolation and management and parallel processing. Finally, the systems are typically event-driven. Changes in state are triggered by selected signals/data values toggling or crossing a threshold. State changes cause actions which may be realized by mechanical or other means on the part of other (possibly non-software) participants in the integrated system.

Most of the system engineering process is the definition of the states, the signal thresholds, and the actions caused by state changes. Since the embedded software is integrated with specialized hardware, there is a strong interdependency between the hardware and software subsystems. Small changes in the hardware configuration can potentially cause major changes in software. In addition, the software may need to be developed in the absence of a hardware implementation--evolving only against the hardware specification.

Desirable features of a software engineering approach for embedded systems are:

- o Generation of easily modifiable software to accommodate continual changes in requirements
- o Early and quick implementation of software to allow multiple implementation for comparison
- o Software conducive to validation and testing.

2.2 Generally Used Software Process View

The generally used and accepted software process view is that of Boehm's waterfall (BOEH76).

Two fundamental problems with this process view are (1) its inherent sequentially, and (2) the fact that it produces only one implementation. The waterfall assumes that upon entering design, one has a fairly firm requirements specification. There is a very small tolerance for changes in the requirements phase. Although backward arrows exustm supposedly to depict the ability to iterate as needed, having to regroup and return to an earlier phase is the sign of a project in deep trouble. Such a move basically precludes the possibility of bringing a project in on cost and schedule budgets. Very few managers are willing to endorse this strategy. Instead, one of several other strategies are activated: (1) the change is postponed to be performed as maintenance, or (2) the change is "patched" into the current version of the system (design or implementation). In one case, this assures delivery of software which doesn't meet the actual requirements. In the other case, this means delivery of a lower-than-desired quality product.

The second problem with this process view is that it is aimed at the development of a single implementation. Even assuming that the requirements have not changed during development, embedded systems are sufficiently customized efforts so as to guarantee that the first implementation will not be the product that the user/customer really wanted. However, he may not realize this until he sees at least one version that he doesn't like.

These problems summarize to the following goal: Reduce the incremental cost per implementation. This may be done by speeding up the availability of the first implementation, thereby freeing up resources for reimplementations, or by

providing facilities which reduce substantially the second and subsequent implementations.

3.0 Alternatives Being Researched

Consider the realm of software engineering research to be broken into two main camps: (1) those pursuing the development of tools and methods which are predicated on the Boehm process view as a fundamental constraint, and (2) which intend to provide software engineering environments via large departures from the traditional process view. The latter group's approaches may be termed "radical innovations".

First, we assert that the radical innovation approaches offer the promise of massive software engineering improvements. At best, traditionalists can offer incremental improvement. In fact, history has shown that the introduction of automated aids and methodologies without changing the process framework has a very small percentage impact on productivity or product quality.

Second, we have examples of how the introduction of radical innovation has been exceedingly successful in several domains, most notably business file/data base processing.

Three alternative approaches to radical innovation will be surveyed in this section. The three research thrusts vary in their tactics insofar as changing the software development process view is concerned. These three are selected as being the most promising radical innovations of which we are aware. They are:

- o Program transformation; process automation
- o Functional languages
- o Domain specific functional commonality

3.1 Program Transformation; Process Automation

The two research efforts described here are led by Cheatham (CHEA81) at Harvard and Lehman (LEHM81) at the Imperial College of Science and Technology, London. Both approaches are based on providing transformations between consecutive representations of a program. Lehman commences at a more fundamental level of questioning Boehm's waterfall view, deriving an alternative process view, and developing the idea of verifiable mechanized translation. Cheatham derives his approach in a more bottom-up fashion, but arrives as a similar concept of program transformation. Other researchers are also looking at automated transformation; these are discussed in both Lehman's and Cheatham's papers.

A fundamental element of both approaches is the existence not only of stepwise--refined system versions, but also the availability of an explicitly stated transformation mechanism which relates consecutive versions. The most attractive features of this approach are the verifiability and maintenance implications.

As Lehman points out: "If the source and object models of this transformation are both formally described and if a verified mechanical transformation mechanism is used, then the precise correspondence between the representations may be guaranteed. Alternatively, the results of the transformation can be verified" (LEHM81) (our emphasis added). With the increasing criticality of embedded systems, the importance of a feasible approach to verification is growing.

Referring now to maintenance, both Lehman and Cheatham point out that if a maintainer has available to him the design sequence (complete with transformations), automatic incremental rederivation of code is enabled. Thus, the program transformation approach's greatest potential is for reducing total life cycle costs rather than development cost/time.

Cheatham offers an additional perspective which is appealing, that of domain specific notational extensions. Thus, program transformation offers the possibility of customized environments, in our view, a powerful and desirable attribute.

A chief disadvantage of program transformation is that observed by Cheatham himself; that it is not a mechanism for rapid prototyping (at least currently). The elapsed time to seeing executable program behavior must be made as short as possible to allow time for redevelopment of the product within the development window.

Finally, both transformation frameworks support the assembly line development of code (reusable software), but the research work itself is process-oriented; functional commonality is left as a domain-specific problem. Cheatham and Lehman are planting the seeds for a long-range software solution. However, unless the functional commonality aspect is a complimentary research effort, the transformation frameworks solve only part of the problem.

3.2 Functional Languages

A second radical innovation is proposed by John Backus (BACK81). Backus argues that much of the software problem is created by the basic architecture of general purpose computers and their languages. A fundamental programming approach offers powerful program-forming operations and supports exploitation of parallel processors.

As with program transformation, a chief advantage of functional programming is verification. The program forming operations allow general algebra theorems to be applied, enabling a long-term goal program proving.

Much of Backus' work deals with the relationship between execution architecture and the software, and how the software may be structured so as to exploit advances made possible by VLSI. In some ways, Backus' work is more radical than Cheatham and Lehman, because it presupposes a change in implementation medium. Backus is however, dealing with the limited spectrum of code development and execution, rather than with the entire life cycle. The program forming operations are very powerful. The context contributes to the use of reusable code. It also contributes to rapid implementation and re-implementation, once a minimum set of elementary building blocks is available on which to do program forming.

Although functional programming provides a means for supporting domain-specific functional commonality (building blocks), it leaves the non-trivial problem of identifying and developing the elementary functions.

3.3 Domain-Specific Functional Commonality

As mentioned in Sections 3.1 and 3.2, a necessary component for both the program transformation and functional programming approaches is the availability of acceptable elementary functional building blocks. It is our assertion that these building blocks are to a large degree domain-specific. A crucial question is: What is the domain?

An example will illustrate this. Exploitation of functional commonality has been done very well in the business data processing community. This occurred as early as the 1960s, with the introduction of general-purpose file-accessing packages. These packages are differentiated based on the file-access paradigm (sequential, direct, indexed, etc.). In the late 60s and early 70s, commonality was again exploited in the introduction at a higher level of abstraction, data base and file management systems. In these systems, the file access paradigm is a user-selected parameter. The user may define his file and data base structures and rely on the management system to supply the majority of the procedural file access. In the 70s up to the present, work has been done on introducing yet another level of programming abstraction, that of coding-by-form or automatic programming. Procedural level programming is virtually eliminated for a large class of applications.

The key factor to the success of this approach is the identification and exploitation of functional commonality within a domain-file processing. Very different results would have been experienced if the domain had been identified differently. Consider, for example, if the domain had been constrained to "payroll processing". Certainly, the constructs at the automatic programming level would be much different. They would probably be more powerful in the sense that they could be much more specialized, however, they would be usable in a very narrow niche of applications.

Reusability in embedded systems has, to date, followed much the same course as the imaginary parallel scenario described above. Because the software groups are typically appended to the system product group, they are usually very domain specific. So, the software groups are *attempting* to identify and exploit commonality within a very narrow domain niche, such as autopilots (common parts are control laws) and sonar processing (common parts are algorithms). The likelihood of reusing a sonar algorithm in an autopilot is very low.

Because of this parochial view of domain, the concept of reusability, although being very attractive, has achieved little success. It is our assertain that the possibility exists for exploiting commonality increase substantially if one broadens the domain carefully.

Second, we assert that a rule-based processing approach is only one possible generalized program construct applicable to control-driven systems. It is the goal of our research to determine if a rule-based processing model is one acceptable concept. Corallaries to this are: If a rule-based processing view is acceptable, what applications are in the control-driven systems domain? If the rule-based processing view is not feasible, what are some alternatives? What are the variations in the rule-processing model to allow adaption to specific applications?

These are the questions being addressed in our research. Details will be introduced in Section 4 to the extent that they are known today. We emphasize two facets of this effort:

(1) In the absence of the availability of the more ambitious frameworks, offered by Lehman, Cheatham, and Backus (among others), having a domain-specific processing construct can be a tremendous aid to practicing software engineers.

(2) In combination with the program transformation and functional programming approaches, having defined elementary building blocks will enable rapid insertion of these powerful techniques into general use.

4.0 Functional Commonality--Rule-Based Processing

This section discusses the appropriateness of representing the behavior of typical embedded, real-time software systems as rules, what this implies for the software structure and for the development process, and a sampling of specific domains for which this problem view may be appropriate.

Notice that the acceptability of rules as a model for functional behavior is not the only issue involved with this concept. There are more substantial issues surrounding real-time execution feasibility. These research issues will be touched on in Section 5.

4.1 Rule-Based Software Architecture

The central idea is that one possibility for functional commonality is a software architecture such as that shown in Figure 4-1.

The common parts are those in the "General Rule Processor" box. In reality, one may desire a selection of components to serve as the processes for this function. The "Rule Retrieval" component accesses the rule base according to some strategy for retrieving rules to be considered for execution. It is this component which offers the most potential for performance tuning, by introducing smarter heuristics for rule base accessing.

The "Rule Execution" component evaluates the rules. This may involve changes to the global state data, require other rules to be invoked, etc.

The "Rule Base" is the organized collection of rules which govern the behavior of the embedded system. The rules may be organized into rule sets or may be a single monolithic set. The selected organization will have performance implications.

The "State Data" is the mechanism of communications between the expert system and the remaining system software or hardware. Examples of data which might be placed in the state data space is a command for a particular sensor reading. The sensor, in turn, would use the state data space to return its value.

Ideally, the system engineer would be required to input:

- o An organized rule base
- o Selection of rule processing components
- o Structure of the state data space
- o Remainder of the hardware/software system

A knowledgeable user could implement mission-specific heuristics for the rule retrieval and/or rule execution components. The development system should be knowledge-based in the sense that it knows the software architecture concepts and can recognize acceptable and unacceptable user-specified configurations. More advanced environments could help the user optimize his implementation.

4.2 Implications for the Software Process

Assume for a moment the following scenario. The software engineer has available to him the facilities to load a ruleset, a collection of control (rule-processing) modules which can be mated with the ruleset, and suitable test harnesses for evaluating the software's behavior. Additionally, assume that a mechanism is provided to download the rules from the development computer to the target. What the user has at this point is not just a rapid prototyping environment, but rapid implementation. Having access to a low-cost, low-effort implementation could radically change the software development process and substantially reduce the amount and cost of post-installation maintenance. The concept is that sufficient uncertainty exists in the expectations for a system to virtually eliminate the possibility that the first implementation will be satisfactory. Thus, the development process becomes one of creating a sequence of alternative implementations, evaluating the individual implementations, and eventually arriving at one which is judged to be acceptable.

The ability for the AI community to achieve this iterative programming approach is based primarily on the functional programming nature of LISP. Adding generally usable functions to LISP has the effect of extending the language to more and more abstract levels. Generalizing the control processing into rule evaluation is one of the outgrowths of this trend. This incremental nature of LISP allows reimplementation without a complete rework. By replacing selected functional building blocks, one can obtain different functional behavior or performance levels.

It is this concept of facilitating rapid implementations that should be borrowed from AI. The direct effect is that we can increase substantially the reuse of common parts and substantially decrease the level of effort required to rework, whether during the initial development or after installation.

4.3 Supporting Evidence

The example set of expert systems implemented to date is populated primarily with "consultant" type systems. These systems assume the existence of a human user who will draw on the support of the expert system for decision making support. In addition, most of contemporary expert systems are "load-and-go" processing. Data is introduced to the system at the beginning of the run and the expert system processes until a solution is reached. Few existing systems have the ability for an external entity, such as a sensor, to interrupt the processing with new data.

Expert systems have been applied successfully to problems which are functionally within the control-driven domain (See 4.4). These have been sufficiently successful to warrant further investigation. The example systems are particularly useful in establishing the requirements for the development environment, and the processing heuristics, for a baseline description of the interface between an expert system and the remainder of the system, and to suggest issues of downloading to an embedded target.

In addition, some work has been done on evaluating the use of finite state machine and frames for implementing embedded systems (GOLD78), (RANG80), (DESA78). There is sufficient similarity between these concepts to support an assertain that a rule-processing framework is worth considering.

4.4 Application Domains

A prototypical application of this class is HASP and its successors at Stanford University and elsewhere. These systems implemented sonar classification (NII78). Additional work has occurred elsewhere in the embedded systems community (EVER84), (GRIE84).

Currently, we are building expert systems selected from two application classes: an embedded controller and a classification/diagnosis expert. The diagnosis expert is similar to that developed by Schlumberger for oil well log analysis (DAVI81). Our intent is to have a high level of interaction between the application and technology efforts, that is, the SOA technology will be advanced in areas defined as high priority by our applications development. In turn, improvements in the application implementations are fueled by the new components supplied from the technology efforts.

5.0 Research Directions

It is our intention to capitalize upon existing and completed research insofar as is possible. That is, we expect our research to be performed primarily in the areas of:

- o Problem specific techniques
- o Expert system interfaces
- o Real-time, embedded issues.

Elaboration of what is meant by each of these issues is discussed in Section 5.2.

This means that we are intending to adopt/build on artificial intelligence and software methods research in the following topic areas: Knowledge representation, AI software architecture, Natural language, and AI language theory.

We intend to limit our work in these areas to that needed to customize approaches to our selected domains and/or to achieve our system performance goals. We do expect to contribute substantially to the state of the art in several of these areas; the work will be requirements, rather than theory, driven.

A direct consequence of this strategy is that we intend to utilize as a starting point an existing expert system engineering environment. The following section describes our analysis and selection.

5.1 Expert System Engineering Environment

The ideal expert system engineering environment includes a software architecture and tools to support building application systems

constructed in compliance with this software architecture. A software architecture is the knowledge representation and corresponding control processing.

One or more knowledge representations are available in commonly available expert system engineering environments (ESEEs). The two most common are rules and frames. As argued in Section 4, rules are the most popular candidate for representing embedded real-time systems. The availability of rules as a knowledge representation is not an overly constraining requirement.

An ESEE may supply one or more processing components which operate on a given rule base. An ideal ESEE contains more than one control processor, so that an application builder may experiment with different control processors without substantially changing his rule base. In addition, an ideal ESEE allows the user to supply his own application needs (particularly performance).

Most expert systems built to date are consultant systems. That is, their purpose is to interact with users in the solving of a problem. Embedded, real-time systems are typically not consultative. On the contrary, the ideal ESEE would support facilities such as the ability to process signal interrupts in preference to a heavy emphasis on the human interface to the application expert system. (Notice this is not saying that the user interface to the ESEE is unimportant.

Our initial starting point for a set of ESEEs were the eight surveyed by Hayes-Roth and others (HAYE83). Our first cut was done via study (no experimentation) of the analysis in Hayes-Roth and of cited documentation. Our leading candidates at that point were: AGE from Stanford University (AIEL80), HEARSAY-III, and OPS-5 (FORG81).

AGE was (and is) particularly attractive because (1) the concept of AGE is to construct an ESEE via commonly-used building blocks (including the possibility of user-provided ones), (2) it supports several control processing strategies, (3) it supports the concepts of "event" and "expectation" and (4) it allows the user to invoke procedures (written in Interlisp). This last capability allows us to build specialized interfaces to the user and other systems such as simulation.

We determined then, to experiment with OPS5, AGE, and a third system, EXPRESS, which had been developed by Honeywell's Hemel Hempstead group (JACO83). EXPRESS offered and added possibility of allowing a LISP-PROLOG tradeoff in that EXPRESS as a PROLOG-based ESEE.

Since our experimentation began, we have evaluated the documentation on several other commercially available ESEEs. We have not added them to our experimentation plan. The primary reason is that most ESEEs are targeted to the consultation domain of expert systems and are not sufficiently flexible to allow their use in our problem areas of interest. A secondary reason is the availability of source code to support modifications.

A final note on our discussion of generally available ESEEs is that we do not expect to find an ESEE which we can use in our applications without substantial modification and supplementation. This will become clear as we present our research plan. However, it is highly desirable for us to be able to utilize an existing operational ESEE as a prototype.

5.2 Research Issues and Plan

Our research objectives are addressing two basic questions:

- Is it reasonable to represent a significant portion of applications within our problem domain as rules (or possibly rules combined with an additional knowledge representation)?

- What is the architecture (hardware and software) required as a vehicle for this approach to meet the system level performance requirements?

During late 1983 and through 1984, work on the first question was undertaken. OPS5 was obtained from CMU and installed on the central Multics facility. the AGE system required rehosting from a DEC-10 to a VAX. This was accomplished by February, 1984.

Prototyping of a selected "typical" embedded controller was also begun in 1984. Early results with OPS5 were disappointing in terms of its appropriateness for use in our environment. Early results with AGE are much more encouraging, despite the fact that the system itself has bugs and response time problems. The reasons for the encouragement are: AGE's provisions for event and expectation driven framework; the

ability to alter control strategies without redoing the rule base; the high level (abstract) nature of the user interface; and the component approach to the system's construction. Work is continuing on a prototype controller using AGE.

Work has begun on the issues surrounding real-time use of an expert system. These issues include:

- o How to download the system to the execution environment, including the issues of target language.
- o How to obtain real-time execution performance, particularly through reimplementation of the rule selection and execution components.

6.0 Conclusion

The success of reusable software process views is dependent on the appropriate selection of common functional parts. Our research deals with this aspect of reusability. One candidate part perspective is to separate knowledge and control; a view used in many AI systems. The suitability of this architecture for embedded systems developers is contingent upon achieving adequate performance. This is the thrust of Honeywell's current research program.

BIBLIOGRAPHY

(1) (AIEL80) N. Aiello, C. Bock, H.P. Nii, and W.C. White, "The Joy of AGE-ing: An Introduction to AGE-1 System," Heuristic Programming Project, Computer Science Dept., Stanford University, Stanford, California, August 1980.

(2) (BACK81) J. Backus, "Is Computer Science Based on the Wrong Fundamental Concept of 'Program'?, An Extended Concept", in Algorithmic Languages, deBakker and VanVliet (eds), North-Holland, 1981.

(3) (BACK82) J. Backus, "Function-Level Computing", IEEE Spectrum, August 1982, pp 22-27.

(4) (BOEH76) B. Boehm, "Software Engineering," IEEE Transactions on Software Engineering, December 1976.

(5) (CHEA81) T. Cheatham, G. Holloway, and J. Townley, "Program Refinement by Transformation," IEEE 1981.

(6) (CHOW78) T.S. Chow, "Testing Software Designs Modeled by Finite State Machines, "IEEE Transactions on Software Engineering, VOL SE-4 No. 3, May 1978, pp. 178-187.

(7) (DAVI81) R. Davis, H. Austin, I. Carlbom, B. Frawley, P. Pruchnik, R. Schneiderman, J>A> Gilreath, "The Dipmeter-Advisor: Interpretation of Geological Signals, "Proceedings of IJCAI-81, pp. 846-849.

(8) (DESA78) R.J. DeSanto, "Using Finite-State and Structured Design Techniques for Embedded Software Design," NAECON 78, May 1978, Dayton, OH, pp. 236-241.

(9) (EVER84) D.C. Evers, D.M. Smith, C.J. Staros, "Interfacing an Intelligent Decision-Maker to a Real-time Control System, SPIE Proceedings, Vol. 485, May, 1984.

(10) (FORG81) C.L. Forgy, "OPS-5 User's Manual," Dept. of Computer Science Carnegie-Mellon University, Report CMU-CS-81-135, July 1981.

(11) (GOLD78) J. Goldberg, Hierarchical System Development, SRI Project 4403-22 SRI International, Menlo Park, CA Jun. 1978.

(12) (GRIE84) J.H. Griesmer, et al., "YES/MVS: A Continuous Real-Time Expert System," AAAI Proceedings, August 1984.

(13) (HAYE83) F. Hayes-Roth, D. Waterman, and D. Lenat, Building Expert Systems, Addison-Wesley, 1983.

(14) (JACO83) P. Jacobs "A Review of Expert Systems or Rules Brittania," Honeywell 7th Int'l Conference on Software Engineering, April, 1983.

(15) (LEHM81) M.M. Lehman, "Programming Productivity - A Life Cycle concept, Proceedings IEEE Compcon, Fall, 1981.

RESUME

NANCY M. GIDDINGS

SECTION CHIEF
HONEYWELL SYSTEMS AND RESEARCH CENTER

Research Expertise

Software Engineering
Artificial Intelligence

Education

All but dissertation PhD, Computer Science, Iowa State University, 1974
MS, Computer Science, Iowa State University, 1973
BS, Mathematics, University of North Dakota, 1971; Phi Beta Kappa

Experience

Ms. Giddings is currently the Research Section Chief of the Software Technology Section at S&RC. The section consists of 10-15 engineer/scientists, most of whom have advanced degrees in Computer Science. The Software Technology section's two long term research topic areas are applied artificial intelligence and Ada target execution environments (run-time and compiling systems). The mid-term research is in traditional Ada environments, particularly testing/V&V. The short term efforts are directed at filling specific Honeywell tool/methodology needs, such as a knowledge-based text editor.

The AI long-term thrust is particularized to embedded, real-time expert systems. Research topics are software architecture, real-time performance, and development environments. The target environments thrust is developing a distributed Ada compiling and run-time system, as well as a multi-targeted Ada software/microcode compiler.

Prior to becoming Section Chief in May, 1984, Ms. Giddings was the project lead on the AI thrust and participated on the near and mid-term efforts. She was the project leader of an 1983 IR&D project to install and experiment with an AI, LISP-oriented toolset. In addition, she won a 1983 Initiatives grant to study and experiment with the feasibility of applying knowledge-based techniques to software development environments.

She managed a $560K software development program over 18 months which ended successfully in November 1982. This was project to apply software simulation to test program set development in ATE. She has participated technically on secure computing and Ada projects.

Ms. Giddings has managed or helped manage the following proposals:

VHSIC Phase 2 study, VHSIC Architectures for AI, Military Computer Family Operating System (MCFOS), Secure Data Base Management System, and LogLisp Programming System. In addition, she prepared the 1983 IR&D brochure on software.

Prior to joining Honeywell, Ms. Giddings was an assistant professor at the University of Wisconsin, River Falls; a software project manager at the North Dakota Employment Security Bureau, and the manager of the data processing section at the North Dakota State Highway Department.

Publications

(1) "A Rule-Based Software Design Evaluator," with T. Colburn, to be published in the Proceedings of ACM 84, October 1984, San Francisco, CA.

(2) "Software Methods Meet AI," Proceedings 7th Honeywell International Conference on Software Engineering, Minneapolis, MN, May 1984.

(3) "Quantifying Software Designs," with J. Beane and J. Silverman, Proceedings of 7th International Software Engineering Conference, Orlando, FL, March 1984.

(4) "An Approach to Design-for-Maintenance," with J. Silverman, J. Beane, IEEE Software Maintenance Workshop, Monterey, CA, December 1983.

(5) "The Software Problem," Scientific Honeyweller, June 1983.

(6) "A Software Engineering Experiment: Using a Component Interconnection Language to Capture the Software Structure of a Flight Control System," with J. Beane, J. Silverman, Honeywell S&RC Technical Report, April 1983.

(7) "A Component Interconnection Language for Evaluating Software Design Quality," with J. Silverman, J. Beane, Honeywell S&RC Technical Report, March 1983.

(8) "The Maintenance and Design Implications of Viewing the Software Structure as an Interconnection of Components," with J. Silverman, J. Beane, Honeywell S&RC Technical Report, February 1983.

(9) "Software Engineering Environments," with J. Silverman, Honey Report, 1981.

(10) Department of Labor Data Base Study, North Dakota Employment Security Bureau, 1978, Volumes 1-5.

(11) "The Automation of the Computer Field," Report to the Director of the Department of Accounts and Purchases in North Dakota, 1976.

(12) "An Algebraic Interpretation of the Halting Problem," Master's Paper, Iowa State University, 1973.

Professional Activities

Phi Beta Kappa
Outstanding Young Women of America - 1976, 1977, 1978
Alpha Lamba Delta, freshman womens' honorary
President's List and Dean's List, University of North Dakota all semesters

A Process View for Real-Time Systems

Reusable Components Based on Separating Knowledge & Control

Nancy Giddings
Honeywell Systems & Research Center
Minneapolis

Reusable Components for Real-Time Systems

Motivation: Real-time, embedded systems becoming more complex (intelligent?)

Validation problems escalating

Maintenance problems escalating

Main Points

- Reusable components have been successful in certain domains, but leverage in the real-time embedded community has not been realized.

- One of the main reasons for this is that software engineers look at the wrong granularity of abstraction when tackling reusable parts.

Two Example Domains at Honeywell

Image Research Lab (~40 persons)

Reusable parts are algorithms

New systems are composed typically of 85% old parts and 15% new

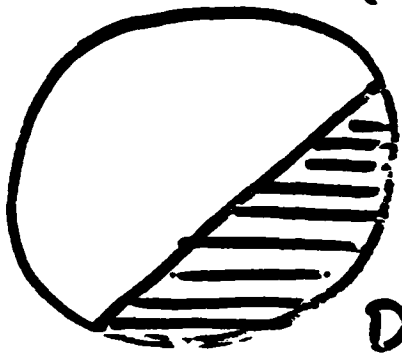
Flight Control Group (~6 persons)

Reusable parts are control laws

Typically represent 30-35% of flight control system

Economical Scenarios

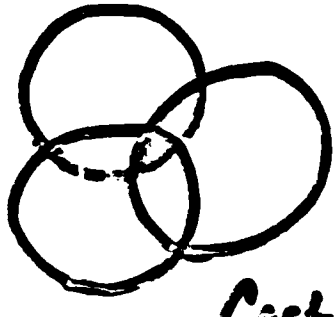
1. Domain / Number of Projects is large.



Cost of preparing reusable parts prorated across many similar projects

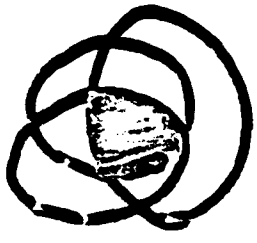
Domain

2. Domains are small, but there is a potential for common reusable parts.



Cost of reusable parts is shared by domains

or:



In Honeywell

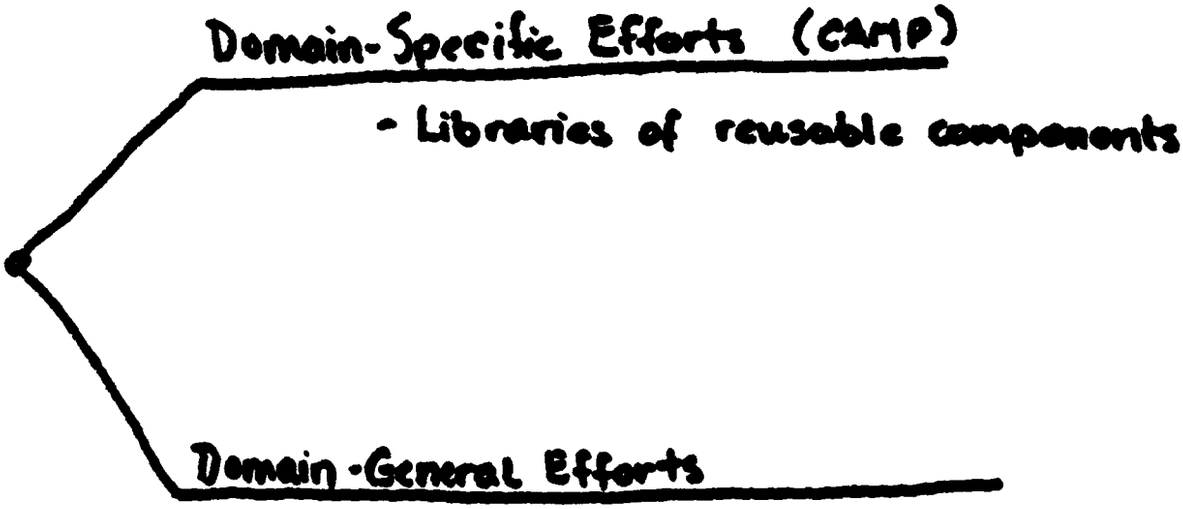
- Domains tend to be small and change rapidly in size and scope.

- Option 2 is the economically preferable one

- Requires raising the level of abstraction so one can see the inter-domain common parts

- One idea is based on separating knowledge and control, a paradigm borrowed from AI

Program Strategies



VARIATIONS OF A REUSABLE SOFTWARE COMPONENT

Dr. J. Kaye Grau

Harris Corporation

Software is constantly being reused. However, creative reuse of a software component developed previously for use in another system to perform a context-dependent function in a very specific and potentially highly-coupled environment is certainly a challenge to any software engineer. As a result, research into methods and tools to improve the reusability of software components and thereby the productivity of the software engineer has potentially high pay-offs. In this paper, a survey of current literature is presented with classification of reusable components into two categories: opaque and transparent. Then, some of the possible variations of a transparent component are examined. Finally, a position is reached that automatic support for variation generation from a transparent component is required to make software reuse practical.

Peter Wegner (WEG 84) recently observed that there are four different kinds of component reusability that contribute to software productivity. He said that they may be reused

- o in a variety of applications
- o in successive versions of a given program
- o whenever programs containing the component are executed
- o by being repeatedly called during program execution

The latter three are the common forms of reusability; for an analogy, compare Wegner's list with the reuse of a spoon. Most everyone over the age of one knows how to use a spoon repeatedly to bring food to their mouth during one meal; to use a spoon at every meal to help with eating; and to use a soup spoon to eat soup and a teaspoon to eat ice cream; however, the really creative uses of spoons such as for making jewelry and as a musical instrument require the application of the spoon for uses that it was not originally invented. Imagine that you have been given an assignment to use a spoon in a new and creative fashion. You really have two choices: 1) use the spoon without physically changing it for a new function or in a new environment, or 2) physically modify it by bending or melting or whatever to create a new object made from the spoon. Needless to say, creative reusability is the only form of reusability that will be addressed in this paper.

A survey of current literature on creative software reusability results in a classification of reusable components into two categories:

(1) opaque components

(2) transparent components

Opaque components are components which one cannot see into and therefore cannot be modified easily. We have all experienced opaque software; commercial operating systems, commercial compilers, and most commercial software which is sold in object form or in an encrypted form fall in this category. These components have well defined interfaces and an anticipated result. But how many times are the results exactly what was expected? We have all written work-arounds for compiler bugs and operating system bugs. So reusing an opaque component in a creative fashion requires "tricking" it, or by using a preprocessor or a post-processor.

One example of opaque component reusability can be observed in Unix. Kernighan, in a recent article on reusability in Unix, stated: "...this trivial example is typical of Unix use: two programs are connected transiently to do a job that is worth mechanizing but not worth writing a special program for....people routinely use the capabilities of the shell to cover up defects in existing programs or to combine them into new ones; it is much easier than writing a new program

from scratch." (KER 84)

There are obvious problems with trying to reuse a software component in an environment or for a function for which it was not originally designed. Standish recently commented: "...it is certainly the case that some software components are too specialized and concrete to be reusable." (STA 84) Cheatham agrees: "It is our belief that, even with the relatively advanced modularization facilities provided by Ada, the extensive reuse of concrete Ada programs is unlikely. The problem is that programs in any concrete high-level programming languages are the result of mapping from some conceptual or abstract specification of what is to be accomplished into very specific data representations and algorithms which provide an efficient means for accomplishing the task at hand." (CHE 84)

With opaque type reusable components, some of the obvious technical problems which require resolution are:

- o As software development proceeds in a top-down manner, how can potentially useful off-the-shelf reusable components be recognized?
- o How do the practical aspects (i.e., cost and schedule) of "make-or-buy" decision influence reusability? What is the cost trade-off between a one-time development cost vs. life-cycle duplicate copy costs?
- o What technical factors need to be considered in a "make-or-buy" decision (e.g., performance, influence on rest of system, design, potential execution side-effects)?
- o How can the components be coupled into the system correctly?
- o How can we be certain that the freshly coupled component will perform reliably in its new environment?

Litvintchouk and Matsumoto have also recognized the need to study coupling: "The very concept of 'reusability' must be defined more rigorously, in terms of the dependence of the component on enclosing or higher level environments. It should be possible to develop metrics for the measurement of such component dependence, enabling quantification of potential reusability." (LIT 84)

Transparent components, i.e., components whose internals may be easily viewed and thus potentially modified, may be used in the same fashion as an opaque component. Additionally, they may be modified, used as patterns for composition of new yet similar components, and used as teaching aids.

Programs have been and are being developed to help mold transparent components into usable software:

- o application generators with fourth generation languages which link together and customize components within a well-defined application area
- o general purpose formal specification transformation/automatic code generation

Horowitz and Munson in "An Expansive View of Reusable Software" present an effective comparison of the currently available programs which support the molding of transparent reusable components into functional software systems. (HOR 84)

Transparent components are useful teaching mechanisms. Standish (STA 84) observes that "the successful practice of software reuse appears to help considerably in the teaching of certain kinds of computer science courses....One view of the teaching of computer science is that it involves identifying and presenting useful abstractions--those which, at best, will be intellectual tools serviceable for a lifetime."

Whether the component is opaque or transparent, the degree of reusability is improved if higher levels of abstract descriptions are available. Matsumoto discussed the relationship between reusable components and levels of abstraction including both requirements and design. (MAT 84) Test sets

and validation suites also increase the reusability. Clear definition of the binding, interface, and/or coupling with the external environment also adds to a component's reusability.

The creative reuse of a component, either an opaque or a transparent component, generally requires modification of a component itself, a modification of the way in which the component is used, or a modification of the environment in which it is to execute. The remainder of this position paper will focus on the potential modifications of a transparent component to produce component variations. A component variation has the same functionality and capability of doing a job but goes about performing the job in a different manner, e.g., faster, more reliably or more accurately.

Litvintchouk and Matsumoto recognized the potential of component variations: "Components which have the same externally viewed static semantic behavior but differ in various performance criteria can still be grouped together in the component library. In this case, a criterion for retrieval based only on static semantic specification will retrieve from the library a range of possible subsystem configurations, all semantically valid. Then, other techniques (possibly heuristic search) can be applied to these possible configurations to find the one which is 'optimal' according to the specified performance criteria." (LIT 84)

An obvious variation is based on sizing and timing. A frequent trade-off which must be made in designing software is between making a program memory resident and fast versus smaller with many disk accesses and thus slower. Thus one variation of a component could be a speedy, memory resident version, while another variation could be a small version with disk accesses and overlays built into it.

Another potential form of variation is based on quality factor variation. An extremely reliable component variation, an easily maintainable component variation and a highly interoperable component variation as well as other quality factor variations could be developed and stored in a component's variation library. Depending on the quality requirements of the system being constructed, the component with the proper quality could be selected from the variation library.

Variations of a component could also be developed with different binding mechanisms. For example, a component variation could have a task and entry binding, another variation a procedure binding, another variation a function binding, and another variation a mere begin block for incorporation in-line in a program.

Another potential characteristic which could result in another set of variations is degree of parameterization. For example, the size of a array can be passed as a parameter or it can be "hard-coded" within the component. Variations of a component with different parameter lists could be developed and stored in the component's variation library.

The type of coupling of a component with its parental environment is another potential source for variations. For example, a semaphore component can be written which either makes use of a shared semaphore or uses an object oriented approach and encapsulates the semaphore within itself. Similarly, a component variation could be developed which is intended to inherit data from its encapsulating parent while another variation could be an independent component with no inheritance required. Yet another potential variation is in the style of repeatability of the component. One variation could be nonreentrant while another be reentrant. Similarly, one variation of a component could be recursive while another variation could use iteration to accomplish the same task.

The interactive ability of a component may produce yet another type of variation. A component can be designed to interact with a user or to perform its task independently in a background fashion with no user interaction.

For components which are based on decision tables, decision trees, or other selection-intensive logic, the choices may be stored as data or hard-coded. The data-driven variation is certainly more flexible than the hard-coded version, but hard-coded versions may be faster and more reliable.

Finally, for purposes of this position paper, the last variation to be identified is the degree of specificity or generalization of the component. For example, if printing the sum of 2 and 2 was a requirement, a component could be implemented as a single line:

```
print "2 + 2 = 4"
```

However, it has limited potential for reusability. A more generalized variation of this component could be developed which would add any counting number to itself and return the result for printing. An even more generalized variation of this component could be developed which would add any two integer numbers. The next level of generalization might be a component which would handle any specified binary operation on any two parameters. More general than that could be a component which could parse and interpret any general equation and evaluate it to produce the desired result. Therefore, each level of generalization/ specialization can be a variation of a component since each would accomplish the required task.

Now that many types of variations of a single component have been identified, a conclusion can be drawn that in general the number of reusable components and their variations can quickly reach an astronomical number. In addition to the first order variations identified above, second, third, and so forth orders of variations of components can be created. For example, a memory-resident, reliable, task bound, highly parameterized, object oriented, recursive, interactive, data driven, generalized (ninth order) variation of a component might be required to meet all the system requirements. The problems of storing, locating, identifying, retrieving, maintaining, and other logistic issues are obvious.

An automated system which would provide the user with a specified version of a component could be accomplished in two different ways. The most obvious automation is a large database of component variations with tools which allow the user to select one or more variations, modify and merge them to create the needed variation and then store the newly created variation back into the database. The alternative automation scheme is certainly intriguing. A knowledge-based variation generator could be developed which given a transparent component and a set of variation requirements could automatically produce the needed variation of a component. The variation generator, seemingly beyond today's technology, has the potential of making component reusability practical.

BIBLIOGRAPHY

(1) (CHE 84) Cheatham, T.E. "Reusability Through Program Transformations", IEEE Transactions on Software Engineering, Volume SE-10, Number 5 (September 1984), pp. 589-594.

(2) (HOR 84) Horowitz, E., and Munson, J.B. "An Expansive View of Reusable Software", IEEE Transactions on Software Engineering, Volume SE-10, Number 5 (September 1984), pp. 477-487.

(3) (KER 84) Kernighan, B.W. "The UNIX System and Software Reusability", IEEE Transaction on Software Engineering, Volume SE-10, Number 5 (September 1984), pp. 513-518.

(4) (LIT 84) Litvintchouk, S.D., and Matsumoto, A.S. "Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification", IEEE Transactions on Software Engineering, Volume SE-10, Number 5 (September 1984), pp. 544-551.

v.np (MAT 84) Matsumoto, Y. "Some Experience in Promoting Reusable Software: presentation in Higher Abstract Levels", IEEE Transactions on Software Engineering, Volume SE-10, Number 5 (September 1984), pp. 502-512.

(5) (STA 84) Standish, T.A. "An Essay on Software Reuse", IEEE Transactions on Software Engineering, Volume SE-10, Number 5 (September 1984), pp. 494-497.

(6) (WEG 84) Wegner, Peter "Capital Intensive Software Technology", IEEE Software, Volume 1, Number 3 (July 1984), pp. 7-45.

RESUME

DR. J. KAYE GRAU

Dr. Grau is scheduled to enter the employment of Software Productivity Solutions, Inc. in February 1985, prior to the Workshop on Reusable Components of Application Software.

HARRIS CORPORATION, Government Satellite Communications Division, Melbourne FL.

Dr. Grau is currently on staff to the Naval Extremely High Frequency Satellite Program (NESP), responsible for interface validation. The effort involves designing a complex real-time simulator that will exercise 23 system interfaces. She previously developed a complete set of 10 NESP software standards, covering the entire life cycle. The approach included the use of an Ada PDL for development, even though the implementation could not be in Ada.

Dr. Grau has been the focal point for Ada methodology in the corporation, transferring the technology to numerous internal and DoD developments. She was responsible for establishing and executing a sector-wide Ada training program. She has provided consulting, evaluations and training in Ada for numerous programs and proposals. She provided an Ada technology assessment for NASA for application on the space station. Dr. Grau was awarded an engineering award for her contributions in Ada technology. She is editor of Ada Letters, the internationally recognized Ada publication.

Previously, Dr. Grau was Group Leader of Methodology, responsible for researching and applying advanced software techniques and tools. She has provided software methodology support to several major software developments. Among the methodology products provided in this capacity are the Software Development Plan, Software Design Standards and Conventions, Computer Resource Plan, Software Documentation Standards, Coding Standards and project-specific training.

She was a major contributor to the definition of Harris' Integrated Software Methodology (ISOMET) and has been instrumental in the writing of a generic Computer Program Development Plan (CPDP). She was the principal author of the Harris Ada Process Description Language Guide and is currently active in the IEEE Ada as a PDL Working Group.

UNIVERSITY OF CENTRAL FLORIDA, Orlando, FL

Dr. Grau was Assistant Professor of Computer Science, teaching a wide range of department courses including operating systems, structured programming and database design. In addition to teaching, she was system manager of the Computer Science Department's VAX laboratory and faculty advisor for the student chapter of the Association for Computing Machinery. Her primary area of research was in software engineering and tools.

EDUCATION

UNIVERSITY OF MISSOURI-ROLLA, Rolla, MO
Ph.D Computer Science
M.S. Computer Science

CENTRAL MISSOURI STATE UNIVERSITY
B.S. Mathematics

PUBLICATIONS/PRESENTATIONS

(1) "Ada Design Language Concerns," Second Annual Conference on Ada Technology, 1984.

(2) "Ada-Based Design Methodologies," Session Chairman, AdaTEC Conference, June 1983.

(3) "The Use of an Ada PDL," AdaTEC Conference, February 1983.

(4) "Compilability of Ada PDL," AdaTec Conference, June 1982.

(5) Ada Process Description Language, Harris Corporation, May 1982.

(6) "Application and Comparison of Algorithms for the A11 Shortest Paths Problem," Ph.D Dissertation, 1979.

(7) "Precise Interstop Distances Using a Digitizing Tablet," ORSA/TIMS Conference, 1976.

(8) "Minimizing Transportation Cost Using Computerized Routing," MEC/UMR Conference on Energy, 1976.

(9) "An Analysis of Finite-Queue, Multiple-Server Facilities," ORSA/TIMS Conference, 1975.

(10) "An Analysis of a Network of Finite-Queue, Multiple-Server Facilities," Master's Thesis, 1973.

PROFESSIONAL ASSOCIATIONS

Editor, Ada Letters
Association for Computing Machinery (SIGSOFT SIGADA)
IEEE Computer Society

Creative reuse of software is a challenge to any software engineer.

Peter Wagner* observed that there are four different types of reusability:

- repeated execution of a procedure/block of code
- repeated execution of a program
- successive versions of a given program
- "creative reuse" in a variety of applications

* Wagner, Peter, "Capital Intensive Software Technology," IEEE Software, Volume 1, Number 3 (July, 1984), pp.7-45.

Two types of software components are creatively reused.

- Opaque Software Components:
  - — cannot see internals
  - -- distributed in object module/encrypted form
  - -- operating system is a prime example

- Transparent Software Components:
  - -- can see internals
  - -- can modify internals
  - — Knuth's books are examples

Opaque software components cannot be internally modified for reuse.

To reuse an opaque component in a manner for which it was not originally designed, the programmer manipulates it by:

- faking its input parameters
- enclosing it with a preprocessor/postprocessor (e.g. UNIX pipes.)

Reuse of opaque components introduces the following problems:

- identification/selection
- make or buy decision
- coupling
- reliability

Transparent components can be reused opaquely or can be modified and a component variation created.

Potential forms of variation include:

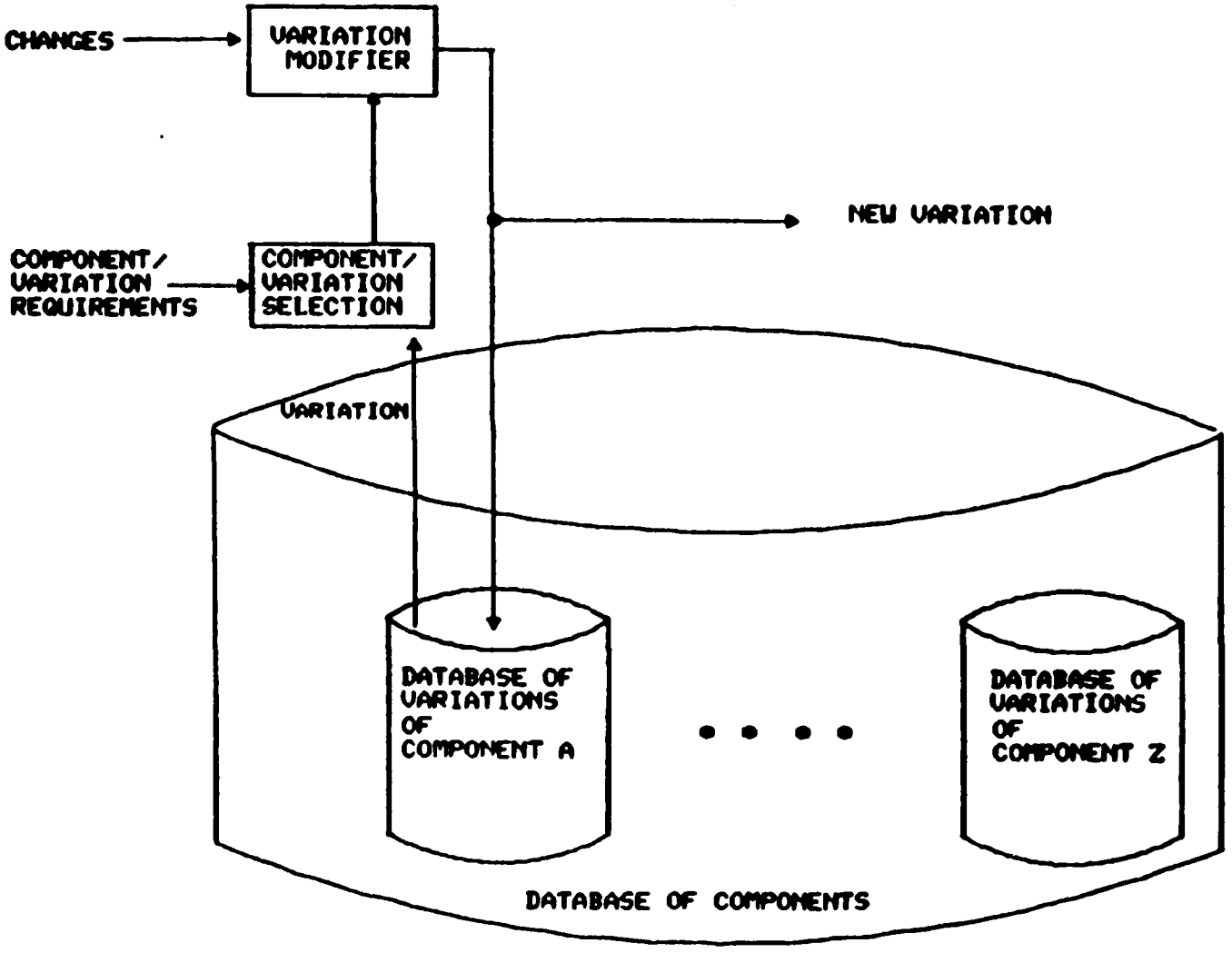
- Size vs. time
- Quality factors
- Binding mechanisms
- Degree of parameterization
- Coupling
- Repeatability
- Interactive vs. background
- Code vs. data driven
- Generalization/specification

To meet realistic system requirements, an nth order variation of a component is needed.

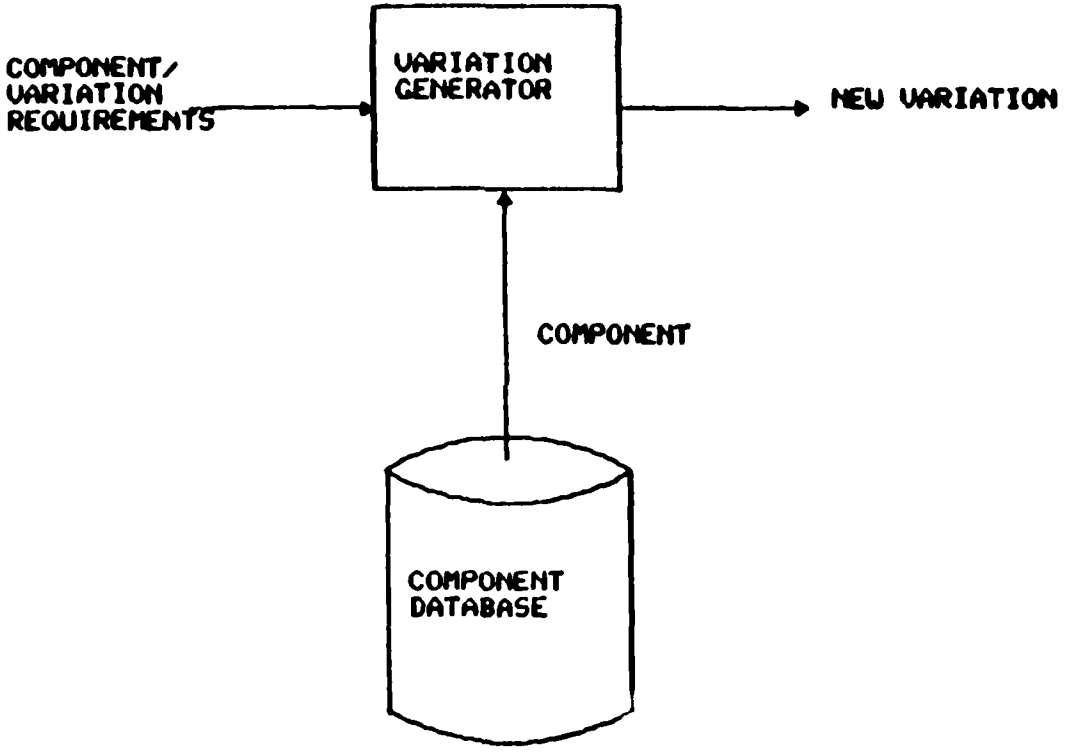
For example, a ninth order component might be required to be:

- memory resident
- reliable
- concurrent
- highly parameterized
- object-oriented
- recursive
- interactive
- data driven
- generalized

One possible form of automation for reuse of transparent components would have a large data base of variations.



An intriguing, revolutionary alternative is a variation generator.



SOFTWARE VALIDATION OF SIGNAL PROCESSING SYSTEMS AND ITS IMPACT ON REUSABILITY

Michael R. Miller
Hans L. Habereder
L.O. Keeler

Hughes Aircraft Company
Fullerton, CA 92634

Abstract

As computers and other digital processors become more powerful and cost-effective as compared to other implementation alternatives, software development will continue to increase in importance in all major new system developments. The reuse of application software has significant potential in terms of reducing cost and development time for mission critical applications. In considering the potential use of reusable software components for weapon systems, the functionality of the component, and other factors such as reliability and maintainability should be investigated. These, as well as other factors, underscore the necessity of a well conceived and executed Validation plan. Validation of software components represents a critical part of the overall system validation problem and the use of automated tools in performing software validation will become increasingly important.

The purpose of this paper is to describe an integrated design and test method used on signal processing software developed at HUGHES, and to provide information on the relative cost savings associated with the use of simulation during software development and the reuse of software components. The unique difficulties encountered in the reuse of signal processing software will be explained along with some techniques which have been used at HUGHES to overcome many of these difficulties.

Characteristics of Signal Processing Hardware and Software

Surveillance systems such as sonar or radar systems require real-time processing of large amounts of input data by means of signal processing operations such as beamforming, digital filtering, fast fourier transforms, digital correlation, noise normalization, and thresholding. This processing is highly computational and requires throughput rates which are typically two orders of magnitude greater than general purpose applications.

Many signal processing algorithms are iterative and/or perform identical processing on large amounts of data. The logical structure is simple, typically requiring only IF-THEN-ELSE, CALL, and DO-WHILE constructs. Data dependent branching probabilities are approximately 4 orders of magnitude smaller than those found in general purpose programs. These characteristics of signal processing algorithms are utilized by specialized hardware and software to meet the large throughput demands. The redundant and simplistic characteristics of signal processing algorithms are exploited through the use of parallel processing and pipelines. Parallel processing is the use of duplicate hardware to perform identical processing on separate data paths simultaneously. Pipelining is a technique in which a sequence of instructions is executed simultaneously in an assembly line manner.

Signal processing software encounters two major obstacles when considered for reuse. First, the software is usable only on a specific hardware design and generally cannot be hosted on other machines. Second, the software is usually designed for such a specific application that it is not useful anywhere else. To realize the capabilities of parallel or pipelined architecture, the software is tailored to very specific functions. Through the use of table driven firmware,

modular design and use of simulation in testing, the generalization and reliability of software can be increased to allow reuse without sacrificing the efficiency of many common signal processing functions.

As a background for discussions that will follow, a brief explanation will now be given about how typical signal processing hardware and software differs from general purpose hardware and software. The operational software of the typical signal processor consists of firmware (or microcode) and high order language (HOL) code which defines the flow and structure of the program. The horizontal microcode primitives are routines which control the flow and manipulation of data through the pipeline. This is where the redundant characteristics of signal processing algorithms are combined with the specialized pipelined hardware architecture to achieve high throughput. The programming process is highly complicated and is slow and costly making the reuse of microcode highly desirable. Each primitive performs a simple function such as the adding or multiplying of two buffers of data, and is controlled by a table of pointers and parameters which indicate the desired processing and the location and size of the data to be processed. This table driven feature of microcode increases reusability by making the primitives more general purpose.

The HOL code implements the simple constructs needed for signal processing applications and contains the structure, control, and flow of the entire program. It calls the microcode primitives in the appropriate sequence to perform the signal processing algorithms and supplies each primitive with the table of parameters it needs. The HOL code is much less complicated than the microcode and is analogous to a high level language which uses a software defined instruction set. It facilitates the development of large complex programs through structured design.

Hughes SP Software Validation Process

Signal Processing software is developed at HUGHES by integrating both the design and test phases through the use of simulators (computer programs which model a software or hardware design) at each level of development. With simulators, designs can be tested from the top down. Before the design proceeds to the next level it is tested to identify oversights, determine the feasibility of uncertain design parameters and to measure system performance. This helps eliminate errors as soon as they are generated rather than allowing them to remain in the system and waste further development efforts. Software validation testing (which typically contributes to over 50% of signal processor software development costs?1O) is much simpler because most errors (particularly the big ones) have already been corrected. Benefits continue after delivery because the software contains less errors and has received fewer modifications and is therefore more reliable. The increased reliability encourages reuse of the software.

A little testing during the design saves a lot of testing later on. Considerable reductions in development costs can be obtained by detecting errors early. The cost of correcting software errors increases rapidly with respect to the development of a project. One source has determined the following relationship of error correction costs and project development.(*2)

Phase of Project Development	Cost of Correcting Software Errors
o Design	1 Monetary Unit
o Code	6.5 Monetary Units
o Test	15 Monetary Units
o Release	67 Monetary Units

To show the importance of early error detection this data will be used to generate a simple example. Assume that there are 100 errors in a software system. Assume that at each phase of development we can increase the detection of remaining errors from 50% to 60%.

	50% EFFICIENCY (COST) X (# ERRORS)	60% EFFICIENCY (COST) X (# ERRORS)
Design	1 X 50 = 50.0	1 X 60 = 60
Code	6.5 X 25 = 162.5	6.5 X 24 = 156
Test	15 X 13 = 195.0	15 X 10 = 150
Release	67 X 12 = 804.0	67 X 6 = 402
	Total Cost = 1,211.5	Total Cost = 768.0

$$\frac{1211.5 - 768.0}{1211.5} = 0.37$$

This shows that an increase in error detection ability of only 10% could result in a 37% reduction in error correction costs!

Early error detection and sound, reliable engineering are the primary motives for the use of simulation in a software development methodology which integrates the design and test phases (Figure 1). At each level of design simulators provide estimates of performance without requiring the design of lower levels.

2 Implementing Software Inspection Course notes, IBM Systems Sciences Institute, IBM Corporation 1981.

1 Randall W. Jensen and Charles C. Tonies, Software Engineering, Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1979 pg. 331.

The functional simulator as shown in Figure 2, is the basic design tool for the systems engineer, providing him with rapid feedback on the feasibility and efficiency of new or uncertain designs. Information regarding processor loading, bus loading, throughput, memory loading, and memory utilization can be obtained. The simulator may include the capability to simulate the environment in which the system is intended to operate, and thus verify that the system will be able to perform the function required by the customer. Use of the functional simulator forces the complete definition of each functional block and interface, and helps uncover costly oversights in the top level design. It allows for the testing of the designs rather than the implementations by utilizing the simplicity and convenience of high order programming languages. Different algorithms are analyzed and design tradeoffs are determined quickly. Problems can be identified early and the designs perfected prior to their costly implementation at more detailed design levels. Instead of allowing an incomplete interface or a bad algorithm to be implemented in HOL code or microcode, many such errors can be detected and corrected soon after their origination. This saves money and improves the quality, clarity, maintainability, and reliability of the code as it reduces the frequency of its modification.

The signal processing subsystem development encompasses the implementation of the required signal processing functions in application software (consisting of HOL code) and microcode primitives which will be delivered to run in the weapons system. During this level of design, the signal processing tasks are partitioned into subtasks and the needed microcode primitives are defined. The high level behavioral instruction simulator as shown in Figure 1 performs a simulated execution of the HOL code by calling the appropriate sequence of microcode primitives which are each implemented using a high order programming language but perform arithmetic in the exact manner of the target machine.

Sets of test data (known as "test vectors") corresponding to the inputs and outputs of each functional block in the subsystem are generated by the functional simulator or are recorded from line scenarios. These test vectors provide the input and expected output data used to test HOL code routines on the instruction simulator. These tests can be performed prior to integration and prior to the development of the microcode primitives, however, some tradeoffs exist which forbid the complete isolation of the HOL code and microcode design processes. The High Level instruction simulator allows for the comparison of different HOL code designs, provides more detailed loading and throughput estimates, and supports debugging of HOL code by allowing the tester to examine the simulated coefficient and data memories during the simulation. This simulator also used to develop test vectors for the microcode routines.

The microcode behavioral simulator is an essential element of the signal processing software design and test system. It is initially created during the architecture design of the signal processor hardware at the register transfer level (RTL), and leads to the gate level design. It models the complete detail of the architecture needed to execute microcode in a manner identical to the hardware itself. Every data path and its associated time delay is simulated along with the contents of every register. It provides extremely effective debugging capabilities by allowing the microprogrammer to inspect the contents of any register during any given clock cycle of the simulated execution of the program. Additional features such as the flagging of registers whose contents changed from one clock to the next enhance the simulator's usefulness.

The importance of the microcode simulator in the validation process lies in its effectiveness in detecting errors. Microcode is highly complex because it requires timing critical microcode programming of a pipelined data path. Every

attempt is made to exploit characteristics of the specialized hardware which will result in greater efficiency. The simultaneous execution of different functions throughout the pipeline makes the microcode difficult to document and maintain, consequently modifications often become rewrites. The microcode simulator provides the capability to perform testing on the primitives during design so that bugs can be eliminated while the program is still fresh on the programmer's mind.

Typical microcode primitives are small and the functions they perform are kept simple so that despite the complexity of microcode development, they can be tested thoroughly using the microcode simulator. Because of the low data dependent branching probabilities of signal processing algorithms, the primitives rarely contain more than three branches hence all the possible logical paths and data extremes are readily testable. Test vectors generated during higher level simulation may be used to verify the functionality of a primitive under test. This comprehensive testing leads to highly reliable components with error densities typically 4 times lower than general purpose HOL software where such techniques are not applied.

Following the microcode simulation, the application code is integrated and validated by executing it on the actual hardware and using test vectors identical to those used to validate the HOL code on the instruction simulator. This testing insures that the functional blocks interface correctly, the microcode is performing as required, and the overall subsystem is exhibiting the prescribed input/output characteristics. After these tests are completed the SP applications package is integrated with other subsystems and the overall system is validated using system level test vectors. Finally, the weapons systems is taken into its real environment for live testing, to measure performance, detect errors, and obtain data for analysis and design improvements.

In summary then, Signal Processing software is complex and is expensive because of its specialization to hardware and the high levels of efficiency required. Simulators are effective debugging tools and allow the testing of high levels of design prior to their implementation at low levels. They support the simultaneous development of applications software by many programmers when target hardware is unavailable or limited. Error detected earlier are less costly to correct and the quality of the lower level designs improve because they are modified less. The reduction of latent errors in the system prior to integration speeds up more costly validation and system testing, saving time and money. A similar use of multi-level simulation is often used in the development of VLSI (Very Large Scale Integrated Circuit) and VHSIC (Very High Speed Integrated Circuit) hardware where redesign after fabrication is prohibitively expensive.

One important side benefit of simulation during software development is its usefulness in analyzing new algorithms and problems encountered during validation and system testing. On one HUGHES project where a fixed point signal processor was used, identical test data was run both on an instructional simulator and a floating point functional simulator and then compared. This provided information about the effects of the fixed point arithmetic on the performance of the algorithms. Data obtained during non-simulated environment testing can be analyzed on the simulator to determine if a problem encountered during was due to a hardware malfunction, a software error, an algorithm or implementation insufficiency, or some factor in the environment. New algorithms created to improve system performance can be tested on the simulators against test vectors which incorporate non-simulated environment data. Thresholds and other design parameters can be adjusted for optimal performance in the real environment.

Another benefit of the integrated design and test method using simulators occurs in the reuse of software. The high reliability of microcode primitives when tested on the microcode simulator coupled with their flexibility obtained because they are table driven, makes them prime candidates for reuse. Nearly all signal processing applications require basic functions such as subtraction or complex multiplication and many applications perform other common functions such as spectral analysis, thresholding, and FIR (finite impulse response) filtering. Because of the high cost of developing microcode, reuse becomes very desirable. A library of microcode primitives developed on one project can be used for the next application. If new microcode primitives are required they can be designed, tested and added to the library. Thirty-five percent of the microcode primitives used on a recent project at HUGHES were designed and tested during a previous project. Every one of these reused routines has provided 100% error free service, and

none of them have incurred any error correcting costs.

Designing for Reuse

Experience at HUGHES with the reuse of signal processing software, particularly microcode, has resulted in the formulation of some design principles which encourage reusability. The first two principles have already been explained but will be listed for completeness.

1. Table Driven Primitives

By making the microcode primitives parameter driven, they are more general purpose and can be used for a greater variety of applications.

2. Comprehensive Testing at the Module Level

Through the use of a microcode simulator, highly reliable testing may be performed reducing maintenance and error correction costs of reused code.

3. Modularity

Software modules which are small in size and perform a single simple function are more likely to be used again. Large numbers of small, single function routines can be arranged in many different sequences to perform a variety of functions. Small and simple modules are much easier to test, particularly when implemented in microcode where exhaustive testing is crucial to reliability. The number of extremal conditions which require testing increases dramatically with the size or complexity of the primitive. Routines which perform a single simple function are more likely to be defined, understood, and documented properly and considered for use during another design.

Increased flexibility and reliability within a project can be obtained by considering the entire software task during the process of partitioning tasks. For example, the implementation of a radix 64 FFT by three successive executions of a radix 4 butterfly requires the testing of only one radix 4 microcode primitive and provides flexibility to implement any FFT with a power of 4 radix. A radix 4 butterfly is much smaller and simpler to implement than the radix 64 FFT and it can be tested far more easily. An insignificant increase in execution time occurs because the butterfly must be executed three times, but this is outweighed by the other advantages. Thus, by exercising modularity in the design, reusability is fostered by improving flexibility and reliability.

4. Complete Definition

Documentation on each software module should include a functional definition which accurately describes the function of the module to someone unfamiliar with it. The definition should be self sufficient (not requiring inspection of the code), explain each element in the parameter table, and should describe the module's behavior on all extremal values. An example of such documentation is given in Appendix A.

5. Standarized Interfaces

Standardization and generalization of software interfaces encourages reuse of routines both within and between projects. Modules with parameter driven input and output data specifications are more flexible allowing for their use in multiple arrangements and applications.

6. Performance Records

Comprehensive records of the test vectors used during validation tests should be kept for every software component, describing which functions were tested and how the tests were performed. Knowledge of a module's performance (particularly error rate) and the conditions under which service was given, along with its history of reuse is helpful in determining the reliability and transportability of software being considered for reuse.

7. Library Configuration

A firmware library consisting of each microcode primitive and its documentation should be configured so that programmers within and between projects can reuse each others work.

An Example of the Effectiveness of Simulation Testing and Software Reuse in Signal Processing Applications

Although the complete effects of software reuse and simulation during design are not easily measured, some estimates have been made for a recent HUGHES project in which the signal processing subsystem was delivered on time and within budget. In this project a signal processor was embedded in a guidance and control system to perform multimode high throughput signal processing tasks. Table I shows the estimated savings achieved due to simulation testing and microcode reuse.

TABLE I

Estimated savings which occurred on a recent project at Hughes due to reuse of microcode and the use of simulators to perform tests on designs at each level.

Source of Subsys.	Additional cost if Software testing not performed		of S.P had not been reused and simulation Monetary Units Budget
		Coding of reused microde	30.2
7% Error correction at SP Subsystem Level	67.2	16% Error correction at system test level*	117.1
27%		TOTALS	214.5
50%			

*System testing was not included in the SP subsystem budget.

Thirty-five percent of the microcode primitives used in this project were lifted directly from the previous project on which the same signal processor was used. This reuse reduced the microcode design effort by an amount equivalent to 7% of the entire SP software development budget. Each of these proved to be 100% reliable as none of them had any errors reported against them. Combined with the use of simulators to test the designs at each development level, this improved reliability achieved a 46% reduction in software module error densities! This reduction in the cost of correcting errors saved another 16% of the signal processing budget. The largest cost reduction occurred during system level testing in which the decrease of latent errors in the signal processing software resulted in savings to the project which totaled over 27%. (These savings occurred outside the SP subsystem budget.) A net total of savings to the project of 50% of the SP budget can be attributed to the reuse of microcode and the use of simulators to test software designs at various levels of the development process.

Summary and Recommendations

Significant reductions in development costs have been achieved at HUGHES through the reuse of firmware and the use of automated validation tools. The specialization of signal processing software needed to utilize the hardware's ability to exploit the redundant and simplistic characteristics of signal processing algorithms seems to oppose its reuse. By making microcode modules table driven and through the use of modular programming techniques, a library of reusable firmware components can be made which exhibit the flexibility needed for reuse while maintaining the specialization needed for efficiency. Simulation at the system and software levels validate the designs from the top down and reduce error correcting costs by detecting errors early. Software modules which are made small and simple can be tested thoroughly and increased reliability obtained.

Bibliography

(1) Horowitz, e., J.B. Munson, "An Expansive View of Reusable Software", IEEE Trans. On Software Engineering, Vol. SE-10, Number 5 (September 1984), pp. 477-487

(2) Jensen, R.W., C.C. Tonies, Software Engineering, Englewood Cliffs, New Jersey, Prentice-Hall, 1979.

(3) Jones, T.C., "Reusability in Programming: A Survey of the State of the Art", IEEE Trans. On Software Engineering, Vol. SE-10, Number 5, September 1984, pp. 488-494.

(4) Pian, C.K., H. Habereder, "Signal Processing Through Macro Data Flow Architecture," NAECON 85 Proceedings, to appear.

(5) Trujillo, E., H. Habereder, "Multi-application Signal Processing Architectures", NAECON 84 Proceedings, Vol. 1, pp. 190-200.

APPENDIX A.

An example of a description for a typical microcode primitive

****

FIRN

GENERAL DESCRIPTION

FIRN is a N TAP FIR (finite impulse response) filter. The function is realized by the following equation:

$$Y(j) = \sum_{k=1}^{K=N} C(K)*X(j+K-1) \quad j=1,M$$

where X(j) and Y(j), are the respective inputs and outputs to the FIR filter. In order to prevent overflow, scaling should be incorporated into the FIR filter coefficients.

STORAGE, PROCESSING AND TIMING

The number of instructions: 20
The number of clocks : 15 + M * (B + 4N), where N is the number of filter coefficients and M is the number of output elements in one channel of the output buffer.

INPUT:
Register Usage:

CS Address	AG Reg.	Function
	AGF	Return address;
	AGE	Label FIRN01;
	AGD	Parameter table address;
	AGC	Label FIRN02
	AGB	INNER LOOP COUNTER
	AG7	CSPTR,CS BUFFER POINTER
	AG6	INPTR, WS BUFFER INPUT POINTER
PLSA	AG4	INTOP, initial address of WS filter input buffer
+1	AG3	CSTOP, start address of filter TAP's in CS
+2	AG2	OUTTOP, initial address of WS filter output buffer
+3	AG1	N, number of TAP's in filter
+4	AG0	M, number of elements in one channel of output buffer

DATA BUFFERS
WS FIR filter input buffer
CS FIR filter TAP's table

OUTPUT:

WS FIR filter output buffer

PROGRAMMER: MAX THE MICROCODER
DATE CODED: 7 APR 1982
DATE OF LAST REVISION: N/A ll 6.5i

RESUME

J.L. HANS HABEREDER

MANAGER
SIGNAL PROCESSING DEPARTMENT
Data Systems Division

EDUCATION

MA, Mathematics, Astronomy Washington State University Award years: 1969, 1971

BA, Mathematics University of California, Riverside Award Year: 1967

EXPERIENCE

8 years at Hughes

As manager of the 110 man Signal Processing Department, he is responsible for development of sonar systems and signal processors including hardware, software and applications engineering for radar, sonar, communications and electro-optical systems.

Previously, as technical director of the EMSP program, Mr. Habereder has led the Hughes technical effort to define the hardware architecture and software concepts for EMSP. Prior to this as head of the Signal Processing Analysis Section, Mr. Habereder's organization was responsible for the development of Minipro application programs for MK48, ADCAP, AN/SQS-53, LANTIRN, and TIES. His responsibilities also included the development of state-of-the-art support software for signal processors, signal processor diagnostics, fault detection/fault isolation firmware, and signal processor architecture.

Mr. Habereder also participated in system design, signal processing, and mathematical algorithm development. He performed analyses of voice data compression techniques, transform domain data analyses, mathematical modeling, and software reliability.

Earlier, at the Technische Ingenieur Schule in West Germany, Mr. Habereder served as professor of mathematics and astronomy. He taught computer architecture and software reliability courses. He also taught a wide range of mathematics, physics, and astronomy (including radio astronomy signal processing) courses.

PUBLICATIONS

Author of 6 papers in the field of information science, signal processor architecture, data flow technology, astronomy, an introductory text on astronomy, and a paper on education.

ACTIVITIES

Member of Pi Mu Epsilon, Signal Processing Curriculum Advisor, California State University at Fullerton, Instructor of Hughes sponsored Advanced Technical Education Program courses in Engineering Mathematics and Discrete Systems Analysis.

HUGHES

SOFTWARE VALIDATION OF SIGNAL PROCESSING SYSTEMS AND ITS IMPACT ON REUSABILITY

HANS HABEREDER,
SIGNAL PROCESSING DEPARTMENT

SIGNAL PROCESSORS

HUGHES

SIGNAL PROCESSOR	GP COMPUTER
• TAILORED ARITHMETIC CAPABILITY (4 ADDS/2 MULTIPLIERS)	• STANDARD ARITHMETIC (1 ADD 1 MULTIPLY-MULTIPLIES OFTEN DONE VIA ADDERS)
• CACHE MEMORIES	• BULK MEMORIES
• HIGH THROUGHPUT & CLOCK RATE 30 Mhz → 240 Mhz	• 500KIPS - 4MIPS
• ISA IS PIPELINED	• NON PIPELINED
• MEMORY ACCESS IS CONVOLUTED	• LINEAR
• TAILORED I/O, HIGH THROUGHPUT	• GENERAL PURPOSE I/O, MEDIUM THROUGHPUT
• MULTIPLE ADDRESSABLE MEMORIES	• SINGLE ADDRESSABLE MEMORY

SIGNAL PROCESSOR vs GP COMPUTER SOFTWARE

HUGHES

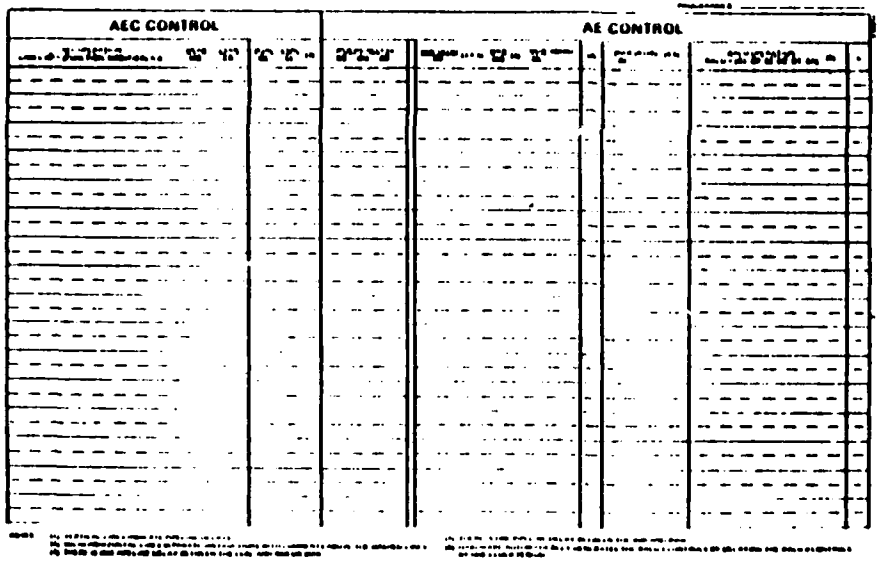
	SIGNAL PROCESSOR	GP COMPUTER
PURPOSE:	HIGH THROUGHPUT	EASE OF PROGRAMMING
PROGRAMMABILITY	MAINLY MICROCODE, SOME ASSEMBLY DEPENDING ON THE ARCHITECTURE, (MANY SIGNAL PROCESSORS ARE ONLY PROGRAMMABLE AT THE MICROCODE LEVEL) OR HOL	HOL
SUPPORT SOFTWARE	LIMITED	EXTENSIVE
COST OF SOFTWARE	VERY HIGH	MEDIUM
ISA	PIPELINED	LINEAR
FIXING BUGS IN PROGRAMS	THROW AWAY AND START OVER	PATCH
DISCIPLINE REQUIRED TO PROGRAM	SIGNAL PROCESSING SPECIALIST WITH HARDWARE EXPERTIENCE	PROGRAMMER
TYPICAL CODE	HIGHLY CONVOLUTED	HIERARCHICALLY DECOMPOSED
PROGRAMMING PRODUCTIVITY	45 LINES/MM	400 LINES/MM

MAKING SIGNAL PROCESSOR MICROCODE REUSABLE

HUGHES

SIGNAL PROCESSOR

MICROPROGRAMMING STATE TABLE



MICROCODE PRIMITIVES:

- TABLE DRIVEN
- STANDARD PARAMETER LISTS
- PARAMETERS STORED IN SEPARATE MEMORY
- THOROUGHLY TESTED AND DOCUMENTED

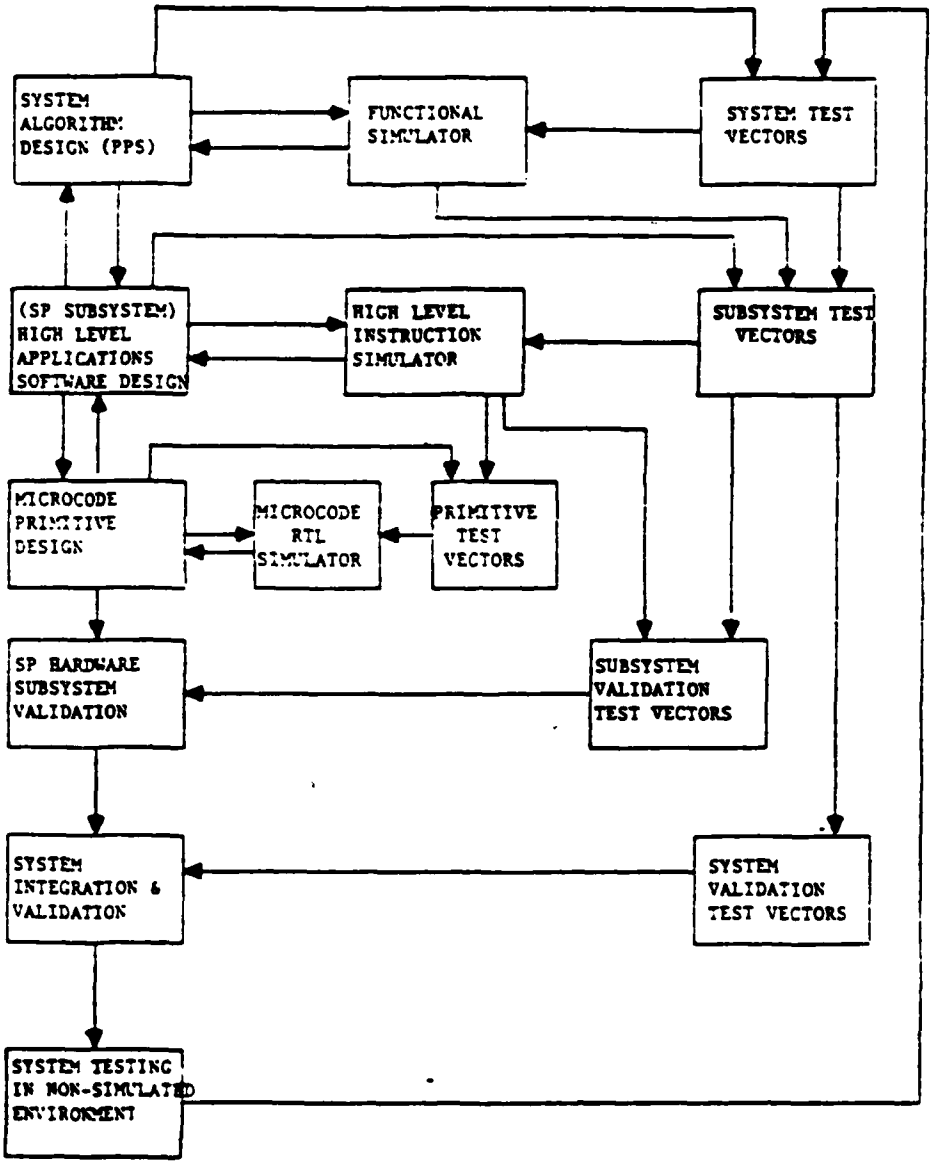
A MICROCODE PRIMITIVE LIBRARY ENTRY IS AN "INSTRUCTION" IN THE SIGNAL PROCESSOR HOL

HUGHES

```
                              FIRN

GENERAL DESCRIPTION

      FIRN is a N TAP FIR (finite impulse response) filter.  The
    function is realized by the following equation:

                          N-1
                Y(j)= SUM  C(n)*X(j-n-1)      j=1,M
                          n=0

    where X(j) and Y(j), are the respective inputs and outputs
    to the FIR filter.  In order to prevent overflow, scaling
    should be incorporated into the FIR filter coefficients.

STORAGE, PROCESSING AND TIMING

         The number of instructions:  20
         The number of clocks       :  15 + M + (8 + 4N), where M
    is the number of filter coefficients and M is the number of
    output elements in one channel of the output buffer.

INPUT:
  Register Usage:

  CS address      AG Reg.     Function
  ----------      -------     ------------------------
                    AGF       Return address
                    AGE       Label FIRM1:
                    AGD       Parameter table address
                    AGC       Label FIRM2
                    AGB       INNER LOOP COUNTER
                    AG7       CSPTR-CS BUFFER POINTER
                    AG6       INPTR, MS BUFFER INPUT POINTER
      PLSA          AG4       INTOP, initial address of MS filter
                              input buffer
        +1          AG3       CSTOP, start address of filter TAP's
                              in CS.
        +2          AG2       OUTTOP, initial address of MS filter
                              output buffer
        +3          AG1       N, number of TAP's in filter
        +4          AG0       M, number of elements in one channel
                              of output buffer
  DATA BUFFERS
    MS FIR filter input buffer
    CS FIR filter TAP's table

OUTPUT:

    MS FIR filter output buffer

 PROGRAMMER: ASK THE MICROCODER
 DATE CODED:  7 APR 1982
 DATE OF LAST REVISION: N/A
```

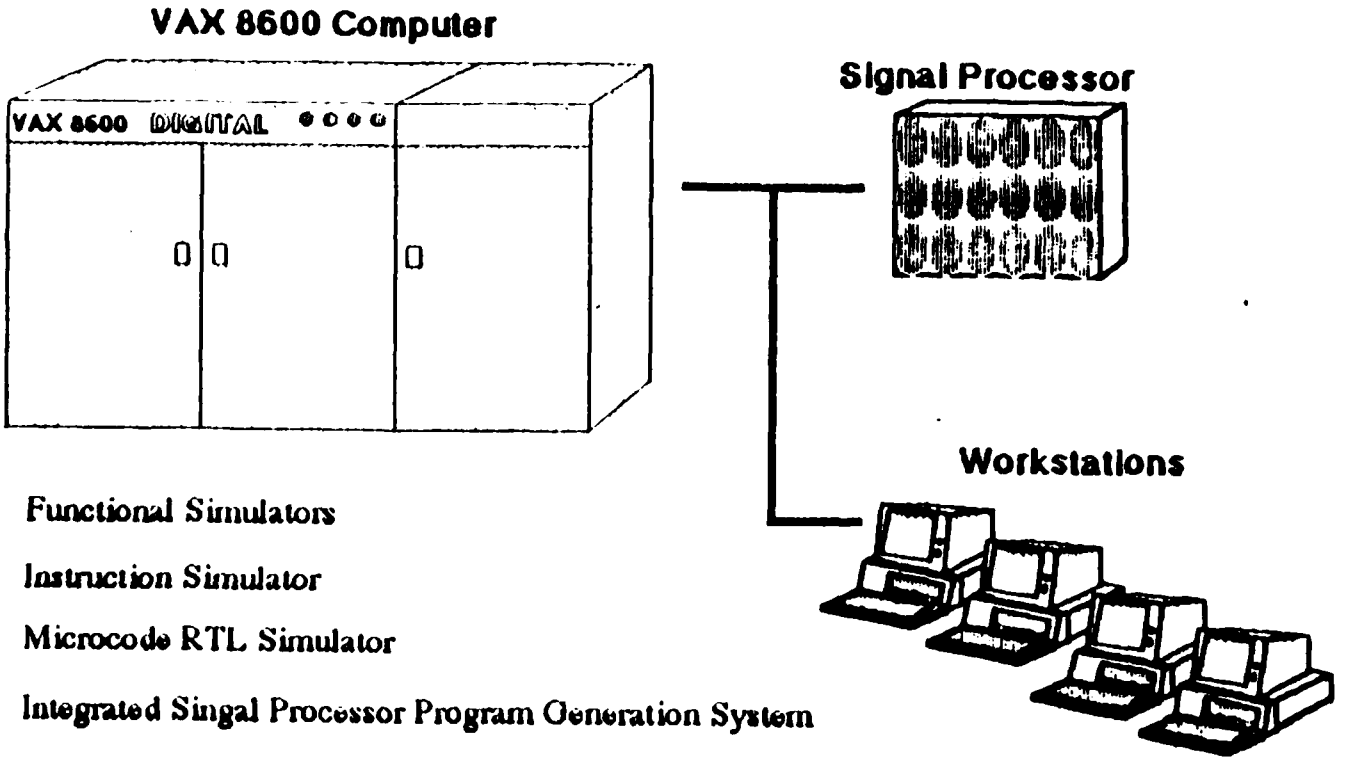
INTEGRATED DESIGN AND TEST METHOD FOR SIGNAL PROCESSING SOFTWARE DEVELOPMENT

HUGHES



Mainframe-based simulation and analysis tools minimize design errors

HUGHES AIRCRAFT COMPANY
Ground Systems Group



Functional Simulators

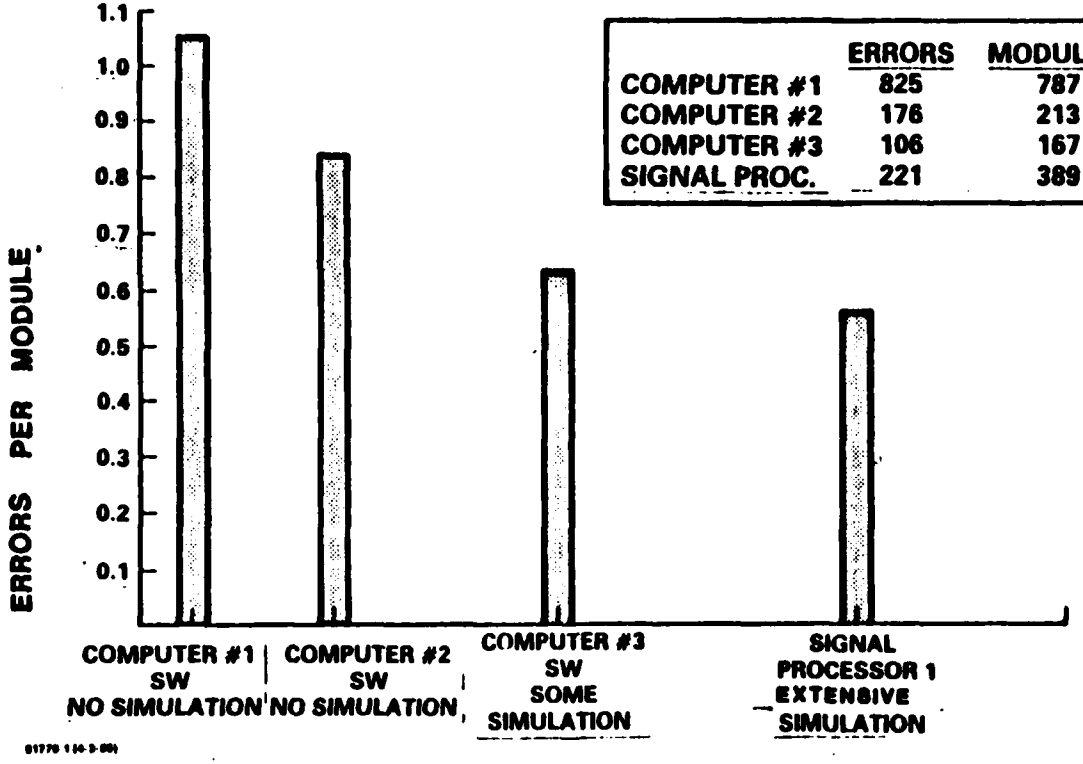
Instruction Simulator

Microcode RTL Simulator

Integrated Singal Processor Program Generation System

REDUCTION IN ERRORS THROUGH USE SIMULATION

HUGHES



	ERRORS	MODULES
COMPUTER #1	825	787
COMPUTER #2	176	213
COMPUTER #3	106	167
SIGNAL PROC.	221	389

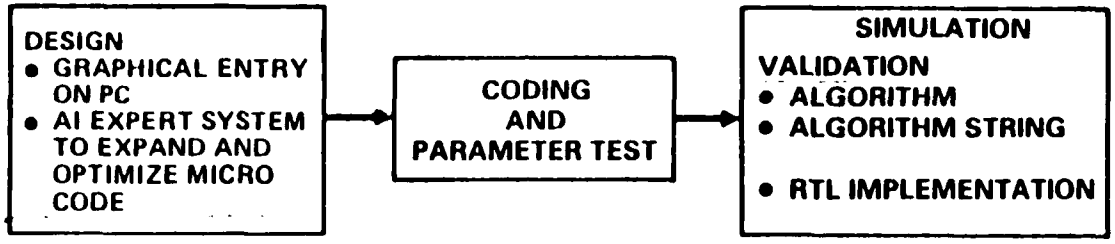
$ BOTTOM LINE SAVINGS ATTRIBUTABLE TO SIMULATION ON THE SIGNAL PROCESSOR

HUGHES

TOTAL # OF SP ERRORS	221
TIME REQ'D TO FIX EACH ERROR	3.5 MAN DAYS
NORMALIZED DIFFERENCE OF ERROR DENSITIES	0.84
NO SIMULATION vs SIMULATION	
$ SAVINGS ATTRIBUTABLE TO SIMULATION	$211K
$ SAVINGS IN SYSTEM TEST	$168K
FIELD TEST	$200K
(5 RUNS AT $40K/RUN)	
TOTAL $ SAVINGS	$579K
ORIGINAL COST OF SP SOFTWARE	$1,350K
PERCENT SAVINGS	42.9%

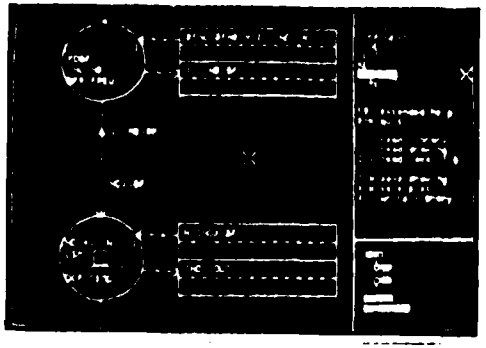
FUTURE REQUIREMENTS FOR LOW COST REUSABLE SIGNAL PROCESSING SOFTWARE

HUGHES



NEAR TERM REQUIREMENTS

- GRAPHICAL ENTRY OF SIGNAL PROCESSING MACROS
- EXPERT SYSTEM TO AUTOMATICALLY GENERATE OPTIMIZED HORIZONTAL MICRO CODE



REUSABLE SOFTWARE IN SIMULATION APPLICATIONS

Frederic D. Heilbronner

Technical Manager for Microtechnology
Advanced Technology
2711 Jefferson Davis Highway
Arlington, VA 22202
(703) 553-7128

Any developer of software for a mission-critical embedded resource is faced with the problem of representing the operational environment of the software for design, development, testing, validation and verification purposes. As very few embedded resources (almost none) operate entirely in isolation, software developers, designers, researchers and testing agents generally turn to simulation software as the best way of representing the operational environment. Indeed, this concept holds true for the majority of hardware development as well.

The problem, as the development community operates today, is that each time a simulation need is identified, a new piece of simulation software is usually developed. Examples of this practice abound. Each development and testing facility (even facilities within the same service) has its own, home-grown, curved-earth geometry target generation programs. There are at least three known simulations of the SDC MK72 radar in existence, each one separately developed at great cost. In these cases, the results output from each simulation program should be identical. Curved-earth geometry is the same whether used to generate missile tracks or aircraft tracks. Similarly, the SDC MK72 radar only produces one set of output signals, whether they're being used as a direct feed to a combat direct system or to a radar scope for a human operator to interpret. Research has shown that the development and maintenance of simulation software runs to millions of dollars at each of several facilities, representing wasted effort and wasted dollars.

The real-time simulation applications requirement provides a unique opportunity for application of the reusable software concept. I propose the development of a Universal Simulation Library (USL), to be a repository of the accepted simulation program for each application. As part of this program, *each time a new piece of hardware* or software is delivered to the Government, the developer would be required to supply a program which would simulate the interfaces with that resource. These simulation modules could then be supplied to any development program, test facility or training school much *like GFE is new furnished for* manufacturing environments. The modules would all be designed to reusable software specifications and written in Ada, with source code supplied. This policy would enable reusability of the modules in any Ada Programming Support Environment (APSE), merely by obtaining the appropriate source code and re-compiling with that particular (APSE's Ada compiler. Thus, the USL concept would have tremendous impact in a number of areas. Time and cost savings would be realized in design, development, manufacturing, validation, training and configuration management.

In the design and development phase of the acquisition of embedded resources, much of current efforts are focused on developing the simulated environment for the resulting resource to operate in.

This requirement lengthens lead times, increases effort and diffuses focus away from the primary job at hand. If the designer/developer could be assured of being able to put together the required simulated environment utilizing GFE programs from the USL, then any given design and development effort would be significantly reduced in scope and therefore in dollars. This would appear to be a prime area for initial large scale application of the reusable software concept. As much of the design and development work is funded on a cost reimbursable

basis, control of scope and level of effort is much more easily attained. The contracting authority can enforce a requirement to use any applicable USL supplied modules and to specifically exclude any costs associated with duplicating any simulation capability already available from the USL. A further benefit of this approach is that the procuring activity is assured that any resulting design is based on sound operating environment considerations, without needing to extensively review a contractor developed simulated environment. Analogies to this method exist through industry. One example is the common multi-meter used for testing electronic components. These are standard measurement tools, which can simulate, for example, the load placed across a resister to determine inherent resistance. When resistors are procured, the Government does not go out to the manufacturing facility, review the test procedures and inspect the design and operation of the manufacturer's test equipment. Rather, the procuring documents call for the manufacturer to supply "15 Ohm resisters" or similar language. This is then accepted as the standard by which the product will be manufactured and supplied. The same concept should be applied to embedded resources, i.e., "one target identifier utilizing input from an SDC MK72 radar". Thus, the manufacturer knows that a track generator and an SDC MK72 simulation program can be obtained from the USL which will provide the correct inputs for the target identifier. He does not first have to either program a track generator and SDC MK72 simulator nor procure an actual SDC MK72 to be able to develop the product.

Validation of embedded resources, encompassing evaluation, integration and test activities, requires that system performance be evaluated in a realistic environment. Thus, any validation activity focused on embedded resources becomes heavily dependent on simulation software for task accomplishment. Currently, validation commonly takes place as part of the verification and validation (V&V) effort. Frequently, this task is performed by an independent activity (IV&V). Currently, when V&V work is undertaken, the V&V agent must first develop the simulation package. This requirement has led to circumstances discussed previously, where each testing facility has its own independently developed simulation software for each given requirement. Thus, by making fully developed, well documented, mature simulation packages available under the USL concept, savings could be realized at each test facility, as well as improving work flow and ensuring the quality of testing efforts.

After testing, training activities have the greatest requirement for simulation applications. Due to the inherent difficulties surrounding training, particularly those involving combat situations, simulation of the operational environment becomes crucial to the success of the training effort. This problem is particularly evident in the area of target generation and fire control. Thus, each new training application currently has its own, custom developed simulation package developed, even though the particular application may be utilizing other resources which have been simulated multiple times in the past. The USL concept would not only make the development of training environments less costly, but would assist in assuring higher quality, consistency, and reduced development lead time.

The final major area of impact is the configuration management function. Currently, if a change is made in the way a particular resource operates, any number of simulation packages must be modified. In the example outlined previously, if the standard output of the SDC MK72 radar is changed in some way, a minimum of three separate simulation programs, at three separate locations, must be modified, using three separate systems and programming staffs. Not only does this approach cost at least three times as much as it should, it increases the likelihood of error and of mismatched configurations by a factor of three.

In sum, by using real-time simulation applications as an initial, large scale target for the implementation of the reusable software concept, several benefits would be realized. First, the application can be tightly controlled, due to its visibility in the contracting process. Second, the sources of "deposits" in the library are clearly identified and linked with other product deliveries. Third, the applications have a high degree of visibility in the development community, giving the opportunity to solve an existing, rather difficult problem with reusable software, thus "proving" the concept and encouraging its use in other areas.

RESUME

b. "FREDERIC D. HEILBRONNER"

Technical Manger for Microtechnology
Advanced Technology
2711 Jefferson Davis Highway
Arlington, VA 22202
(703) 553-7128

SKILLS SUMMARY

o	Accounting	11 Years
o	Financial Management	5 Years
o	C/SCSC	4 Years
o	Computer Programming	11 Years
o	Systems Analysis and Design	6 Years
o	Scheduling (PERT,CPM)	5 Years
o	CPR Analysis	3 Years
o	Program Management	5 Years
o	Proposals	4 Years
o	ARTEMIS	5 Years
o	Microcomputers	3 Years

EXPERIENCE

Currently, Deputy Project Manager, LVT7A1 Program and Technical Manager for Microtechnology. Responsible for the day-to-day schedule, financial and performance aspects of the Company's support to PMS310 on the LVT7A1 Program, reporting to the Company's Project Manager. Responsible for establishment of a Microtechnology Group to support the development of microcomputer applications in the areas of logistics, financial management and engineering. Eleven years of experience in accounting, computer programming and general management in both commercial and government environments. Five-plus years experience in project and program management, specializing in the analysis, design, implementation and operation of automated management information systems (MIS) for scheduling, financial management and other project related functions. Certified Public Accountant in the Commonwealth of Virginia.

Reviews all LVT7A1 Program deliverable products to assure high quality of final documents. Prepares resource utilization plans for meeting contract objectives. Participates in preparation of Cost Performance Report (CPR) Analysis based on monthly CPR submitted by hardware contractor. Performed complete MIS analysis, design and implementation during project start-up. MIS includes automated CPR analysis (with complete graphics), GFE/GFM status tracking, vehicle location, transportation schedule and shipping instructions, ECP status tracking and financial management systems.

Performed system review and prepared system master plan for an automated project management system for the Army's VIABLE Project Management Office. Directed efforts of three programmers/analysts to perform complete redesign and reprogramming of existing applications and design and implementation of new applications. Achieved average reduction of fifty percent in turnaround time in existing applications and tripled automated functions with no increase in personnel or machine capacity.

Developed a complete C/SCSC compliant performance measurement system for the Army's prime contractor on the Division Level Data Entry Device (DLDED) program. System was based on ARTEMIS and included complete labor distribution, scheduling and costing functions required to generate DODI 7000.1 0 compliant CPRs.

Performed system analysis, design, implementation and operation of an automated project management system for the Department of State's Security Enhancement Program. System functions included project scheduling, financial management and complete graphics. System was ultimately adapted for use by the Multinational Force and Observers in preparation for the peacekeeping mission in the Sinai, as well as the Department of State's Public Access Controls Project. Other work for the Department included development of a Case Tracking System for the Office of Investigations and redesign of the schedule for the Moscow Embassy project.

Responsible for all management aspects of small professional services company, including financial, technical and administrative matters. Performed all marketing functions including sales presentations, proposal writing, new business plan development and client liaison for performance-based marketing. Ensured all contractual requirements were met, as well as all local, state and federal reporting and tax matters. Negotiated all contracts and other financial arrangements, including credit lines and equity and debt instrument placements. Responsible for over one million dollars in sales in three year period and corporate growth from initial staff of one to staff of eight professionals.

Performed ARTEMIS system management and programming duties for the Saudi Naval Expansion Program Operations and Maintenance Augmentation project. Responsible for analysis, design, implementation and operation of the manning-based Forecasting and Estimating System. Other duties included maintenance programming for payroll and labor distribution systems.

Staff accountant specializing in government and not-for-profit entity audits using automated techniques. Further public accounting experience in records and taxation for estates and trusts and automated preparation of tax returns. Prepared statistical analyses of rail freight volume for line continuation case regarding the Delaware and Hudson Railroad for the Federal Railroad Administration.

EMPLOYMENT HISTORY

Advanced Technology, Inc. (12/84 - Present), Technical Manager, Deputy Project Manager.

AccuSystems, Inc. (12/80 - 12/84), Principal Analyst

HBH Company (10/79 - 12/80), Programmer

Deloitte, Haskins and Sells, CPAs (6/79 - 10/79), Staff Accountant

S. Walter Kaufman and Company, CPA (6/73 - 6/79), Staff Accountant

EDUCATION

BS Accounting/MIS, University of Virginia, 1979

ARTEMIS Project Management Course, 1980

Coursework in Pension and Profit Sharing, Virginia Society of CPAs, 1983

Coursework in Auditing and Small Business Accounting, American Institute of CPAs, 1984

SPECIAL SKILLS

Certified Public Accountant, Commonwealth of Virginia

Knowledge of variety of computer hardware:

IBM 303x, 360/370,	HP1000,2000
PC and compatibles	Wang OIS, VS
Apple	CDC Cyber
Osborne and CP/M machines	

Knowledge of variety of computer software:

BASIC	DBASE II & III	WordStar
COMBOL	Knowledgeman	CPM
FORTRAN	Lotus 1-2-3	MS-DOS
ALC	SuperCalc	Apple - DOS
ARTEMIS	MultiPlan	JES 2/3
Panvalet	Crosstalk	RTE IV

PROFESSIONAL AFFILIATIONS

Project Management Institute American Institute of Certified Public Accountants Virginia Society of Certified Public Accountants

Presentation to the Application Workshop
of the

Software Technology for Adaptable, Reliable Systems (STARS) Program

Naval Research Laboratory
April 1985
by
Fred Hellbronner
Technical Manager for Microtechnology
Advanced Technology, Inc.

Reusable Software in Simulation Applications

Why Choose Simulation Applications?

- Focus on Practical Implications
- Immediate and Direct Cost Savings
- Enhanced Product Quality
- Growth in Impact Over Time
- Easily Controlled
- Highly Visible

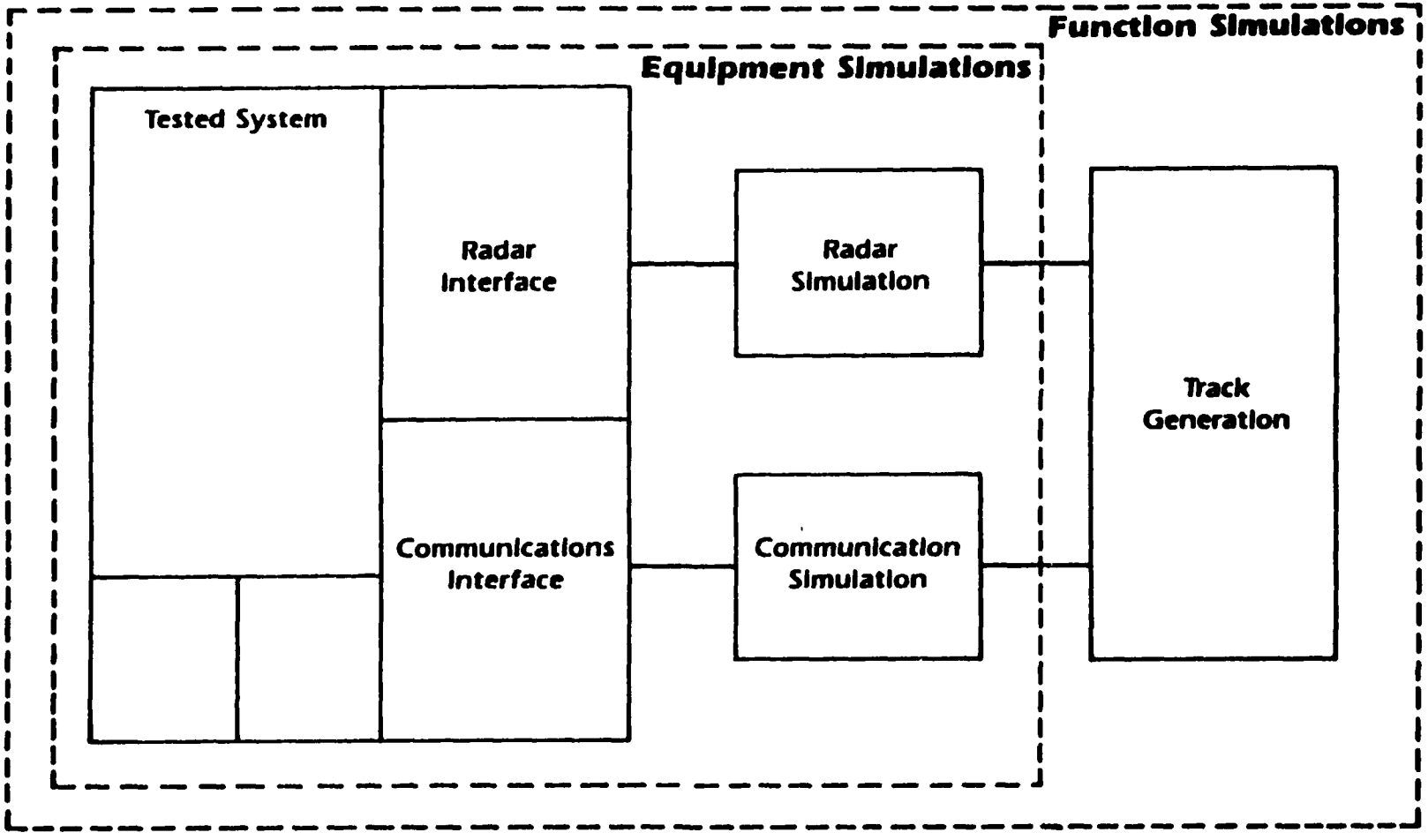
What Is Meant by "Simulation Software"?

The Applications Under Consideration Are **Real-Time, Interactive Simulations** of Equipment and Functions for the Purposes of **Testing** and **Training**

Simulation Applications Focus on Two Areas

- Equipment (Hardware)
  - Radars
  - Electronic Warfare Systems
  - Data Converters
  - Guided Missile Launchers
  - Manned Consoles
  - Navigation Sensors, Etc.
- Functions
  - Track Generation
  - Communications
  - Flight Dynamics
  - Weapons Systems Mission Profiles
  - Stabilization
  - Heat Transfer, Etc.

Generic Simulated Environment

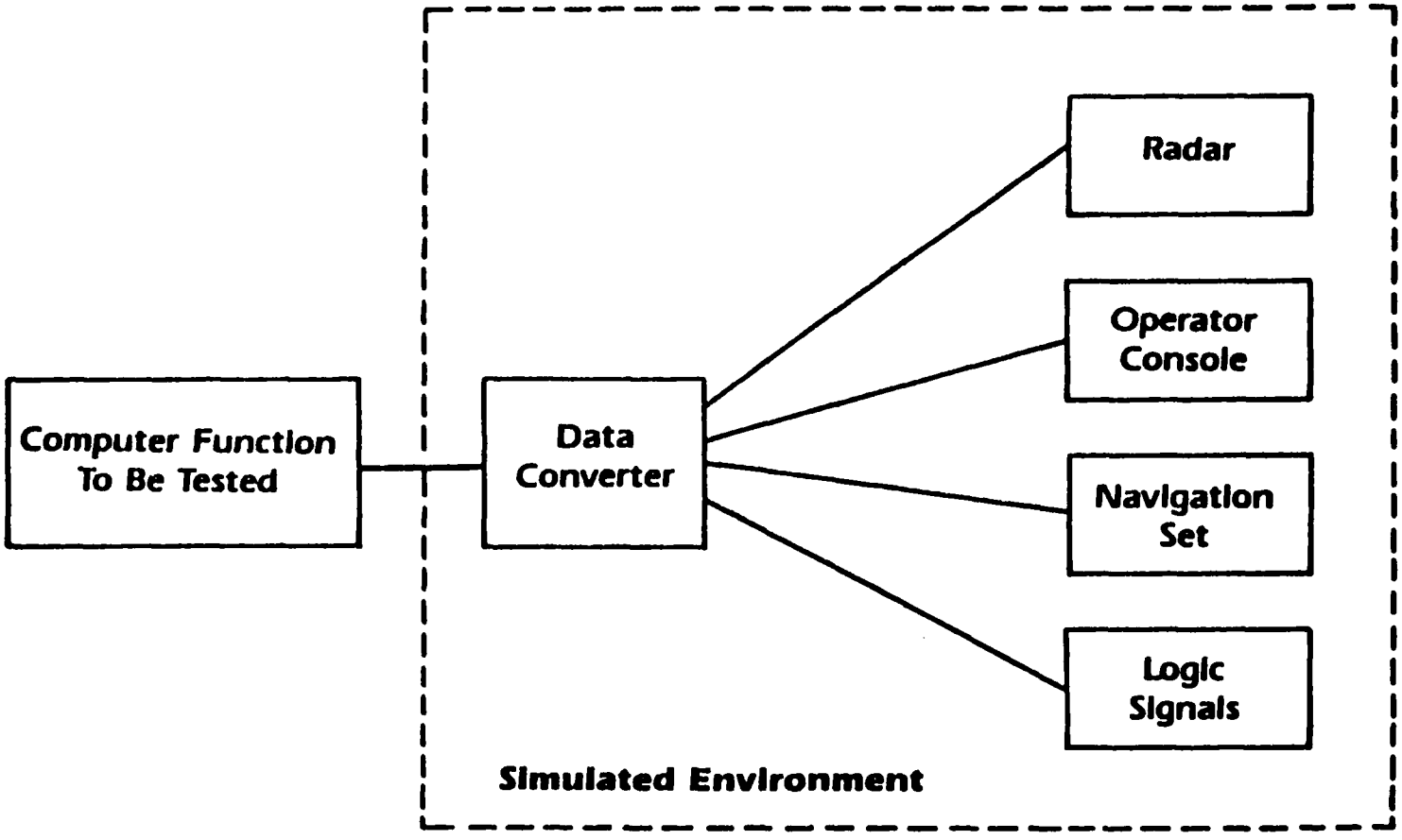


Simulation Impact on Software Development Programs

Up to 40% of the Resource Dollars:

- Testing
- Personnel Training
- Verification

Hypothetical Simulation Example



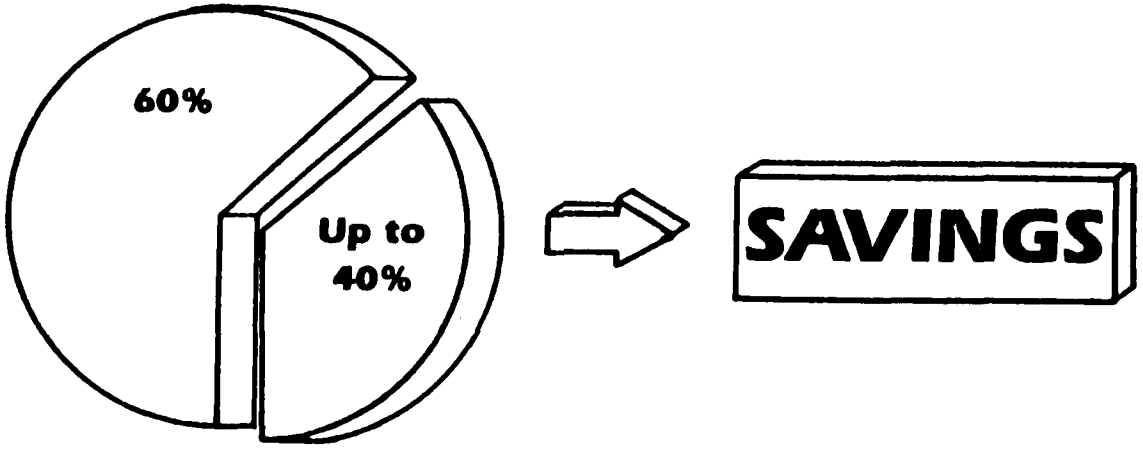
The Current Problem

Each Time a Simulation Need Is Identified, The Simulated Environment Is Re-created From the Beginning

The Solution

By Making Simulation Packages Modular and Reusable, Significant Savings Can Be Realized

Software Development Costs



Implementation

What To Do

- Simulation Packages Procured In Parallel With End Products
- Supplied to System Developers as GFE

How To Do It

- DOD Task Force
- Regulations and Instructions
- Standards Definition
  - Documentation
  - Interface
- Clearinghouse With
  - Repository
  - Distribution Mechanism
  - Maintenance Capability
- Educational Program
- GFE Orientation

REUSABLE SOFTWARE - A CONCEPT FOR COST REDUCTION

Christine M. Anderson
Marlow Henne*

Air Force Armament Laboratory, AFATL/DLCM
Elgin Air Force Base, Florida 32542

Summary

The cost of military computer systems is increasing rapidly. It is alarming to consider that not only the total cost of military computer systems is increasing, but the percentage of the total cost attributed to software is also increasing. Several underlying causes for this increase are discussed. While the new United States (US) Department of Defense (DoD) standard high order language, Ada, will significantly help to reduce the software cost growth, other solutions must also be sought. Reusable software component technology and associated parts composition systems are presented as possible solutions. Recommendations for future research are provided.

1. Software Cost Growth

Much growth in the power and sophistication of US defense systems is due to the extensive application of computers. Almost every defense system fielded today contains computers whose software performs mission-critical functions (1). Most of these are embedded computers. An embedded computer can be thought of simply as an integral component of a larger system. Defense systems or subsystems using embedded computers include sensors, electronic warfare systems, weapon system control, communications, command and control, navigation, and target acquisition.

The cumulative inventory of DoD embedded computers is projected to grow from 10,000 in 1980 to nearly 260,000 by the end of the decade. While hardware quantity is increasing rapidly, the percentage of the total dollar attributed to software will increase from 65 percent of the total in 1980 to 85 percent of the total by 1990. This translates to a staggering projection of $32 billion (absolute dollars) for annual DoD embedded computer hardware (2).

There are several underlying causes for the increase in software cost over the past decade and the projections cited above. A prime reason is the large variety of programming languages what has evolved. By 1974, there were over 400 languages, dialects and subsets of which few supported modern methodologies for structured program development, making maintenance (accounting for 65% of the software life cycle cost (3)) a nightmare (4).

Another contributing factor to the increased expenditure in DoD embedded computer software is inherent in software itself--its flexibility. DoD has been increasingly exploiting software's flexibility in developing modern weapon systems. As reported by the USAF Scientific Advisory Board, software "can embody and implement abstract operational concepts; it has no manufacturing cycle or cost; it can be modified quicker and cheaper; it does not wear out; and it can incorporate new features and functions in an evolutionary fashion without major investment in new systems and hardware" (5).

The Air Force F-111 program illustrates this point. The table below compares similar capabilities (additional offset aim pointer and updated weapon ballistics) implemented through hardware on the F-111 A/E and in software on the F-111 D/F. Given an existing software support facility, the savings due to making the changes via software rather than hardware have ratios of about 50:1 in cost and 3:1 in time (6).

Modification	Via Hardware	Via Software
Cost/Time Ratios		
#1	$5.28M/42 mo	$0.10M/16 mo
52.8:1/2.6:1		
#2	$1.05M/36 mo	$0.02M/10 mo

*Mr Henne is now at Harris Corporation, GISD Software Operations, Melbourne, Florida 32901

52.5:1/3.6:1		
#3	$8.00M/78 mo	$0.02M/15 mo
400:1/5.2:1		

Another cost contributor is the fact that software development is labor-intensive. Typically each line of a computer program is written by hand. Unfortunately, while the demand for software is increasing, the number of qualified personnel is not increasing as rapidly. The Air Force Scientific Advisory Board reported that there is a 4% annual increase in qualified software personnel, a 4% annual increase in software productivity (based on current methods) and a widening gap based on a 12% annual increased demand for new computer software (5).

In more absolute terms, the shortfall between supply and demand, currently measures 50,000-100,000 programmers, and may rise to 1.2 million by 1990 if remedial measures are not taken (7).

Another cost contributor is the problem of building reliable software. Instances of software reliability problems include false alerts for the North American Defense (NORAD) system, space shuttle's on-pad launch delay, and test missiles (and even airplanes) hitting mountains they were programmed to fly over (5). The criticality of the reliability problem is made clear by C.A.R. Hoare's warning: "The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus. It may be a nuclear warhead exploding over one of our own cities" (8). Developing reliable mission critical software where errors can translate into life or death situations is a time consuming and costly process.

To briefly summarize, the reason for the tremendous cost growth in defense embedded computer software is multifaceted. Contributing factors include: the large number of existing computer languages currently being used; the increased exploitation of software's flexibility resulting in its increased utilization; the labor-intensive nature of software resulting in a shortage of qualified software engineers; and the complexity associated with meeting the reliability requirements of mission critical software. Solutions to all but the second factor can be, and currently are being sought. The second factor is really a reflection of our "age of information" rather than a problem.One of DoD's more notable technological and managerial solutions is the development and standardization of one high order language for mission-critical computer systems--Ada. While Ada has a tremendous potential to reduce costs in all phases of the software life cycle, Ada alone is not a panacea for DoD's software problems. We cannot hope to meet the software needs of the 1990s and beyond if we continue to develop Ada software on a line-by-line basis. One approach is to develop the concept of software reusability using Ada as the cornerstone.

2. Software Componentry

The reuse of software components has the potential of reducing the shortfall of required software engineers, while increasing software reliability.

There is a great deal of interest in software component technology. The following discussion represents a collage of thinking on the subject that has been generated over the past several years.

"Having reusable software available can significantly reduce system development time. The more software that can be obtained off-the-shelf, the less new software that must be created. The risks involved are thus reduced, since off-the-shelf software should already have been well tested and debugged. Reduced time and risk enhance the probability that a major resource savings, ultimately reducing the required DoD software investment." - Commander J. Cooper, "Increased Software Transferability Dependent on Standardization Efforts," Defense Management Journal, October 1975.

"Software reuse saves development time and money, and field proven software is more reliable." - Strategy for a DoD Software Initiative, 1 October 1982.

"The Introduction of reusable software components can significantly relieve the resource demands thus assuring continued responsiveness to new threats through the introduction of new or enhanced weapon systems". - Report of the DoD Joint Service Task Force on Software

Problems, 30 July 1982.

Recommend the Air Force "Initiate a set of formal laboratory programs to define and exploit opportunities for software standardization through reusable software packages in selected application areas." - Dr. Edwin B. Stear, former Chief Scientist of the Air Force in a Briefing to General Marsh, AFSC Commander, 14 March 1983.

"Achieving reusability in mission software represents a good opportunity for dramatic productivity gains since using existing software in lieu of new software not only saves money but also saves documentation costs and test costs. And since the software has already been verified, it increases the quality of the new system." - The Army Science Board 1983 Summer Study on Acquiring Army Software.

2.1 A Hardware Analogy

There is nothing unique about component technology: it has existed for years in the digital electronics world. The uniqueness emerges when we apply this technology to software. To a great extent, it can be said that today's software design process has not progressed much beyond the early digital design methodology in the digital hardware industry.

In the late 1950's and early 1960's low level components, such as vacuum tubes, transistors, resistors, and capacitors, existed for the design engineer to use for his circuit design. A strong analogy can be drawn between these and the assembly language statements used by the software engineers of today. Each time a particular kind of circuit, such as a gate or flip-flop was needed, the engineer had to select the type of circuit and the components' values which would be used. When large quantities of a particular kind of circuit were needed on a project, the engineer could reuse his design for that project. The design of each circuit for large projects was such a shore, individual engineers kept earlier designs and tried to borrow from them on new projects whenever possible. This also led to published design compendia of circuits much as may be found today in collected algorithms and published math or utility routines.

Enterprising companies realized that significant markets existed for certain low level digital building blocks and began to develop circuit cards which contained individual gates, flip-flops, etc. These circuit cards (modules) made a significant contribution to the design of systems. By using these ready made modules, the designer was freed to do more productive system design. While these modules were helpful, a significant problem still existed when trying to use modules from different suppliers. Each supplier chose his own voltage levels for logic "one" and "zero", and other interface details were often different.

This level of modularity is close to that which we are approaching today in software. We have standard math and utility routines which exist in libraries but little else is available for general reuse.

Once the advantages of reusing hardware designs became evident, the next step was to improve the production technology for hardware modules. As integrated circuits (ICs) became possible, computer and general digital components represented the vast majority of units built. This, of course, was due to the modular nature of digital designs with their many common components and to the design methodology which had already begun supporting reusable module designs. ICs enabled entire flip-flops and other circuits, which had previously taken an entire circuit card, to be fabricated on a single IC. As production technology improved, more circuits were placed on individual ICs to form intermediate modules such as counters, adders, etc. Today, we find Very Large Scale Integrated (VLSI) circuits which represent large portions of the overall system. Since these circuits are so large in scope, most systems may not use more than one of a given circuit. Even though we may only need one circuit of a given kind, production techniques and reusability have reduced the cost of these complicated circuits to a level where they are competitive to use for a variety of applications. A good example of this is the microcomputer. Today we find full-scale digital computers, on a single IC, performing timing tasks for small appliances which previously used analog circuits or mechanical timers.

If the current trend continues toward software module production and reuse, we should see the size and use of common software

modules expand rapidly, paralleling the development of hardware modules.

Thus, an analogy can be made between the inventory of software components and the inventory of prefabricated circuits available from semiconductor manufacturers. The software developer resembles the computer designer, who determines the gross system structure and the interconnections between circuits but relies on the prefabricated components for low-level operations. Over the past few years, larger components have been developed. A similar technology is needed in the software world (9).

2.2 Software Componentry Methodology

Past critics have maintained that the software reusability concept has not flourished because individual concrete programs are too specialized to be reused. However, with the advent of Ada, we now have the means to more easily construct software components at a more abstract level. Ada's package feature will permit the creation of software packages analogous to sealed hardware components that consist of an external interface or specification and an internal body which the user cannot alter. Ada's generic feature extends the package concept to include a parameterization facility for tailoring packages to particular needs.

Additionally, Ada's strong typing imposes constraints on module interconnections and allows consistency between formal parameters of module definitions and actual parameters of module invocations to be enforced at compile time (10). These features provided by Ada for reusable software components are richer than those of its predecessors.

Studies investigating methodologies for applying Ada to develop reusable software components are only now being initiated. Thus the technique involved in developing generic packages is not well defined and almost no implementation experience exists. A recent Air Force study identified criteria which impact the reusability of software. Chief among these are application independence, modularity, simplicity and code self-descriptiveness (11).

Application independence can best be achieved via generalized data structures such as parametrically described arrays; minimal use of machine dependent constructs such as machine language code, microcode and specific I/O features; and implementation of well chosen common functions.

The components should be encapsulated in such a way that their external usage is completely defined by an interface specification, which is physically distinguishable from the implementation portion. This interface should be as firm and well defined as that for the hardware interconnection to an integrated circuit. Further, components should be orthogonal. This means that, unless contraindicated by their interface descriptions, they may be used in each other's presence (12).

Simplicity in both design and implementation, will facilitate program understanding and modification. The quantitative counts (number of operators, operands, nested control structures, nested data structures, executable statements, statement labels, decision points, parameters, etc.) will determine to a great extent how simple or complex the source code is.

Component source code should be as self-descriptive as possible. One approach to achieving this goal is to embed compilable program design language (PDL) in the source code, thus insuring up-to-date cross-correlation between design, code and documentation. By using valid Ada procedure names and declarations in addition to commentary in the PDL design, rigorous checking of the PDL can be performed.

In order to encourage use of software components once they are implemented, a systematic approach to accessing and combining particular instances of components must be pursued. This approach focuses on parts composition system technology.

3. Parts Composition Technology

A parts composition system supports the building, testing, and optimizing of programs using reusable components. This system must include, at a minimum, cataloging and retrieval facilities, a language to compose parts, a warehouse of parts, and an editing and testing facility. The Japanese have reportly achieved up to an 85% reuse rate in their software factories by

using these currently available information retrieval techniques (13). Toshiba's Fuchu Works Software Factory, specializing in real time applications, averages 2870 instructions per programmer per month (14), compared to a U.S. software productivity rate of 75 to 280 lines per programmer per month (15). Toshiba's productivity is due in large part to software reuse. A mature parts composition system will include thousands of software parts available in a common environment. In theory, a software engineer can attempt to combine any two available parts so the system must provide robust mechanisms to insure reliable and meaningful parts composition (16).

Parts composition systems may range from manual and semiautomated software parts catalogs to more advanced automated systems, systems that may even include artificial intelligence (AI) (i.e., expert systems).

An expert system is a man-machine system with specialized problem solving expertise. The first generation of AI focused on defining a general mechanism of intelligence for expert systems. The current perspective holds that the true power of the expert system comes from the knowledge it possesses, not from the particular formalisms and inference schemes it employs (17). Thus, it is essential to capture the knowledge of the application domain to be modeled.

While various domains of knowledge are being studied and common functions extracted for later component implementation, parallel investigations of methods to organize, index, describe, and reference software components must also be pursued. Further in conjunction with all of these activities, studies aimed at producing more advanced user friendly systems that change data and algorithmic representations into code, that is, software generator systems must continue.

The following example describes one scenario of a user interacting with a knowledge-based parts composition system. Assume the user is interested in locating a guidance algorithm for a particular armament electronics (harmonics) application. He asks the system for retrieval of all generic classes of guidance software that meet his requirements. The system may actually solicit (via leading questions) the requirements from him.After reviewing these generic classes, he asks the system to retrieve more detailed descriptions of certain components. Following the examination of these, a more detailed review of the specifications associated with a select few components is performed. Finally, the actual component code is examined. This retrieval mechanism can be made increasingly intelligent by providing facilities for querying the user for additional information if no component is found, by searching for related components or by custom tailoring components to match the user's request.

This example is still at a fairly low level of automation. A more advanced parts composition system would allow the user to describe (in a user oriented high level specification language) an entire subsystem's requirements (e.g., autopilot for a particular type of weapon). The system would query the user concerning critical aspects of the subsystem and then proceed to retrieve, customize and compose the necessary software.

There are several parts composition systems commercially available that offer varying degrees of aid to the user. Thus far, none have been applied to the development of weapon system software. An evaluation of these systems should be performed.

4. A Related Effort

The U.S. Air Force Armament Laboratory has just initiated a program that addresses software component technology and supporting parts composition systems. The program, Common Ada Missile Packages (CAMP), features two related study efforts: a commonality study and a parts composition study. The objective of the commonality study is to investigate the feasibility of developing reusable Ada components for armonics systems. The approach is to examine existing missile software and/or associated documentation in order to identify candidate common functions for component implementation. The second study, to be performed concurrently, features an investigation of current parts composition system technology and recommendations regarding the most practical approaches for achieving both near-term and long-range benefits. Based on the results of the studies, a follow-on implementation phase ill commence aimed at

developing a parts composition system and associated reusable software armonics components.

5. Conclusion

In closing, it should be stressed that software componentry will not evolve quickly or cheaply. Both mental and organizational road blocks must be overcome. However, the technology holds such a tremendous potential for slowing the cost growth in DoD mission-critical software, further research is imperative. This research should be aimed at joining parts composition technologists, who are often from academia, with DoD mission-critical specialists in order to achieve a fruitful blending of composition techniques and DoD knowledge domains.

REFERENCES

(1) Martin E.W., "The Context of STARS," IEEE Computer, November 1983.

(2) Electronics Industries Association Government Division, "DoD Digital Data Processing Study - A Ten-year Forecast," October 1980.

(3) Grove H.M., "DoD Policy for Acquisition of Embedded Computer Resources," Concepts, The Journal of Defense Systems Acquisition Management, Autumn 1982 Volume 5, Number 4.

(4) Deutsch R., "JOVIAL: The Air Force Software Solution in the Years Before Ada," Defense Electronics, October 1982.

(5) USAF Scientific Advisory Board, "Report on the High Cost and Risk of Mission-Critical Software," December 1983.

(6) DoD, "Report of the DoD Joint Service Task Force on Software Problems," July 1982.

(7) Boehm B.W., Standish T.A., "Software Technology in the 1990's," Appendix to Software Initiative Plan, October 1982.

(8) Hoare C.A.R., "The Emperor's Old Clothes," Communications of the ACM, Volume 24, Number 2, February 1981.

(9) Wasserman A.I., Gutz S., "The Future of Programming," Communications of the ACM, Volume 25, Number 3, March 1982.

(10) Wegner P., "Varieties of Reusability," Workshop on Reusability in Programming Proceedings, Sponsored by ITTT Programming, Stratford, Conn., 7-9 September 1983.

(11) Presson P.E., et al., "Software Interoperability and Reusability," Boeing Aerospace Company under contract to Rome Air Development Center, Griffiss AFB, NY, NY, RADC-TR-83-174, July 1983.

(12) Spector D., "Language Features to Support Reusability," SIGPLAN Notices, ACM, Volume 18, Number 9, September 1983.

(13) McNamara D., "Japanese Software Factories," NSIA Conference, Arlington, VA, May 1984.

(14) Kim K.H., "A Look at Japan's Development of Software Engineering Technology," IEEE Computer, May 1983.

(15) Zelkowitz M.V., Yeh R.T., Hamlet R.G., Gannon J.D., Basili V.R., "Software Engineering Practices in the US and Japan," IEEE Computer, June 1984.

(16) Rice J.R., Schwetman H., "Interface Issues in a Software Parts Technology," Workshop on Reusability in Programming Proceedings, sponsored by ITT Programming, Stratford, Conn., 7-9 September 1983.

(17) Hayes-Roth F., Waterman D.A., Lenat D.B., "Building Expert Systems, Addison-Wesley Publishing Company, Inc., Reading, Mass, 1983.

(18) Berard E.V., "Ada Education A Moving Target," Defense Science & Electronics, May 1984.

(19) Dolotta E.A., et al., Data Processing in 1980-1985, John Wiley & Sons, NY, NY, 1976.

(20) Boehm B.W., Software Engineering Economics, Prentice-Hall, Englewood-Cliffs, NJ, 1981.

(21) Standish T.A., "Software Reuse," Workshop on Reusability in Programming Proceeding, Sponsored by ITT Programming, Stratford, Conn., 7-9 September 1983.

(22) Bunyard J.M., Coward J.M. "Today's Risks in Software Development Can They be Significantly Reduced?" Concepts, The Journal of Defense Systems Acquisition Management, Volume 5, Number 4, Autumn 1982.

(23) "Missing Computer Software: A Bottleneck Slows New Applications, Spawns a Booming New Industry," Business Week, September 1, 1980.

RESUME

A. MARLOW HENNE

Senior Associate Principal Engineer, April 1985

Marlow Henne joined Harris in June 1984 as Group Leader of the Methodology and Language Group. His duties include planning and direction of development activities in the areas of requirements analysis, process description languages, rapid prototyping, automatic document generation, Ada technology and related compiler and environment activities. He also serves as a Distinguished Reviewer on the Evaluation and Validation Team sponsored by the Ada Joint Program Office and is a consultant for the U.S. Air Force to NATO/AGARD on Ada for real-time guidance and control applications.

PREVIOUS EXPERIENCE

Prior to coming to Harris he was Chief of the Computer Technology Section of the Air Force Armament Lab. Duties there included planning and directing the embedded computer policy and technology for the Armament Lab. Day-to-day duties included directing the development of an Ada compiler and tools for real-time guidance and control of tactical missiles, computer architecture design, interprocessor networking studies and simulations, and distributed processor architecture design, and the supervision of military and civilian engineers involved in hardware and software design. He originated the first Reusable Ada Software Packages effort funded by the STARS project and a VHSIC missile processor project. Duties of that position also included serving as the Embedded Computer Resource Focal Point for the Armament Lab. and VHSIC Focal Point for Eglin AFB. He participated in writing the Air Force (AFSC) Ada Introduction Plan for Mission Critical Computer Systems and the Air Force STARS plan. He led a project team which developed the computers used in a Conventional Cruise Missile which were also modified to be used in the F-16 upgrade. While there, he also served on several advisory panels to Air Force Systems Command, NATO and JTCG.

He has also served as a Adjunct Professor at the University of West Florida and Troy State University, teaching graduate and undergraduate courses in computer architecture, data communications, operating systems, and Ada. In 1984 he presented Ada at a short course at the University of Southern California.

At Metric Systems Corporation, he was a Senior Systems Engineer, reporting directly to the Vice-President, responsible for the design of real-time Radar signal processors, computer controlled data transmission systems, real-time industrial power control systems, and industrial plant monitoring systems. In this position he was also responsible for marketing the company's capability to the federal and state governments as well as industrial customers. Responsible for establishing a network of field marketing representatives, and directing their work.

As a Physicist with the Naval Coastal Systems Lab., he designed navigation and mine-hunting systems using Radar, Sonar, Laser and computer techniques.

In addition to the above technical experience, he has served as President of two corporations dealing in construction and land development.

EDUCATION

Candidate for Ph.D Computer Science - Florida State University

graduation expected - Fall 1985
M.S. Computer Science - Florida State University - 1982
Graduate study in Business Administration - U. West Florida - 1969-71
B.S. Physics - Florida State University - 1962

PROFESSIONAL AFFILIATIONS

IEEE - (Computer Society)

SECURITY CLEARANCE

Secret (NATO)

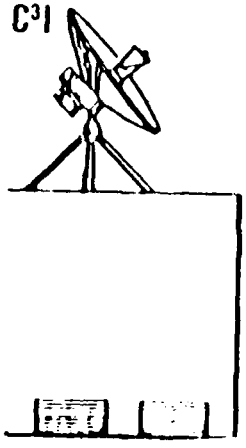
REUSABLE SOFTWARE

— A CONCEPT FOR COST REDUCTION

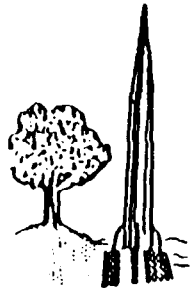
CHRISTINE M. ANDERSON
MARLOW HENNE
AIR FORCE ARMAMENT LABORATORY
EGLIN AIR FORCE BASE, FL 32542
U.S.A.

EMBEDDED COMPUTER

C³I



ARMONICS

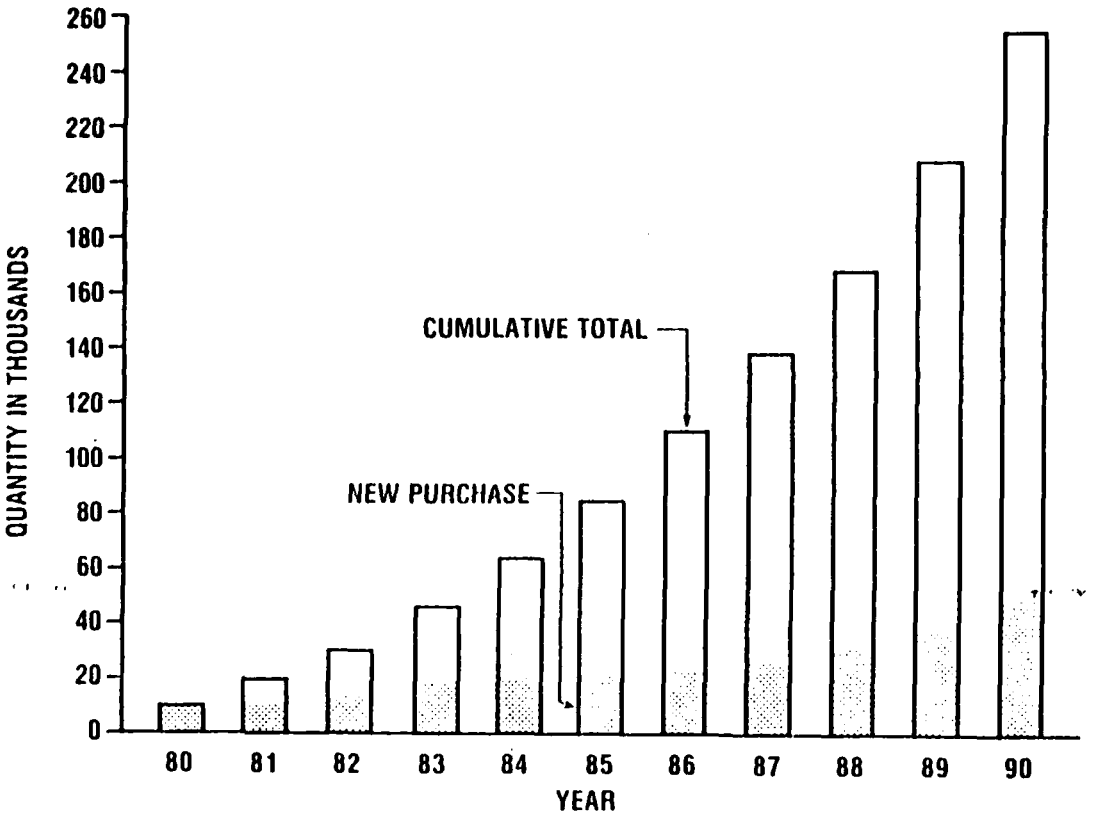


AVIONICS

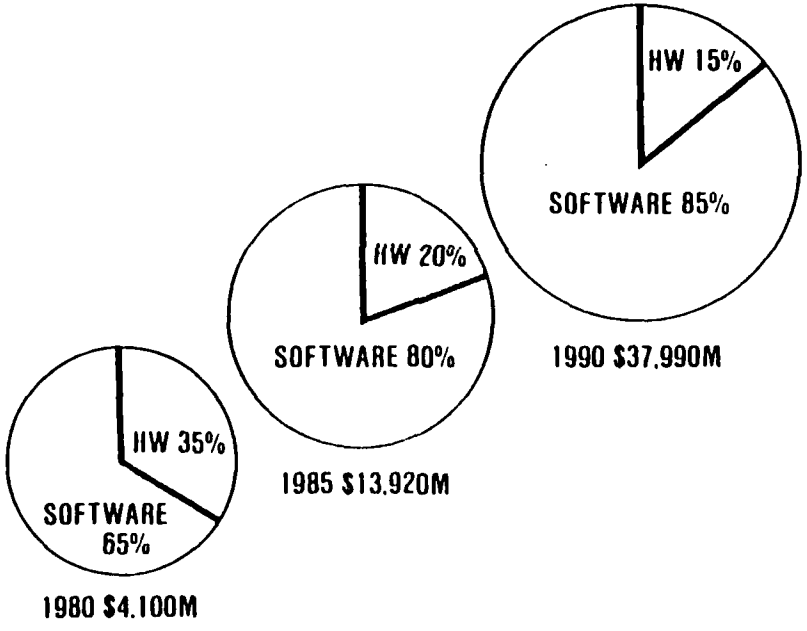
SPACE

A COMPUTER INCORPORATED AS AN INTEGRAL PART OF, DEDICATED TO, OR REQUIRED FOR DIRECT SUPPORT OF, OR FOR THE UPGRADING OR MODIFICATION OF MAJOR OR LESS-THAN-MAJOR SYSTEMS.

DOD EMBEDDED COMPUTERS



DOD EMBEDDED COMPUTERS
HARDWARE VS SOFTWARE EXPENDITURES



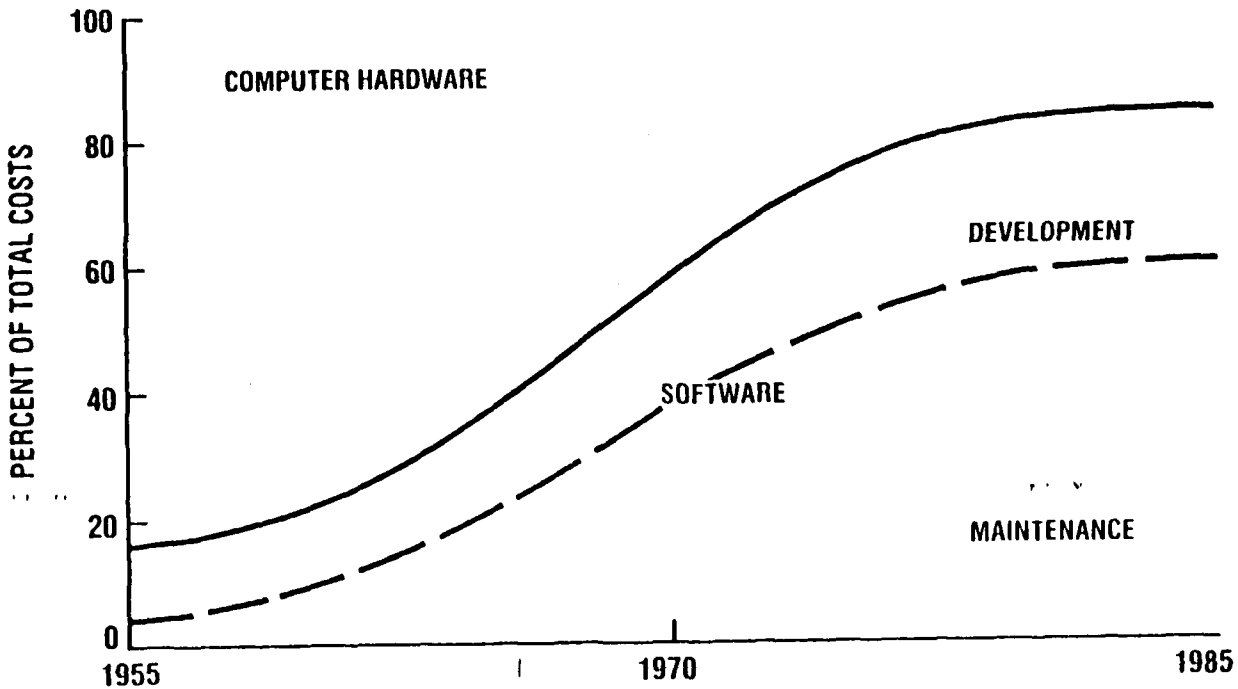
REASONS FOR SOFTWARE COST GROWTH

- PROLIFERATION OF PROGRAMMING LANGUAGES
- EXPLOITATION OF SOFTWARE'S FLEXIBLE NATURE
- LABOR-INTENSIVE NATURE OF SOFTWARE ENGINEERING
- COMPLEXITY OF MEETING RELIABILITY REQUIREMENTS ASSOCIATED WITH MISSION CRITICAL SOFTWARE

PROLIFERATION OF PROGRAMMING LANGUAGES

BY 1974, OVER 400 LANGUAGES, DIALECTS AND SUBSETS
➡ MAINTENANCE DIFFICULT AND COSTLY

DOD COMPUTER SYSTEM LIFE-CYCLE COST TRENDS



EXPLOITATION OF SOFTWARE'S FLEXIBLE NATURE

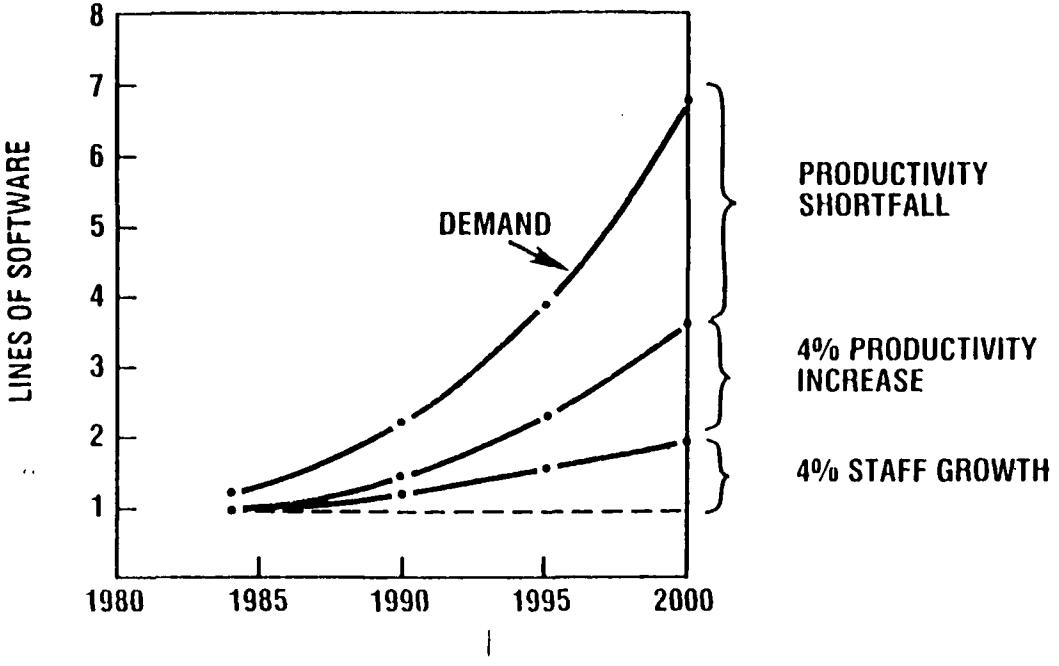
MODIFICATION	VIA HARDWARE	VIA SOFTWARE	COST/TIME RATIOS
#1	$5.28M/42 MO	$0.10M/16 MO	52.8:1/2.6:1
#2	$1.05M/36 MO	$0.02M/10 MO	52.5:1/3.6:1
#3	$8.00M/78 MO	$0.02M/15 MO	400:1/5.2:1

USAF F-111 MODIFICATIONS

INDUSTRY ABILITY TO CREATE SOFTWARE

LABOR INTENSIVE NATURE OF SOFTWARE ENGINEERING

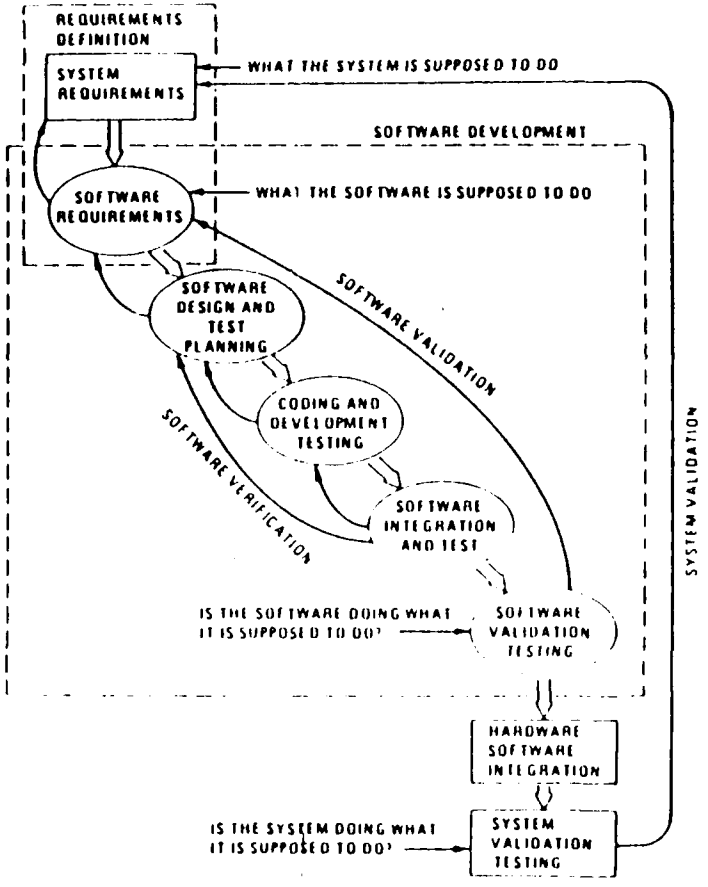
➔ RESULTS IN SHORTAGE OF QUALIFIED SOFTWARE ENGINEERS



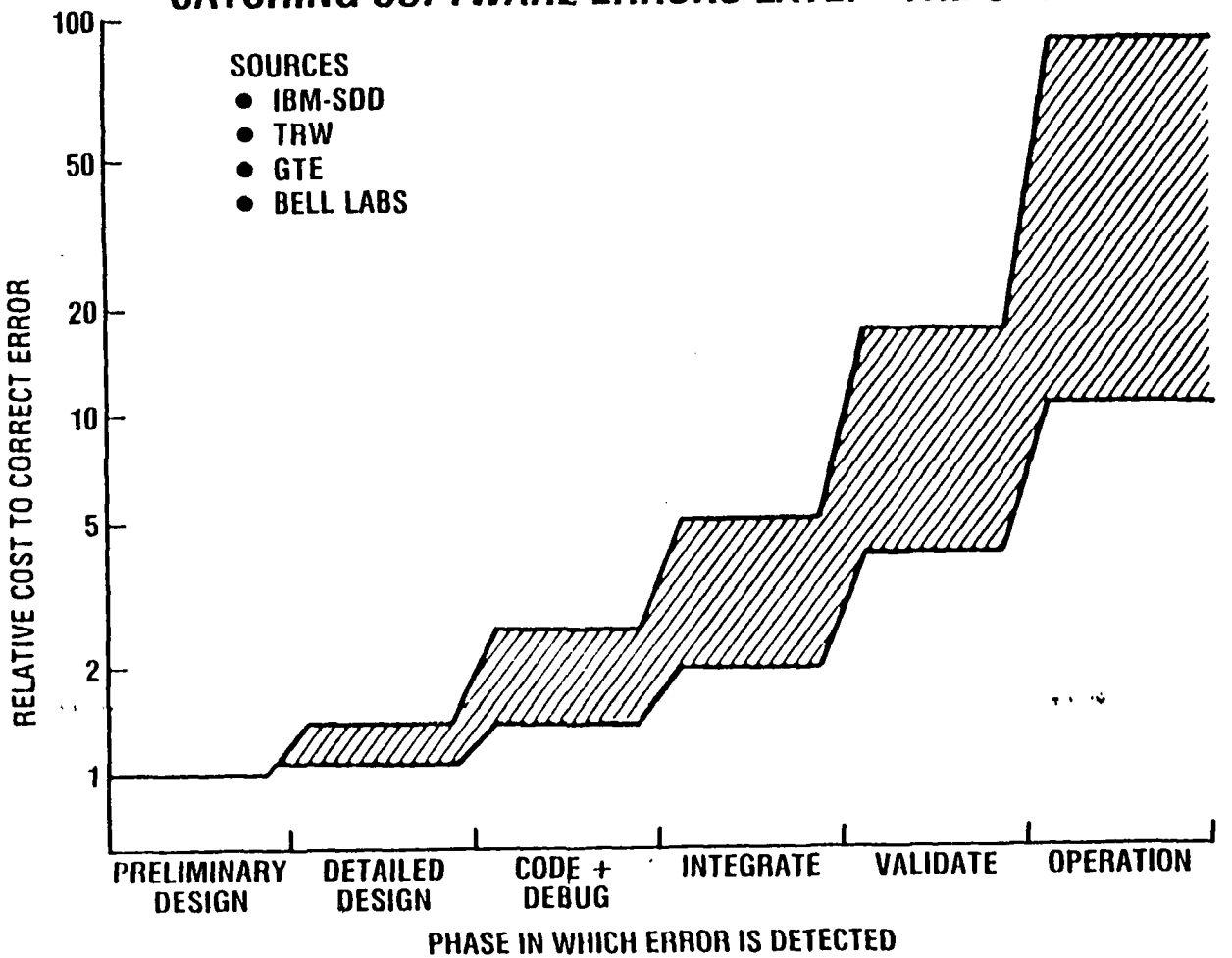
COMPLEXITY OF MEETING RELIABILITY REQUIREMENTS

INCREASED TESTING REQUIRED

SOFTWARE VERIFICATION AND VALIDATION



CATCHING SOFTWARE ERRORS LATE: THE COST



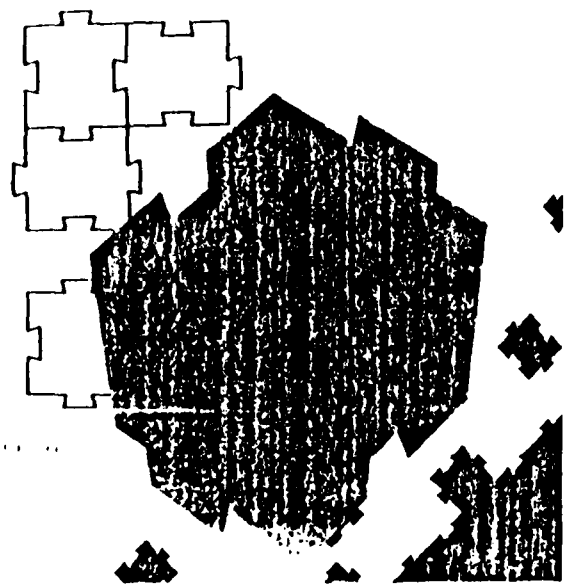
ONE SOLUTION

Ada

- Ada — NAMED FOR AUGUSTA ADA BYRON FIRST PROGRAMMER, WORKED ON CHARLES BABBAGE'S ANALYTIC ENGINE IN THE MID-1800'S
- THE Ada JOINT PROGRAMMING OFFICE (AJPO) MONITORS CURRENT Ada DEVELOPMENT ACTIVITIES
- "Ada" IS TRADEMARKED TO PREVENT SUBSETS OR SUPERSETS

ANOTHER SOLUTION

SOFTWARE COMPONENTRY



SOFTWARE BUILDING BLOCKS FOR

MISSION CRITICAL SYSTEMS

COMPONENT TECHNOLOGY

HARDWARE COMPONENTS	SOFTWARE COMPONENTS
CIRCUIT CARDS	MATH & UTILITY ROUTINES
INTEGRATED CIRCUITS	APPLICATION SPECIFIC ROUTINES
VLSI CIRCUITS	GENERICALLY DESIGNED SUBSYSTEMS

PARTS COMPOSITION TECHNOLOGY

OBJECTIVE

TO ENCOURAGE THE USE OF SOFTWARE COMPONENTS BY SUPPORTING THE BUILDING, TESTING, AND OPTIMIZING OF PROGRAMS USING REUSABLE COMPONENTS.

SOFTWARE COMPONENTRY METHODOLOGY

- USE Ada'S FEATURES
  - PACKAGE
  - GENERIC
  - STRONG TYPING
- EMPLOY REUSABILITY CRITERIA
  - APPLICATION INDEPENDENCE
  - MODULARITY
  - SIMPLICITY
  - CODE SELF-DESCRIPTIVENESS

PARTS COMPOSITION SYSTEMS

MANUAL → AUTOMATED

PARTS CATALOG

WEAPON SYSTEM DESIGNER'S SOFTWARE ENGINEERING ASSISTANT

INFORMATION RETRIEVAL SYSTEMS

SOFTWARE GENERATOR SYSTEMS

- RAPID PROTOTYPING
- CUSTOM TAILOR.NG
- EXPERT SYSTEMS

COMMON Ada MISSILE PACKAGES (CAMP)

CAMP IS A USAF ARMAMENT LABORATORY PROGRAM AIMED AT INVESTIGATING THE FEASIBILITY OF DEVELOPING REUSABLE SOFTWARE FOR ARMAMENT ELECTRONICS (ARMONICS) APPLICATIONS AND A SUPPORTING PARTS COMPOSITION SYSTEM.

SUMMARY

- MISSION CRITICAL SOFTWARE COST GROWTH MUST BE HARNESSED
- POTENTIAL SOLUTIONS INCLUDE DEVELOPMENT OF A ROBUST SOFTWARE COMPONENT INVENTORY BASED ON Ada AND DEVELOPMENT OF SUPPORTING PARTS COMPOSITION SYSTEMS
- RESEARCH AIMED AT JOINING PARTS COMPOSITION TECHNOLOGISTS AND MISSION CRITICAL SPECIALISTS IS IMPERATIVE

A UNIFIED SYSTEMS ENGINEERING APPROACH TO SOFTWARE REUSABILITY

Ted Hobson

presented to
Workshop on Reusable Components
of Application Software

by
Computer Sciences Corporation

CSC
COMPUTER SCIENCES CORPORATION

- NSIA STUDY ON REUSABLE SOFTWARE
  (RSIP SYMPOSIUM - NRL - 16 JANUARY 1985)
  - SIGNIFICANT FACTORS IN SOFTWARE REUSE
    - ACQUISITION PROCEDURES
    - SYSTEMS ENGINEERING TECHNIQUES
    - SOFTWARE ENGINEERING TECHNIQUES
    - CONFIGURATION MANAGEMENT
    - EDUCATION/TRAINING
    - STANDARDS AND PROCEDURES
    - MANAGEMENT COMMITMENT

AN APPROACH:

THE

STANDARD PROCESSING MODULES (SPM)

PROGRAM

CSC
COMPUTER SCIENCES CORPORATION

- TOPICS OF DISCUSSION
  - REUSABLE COMPONENT DEFINITION
  - MODULE SPECIFICATION
  - MODULE VALIDATION
  - CONFIGURATION MANAGEMENT PROCEDURES
    - LIBRARY
    - SPM ADMINISTRATION
    - LOGISTICS FOR REUSE
    - CHANGE CONTROL
  - INCENTIVES FOR USE
- QUESTIONS AND ANSWERS

THE SPM PROGRAM

- OBJECTIVES:
  - INCREASE GOVERNMENT VISIBILITY DURING SOFTWARE DEVELOPMENT
  - IMPROVE SOFTWARE QUALITY
  - IMPROVE SYSTEM INTEROPERABILITY
  - IMPROVE SOFTWARE DEVELOPMENT AND MAINTAINABILITY

- APPROACH:
  - PARTITION FUNCTIONAL COMMONALITY INHERENT IN LIKE SYSTEMS (E.G., COMMAND AND CONTROL, NAVIGATION, COMMUNICATIONS)
  - DOCUMENT PARTITIONED MODULES IN A FORMAT READILY USEABLE BY SYSTEM DEVELOPERS
  - ESTABLISH A MODULE REPOSITORY
  - ESTABLISH CONFIGURATION AND INVENTORY MANAGEMENT PROCEDURES TO CONTROL MODULE USE AND DEVELOPMENT

CSC
COMPUTER SCIENCES CORPORATION

- SPM EXPERIENCE
  - 1979 - IDENTIFY EXISTENCE OF FUNCTIONAL COMMONALITY IN APPLICATION DOMAINS
  - 1980 - SELECT STRATEGIC COMMUNICATIONS AS TARGET DOMAIN
  - 1980 - ESTABLISH GENERIC FUNCTIONS
  - 1981 - DEVELOP TRIAL MODULE
  - 1981 - SOLICIT GOVERNMENT/INDUSTRY COMMENTS
  - 1981 - ESTABLISH SPECIFICATION FORMAT AND CONTENT
  - 1982 - DETERMINE MODULE USE AND DEVELOPMENT CRITERIA
  - 1982 - INITIATE CONFIGURATION MANAGEMENT PROCEDURES
  - 1983 - ESTABLISH ADMINISTRATION PROCEDURES
  - 1984 - APPLY SPM METHODOLOGY TO ACTUAL PROGRAMS
  - 1985 - SUPPORT THE REUSABLE SOFTWARE CONCEPT

THE SPM PROGRAM

DECISIONS

- LEVEL OF STANDARDIZATION
- METHOD OF SPECIFICATION
- PROCEDURES FOR CONFIGURATION AND INVENTORY MANAGEMENT

WEIGHTS

- ACCEPTANCE
- UNDERSTANDING
- USEFULNESS
- SIMPLICITY

CSC
COMPUTER SCIENCES CORPORATION

REUSABLE COMPONENT DEFINITION

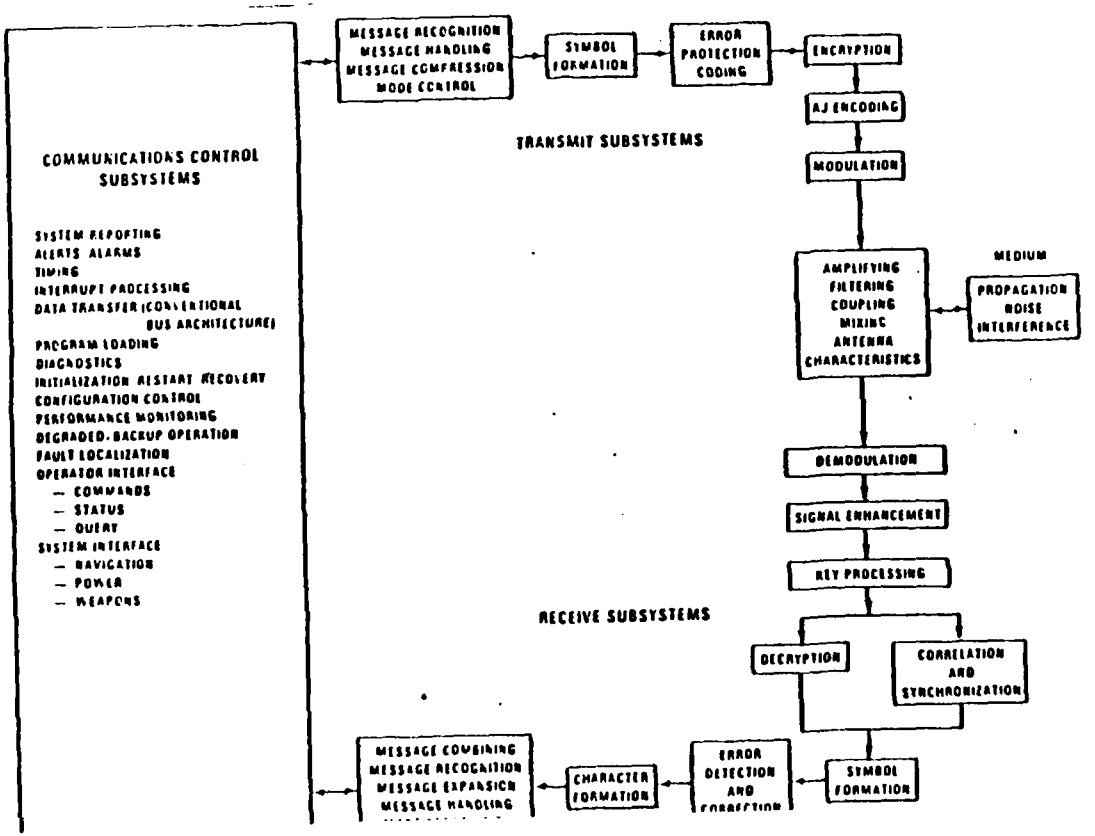
THE SPM PROGRAM

SYSTEM COMMONALITY STUDIES

- MESSAGE PROCESSING
  - TACAMO MESSAGE PROCESSING SYSTEM (TMPS)
  - MODULAR INTERFACING AND MULTIPLEXING SWITCH (MIMS)
  - TRIDENT INTEGRATED RADIO ROOM (IRR) CONTROL, MONITOR, AND TEST (CMT) SUBSYSTEM
  - INTEGRATED SUBMARINE AUTOMATED BROADCAST PROCESSING SYSTEM (ISABPS)
  - MEECN MESSAGE PROCESSING MODE (MMPM)
- SIGNAL PROCESSING
  - VERDIN
  - ENHANCED VERDIN SYSTEM (EVS)
  - TRIDENT IRR VLF/LF SUBSYSTEM
  - CONSOLIDATED VLF (CVLF) SYSTEM

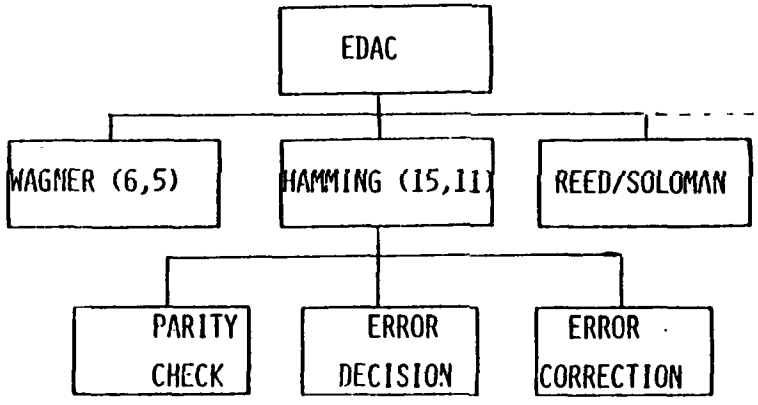
THE SPM PROGRAM

GENERIC PARTITIONING



THE SPM PROGRAM

- LEVEL OF STANDARDIZATION



GENERIC FUNCTIOI

TYPE OF ALGORITHM

SUBFUNCTION ROUTINES

- SELECTION -- TYPE OF ALGORITHM
- WEIGHT -- USEFULNESS

CSC
COMPUTER SCIENCES CORPORATION

MODULE SPECIFICATION

THE SPM PROGRAM

- METHOD OF SPECIFICATION
  - TEXT
  - FLOWCHARTS
  - PROGRAM DESIGN LANGUAGE (PDL)
  - COMPUTER PROGRAM CODE
- SELECTION -- PDL AUGMENTED WITH TEXT
- WEIGHTS -- ACCEPTANCE
  -- UNDERSTANDING
  -- USEFULNESS
  -- SIMPLICITY

FINAL DETAILED SCSPM

SECTION 1 – SCOPE

SECTION 2 – APPLICABLE DOCUMENTS

SECTION 3 – REQUIREMENTS
- NARRATIVE DESCRIPTION
- DATA REQUIREMENTS
- PDL
- VERIFICATION TO SOURCE DOCUMENTATION

SECTION 4 – QUALITY ASSURANCE
- DELINEATE QUALITY ASSURANCE RESPONSIBILITIES
- VALIDATION PROCEDURES FOR QUALIFICATION
- TEST DATA

SECTION 5 – PREPARATION FOR DELIVERY
- NOT USED

SECTION 6 – NOTES
- AMPLIFYING REMARKS
- EXISTING IMPLEMENTATIONS
- DEFINITIONS

SECTION 10 – APPENDIXES, IF NEEDED

MODULE VALIDATION

CSC
COMPUTER SCIENCES CORPORATION

- MODULE VALIDATION
  - CONFORM TO EXISTING ACQUISITION PROCEDURES
  - USE TEST RESULTS OF DEVELOPED SOFTWARE
  - INCLUDE VALIDATION DATA IN MODULE SPECIFICATION
  - PROVIDES PRE-TESTED DESIGN
  - DEVELOPER CAN TEST IMPLEMENTATION PRIOR TO FORMAL TESTING

CSC
COMPUTER SCIENCES CORPORATION

CONFIGURATION MANAGEMENT PROCEDURES

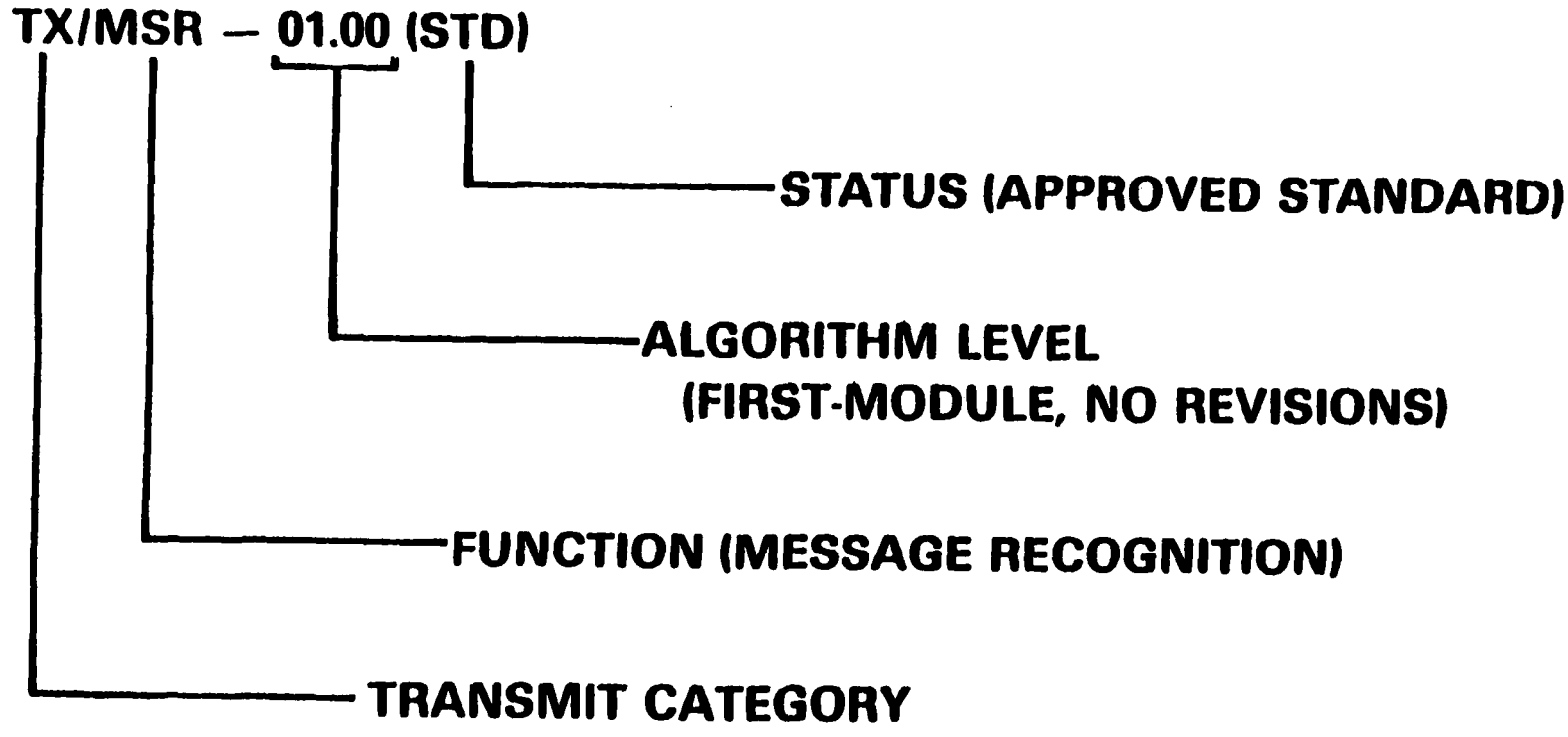
THE SPM PROGRAM

- CM PROCEDURES
  - ORGANIZATION
    - TECHNICAL -- SYSTEMS ENGINEER
    - PROGRAM MANAGEMENT -- CCB ACTIVITY
  - DOCUMENTATION
    - REQUIREMENTS -- MIL-STD FORMAT
      -- USE AND DEVELOPMENT CRITERIA
    - PROGRAM -- ORGANIZATION MANUAL
      -- RESPONSIBILITIES AND CONTROL
      -- PROJECT ENGINEER HANDBOOK
      -- ACQUISITION PROCEDURES
      -- POINTS-OF-CONTACT
  - INVENTORY CONTROL
    - REPOSITORY
    - SPM SPECIFICATION LISTINGS
    - INDEXING SCHEME

THE SPM PROGRAM

- DOCUMENTATION
  - REQUIREMENTS FOR USE AND DEVELOPMENT OF STANDARD PROCESSING MODULES
    - SPM OVERVIEW
    - CRITERIA FOR USE AND DEVELOPMENT
    - SPM SPECIFICATION FORMAT REQUIREMENTS
    - PDL STANDARDS
  - ORGANIZATION MANUAL
  - PROJECT ENGINEER HANDBOOK
    - WHO TO CONTACT
    - HOW TO INCORPORATE SPM's IN ACQUISITION PACKAGE
    - DESIGN REVIEW SUGGESTIONS
  - SPM APPLICATION GUIDES
    - LISTINGS OF SPM APPROVAL/DEVELOPMENT STATUS
    - INDEXING EXPLANATION
    - SPM SPECIFICATIONS

EXAMPLE


TX/MSR – 01.00 (STD)
STATUS (APPROVED STANDARD)
ALGORITHM LEVEL
(FIRST-MODULE, NO REVISIONS)
FUNCTION (MESSAGE RECOGNITION)
TRANSMIT CATEGORY

RECEIVE (RX) MODULE FUNCTIONS

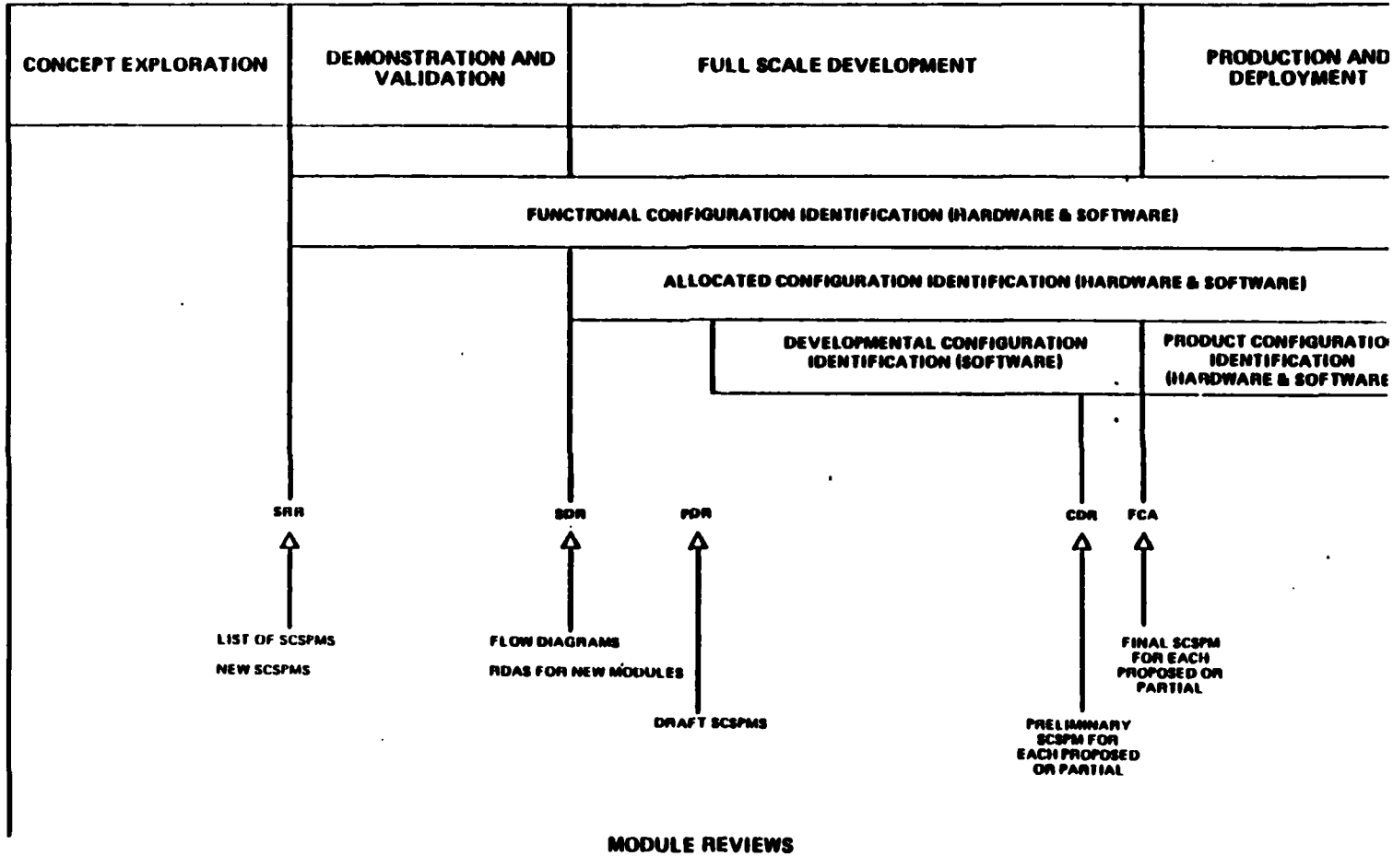
• DEMODULATION	DEM
• SIGNAL ENHANCEMENT	SGE
• KEY PROCESSING	KEY
• DECRYPTION	DCY
• CORRELATION AND SYNCHRONIZATION	COR
• MODE RECOGNITION	MDR
• SYMBOL FORMATION	SMF
• ERROR DETECTION AND CORRECTION	EDC
• MESSAGE COMBINING	MCB
• CHARACTER FORMATION	CHF
• MESSAGE RECOGNITION	MSR
• MESSAGE EXPANSION	MSX
• MESSAGE HANDLING	MSH

CVLF DIGITAL DATA RECEIVE SET MODULES

FUNCTION	MODULE	INDEX
DEMODULATION	ANALOG SIGNAL PROCESSING	RX/DEM-03.00
	DIGITAL SIGNAL PROCESSING	RX/DEM-04.00
SIGNAL ENHANCEMENT	DEPTH COMPENSATION	RX/SGE-01.00
	NOISE REDUCTION CONTROL	RX/SGE-02.00
KEY PROCESSING	MULTICHANNEL	RX/KEY-01.00
	SINGLE CHANNEL	RX/KEY-02.00
	DEMULTIPLEX (NATO)	RX/KEY-03.00
	SINGLE CHANNEL DECRYPTION (NATO)	RX/KEY-04.00
DECRYPTION	MULTICHANNEL DECRYPTION	RX/DCY-01.00
	SINGLE CHANNEL DECRYPTION	RX/DCY-02.00
CORRELATION AND SYNCHRONIZATION	COMPLEMENTARY ORTHOGONAL	RX/COR-01.00
	32-ARY	RX/COR-03.00
	CSK	RX/COR-05.00
	WAGNER	RX/COR-06.00
	FIBONACCI (NATO)	RX/COR-08.00
	COHERENT FSK	RX/COR-09.00

CSC
COMPUTER SCIENCES CORPORATION

- ACQUISITION PROCEDURES
  - TRACK MODULE DEVELOPMENT THROUGH FORMAL REVIEWS
  - INCLUDE MODULE INFORMATION IN SOFTWARE SPECIFICATION DOCUMENTATION
  - PACKETIZE SOFTWARE INFORMATION ACROSS ENGINEERING DISCIPLINES
  - FORCE REQUIREMENTS TRACING IN DOCUMENTATION PARAGRAPHING SCHEMES


CONCEPT EXPLORATION
DEMONSTRATION AND VALIDATION
FULL SCALE DEVELOPMENT
PRODUCTION AND DEPLOYMENT
FUNCTIONAL CONFIGURATION IDENTIFICATION (HARDWARE & SOFTWARE)
ALLOCATED CONFIGURATION IDENTIFICATION (HARDWARE & SOFTWARE)
DEVELOPMENTAL CONFIGURATION IDENTIFICATION (SOFTWARE)
PRODUCT CONFIGURATIO IDENTIFICATION (HARDWARE & SOFTWARE
SRR
SDR
PDR
CDR
FCA
LIST OF SCSPMS
NEW SCSPMS
FLOW DIAGRAMS
RDAS FOR NEW MODULES
DRAFT SCSPMS
PRELIMINARY SCSPM FOR EACH PROPOSED OR PARTIAL
FINAL SCSPM FOR EACH PROPOSED OR PARTIAL
MODULE REVIEWS

- INCENTIVES FOR REUSE
  - REDUCED TIME AND COST
  - SIMPLIFIES CONFIGURATION MANAGEMENT
  - ACCOMMODATES SYSTEMS ENGINEERING TECHNIQUES
  - ACCOMMODATES SOFTWARE ENGINEERING TECHNIQUES
  - PROVIDES MECHANISM FOR TRAINING/EDUCATION OF NEW EMPLOYEES
  - PROVIDES LIBRARY OF INFORMATION TO INCREASE COMPETITION

ISSUES IN REUSING SOFTWARE

Richard A. Howey
Lynn M. Meredith

Sperry Corporation
St. Paul, MN 55164-0525

1. INTRODUCTION

The goal of widespread reuse of software has so far been elusive. While the potential of reusable for increasing programmer productivity has been recognized for a long time, Sperry, and most other companies, do not have an organization-wide process for reuse or a library of reusable components. Why?

Appendix II.13 of (DOD82) described several obstacles to software reuse:

(1) Software is rarely designed for reuse

(2) Reusers have difficulty finding software to reuse

(3) Reuse is risky because the software may not work

(4) Proliferation of programming languages

(5) Lack of standards

(6) Inappropriate procurement practices

It is our contention that much of the technology which is needed to overcome the first four obstacles either currently exists or is now being developed. However, DoD contractors have little incentive to apply the technology for reuse. In addition, obstacles 5 and 6 remain largely untouched. If DoD procurement practices are altered, standards are developed, and incentives are adequately addressed, then innovative new technology can be employed and refined to make reuse a reality.

This paper discusses important existing ingredients for supporting reusability, identifies key obstacles, and recommends strategies toward resolving them.

2. EXISTING TECHNOLOGY

Sperry recognizes the potential of reusable software. We appreciate the opportunities offered by Ada?*O to support reuse. However, through our experience with Ada and Ada PDLs, we also recognize its limitations. We have also studied and exercised other state-of-the-art methodologies which give serious treatment to reusability.

The following paragraphs briefly summarize some technical developments which the authors believe have contributed greatly to the field.

2.1 Ada Related Developments

The Ada language has taken a large step toward the goal of reusing software. Examples of this are Ada packages and generic packages. Packages can be written to solve problems that occur often and used directly by other developments that encounter the same problem (similar to subroutines). Generic packages are a more powerful concept in that they allow the general algorithm to be written independent of the context of a specific problem. Context-specific characteristics are expressed by generic parameters which, by instantiation, are applied to the general algorithm to transform it into a solution to the specific problem at hand. This is not, however, enough to solve the whole problem..

2.1.1 Environment Support for Reusability

Ada environments are being built by organizations that facilitate reuse of Ada packages and programs. The Distributed Software Engineering Control Process (DCP) is a classic example (DCP). The strengths of the DCP approach lie in its claims that in order to develop reusable software the developers must have the intention to do so and having a development environment that encourages production of reusable software is helpful. The latter claim is made on the basis that Ada, while it contains constructs in support of reusability, does not enforce such practice. The third problem, recognized by DCP, is that producing reusable software is not enough; people, other than the developers, must be able to find the software in order to reuse it.

The DCP approach does contain properties which support reusability. The PDL developed by DCP captures information, that is normally available during the design activity, and retains that information in the encyclopedia in order to assist in the search process. These two features, along with the categories they have developed for the abstractions which packages represent, are useful concepts that, we believe, would be useful in the final solution. We also like the fact that the PDL constructs appear as Ada comments.

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

What does bother us about the DCP approach is that the abstraction categories are not strong enough to support the application point of view. For example, if an application needs, say a radar processing capability, which abstraction category (or categories) does it look for? In addition, we are not convinced that simple abstractions will work well in all cases.

2.1.2 Formal Annotations to Describe the Effect of a Part

The issue of how much a reuser needs to know about the rationale for a module's design or the methodology used when it was produced, has raised the issue of where does this type of information go? In response to this, more formal languages have been proposed. Anna (ANNA84) is one example of a formal specification language.

Anna provides a vehicle in which information about the design of a module can be captured and unambiguously expressed using ideas from boolean algebra and first order logic. Unambiguous expression of the properties and effect of the module does support reusability. In addition Anna offers the ability to analyze and verify the part, because the annotation can be compiled and executed. This is a useful concept for software reusability since it is desirable (if not necessary) to be able to analyze and verify that the "borrowed" software will work in its new surroundings. In addition, if the part needs to be tuned, its current limitations are precisely described in Anna.

There are, however, some sizable drawbacks to the Anna approach. While this type of rigor is certainly appropriate in some situations, it becomes unacceptably difficult even for moderately-sized programs. In addition, it may require a skill level that is not possessed by the majority of software professionals.

2.1.3 Tighter Control over the Relationship Between Part and System

There is another point of view on the subject of how much needs to be known about a module's relationship to other modules in a system, as well as to its physical implementation. Some approaches claim tighter control is necessary. A prime example of this point of view is Programming-in-the-Large (WOLF84).

The general problem here is that there is a difference between the design of one single module and a system of modules and that the "all-or-none" rules of Ada packages may not be strong enough to control modularity and interfaces. The basic ideas of this approach, access and provision, do provide more precise information about a module's relationship to its surroundings. When reusability is the issue the reuser certainly needs to know when these relationships are violated. For example, using the provision concept does allow the designer to distinguish between the provision of the name of a type and the provision of the representation of that type. In support of reusability this is certainly a promising feature, that cannot currently be accomplished with Ada. In addition this approach supplies tools that may be used to analyze the relationships and interfaces between modules. There are, however, some situations where tighter control is not desired.

2.2 Separation of Logical and Physical Design

One view of reusability is that source code which implements a specific data transformation is often embedded in other types of code (e.g., error detection and recovery, data declaration, input/output mechanisms, etc.) which destroys its reusability. The Distributed Computing Design System (DCDS) implements a strategy for reuse based on this viewpoint (ALF081).

DCDS separates issues of logical module design from physical design. The logical design of a module (note: module refers to a logical algorithm and not to a physical structure such as a procedure or task) which defines its algorithm and inputs and outputs is specified in one language. The collection of modules into physical structures such as procedures, functions, tasks, and packages and the definition of the physical interfaces (e.g., parameter passing, global data) are specified in another language. It is the logical designs of modules that DCDS defines to be the reusable component.

The DCDS approach allows the logical module designs to be reused in many different ways. In one system, a module may be a procedure. In another, it could be a task. In one system, data may be passed from one module to another as parameters. In another, it may be shared as global data. DCDS provides for a smaller core of reusable software which may be reused in more varied ways than is possible with pure Ada source code.

As part of the DCDS development effort, an interesting Ada experiment was performed (ALF083). The purpose of the experiment was to investigate whether logical designs, specified in Ada, could be mapped into various physical designs. This was done by specifying the logical invariant module algorithms as Ada fragments

and using a macro language to specify the physical design.

The Ada code, augmented by the macros, would then be processed by a macro translator which would produce compilable Ada code. The experiment successfully produced ten physical variants of Michael Jackson's telegram problem based on one single set of invariant Ada specifications of the logical modules. The experiment demonstrates the viability of the DCDS approach for Ada programs.

2.3 Application Specific Very High Level Languages

Another view of reuse is represented by very high level languages (VHLL). Commonly used application-specific functions can be built into such languages. An example of this is used on the Enhanced Modular Signal Processor (JONE84).

Signal processing systems can be conveniently represented by data flow graphs. The arcs of the graph represent queues. The nodes are either a predefined primitive operation (such as a Fast Fourier Transform) or a subgraph. For each execution of a node, each incoming queue has an associated minimum threshold of information needed for the node's execution (e.g., the queue must contain 5 entries). When every incoming queue for a given node has reached its threshold, then the node is eligible for execution.

It is in the nature of signal processing systems that the operations represented by the nodes of the graph are highly reusable. Many different systems can be developed by defining new graphs which reuse the same primitive nodes. The language used to describe the graphs can be considered a VHLL for signal processing.

The VHLL approach to reusability represents a very valid concept for this particular application domain. However, it may not be appropriate for every application domain. It is important that any standardization efforts which focus on Ada should not preclude the use of VHLLs.

2.4 Tailoring Source Level Code

Although reuse of software is, in fact, going on to a fair extent, the method employed suffers from three well-known weaknesses: 1) isolation of reusable parts is ad hoc, 2) finding parts is a manual process and therefore time consuming and prone to error, and 3) the result is several versions of the same part, with minor differences; all of which are maintained as if they had no relationship to each other. The Computer-Aided Programming (CAP) development (BASS83) provides a pragmatic solution to these weaknesses.

The principles of their approach are: 1) most programs require little unique code and 2) programmers spend far more time adapting and maintaining programs than writing them (BASS83, p.9). To address these two issues, the approach utilizes the concept of "functionals"; an enhancement of the notion of "frames" originally introduced by Minsky, and a "code adaption process". The "functional" consists of a "template" (the invariant portion of the code) and "meta-statements" (which describe the context-specific information). The approach follows functionals to use other functionals in a hierarchical (tree-like) fashion. The code adaptation process then traverses the tree, from leaves to the root, translating "meta-statements" to context-specific source level code. When the root of the tree is reached the conversion of the "functional" to a specific function (i.e., the finished program) is complete. The system has been successfully used for Cobol and Dibol implementations of actual commercial applications.

The major drawbacks to the CAP approach are: 1) CAP offers no method to locate the needed part, 2) we have reservation about whether or not the "meta-statements" can be effectively utilized for embedded real-time applications, and 3) the distinction between logical and physical design is not strong enough.

3. OBSTACLES TO REUSING SOFTWARE

In the previous section we have identified a few of the currently available technical ingredients which demonstrate experience toward making reusability achievable on a wider scale. However, as our choice of ingredients implies, even the technical aspects of reusing software are not solidified.

The section on Ada-related activities (i.e., DCP, Anna, Programming-in-the-Large) points out an interesting and very difficult to overcome, fact that any approach to software reusability must be able to address: that there are many different, and valid, software development opinions and practices that are in support of reusable software. The first thing that is needed is a definition of what constitutes a reusable part. This definition must allow for variations of certain notational constructs. For example, the EFFECT concept can be addressed by ANNA in some cases, the DCDS PDL in other cases, and even english in others. All points of view are valid.

This uncovers an extremely important point: definitions and guidelines (rules) must be developed, so that the variation in notation is properly selected, in order to make reusability a reality. We say definitions because the term "software parts" does not mean the same thing to all people. More importantly, as is clear from some of the discussion above (i.e., DCP, CAP), source text is not the only datum to be collected on a "software part".

The second issue to be dealt with whether or not a software part, developed by another company for another system, can be inserted into a new system by a new "author" without any change at all. There is considerable doubt that there is, for embedded real-time applications, such a thing as 100% reusability. It is vital that this be both realized and accepted, so that "tuning" a part is an acceptable practice. This observation has some wide ranging implications in terms of maintenance, configuration management, and the incentives offered to developers.

Software developers must be motivated to attempt to reuse software parts. This will not occur on a widespread scale without alteration in the DoD procurement process, including incentives and standardization of the definition of a reusable part.

4. STRATEGIES TO MAKE REUSE REAL

The following paragraphs discuss the primary issues which we believe must be resolved before widespread reuse will become a reality.

4.1 Government standards concerning reuse of software are inadequate.

Paragraph 4.3 of DoD-STD (DOD83) states that, "the contractor is encouraged to incorporate...software developed for other applications (reusable software) into the current software design." Even this standard, intended to become the future tri-service standard, places no specific requirements on reusable software. It is left up to every contractor to do as he pleases.

What should a reusability standard cover? First, and most importantly, it should define just what a reusable component is. In what notation is it recorded? We believe that they key feature required is an appropriate definition of variation. It must be possible to modify reusable components which may not exactly fit the need for which they are being retrieved but do represent a close fit.

Also important to the notation is conciseness and clarity of expression. Once we retrieve a component, we must be able to determine exactly what it does. What are its limitations? How accurate is it? How does it treat boundary conditions?

There should be a standard taxonomy for cataloging reusable components.

We believe that standards should not try to force one single methodology for software development on all DoD contractors. This would not be accepted by the software development community. Instead, standards should be oriented towards defining standard reusable parts which can be used by many different software development methodologies.

Standards must be written in such a way as not to prevent the use of new technology. As new and innovative technologies, such as knowledge based systems, become available, the existing standards should not inhibit its use. Standards should not limit the state-of-the-art.

Some other topics for standardization will be discussed in subsequent paragraphs.

4.2 Establishment of a Government Controlled Library.

While reuse of software within an organization or company is valuable, DoD has more to gain from DoD-wide reuse. Today, when a DoD contractor wants to reuse software developed by another company, this usually requires some kind of licensing agreement.

Executing these agreements is not too bad if you are only licensing a few large components, such as an operating system. However, what would it be like if you wanted to develop a large system reusing hundreds or even thousands of components? The resulting legal hassles would probably make this prohibitive.

We believe that a government operated software library would contribute greatly toward solving this problem. While there will always be some software to which companies will wish to retain the data rights, it should be possible to make contributions to this library attractive. This could be done by purchasing the data rights of all contributed software for a fair price. Once deposited in the library, the software should be freely available to all, or available at a nominal cost. We must also consider who is liable for reused software. If we contribute some software to the library which we developed for, say, an

accounting system and somebody else reuses it in an on-board system on an airlineer, are we legally liable if the airplane crashes? This is a strong incentive for us not to let anyone else near our software. Thus, we believe that depositors to a library should be released from liability. There is a need to reach an equitable balance between government needs for reusability and its contractors' constitutional property rights.

4.3 Incentives to Reuse Software

What does a DoD contractor stand to gain from reusing software? Since his productivity will be higher, he may be able to underbid his competition. However, since his costs are lower, so are his revenues and profits. In a way, current DoD procurement practices provide a strong incentive not to reuse software.

Another problem can occur during the proposal process. If a bidder proposes a substantially lower price than the competition because he intends to reuse a large amount of software, what happens to his cost credibility? He could conceivably lose the contract because his bid is too low to be considered credible.

We believe that a combined "carrot and stick" approach could provide the proper incentives. Standards could provide the stick. They could, for example, require a reusability review somewhere in the software development cycle. This could be a separate review or possibly incorporated into the preliminary design review. It would serve to ensure that the design incorporates reusable components whenever possible. Such requirements should be incorporated into DoD-STD-SDS and MIL-STD-1521B.

Incentive fees for reusing software could be the carrot. Lower costs via reuse of software should result in higher profits.

4.4 Technology Transfer

Simply providing standards, a library, and incentives would help, but we do not believe that it is sufficient. A technology insertion plan must be developed. A plan for using reusability should include appropriate training of all participants.

Once proposed standards, etc., are developed, they should be tried on pilot projects. The results of these projects should be used to improve the standards. If the results are positive, then they will be a powerful inducement for DoD contractors to incorporate this technology into their organizations.

The SEI or similar organization level technology insertion programs may be the best means of effectively inserting this new process.

5. CONCLUSION

If widespread reuse is ever going to become a reality, DoD must take the lead. So far, it has been left up to individual contractors to do as they please, or do nothing at all. The result is that widespread reuse remains only a dream.

The Applications Tri-Service Working Group of STARS is in a good position to remedy the situation. With DoD leadership and strong industry commitment to the working group we are confident that the issues described in the report, plus the many other issues involved, can be resolved.

6. CORPORATE EXPERTISE AND EXPERIENCE

The sperry Corporation is known throughout the world for reliable, ruggedized, militarized computer systems. These systems are designed for all kinds of applications in all environments. As a major supplier of software to military and civilian agencies of governments and to commercial users, Sperry has continually expanded and updated its facilities to maintain innovative capabilities for producing higher quality software and increasing productivity.

6.1 Management and Measurements Perspective

6.1.1 Software Cost Estimation

Sperry has employed industry recognized software cost models for proposal and management estimation purposes since 1979. Sperry has intimate practical working knowledge of the SLIM (Putnam model, QSM, Inc.) and PRICE-S (RCA) cost models. Sperry has studied, and on some occasions uses the Jensen (Hughes) and COCOMO (Boehm, TRW) cost models. Today these models are used on a regular basis, both in the estimation of new development projects, and to gauge the status of work in process.

6.1.2 Software Complexity Measures

Sperry has experimentally pursued software complexity measurement on major software development programs.

For example, Halstead complexity metrics were implemented on a real-time avionics system developed in PASCAL. This data together with

management records of production effort were used to establish a project spectrum of complexity versus production difficulty, down to the program unit level.

In a ground-based control center being developed in CMS-2M the Halstead metrics extraction was supplemented by a measurement of McCabe cyclomatic number metrics. A comprehensive metrics analysis will be performed when this project is completed.

On a number of major projects, error and error rate data are being regularly gathered to further infer empirical measures of software complexity.

6.1.3 Automated Work Breakdown Structure

Sperry has developed an automated work breakdown structure (WBS) generation program as a support tool for software development projects. Downstream planning for utilization of this capability includes integration with: 1) work planning network tooling, and 2) cost modeling software. Complete WBS analysis is considered to be a necessity for effective software cost estimation and project planning.

6.2 Application Perspective

A broad view of the domain of DoD applications is essential to understanding the issues involved in reusing software. Sperry is a leader in the development of real-time tactical command and control systems, logistics support systems, communications systems, and intelligence systems. Sperry has been a developer of DoD application software for command and control, transportation, and communications systems for years. A few examples of Sperry's current contracts demonstrates this broad base:

(1) Marine Air Traffic Control and Landing System (MATCALS)

Sperry has been associated with MATCALS since 1973 in this system consisting of air traffic control, communications, circuit switch, and display software. Other system features include high resolution color graphics, direct radar interfaces, local network interfaces, processor controlled circuit switches, and digital voice conferencing units.

(2) Canadian Patrol Frigate (CPF)

Sperry is developing a distributed command and control system and establishing the facilities to support full scale development and life cycle maintenance for CPF. Operational and simulation software with associated documentation are being developed for this system which features a variety of subsystems including weapon systems, fire control, meteorological systems, ESM and ECM systems, and communication control and monitoring.

(3) Enhanced Modular Signal Processor (EMSP)

EMSP is a multiprocessing network, programmable array processors and special purpose devices to support high performance signal processing applications for the U.S. Navy. This programmable system will support a very high level user interface oriented to the application engineer.

6.2.1 Application of Unique Tools and Capabilities

With its background in the development of command and control systems, Sperry has developed tools in response to application needs. Sperry's experience in simulation and modeling of weapons systems is one example of important application perspective of APSE tools. Examples of Sperry experience in this are alone are:

(1) The ADAM model is one of a series of analytical modeling tools developed for the evaluation of interconnects for distributed systems processing.

(2) The AQUADS model is an analytic model for investigating interactions in distributed database management systems.

(3) The System Performance Assessment Model (SPAM) was developed as a general purpose command and control simulator based on Sperry's experience with weapon system modeling and simulator development. It was implemented in SIMSCRIPT II.5 to be compatible with the majority of existing service owned weapons systems simulators.

6.3 Environment Builder Perspective

For the past several years, Sperry IR&D has been building tools to support Ada. These include a prototype compiler and run-time system, a Code-Generator Generator, reusable

packages, and a hardware architecture simulator.

Sperry develops almost all of its software systems using hosted tools, both batch and interactively, through remote terminals. These tools (editors, compilers, retargetable compilers, design aids, debug packages, instrumentors, documentation systems, etc.) give Sperry the capability to provide customers with a unique expertise in the design and utilization of tools from the perspective of a high technology, production oriented, major defense contractor. MTASS and NSW are examples of Sperry capability to create and sustain environments.

6.3.1 Ada Compiler and Run-Time Research

In an earlier IR&D project, Sperry developed a compiler for a subset of Ada and also developed a machine transportable simulator for executing the generated code. The simulator ran on a Sperry 1100 computer and supported hardware features that greatly aided the execution of Ada programs.

6.3.2 Ada Code-Generator Generator (CGG)

As a result of Sperry's experience with compiler developments, Sperry is conducting research and development directed toward automating the construction of code generators. The approach is based on the following observations on recently developed compilers.

(1) The implementation of the code generation method includes the generation of preset tables which can be automatically produced.

(2) The code generator design is not dependent upon a particular intermediate language structure.

(3) The code generator to be produced is specified in an object independent manner.

The input to the CGG describes the forms the intermediate language would take in being transformed from Diana to the target computer object code. The form of the code-generator produced by the CGG is that of one or more processes with, by a template matching process, cause the data transformation through action routines coded by the programmer and supplied to the CGG. A skeleton code-generator was produced for the AN/UYK-43.

Sperry's current efforts lie in hosting the CGG and its outputs and in specifying the deployed system software environment on the target computer. This involves run-time modeling, distributed software, and hardware modification for support of Ada.

6.3.3 Ada Support Tools

As an IR&D activity, Sperry developed a library of software development tools to support APSE components. An example are packages providing a set of IDL tools. These tools have been designed and implemented to manipulate the data instances of the various user-defined data structure types. These packages were used by the Hierarchical Processor (HSP) of the VHSIC Phase I program.

6.3.4 Hardware Simulator

The architectural extensions project is defining and evaluating modifications to the AN/UYK-43 computer ISA to enhance its support for the Ada programming language. This effort includes development of a software simulator for the AN/UYK-43. Ada is being used both as a design and implementation language for this development. The simulator design facilitates modification to simulate other ISA's.

6.3.5 Machine Transportable Support Software System (MTASS)

Sperry has many years of experience in rehosting software on a wide variety of host machines. A notable example of such a hosted system is the Machine Transportable Support Software system (MTASS). MTASS is an 1100 hosted software system that provides program development tools for the Navy standard minicomputer, the AN/UYK-20, as well as the Navy standard avionics computer, the AN/AYK-14. MTASS consists of the following:

(1) CMS-2M Cross Compiler

(2) FORTRAN IV Cross Compiler

(3) Macro Cross Assembler

(4) Loader and System Generator

(5) AN/AYK-14 Simulator

(6) AN/UYK-20 Simulator.

This set of software tools (which is highlighted by the CMS-2M compiler) was designed, implemented, tested, documented, and is currently maintained and configuration-managed by Sperry. In addition, Sperry is responsible for transferring MTASS to a number of other host systems including IBM 360/370,

CDC Cyber, DEC, and PDP-10.A key concept of the MTASS system is to use the Common Interface Routine (CIR) to localize the host dependent portions of the MTASS system.

6.3.6 National Software Works (NSW)

National Software Works (NSW) is a network operating system which maintains objects for users and provides access to these objects which reside on different host computers connected to the ARPANET. Sperry produced the design of the 1100 tool bearing host software. This experience utilized vast existing experience from other tool bearing host developments on UNIX, TENEX, TOPS 20, IBM 370/VS and MULTICS which created additional expertise in the area of inter-tool data interfaces, process to process interfaces, and end-user interfaces.

6.4 ADA Language Proficiency

Sperry is actively involved in the DoD's common high order language effort, Ada. Sperry staff members have actively participated in reviewing the WOODENMAN and IRONMAN requirements documents for Ada and have contributed as Ada Phase I and Phase II evaluators. Sperry has also contributed to the Ada effort at Irvine, Fort Walton Beach, and Boston. Sperry has also been active in providing the high order language working group with comments and suggestions on the PEBBLEMAN, STONEMAN and MIL-STD-1815A documents.

Sperry has established a task force to integrate with the Ada language and environment efforts. A team from Sperry has participated in the Ada Test and Evaluation phase and has reported test results and language issue reports and comments.

For the Test and Evaluation phase, the team selected a typical display program from the set of TCCF (Tactical Communications Control Facility) application programs and reprogrammed it in Ada.

Sperry personnel have attended Ada language courses taught by the Ada language design team, and in turn, have presented in-house Ada courses.

Sperry personnel attended the Ada Language Test and Evaluation workshop in Boston, and participated in the Ada Language Environment workshop.

Sperry Ada language capability has extended beyond the core group involved in these early Ada language efforts. Ada training has been obtained by both application and environment developers through external sources such as Dick Bolz, Ed Bernard, and ALSYS Ada launch seminars.

6.5 Professional Society Activities

Sperry is actively involved in many related areas of support to the DoD through professional societies, such as the IEEE, EIA and NSIA. In some cases, support is provided directly to DoD agencies, such as the APSE E&V task of AJPO and the reviews of STARS DIDS for RADC.

6.5.1 Participation in EIA Computer Resource Committee

The EIA G-33 Computer Resources Task group was formed in 1976. That year, Mr. J.L. Raveling became the Sperry representative to the G-33 CM/DM committee. Since that time, Mr. Raveling, a staff systems manager with Sperry, has been a highly active participating member in the G-33 task group, and in several years has held positions of vice-chairman and chairman. In 1983 the EIA separated non-CM/DM computer resources activities from G-33 in forming the G-34 Computer Resources Management Committee. Mr. Raveling was G-34 vice-chairman in 1983, and was elected chairman in 1984. Current task areas of the EIA G-34 committee include:

(1) Next generation computer architectures

(2) The STARS program

(3) Review of the JSSEE Operational Concept Document, and the proposed military standard for software support environments

(4) The Software Engineering Institute

(5) Support to computer resource management for the Joint Logistics Commanders

(6) DARPA Strategic Computing and Strategic Initiative program

(7) DoD HOLS (including Ada)

(8) Software quality assurance and reliability

For the past seven years Mr. D.M. Erb, director of software QA, has been a forceful mover for software QA in the G-33, and subsequently G-34, groups. Mr. Erb is the former Chairman and is currently a member of the G-34 software QA committee. This subcommittee has played a major participating role in the rewrite of existing, and development of new software QA

and other computer resource military standards and handbooks.

Sperry has played a strong active role in computer resource CM/DM for over eight years. Currently Mr. J.M. Anderson, manager of systems CM/DM, is a voting member of the G-33 committee for CM/DM. He is supported by Mr. D.R. Willi, principal data manager, in the DM specialty area. Current activities of the G-33 CM/DM Committee include:

(1) Preparation of a DM overview guidebook
(2) Identification of needed CM bulletins (J.V. Anderson - lead)
(3) IEEE software CM standard
(4) CM plan DID improvements
(5) CAD/CAM impacts on CM/DM
(6) Identification of needed DM bulletins
(7) CM plan bulletin.

6.5.2 APSE E&V Task Group

M.J. Meirink and R.E. Sandborgh are distinguished reviewers of the APSE Evaluation and Validation Task Group of the AJPO. The purpose of this task group is the development of the requirements, criteria, and technology to be used in evaluation and validation of Ada programming support environments. Additionally, both are members of the task group's requirements working group.

6.5.3 STARS Measurement Task Force

Mr. R.E. Sandborgh participated in the review and was leader of the general issues committee whose task was to ensure the effort was properly focused. This committee recommended a major reduction in the number of DIDs and focusing of data collection for specific purposes.

6.5.4 JSSEE

Sperry has actively supported and monitored the JSSEE development during the past year. Members of our technical staff have participated in the JSSEE Industry Team and in the review of the:

(1) Plan of action and milestones for definition and preliminary design of a Joint Services Software Engineering Environment (JSSEE), January 1984
(2) Joint Services Software Engineering Environment (JSSEE), Operational Concept Document (OCD), 12 June 1984.

A member of our technical management staff served as a co-chair of the Software Support Environment (SSE) panel at the Joint Logistics Commanders (JLC) Orlando I workshop. The SSE panel report made direct contributions to the JSSEE activities.

6.5.5 DoD-STD-SDS

Sperry has played a leading role in the review and coordination of the JLC's new Software Development Standard (DoD-STD-SDS). This support has been provided over the last two years and encompasses their major reviews of this evolving standard.

6.5.6 The International Society for Parametric Analysis

Mr. E.O. Tilford, director of Software Engineering, is an executive member of the International Society for Parametric Analysis, an organization dedicated to the study and refinement of significant measures of estimation within the scientific and engineering disciplines. In the two immediate past years, Mr. Tilford served as chairman of the software policy and applications committee. He is presently a member of the society's board of directors, and is chairman of its ways and means committee. Representing the society, he has been an active lecturer in software cost estimation. Mr. Tilford also works in the fundamental research of software cost modeling in collaboration with Mr. Larry Putnam, a nationally recognized expert in the field.

6.5.7 IEEE Ada PDL Working Group

From its beginning in 1982, Mr. M.J. Meirink has been a member of the IEEE Project 830, Ada Program Design Language Working group. Mr. Meirink supervises the tools and methodology organizational unit, and is project engineer for the software engineering environment IR&D program. A draft of the IEEE guidelines has been released for review. Mr. Meirink is currently a member of a working group subcommittee charged with focusing upon specific environment issues.

6.5.8 Software Reusability Study

During the summer of 1984, Ms. J.E. Mortison, Software Engineering Staff Consultant, served as group leader of the NSIA study task ISTG 84-2, "Systems Engineering Aspects of Software Reusability" for the U.S. Navy. This study was initiated at the request of the Deputy Assistant Secretary of the Navy for C3I. Current Navy weapon system computer programs were examined to determine factors and issues relating to reusability. Recommendations for the promotion of reusability were a part of the task group's final report.

REFERENCES

(1) ALF081- M.W. Alford, J.E. Irby, J.E. Scott, J.T. Lawson, R.G. Osborne, E.J. Hardy, Distributed Computing Design System Description, TRW Defense and Space Systems Group, August 1981.

(2) ALF083- M. Alford, H. Hart, E. Miller, J. Scott, D. Smith, Distributed Computing Design System Final Report, TRW Defense and Space Systems Group, June 1983.

(3) ANNA84- David Luckham, Friedrich W. vonHenke, An Overview of ANNA A Specification Language for Ada, Technical Report No. 84-265 Program Analysis and Verification Group Report No. 26, Computer Systems Laboratory, Standford University, September 1984.

(4) BASS83- P. Bassett, J. Giblon, Computer Aided Programming (Part I), SoftFair, A Conference on Software Development Tools, Techniques, and Alternatives, Proceedings, pp. 9-22; Arlington, Virginia, July 25-28, 1983.

(5) DCP84- Steve Parish, Andres Rudmik, DCP--Experience in Bootstrapping an Ada Environment, GTE Communications Systems R&D.

(6) DOD82- Department of Defense, Strategy for a DoD Software Initiative, August 1982.

(7) DOD83- Department of Defense, Proposed Military Standard Defense System Software Development (DOD-STD-SDS), December 1983.

(8) JONE84- D.E. Jones, W.J. Shellenbarger, EMSP Common Operating System, A Tool for Developing Real-Time Signal Processing Systems, Proceedings of the 1984 Technical Symposium, Sperry Corporation, May 1984.

(9) MAUR81- Proceedings of the Software Workshop Joint Workshop Logistics Commanders Joint Policy Coordinating Group on Computer Resource Management, Report of the Panel on Software Reusability, PANEL E, Monterey, California, June 1981.

(10) WEIS- Leonard R. Weisbert, A New Approach to Lowering DoD Software Costs, Aerospace and Defense Group, Honeywell, Inc., Minneapolis, Minnesota, date unknown.

(11) WOLF84- A.L. Wolf, L.A. Clarke, J.C. Wileden, An Ada Environment for Programming-in-the-Large, Software Development Laboratory, Computer and Information Science Department, University of Massachusetts, 1984.

RESUME

RICHARD A. HOWEY

Principal Programmer
Sperry Corporation, Defense Products Group

Richard Howey is currently assigned to the Ada Department at Sperry DPG. He is involved in the specification, design, and testing of Ada programming support environments. He is participating on the Sperry 1100 Ada Compiler System testing, the ALS/N proposal for the U.S. Navy, and a study to define an Ada environment for the U.S. Air Force's Project LIBRA.

Prior to the assignment, Mr. Howey spent nine years as a programmer/analyst, and later as a development team leader, in Sperry's Air Traffic Control Systems department. In this capacity he was involved in large scale real-time embedded software development. He also participated in many related studies, proposals, and IR&D activities.

Mr. Howey received a BS degree with highest honors in Mathematics and Computer Science from Lawrence Institute of Technology in Southfield, Michigan in 1975.

SPERRY

ISSUES IN REUSING SOFTWARE

STARS APPLICATIONS TRI-SERVICE WORKING GROUP

REUSABLE SOFTWARE WORKSHOP

9-12 APRIL 1985

RICHARD A. HOWEY
LYNN M. MEREDITH

SPERRY CORPORATION
DEFENSE PRODUCTS GROUP

SPERRY

TOPICS

- CURRENT STATUS OF REUSE
- OBSTACLES TO REUSE
- ISSUES WE NEED TO ADDRESS

RAH/LMM
4/9/85

SPERRY

- ➡ STATUS
  OBSTACLES
  ISSUES

CURRENT STATUS OF REUSE

- o LOTS OF RESEARCH
- o MANY DIFFERENT TECHNICAL IDEAS
- o FEW FORMAL ORGANIZATION-WIDE REUSE PROGRAMS
  - MOST REUSE IS AD HOC

RAH/LMM
4/9/85

SPERRY

STATUS
➡ OBSTACLES
ISSUES

OBSTACLES TO REUSE

- MANY DIFFERENT VIEWS
- DATA RIGHTS
- LACK OF DOD INCENTIVES

RAH/LMM
4/9/85

SPERRY

STATUS
➡ OBSTACLES
ISSUES

MANY DIFFERENT VIEWS

- o SOME VALUABLE ONES
  - ADA SOURCE VIEW
    - ENCYCLOPEDIA OF PACKAGES (DCP)
    - FORMAL ANNOTATIONS (ANNA)
    - PART/SYSTEM RELATIONSHIP (PROG-IN-THE-LARGE)
  - LOGICAL DESIGN VIEW (DCDS)
  - VHLL VIEW (EMSP)
  - TAILORABLE CODE (CAP)
- o MANY VIEWS HAMPER EFFORTS AT REUSE

BUT

- o DIFFERENT VIEWS ARE VALID

RAH/LMM
4/9/85

SPERRY

STATUS
➡ OBSTACLES
ISSUES

DATA RIGHTS

- LEGAL NIGHTMARE
- NEED TO BALANCE CONTRACTORS RIGHTS vs DOD NEEDS

RAH/LMM
4/9/85

SPERRY

STATUS
➡ OBSTACLES
ISSUES

LACK OF INCENTIVES

- REUSE INCREASES PRODUCTIVITY
- SOME NEGATIVE ASPECTS OF INCREASED PRODUCTIVITY
  - LOWER REVENUES AND PROFITS
  - BIDS BELOW CREDIBLE MINIMUM

RAH/LMM
4/9/85

SPERRY

STATUS
OBSTACLES
➡ ISSUES

ISSUES WE NEED TO TACKLE

- STANDARDS
- ESTABLISHING A LIBRARY
- INCENTIVES
- TECHNOLOGY TRANSFER

RAH/LMM
4/9/85

SPERRY

STATUS
OBSTACLES
➡ ISSUES

STANDARDS

- o NEED TO CONTROL PROLIFERATION

  BUT

- o MUST NOT PROHIBIT VALID/INNOVATIVE PRACTICES

- o POSSIBLE APPROACH
  - ANALYZE CURRENT SOFTWARE METHODOLOGIES TO IDENTIFY COMMON POINTS
  - DEFINE REUSABLE COMPONENTS WHICH CAN BE REUSED BY MANY DIFFERENT METHODOLOGIES

- o MAY REQUIRE APPLICATION DOMAIN SENSITIVITY

RAH/LMM
4/9/85

SPERRY

STATUS
OBSTACLES
➡ ISSUES

ESTABLISHING A LIBRARY

- NEED A CLEARING HOUSE FOR STANDARD COMPONENTS
- STANDARDS ARE A PREREQUISITE
- MUST TACKLE DATA RIGHTS ISSUE

RAH/LMM
4/9/85

SPERRY

STATUS
OBSTACLES
➡ ISSUES

INCENTIVES

- o ADD INCENTIVES + REMOVE DIS-INCENTIVES
- o POSSIBLE IDEAS
  - INCENTIVE FEES
  - PROPOSAL EVALUATION CRITERIA
  - STANDARDS
    - DOD-STD-SDS
    - MIL-STD-1521B
- o REUSABILITY REVIEW

RAH/LMM
4/9/85

SPERRY

STATUS
OBSTACLES
➡ ISSUES

TECHNOLOGY TRANSFER

- NEED TO ACTIVELY PROMOTE REUSE
- TRAINING WILL BE NEEDED
- PILOT PROJECTS
- SEI OR SOMETHING SIMILAR

RAH/LMM
4/9/85

SPERRY

CONCLUSION

- LOTS OF GOOD TECHNICAL WORK HAS BEEN DONE
- NO CENTRAL FOCAL POINT
- IF DOD WANTS WIDESPREAD REUSE, DOD MUST TAKE AN ACTIVE LEAD ROLE

RAH/LMM
4/9/85

SEARCHING AND RETRIEVAL FOR AUTOMATED PARTS LIBRARIES

John D. Litke

Grumman Data Systems Corporation

The search for methods that encourage the reuse of software is usually motivated by the need to reduce cost and increase productivity in the construction of software. This desire has a long history that goes back at least to the SHARE group and continues as a dominant motive in the design and adoption of the Ada language. In recent years, an increasing interest in reusable software has come from the desire for rapid prototyping. One approach to rapid prototyping is the construction of elaborate application generators, but we now know that such are very difficult to build for even a narrow span of application. A more promising approach is to build software parts that "fit" easily together, permitting the rapid assembly of parts into a working whole.

Libraries of reliable software have been with us for some time. The IMSL library of mathematical routines is often cited as a good example of a difficult art. Why are such libraries so few and so rarely used? Common excuses that we have all heard are:

I didn't know that the routine was available.
The routines lacks this or that feature.

Aside from the not invented here bias to such feelings, the first reason clearly points to a need for better searching and retrieval methods to encourage use of collections of routines. Anyone who has searched the SHARE or Collected Algorithms of the ACM will attest that the indexing method never seems to match the problem statement. The second reason can be overcome if we can find a way to store and reuse algorithms rather than programs or subroutines. Ada packages and generics are not quite what we want, but rather we need fragment specifications in Ada that are dependent on type and variable elaboration. (1) Rather than trying to solve the second more difficult problem, this position paper discusses the requirements for a searching and retrieval mechanism that will allow efficient use of both completed programs, packages, etc., as well as the searching and retrieval of algorithms. It is our contention that the classical data base approach is not sufficient. Rather we require a method that describes the organization of our knowledge, as well as the knowledge itself. We propose that a delineation of the knowledge to be organized is required before the organizing machinery can be designed and constructed.

The classical approach to a searching and retrieval problem is to design a database and inquiry language, usually with one of the many commercially available database systems. A partial list of items we would desire in such a database is:

Identification:

Routine name, identification number, version, classification codes and sub codes, key words, author, responsible organization, etc.

Environment:

Machine, compiler version that the program has been tested on, I/O, storage and peripheral requirements, usage restrictions, dependencies on other program elements.

These items are typical of the set of specifications one will find in most existing libraries.

However, a useful software library will contain thousands of items, each with many tens of keys. Searching through such a large space of key words is not implemented effectively in todays database systems. For example, suppose I injure my arm and suspect that I may have broken a bone. I need to select a doctor for treatment, so I look up doctors in the yellow pages.

There is no such entry in my yellow pages. I try MD's with no success. Finally, I will find it under "Physicians and Surgeons". This well know naming problem is bearable when there are a few commonly recognized synonyms, but now when there are many. In the list of doctors I need to know which classifications will treat my injury. Osteopaths? Internal Medicine? Orthopedists?. The point is not the search cannot be done, but that it is onerous enough that special motivation or special training is required. Such a circumstance is not conducive to wide spread use of software libraries.

Simply searching on static properties is not enough. We also must ensure that the selected element satisfies behavioral requirements as well. Behavioral specifications require precise information on the transformation functions or rules that the routines implement, the types and variables that are imported or exported and their meaning. To assess the robustness of the library element, the user would want to know the exceptions propagated into and/or out of the element, timing constraints, the range of input values that are handled "correctly", etc. Reliability characteristics are also important, as are behavioral differences on different hosts.

An important difficulty with this wealth of information is that, depending on the application, almost any item could be a search item. Classical databases work well when there are few keys and many elements per key, but in this case there are many multiple keys and few items per unique sets of keys. Further, the range of keys is not stable. For example, we would surely want to key on the types of the input and output parameters. However, types in modern languages are meant to model real world items, so that the span of types is now conceptually infinite, rather than 4-10 different types contained in older languages.

Searching for items with a large number of keys can yield the selection of hundreds of items if only the value of one or two keys is specified. A critical observation here is that we do not want to be more specific by supplying several additional keys, but we want to supply as many keys and values as we know, rank the keys according to our own particular values, and ask the mechanism to select items that "best" match the criteria.

To return to the medical example, in my search I want to select doctors that are "close" to my home. In the particular database of the yellow pages, there is no notion of distance so that ones initial reaction is that the question is unanswerable. However, if close means "in my zip code", then the address information will suffice. Further, if the telephone exchange matches my own, the doctor is probably close to me. This example illustrates that the model used to request information will never exactly match the information model in the database, so that powerful inference mechanisms are required to create a dialogue with the user during the search. The notions of inheritance and hierarchy for organizing such variety can be helpful for reducing the storage required and providing the indexing approach for such a query.

With thousands of possible key types and key values, the user will require the ability to find elements that are "like a" known element. That is, high level searches must allow a user to gradually refine the specificity of his or her request, not by specifying a host of new key ranges, but gradually refining the range of keys considered as a class. This notion of "like a" and class can become extremely subtle in an ideal system. For example, if algorithm is an important search concept for a particular query, than an Automatic Message Handling system that contains a transformational grammar is "like a" compiler and a user than may need to narrow the scope by size of possible symbol table, running speed, etc., before implementation distinctions between the AMH and compiler system become important.

The large variety of types indicates another difficulty with the classical database approach. Such databases require that the taxonomy of the information they contain be stable and specifiable so that the schemas on which they are built can be constructed. Further, the range and syntactic values of each indexable item must be known before the schema can be built. Because a classification scheme is expected to be extensive and detailed, we require a means of maintaining the underlying classification machinery in the face of constant change. This implies that our database machinery cannot contain the classification rules in the schema, but must contain the classification rules in the database itself as well as the data that the rules classify.

Another requirement that must be addressed is missing and inhomogeneous information. (2) We require the means to specify "default" or "probable" information for missing items rather than "not available". Bureaucratic rules are not sufficient here, for we envision a

searching/retrieval mechanism that is adaptive. If we change the meaning of a small, medium, or large string, we cannot possibly go back to all authors or the original source to reclassify all strings. Thus the retrieval mechanism must deal with old and new string classification systems, possibly of incommensurate measure. Further, for searching mechanism to take advantage of missing information, it must first know that the information exists, and then be able to take advantage of anything that is known about the information. For example, if I do not know what machines that a program will run on, I still may know that it will not work on a certain model. One approach is for a database to store all known negative and positive information about a key but this requires a great deal of space. In the absence of information on what machines will not run on, do I conclude that the program will or will not work on a specified model?

From this brief discussion we can see that we not only require a rich classification scheme, but also a rich inferential engine. The structuring mechanisms being used by AI researchers such as semantic nets, first order logic, frames, etc. seem promising, yet they also have difficulties. For example, many of our relationships do not follow first order logic, (3) and yet that logic is the basis of many experimental reasoning systems. Class membership systems have not found effective means to allow multiple class memberships. Refining of class membership criteria as we describe sub-classes is also a known deficiency in some implementations. Further, since the complexity of a useful classification mechanism is large, we need a way to specify a sample object and search for all others that are "like it". The methods of cluster analysis and projective geometry might well be useful.

In summary, a useful searching and retrieval mechanism for reusable software must have the following characteristics that distinguish it from contemporary database systems:

(1) The mechanism must be efficient with data that has many keys and few records per unique key combination.

(2) The mechanism must accommodate successive refinement style searches with many keys specified approximately.

(3) The mechanism must accommodate non-monotonic key relationships.

(4) The mechanism must accommodate highly volatile schemas.

(5) The mechanisms must enable searches by analogy or "best match" criteria.

The variety and complexity of program elements and algorithms makes the ideal solution to searching and retrieval beyond our present abilities. However, graceful growth toward the ideal will ensure that resources applied to present problems are not irrelevant to the more complex future. Several steps suggest themselves.

(1) Define a preliminary taxonomy of software units and a second, allied taxonomy of algorithms. The aim should be to scope the richness and variety that is required with a definite tendency to idealism rather than practicality.

(2) Define ideal inference mechanisms required by the taxonomies from 1. above.

(3) Select subsets of 1 and 2 as subgoals that can practically be constructed with todays equipment.

(4) Provide guidelines to software developers for documentation and classification of early candidates for the software library so that many candidate elements can be entered and the first systems given a significant, realistic test.

References

(1) Bentley, J.L. and M. Shaw, "An Alphard Specification of A correct and Efficient Transformation on Data Structures", Proc IEEE Conference on Specification of Reliable Software, April 1979, pp 222-237.

(2) Moore, R. "Reasoning about Knowledge and Action", Technical Note 191, Artificial Intelligence Center, SRI International, Menlo Park, 1980.

(3) Hewitt, Carl and Peter de Jong, "Open Systems", in "On Conceptual Modeling", Brodie, M. L. Mylopoulos, J. and Schmidt, J. W. eds. Springer Verlag, New York, 1984.

Grumman Data Systems is actively equipping itself to effectively develop software in Ada. This requires that we obtain expertise, hardware, and extensive training capability to make the transition to a new language as smooth as possible. For several years, we have participated jointly with Grumman Aerospace in the AdaTec activities and the Kapse Interface Team from Industry and Academia (KITIA). Over a year ago, we established a development team that we call the Ada Lab to lead the company into the new technology.

The Ada Lab has constructed a major software system that simulates the in-flight refueling of aircraft by a tanker, including not only the control of the actual fuel transfer operation, but also the communications to establish refueling need and the navigation required to attain a rendezvous. The simulation was chosen to explore the implications of Ada on program design methods, to test the independent tasking capability of Ada, and to understand the practical implications of the very strong typing in Ada and its impact on inter-task communication. The resulting system contains over 15,000 lines of Ada and runs on the Data General Ada Development Environment with DEC color graphics display hardware.

To promote the effective production of uniform Ada code, the lab has produced a 47 page Ada Style Guide that is consistent with modern software engineering methodologies. The Ada Lab provides seminars to introduce managers and developers to Ada. These have ranged from brief one hour synopses to comprehensive four hour training sessions. In February 1984, we will furnish all software managers in the company with an Ada Fact Pack that contains an overview paper, the Ada Style Guide, the Ada Language Reference Manual, an Ada glossary, a guide to the Ada literature, a guide to Ada training resources and an index of our own extensive Ada library that contains over 100 catalogued items.

At present, Grumman Data Systems has obtained and used Ada on DEC VAX 11/780 and Data General MV8000 machines using compilers from NYU, DEC, DG, and Telesoft. We are providing time on this equipment for key members of the Grumman Data Systems development team to sharpen their skills in Ada so that we will have knowledgeable and effective programming leaders.

As the corporation addresses the task of integrating Ada skills into the entire organization the Ada Lab team continues to extend our grasp of the new language. We have a program underway to refine and extend our Ada design methodology so that it might be an effective tool on Ada projects. This program will apply the methodology to a second demonstration project and attempt to quantify the utility of the method by the end of 1985. A second team is investigating the issues involved with moving Ada code from one machine to another as we continue to acquire additional diverse hardware and Ada software systems. A summary document is expected by the middle of 1985. A third team is determining feasible approaches to establishing a library of re-usable modules in Ada to further enhance our Ada productivity. This team expects to thoroughly understand the complex issues and to recommend a long term research and development plan by the end of 1985. We expect to participate with others addressing this difficult problem via the DoD STARS program. Finally, we are exploring the issues of using Ada in a fault tolerant environment by chartering a team to develop an Ada compiler on a fault tolerant architecture.

By these and other efforts, Grumman Data Systems intends to have the ability to effectively apply the new Ada language to a broad range of modern software problems.

RESUME

JOHN D. LITKE

John Litke's professional interests include programming languages, software engineering methodologies, human factors, fault tolerant software, and optimizing algorithms.

EXPERIENCE

Grumman Data Systems Corporation - Assistant Director for Software Technology - Present

Responsible for research and development programs in advanced languages (including Ada) database management and software tools.

Photocircuits, Manager of Computer Engineering - 1/80 - 10/84

Designed and developed a graphic based CAE system for NC machine programming with emphasis on human factors, reusable software, and concurrent processing.

Designed and developed an innovative approach to the optimization of NC machine programs. This algorithm saved over $700,000 per year and was an order of magnitude improvement over existing algorithms.

Designed and developed a new costing and manufacturing engineering system based on a message passing design and interpreted specifications. The result was a dynamically configurable software system that could be customized by users with no software engineering involvement.

Designed and developed a new language that extended and enhanced an existing HOL for improved programmer productivity and program maintainability. The language is now in extensive use by sharing via the users groups.

Designed and developed a reusable suite of human interface routines that tolerated and correctly interpreted many forms of errors.

Designed a prototype expert system approach to product engineering for custom printed circuit board manufacture.

Listed in Who's Who in Computer Graphics.

Participated in industry committees to design a graphical exchange specification system that anticipated and was a strong influence on the IGES effort.

Bell Telephone Laboratories, Member of Technical Staff 6/76 - 1/80

Developed and extended software engineering tools to provide a complete environment for development of FORTRAN based code, including a timesharing enhancement to an operating system, compilers, pretty printers, and code analyzers. The result measurably improved productivity in the department.

Designed and developed an operating system for a multi threaded communications enhancement to run on top of an existing real time operating system. It used time slicing, priority scheduling and dynamic memory management to multiplex the communications of up to 15 concurrent processes over one dial

up channel via a DDCMP protocol.

Johns Hopkins University, Instructor and Research Staff 9/65 - 6/76

Taught a wide variety of undergraduate and graduate physics courses. Research speciality was in plasmas and quantum electronics.

Developed efficient and sensitive computer algorithms for delicate line shape analysis in the presence of noise. The innovative algorithm required new ideas in guided min/max searching for a five parameter highly non-linear functional representation.

Designed and constructed computerized data acquisition equipment that for the first time allowed submicrosecond detailed analysis of spontaneously emitted line shapes.

PUBLICATIONS

(1) "EMC Design Considerations for Printed Circuit Boards", presented to the Printed Wiring Symposium, Mijas, Spain, (1984).

(2) "Human Factors in CAD System Design", presented to the Kollmorgan QTI Conference, 1981.

(3) "A Practical Solution of the Traveling Salesman Problem with Thousands of Nodes", CACM, 27, no. 12 :; 1227-1236 (1984)/

(4) "An Alternate Approach to Software Development", IEEE Software Engineering, (in press).

(5) "Software Systems Development", internal Photocircuits publ. JDL-8, 1982.

(6) Department 4142 Time Share System, Case 38649-16, 1988.

(7) RMMS - TFMS/CTMS Interface Specification, BTL internal publ., 1978.

(8) Evaluation of an Experimental Radio Performance Monitoring System, Case 38649-16, 1979.

(9) A New CTMS Database System, Case 38649-16, 1979.

(10) Results of a TFMS Automated Database Field Experiment, Case 38649-23, 1979.

(11) "Numerical Solution of the Boltzmann Equation in Plasmas, and Collisional and Radiative Processes in Ion-Laser Plasmas", proposal submitted to the National Science Foundation, 1969.

(12) "Excitation Processes in an Argon Ion Laser", B.G. Bricks, D.E. Kerr, and John D. Litke, presented to the 24th Gaseous Electronics Counterence, Gainesville, Florida, October 1971.

(13) "Spontaneous Atomic Line Shapes from an Argon Ion Laser Discharge", JQSRT, 12, 411-419, (1977).

(14) "Current Modulation in a Pulsed Argon Ion Laser Discharge", J. of Appl. Phys 48, 1385-6, (1977).

EDUCATION

Ph.D. in Physics - Johns Hopkins University 1976

B.S. in Physics - Massachusetts Institute of Technology 1965

Grumman Data Systems Corporation

GRUMMAN

Searching and Retrieval for Automated Parts Libraries

A Library Is:

An archive

A reference collection

A circulation source

Parts Libraries Will Be:

Large

Heterogeneous

Dynamic

Implementation Problems

Nomenclature

Multiple keys

Variable taxonomy

Missing information

Taxonomy Problems

Programs/subprograms

Packages

Algorithms

Fragments

New Directions

Enrich searching

Constrain variety

To Enrich Searching:

Define a taxonomy

Define inference mechanisms

Provide statistical learning

To Constrain Variety:

Define taxonomy of algorithms

Define interface mechanism

- Self describing data
- Optional context information

REUSABLE SOFTWARE IMPLEMENTATION PROGRAM: RESPONSE TO REQUEST FOR INFORMATION

February 11, 1985

WP-23

SofTech, Inc.
3 Skyline Pl., Ste 510
5201 Leesburg Pike
Falls Church, VA 22041

Contact

John G. McBride

Advanced Programs Manager

Abstract

The Graph Analysis And Design Technique (GADT) is a visually oriented systems development environment that is based on industrial and military techniques of software documentation. It is consistent set of graphics tools that support the systems development process from requirements analysis through implementation. The GADT environment maintains a graphic representation of software at the level of existing manual documentation methods such as SADT?TMO and provides a semantics for the execution of this representation with a library of primitive Ada?TMO units. We believe that this approach has important implications for software reusability at the level of requirements reuse as well as for the development of a library of reusable software components.

Section 1

SUMMARY - REUSABLE SOFTWARE FOR MISSION CRITICAL EMBEDDED SYSTEMS

SofTech, Inc., is pleased to respond to the Reusable Software Implementation Program (RSIP) Request for Information. SofTech has been directly pursuing the development of a software engineering environment for reusable software that will provide automated support for the full range of systems engineering tasks from requirements definition to implementation.

We believe that software must be shared at the requirement level as well as the module/program level. A support environment for an integrated development methodology spanning the entire system life cycle is needed to produce and disseminate software that is shareable at these levels. We also believe that this support environment must be easy to use if it is to become widely accepted rather than resisted. This would encourage cooperation with RSIP goals by making the RSIP method the path of technical least resistance for the project staffs.

We are investigating a Computer Assisted Design workstation for software that supports the Graph Analysis and Design Technique (GADT). This concept evolved from earlier SofTech work in reusable signal processing software for the AN/UYS-1. As will be discussed below, this work brought to light some reusability issues specific to real-time systems and some issues generic to any large application. The GADT workstation would provide a development methodology combining features of a data-flow operating system, the Structured Analysis and Design Technique (SADT), and an Ada based detailed design methodology.

SADT like data-flow system design graphs, would be used by the operating system to specify the topology of an executable system-flow graph. The workstation will support SADT based requirements analysis, and, without loss of continuity, the data-flow requirements graphs would be mapped onto functional system design graphs, and then onto executable system modules. At the point of system functional decomposition where single programs could serve to perform the functions required by graph nodes, data-flow graph description would be discontinued and the terminal nodes would consist of Ada units. The graph edges would consist of typed data queues managed by the graph operating shell at runtime. The queue management system would encourage the production of simple, reusable Ada programs by relieving the primitive processing modules of the need to manage data buffering and program synchronization. These reusable programs could form the basis of a development methodology for reusable software. Machine representable specifications would allow for retrieval and dissemination of software specifications.

Section 2

CURRENT SOFTWARE DESIGN METHODOLOGY VERSUS REUSABLE SOFTWARE

The Phase-I RSIP reports (1,2) indicate that there is no technical impediment to reusing software if reusability is considered during design. However, the issues involved in making software reusable are not well understood at this time and the reports give no clues as to the nature of the design procedures that are required to assure a high yield of reusable modules from a design effort. However, even though the area of reusability is relatively new, some basic points about reusability do seem clear. Perhaps the most important of these is that the amount of reusable software obtained using present methods has been so small that we are lead to wonder if they actually discourage reuse of software.

2.1 Reusability in the Small

The major exception to the paucity of reusable products is support software including both applications packages and operating system software. As numerous applications are implemented in an area, a set of primitive routines are defined over time that become support software and are reused for future developments in the area. These are often distributed by commercial vendors for commercial advantage. For example: signal processing macros, such as Fast Fourier Transform; crosscorrelation; recursive filters; and the like, are provided with commercial array processors to improve the marketability of the hardware. Two disadvantages of this natural evolutionary approach are that it is slow, and that it is designed to benefit the vendors rather than government.

The nature of this type of software may, however, provide some insight into reusability in the small. There are numerous examples of support packages, including the IBM scientific subroutine package, and various signal processing support software packages. These seem to share two characteristics:

- They are comprised of discrete single function routines that can be composed on each other to provide a large variety of complex applications.
- They are input/output free, requiring the calling applications to provide the larger system structure that interconnects these modules.

Two issues in reusability are then: How can the development and capture of domain specific primitives be encouraged during routine applications development?, and; What is it about our present methods that discourages this production and capture? We will argue below that present methods discourage the production of reusable software by failing to partition the system functions in the small, from system structure in the large. Other workers in the field including Stevens (3) and DeRemer and Kron (4) have made the same observation.

2.2 Communicational Cohesion in Present Software Development Efforts Guarantees the Production of Nonreusable System Components

In the present systems development process, the data-flow portions of the systems, implemented in procedural Languages, are communicationally cohesive, in the sense given by Meyers, Constantine and Stevens in (5), with the algorithmic portions. This means that the data-flow management code is intermixed with algorithmic code on the basis of common access to data items. The applications modules themselves must reflect the system structure as well as execute the algorithms. Typically, systems are implemented as a relatively few, large tasks that are unique when viewed at the highest level. This type of cohesion guarantees at the outset that the developed code will not be reusable since a new application will require a new structure and the existing modules will have to be redesigned. To have any hope of developing reusable code, the modules developed must be functionally cohesive. That is to say: "In a functionally bound module, all of the elements of the module are related to a single function." (5) Present methods, then, are seen to encourage the development modules with distinct multiple functions grouped together on the basis of access to

common variables rather than on the basis of the functions actually needed.

2.3 Temporal Coupling in Realtime Systems Reduces the Degree of Reusability

In realtime applications there is an additional problem of time line analysis that must take peak loading into account. This requires that each system function must be dealt with in the context of all of the other functions operating in the system. We refer to this phenomenon as temporal coupling because the existing system functions constrain when a new function can be scheduled on the system operation timeline. Thus, detailed knowledge of the surrounding system is required for maintenance and extension. Small modifications in such an environment often have major implications for the structure of the surrounding system. When and whether a new system function can be added becomes more and more tightly coupled to the software already operating, as the system loading becomes heavier. Major modification or reuse of existing realtime systems is thereby made expensive and difficult.

Our experience with signal processing applications on the AN/UYS-1 Signal Processor indicates that a data-flow paradigm, such as the ASP Common Operational Software (ACOS), for software implementation allows an automated trade-off of memory for knowledge of overall system structure. In present software architectures, signal processing applications are spread out over many tasks. The tasks are multifunctional, complex, and strongly time coupled. They are required to execute in certain intervals, to avoid resource contention with other real-time tasks. This temporal coupling requires a detailed time line analysis of operational systems to be developed, and the usability of a given task depends on the entire system surrounding that task and its resource utilization. This time coupling becomes more and more severe as the processor utilization increases. As a consequence, modifications to existing systems become more and more difficult and expensive to make. By contrast, the ACOS shell runtime scheduler has been demonstrated to adequately manage the runtime complexities due to temporal coupling in exchange for, possibly, increased use of queue memory. It has been demonstrated to do so until a very high level of processor utilization has been reached.

Section 3

THE GADT SYSTEMS DEVELOPMENT ENVIRONMENT

We believe that a unified methodology spanning the entire system life cycle and an associated support environment is needed to reduce analysis, design, and implementation failures. We are developing a Prototype environment, based on the documentation techniques described above that will:

- o Provide automated support beginning at requirements analysis with tools to support SADT, which has proven itself as an ideal method of describing the functional requirements of a broad class of systems;
- o Provide a unified set of tools that allows the work products from each life cycle phase to be used in the subsequent phases;
- o Support transition between design and implementation phases by stepwise refinement until a system is produced;
- o Present the in-process systems analysis and design in a manner which does not overload the information processing capabilities of the human implementers;
- o Define a runtime behavior for the system specification that allows the high level specifications to actually control the execution of the lower level system software.

This environment also provides automated support at the junctures in the phases of the system life cycles, and thereby reduces the chances that the system, as built, will fail to meet its specifications (Figure 3-1). The highest level of this documentation consists of data flow graphs, drawn with the aid of a graph-directed editor. This editor is used to specify the structure of applications in Hierarchical data flow graphs. The runtime behavior of the graphs is given by the computational model first described by Rodriguez (6,7) Karp and Miller (8) and Dennis (9) have given similar models of computation. This model was chosen on the basis of our experience with the ASP Common Operational Software (ACOS), a data flow language SofTech developed for the UYS-1 Advanced Signal Processor (10). The GADT environment combines SADT data flow documentation, with a data flow runtime shell that schedules system software modules based on their data flow specifications. The nodes are viewed as data transforms that are activated when sufficient data is available for processing.

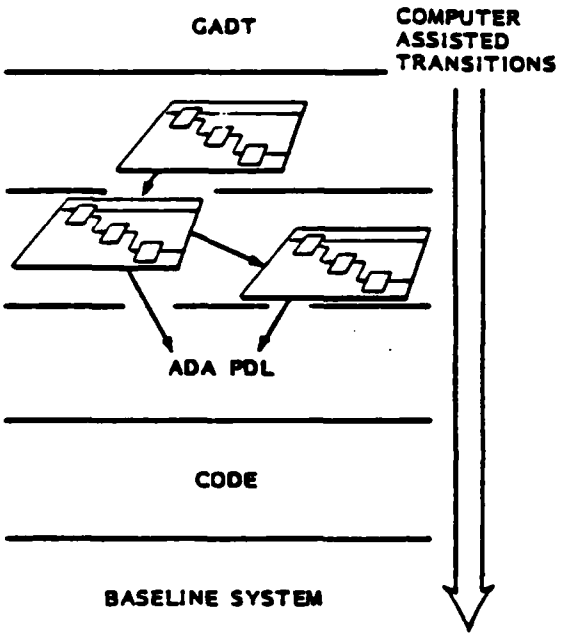


Figure 3-1 The GADT Development Environment

GADT also provides a separation of algorithmic portions of system software from the buffering and synchronization among the nodes. Using GADT, the application designer is concerned with the functions he is required to perform and a declaration of the relationships among them. Implementation details associated with synchronization, dispatching, access control, communication, storage allocation, and process (node) scheduling are handled by the GADT runtime shell using these declarations. We believe, as does Stevens (3), that this approach to development will have beneficial effects on the reusability of the node software.

This follows from the fact that in the present systems development process, the data flow portions of the systems, implemented in procedural Languages, are communicationally cohesive, in the sense given by Meyers, Constantine and Stevens in (5), with the algorithmic portions. This means that the data flow management code is intermixed with algorithmic code on the basis of access to common data items. Modules developed under present methods must implement the system structure as well as that of the processing algorithms. This type of cohesion guarantees at the outset that the developed code will not be reusable since a new application will require a new structure and the existing modules will have to be redesigned. To have any hope of developing reusable code, the modules developed must be functionally cohesive. That is to say: "In a functionally bound module, all of the elements of the module are related to a single function." (5) The GADT environment is designed to facilitate the production of reusable software by separating algorithmic code and the buffering and synchronization processes required to supply the algorithmic codes with data.

3.1 GADT Computational Model

When a GADT specification graph is executed, each node (or activity in the terminology of SADT), is scheduled for execution based on a set of data flow activation rules by the GADT executive (11). This data flow model allows the operation underlying a node to proceed as soon as all data required are available. No explicit synchronization or concern for parallelism is required of the implementer.

GADT implements the edges connecting the process nodes as queues of typed data objects of arbitrary complexity. When a node executes, the required input objects are read from each input queue. After execution, data objects may be consumed by removing them from the queues depending on the node execution parameters. Since the computational model depends only on the state of nodes input queues, a node can execute repeatedly if its input requirements are still satisfied. After a node has executed, the data produced are placed on the indicated output queues.

The graph in Figure 3-2 illustrates this model. Initially the input queue, Q1, to node A contains 1000 data objects, with Q2 and Q3 empty. Node A can execute since its input requirement is met, and it consumes 100 objects. It produces 100 objects for placement on Q2. Node B requires 500 objects on Q2 before it can execute, and consumes all objects read. Therefore, Node B requires node A to execute five times before its input requirement is satisfied. Figure 3-3 shows the indicated execution sequence.

3.2 The Graph Editor

To develop a GADT application, the designer draws the flow graphs using a graph-directed editor that provides interactive commands for rapid graph definition. This editor allows the designer to represent the system structure declaratively rather than procedurally. The designer places each node on the screen, names its underlying function, and connects the nodes to form the application structure. The named functions may represent another graph, or a primitive function drawn from an application library. If the function underlying a node is undefined, the name serves as a place holder for future definition.

The graphics editor, serving as the primary user interface for both applications development and run time debugging, allows a hierarchy of graphs to be created and reused as building blocks. Modifications of graphs may include renaming nodes to specify new underlying functions, deleting nodes, moving nodes to another screen coordinate, and disconnecting and reconnecting nodes. A zoom function allows the user to navigate the specification graphs, or to view the contents of a node, whether it is a subgraph or the procedural description of a primitive function.

One of the design guidelines for the GADT editor is that a maximum of knowledge about the nature of the data flow graphs and their content should be incorporated into the edit command language. The developer can then communicate with the editor about a commonly held, high information content model of the system under development using concise edit commands rather than long textual commands (Figure 3-4). Until now, analysis and design techniques have involved the creation of high information content models of systems on paper to ensure that all parties to the development hold a common model of the system under development. Syntax directed text editors such as that developed for Gandalf (12) have been used to show that substantial effort can be saved by the developer if the editor is knowledgeable about the syntactical structure of the language being edited. The GADT editor applies this principle to data structures representing application software

NODE A		NODE B	
THRESHOLD	= 100;	THRESHOLD	= 500;
READ	= 100;	READ	= 500;
CONSUME	= 100;	CONSUME	= 500;
PRODUCE	= 100;	PRODUCE	= 100;

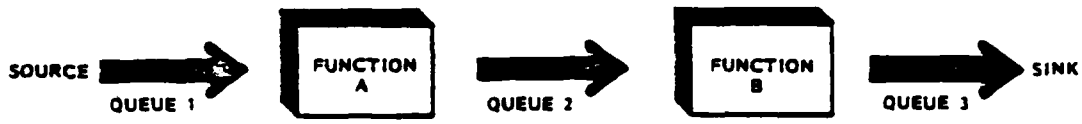


Figure 3-2. GADT Directed Flowgraph

Figure 3-3. Node Execution Sequence

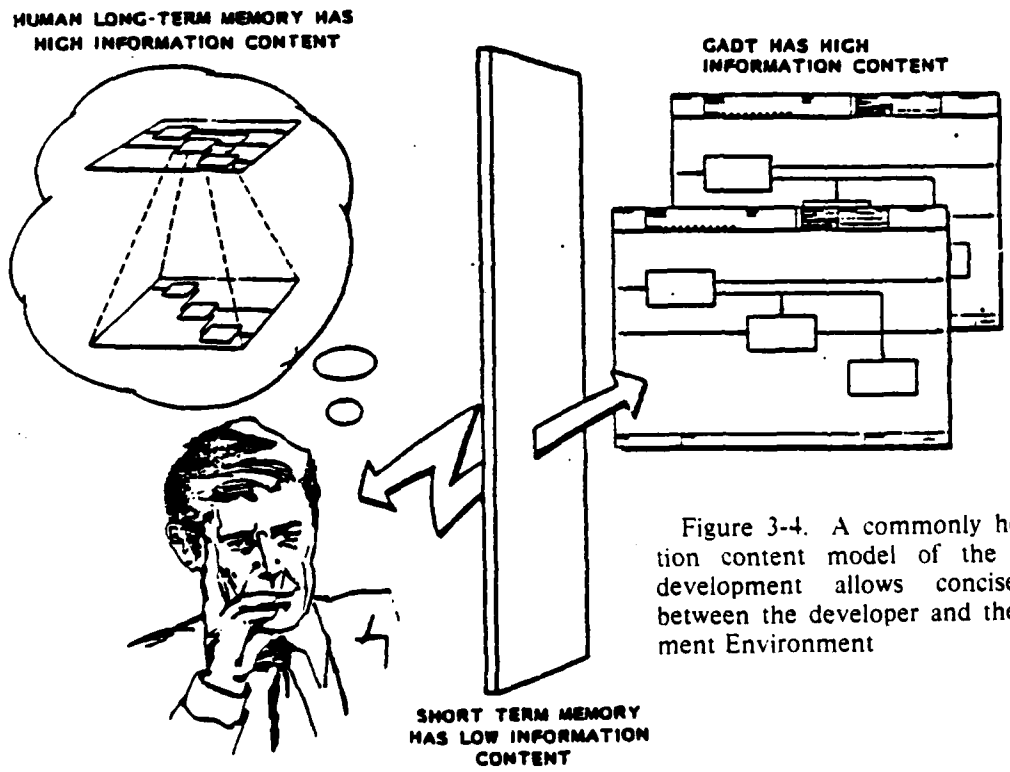


Figure 3-4. A commonly held, high information content model of the application under development allows concise communication between the developer and the GADT Development Environment

specifications to achieve a similar reduction of effort.

Another advantage to this approach is that if the functions underlying the graph nodes have been defined, semantical error checking is possible at edit time. When a node is connected, the data type of the nodes being connected can be checked for consistency. Incompatible data types result in the generation of a connection error and the operation is disallowed. Another form of interactive parameter checking ensures that all required node inputs and outputs are connected. This is equivalent to a higher level language consistency check on parameter passing and could be performed when the graph is saved in the library. The benefits of interactive graph-directed editing and error checking are superior to conventional development since errors are detected and fixed by the developer immediately, while the context of the system is still fresh. Many opportunities for errors are eliminated due to the method and sequence of developing data flow graphs. As Smith et. al. (13) have observed, declarative task description holds the potential to force users to properly structure their task interfaces.

The editor can also be used to enforce the specification of the system within the limits of human cognitive limitations. In his now famous studies on the limits of human perception, Miller (14) pointed out that the human digit span, which is seven plus or minus two, gives an indication that the average person can keep only approximately seven items in short term memory at once. Further, when this memory is overwhelmed, all of the items are lost rather than only those after seven. Therefore, the amount of information presented to a human developer at any one time should be strictly limited, and if too much information is presented useful perception is lost. Figure 3-5 shows a drawing made by a group of users who were presented with too much information during a structured analysis and design. While amusing, this picture shows that when presented with too much information, we can remember only an impression of confusion. This fact has been used in the design of some of the software development methodologies including SADT, GADT, and Yourdon-constantine Structured Analysis.

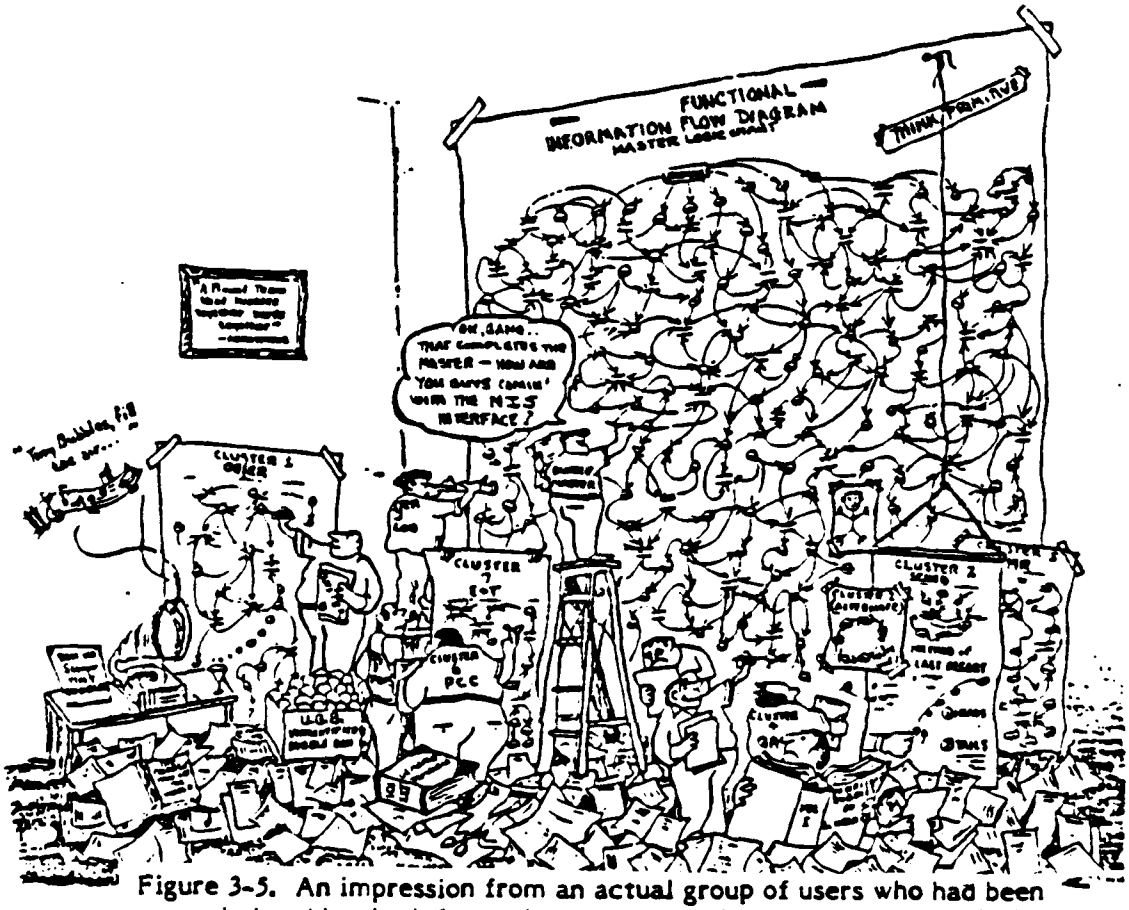


Figure 3-5. An impression from an actual group of users who had been overwhelmed by the information content during a structured analysis

Section 4

CONCLUSIONS

GADT provides CAD support of a data flow systems development environment that we believe will increase system development productivity. It provides a unified methodology for analysis, design, and implementation of software systems and automated tools to facilitate the use of the methodology. The data flow technique used is valuable for modeling system requirements, controlling system complexity, and reusing software components. The graphic specifications are machine-independent and allow the designers to concentrate on the functions of the application. Specification of synchronization, parallelism, and system level type checking are provided in a convenient manner. As described, these features may facilitate the development of software that is reusable at the library modules and requirements levels.

Section 5

BIBLIOGRAPHY

(1) The IBM Corporation, Reusable Software Implementation Program (RSIP) Software Development Methodology Reviews (Draft). IBM Federal Systems Division, 9500 Godwin Drive, Manassas, Virginia.

(2) P. Grabow, W. Noble, C. Huang, Reusable Software Implementation Technology Reviews. Hughes Aircraft Company, Ground Systems Group, Fullerton, California, October 1984, N66001-83-D-0095, FR 84-17-660.

(3) W. Stevens, "How Data Flow can Improve Application Development Productivity." IBM Systems Journal, Vol. 21, No. 2, 1982, pp. 162-178.

(4) F. DeRemer, H. Korn, "Programming-in-the-Large Versus Programming-in-the-Small." IEEE Transactions on Software Engineering, June 1976, pp. 80-86.

(5) G.J. Myers, L.L. Constantine, W.P. Stevens, "Structured Design." IBM Systems Journal, No. 2, 1974, pp. 115-139.

(6) J.E. Rodriguez, "A Graph Model for Parallel Computations." Doctoral Dissertation, Massachusetts Institute of Technology, September 1967.

(7) J. Rodriguez, S. Greenspan, "Directed Flowgraphs: The Basis of a Specification and Construction Methodology for Real Time Systems." The Journal of Systems and Software, Vol. 1, Issue 1, 1979, pp. 19-27.

(8) R.M. Karp, and R.E. Miller, "Properties of a Model for Parallel Computation: Determinacy, Termination, Queueing." SIAM Journal of Applied Mathematics, Vol. 14, No. 6, November 1966.

(9) J.B. Dennis, "First Version of a Data Flow Procedure Language." An unpublished technical memo from the Laboratory for Computer Science, Massachusetts Institute of Technology. MIT/LCS/TM-61, May 1975.

(10) ASP Common Operational Support Software Methodology (ACOS) Benchmark Report. NAVSEA Contract N00024-80-C-7198, Doc. 3140-70.1, 30 August 1982.

(11) J.G. McBride, "Graph Analysis and Design Technique Methodology." An unpublished SofTech Internal Research and Development Program technical memo, September 1984.

(12) D.S. Notkin, and A.N. Habermann, "Software Development Environment Issues as related to Ada." Tutorial: Software Development Environments, A.I. Wasserman, Ed., IEEE Computer Society Press, 1981, pp. 107-135.

(13) R.G. Smith, et. al., "Declarative Task DescrIPtion as a User Interface Structuring Mechanism." Computer (a publication of IEEE), September 1984, pp. 29-37.

(14) George A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Ability to Process Information." Psychological Reviews, Vol. 2, No. 2, March 1956.

RESUME

JOHN G. McBRIDE

Principal Consultant

EDUCATION

B.S., Engineering Physics, University of Oklahoma.

Graduate level Computer Design and Management Course Work, Data Flow Architectures, MIT.

SUMMARY OF EXPERIENCE

Mr. McBride has served in a progression of technical and management roles with experience in digital signal processing, data flow languages, automated software production, SIGINT collection and processing systems. He has led the development of multimillion dollar computer based systems and has held responsibility for a million dollar plus cost center.

Mr. McBride is a specialist in modern software engineering methodologies and has pioneered the development of computer aided design techniques for software development. His activities include strategic planning, briefing of senior DoD officials and promotion of innovative concepts. He has served on several DoD Strategic Planning Committees.

PROFESSIONAL EXPERIENCE

SofTech, Inc. (3 years) Director, Advanced Programs. Responsible for business acquisition, IR&D, strategic studies, special projects and product development for the Washington Division.

Formerly, Director, Signal Processing reporting to the Manager of the Washington Division. Responsible for total performance of the directorate relative to corporate financial objectives. Specific responsibilities include program and project management, business development and planning. Primary efforts have been focused on the development of the Advanced Signal Processor Common Operating System (ACOS) to support a new Navy Standard product for the AN/UYS-1 Signal Processor. ACOS is a new software methodology based on the data flow concept and directed graph theory which promises to reduce signal processing software costs by as much as 50%.

Mr. McBride participated on task forces and ad hoc committees which required direct interfacing with NAVMAT and the Office of the Assistant Secretary of the Navy to promote the funding of ACOS concept development.

He has developed concepts for computer aided design techniques for Ada software development. Concept includes the use of interactive, graphic workstations to represent software systems and automated Ada source code generation derived from the graphic representations.

Litton Systems, Inc. (4 years). Manager of Intelligence Systems. Directed all phases of intelligence systems acquisition and development. Assisted the Government in defining the requirements of an enhanced automated SIGINT analysis system.

Project Manager for the $2.4 million Automated Analysis Aids (A^3) system. A^3 provided the first fully integrated system for DoD for high speed digitizing (40 MHZ), digital signal processing, interactive graphics and relational DBMS. Based upon a dual PDP-11 architecture the A^3 System incorporated a common analyst language, standardized interfaces of file formats and a library of modular signal processing functions.

Project engineer for an Air Force sponsored study to identify the tools and techniques necessary to process and analyze complex ELINT signals collected in the next decade. Developed requirements for a wide band digital analysis system which included high speed digitzer-buffer, interactive graphic display, an array processor and PDP-11/70 controller for the U.S. Navy. Project engineer for ELINT analysis under contract from the U.S. Government.

Lockheed Corporation (1 year). Operations Research Engineer. Staff to the EPM of the Tactical Airborne Signal Exploitation System (TASES). Assessed the impact of system design on operational requirements. Designed ground processing digital ELINT analysis techniques to validate emitter classification and build a precision ELINT data base. Developed ELINT signal processing and classification algorithms to reduce real-time identification ambiguities. Developed models to predict the mean-time-to-intercept of various high interest emitters using step-tuned intercept receivers as influenced by environmental emitter densities, receiver dwell and analysis time, total RF coverage and antenna dynamics of emitters and intercept system. Developed SIGINT threat scenarios to evaluate throughout and system effectiveness.

NOSIC, Naval Intelligence Command (2 years). Intelligence Research Specialist. Analyzed and evaluated current foreign military and space operation, C^3 and tactical doctrine from all source intelligence data.

U.S. Navy - VQ-2 (4 years). Mission Commander/Director of Analysis. Over 2000 flight hours in the EP-3E aircraft with hands-on experience in collection, processing and analysis of SIGINT. Responsible for Squadron level ELINT analysis, and mission intercept reports. Developed synthesized analytical techniques with HULTEC applications.

COMMITTEES

Participated in the ASW Signal Processing Study for the Assistant Secretary of the Navy for Research, Systems and Engineering. Study focused on acquisition strategies during the next 20 years.

Involved in several ad hoc committees associated with NAVMAT and PM-4 of the U.S. Navy.

PUBLICATIONS

"A Computer Aided Design Methodology for Ada Systems," Mar. 84, SofTech
"Complex ELINT Study," Feb. 80.
"An Algorithm for Recognizing Radar Scan Types," June 78.
"Radar Fingerprinting Using Precision PRI Measurements," July 78.

SOFTECH
THE SOFTWARE TECHNOLOGY COMPANY

GRAPH ANALYSIS AND DESIGN TECHNIQUE

(GADT)

AN Ada™ BASED SYSTEM DEVELOPMENT APPROACH

PRESENTED TO

STARS APPLICATIONS WORKSHOP

BY

JOHN McBRIDE

APRIL 9-12, 1985

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

3 Skyline PL, Ste. 510, 5201 Leesburg Pike Falls Church, VA 22041 (703)931-7370

Ada™/SYSTEM INTEGRATION

- WHY AN INTEGRATED APPROACH IS NEEDED
- MAJOR REQUIREMENTS FOR AN INTEGRATED APPROACH
- GRAPH ANALYSIS AND DESIGN TECHNIQUE

SOFTECH

THE PROBLEM WE ARE ADDRESSING

IN THE CURRENT SOFTWARE DEVELOPMENT PROCESS

- DISJOINT TOOLS FOR DEVELOPMENT
- TOO MUCH OPPORTUNITY FOR MISUNDERSTANDING
- EXPLOSION OF COMPLEXITY
- LACK OF ADAPTABILITY AND REUSE
- HARDWARE KNOWLEDGE REQUIRED AT DESIGN LEVEL

vg1358ow/2 2/9/83

SOFTECH

MAJOR REQUIREMENTS FOR AN INTEGRATED APPROACH

- UNIFIED FROM REQUIREMENTS ANALYSIS TO SYSTEM CONSTRUCTION TO SUPPORT STEPWISE REFINEMENT WITHOUT MAJOR DISCONTINUITIES
- PROVIDE HIERARCHICAL LEVELS OF ABSTRACTION TO AID IN COMPLEXITY MANAGEMENT
- DEVELOPMENT RULES WHICH ARE DECOUPLED FROM UNDERLYING SYSTEM ARCHITECTURE

MAJOR REQUIREMENTS FOR AN INTEGRATED APPROACH (CONT.)

- MAINTAIN DESIGN INTEGRITY THROUGHOUT DEVELOPMENT PHASES
- PROVIDE COMPONENT LIBRARY FOR REUSEABILITY AT ALL LEVELS OF ABSTRACTION
- PROVIDE VISUALIZATION OF SYSTEM FUNCTIONS FOR RAPID INTERPRETATION AND UNDERSTANDING BY THE DESIGNERS
- AUTOMATICALLY GENERATE CODE AND ASSOCIATED DOCUMENTATION

GRAPH ANALYSIS AND DESIGN TECHNIQUE (GADT)

- GADT IS A UNIFIED METHODOLOGY FOR DEVELOPING COMPLEX SOFTWARE SYSTEMS
- GADT SUPPORTS CONCISE REQUIREMENTS DEFINITION IN GRAPHIC FORM WHICH ARE AUTOMATICALLY TRANSLATED TO Ada™ SOURCE CODE

SOFTECH

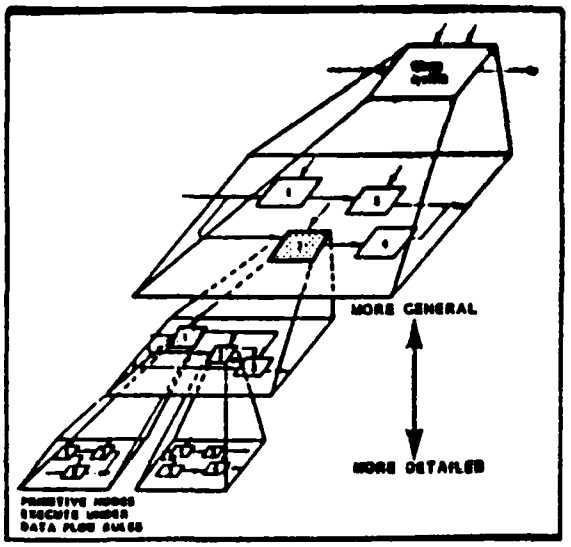
GADT ADVANCES SOFTWARE METHODOLOGY

GADT INTEGRATES:

- STRUCTURED ANALYSIS AND DESIGN TECHNIQUE (SADT™)
- DATA FLOW COMPUTATIONAL MODEL
- A SEMANTICS-DIRECTED GRAPH LANGUAGE EDITOR

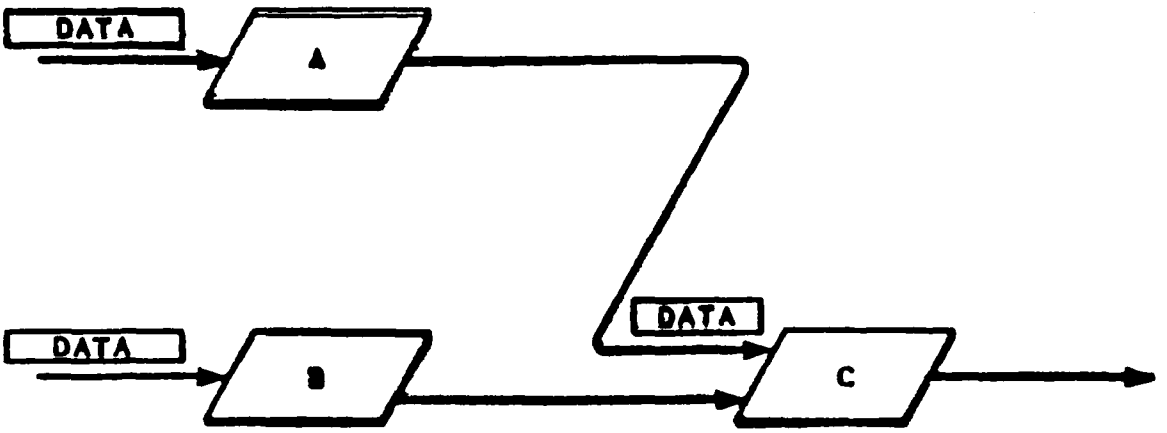
™SADT IS A TRADEMARK OF SOFTECH, INC.

SADT DATA-FLOW DIAGRAMS ARE HIERACHICALLY ORGANIZED TO AVOID INFORMATION OVERLOAD



SOFTECH

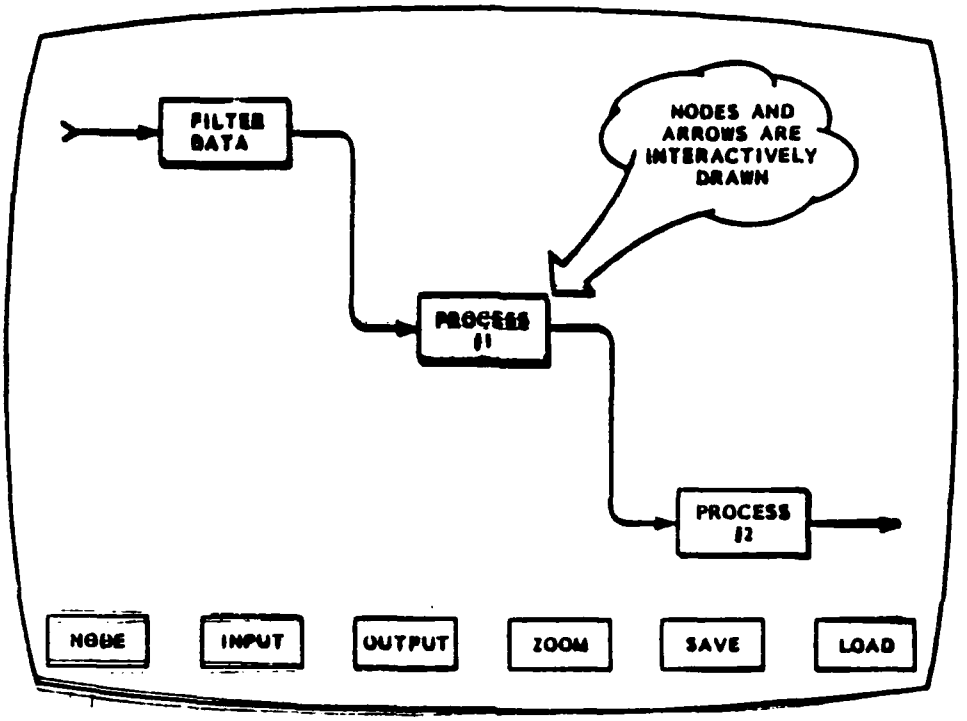
DATA FLOW COMPUTATIONAL MODEL



A NODE IS SCHEDULED TO EXECUTE IF

- ALL ITS INPUT QUEUES CONTAIN SUFFICIENT DATA FOR PROCESSING
- ALL ITS OUTPUT QUEUES CAN ACCEPT DATA

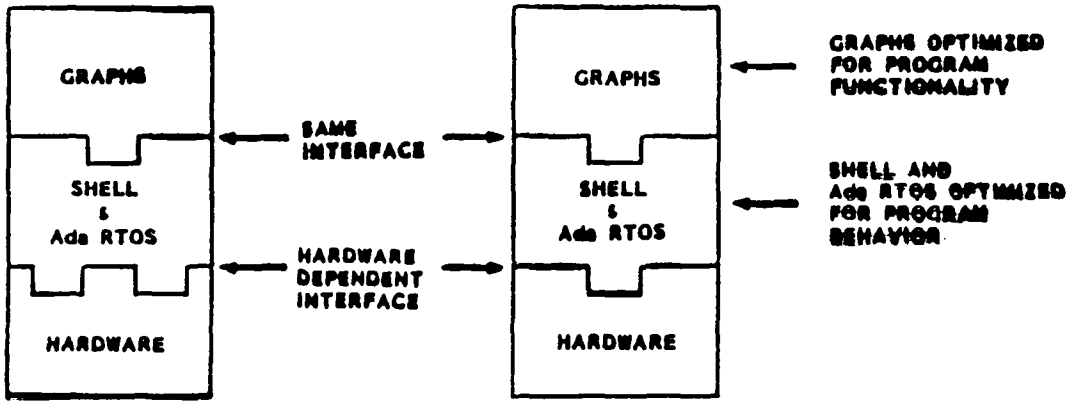
GADT GRAPHIC DISPLAY



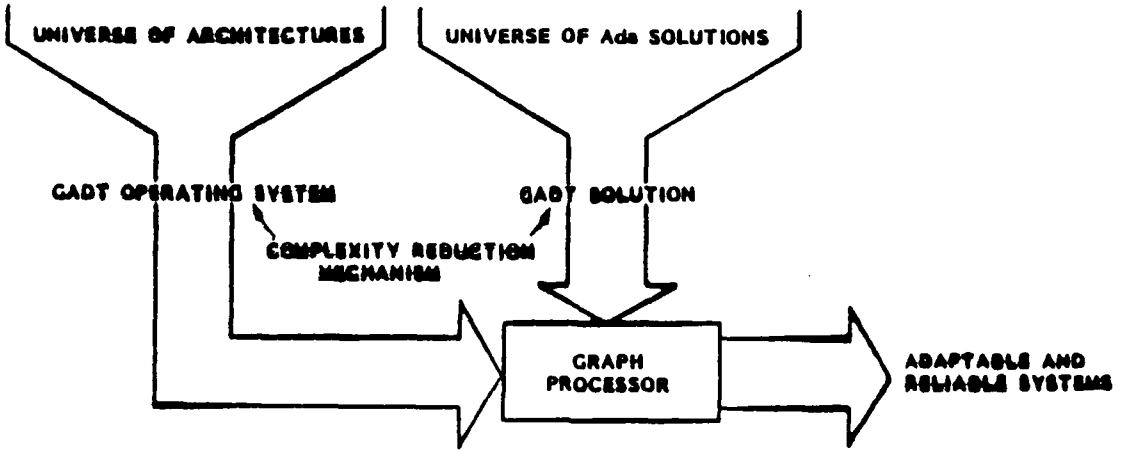
GADT GRAPHIC EDITOR ERROR CHECKING

- STRUCTURAL CONSISTENCY
  - CONNECTIVITY OF INTRA GRAPH NODES
  - CONSISTENCY BETWEEN HIERARCHICAL LEVELS
  - AVOIDING MULTIPLE CONNECTIONS TO THE SAME PORT
- FUNCTIONAL CONSISTENCY
  - DATA TYPE CHECKING BETWEEN SOURCE AND SINK NODES
  - INCOMPATIBLE NODE EXECUTION PARAMETERS (I.E., THRESHOLD < READ)

... GADT PROVIDES TRANSPARENCY OF HARDWARE TECHNOLOGY



ACHIEVING ADAPTABLE AND RELIABLE SYSTEMS



CONCLUSIONS

GADT PROVIDES FRAMEWORK FOR Ada™/SYSTEM INTEGRATION WITH THE FOLLOWING BENEFITS:

- UNIFORM REPRESENTATION FOR IMPROVED COMMUNICATIONS
- AIDS IN COMPLEXITY MANAGEMENT
- HIDES PECULIARITIES OR HARDWARE ARCHITECTURE
- REUSABILITY OF GRAPHS
- AUTOMATED PROGRAM GENERATION

A SOFTWARE DEVELOPMENT METHODOLOGY FOR REUSABLE COMPONENT

Ron McCain
Federal Systems Division, IBM
Houston, Texas

ABSTRACT

Software reusability could potentially provide substantial economic benefits. Large-scale software component reuse, however, will not be possible without a software development approach that emphasizes the production of reusable software components. This paper defines the characteristics of reusable software and proposes a software development methodology that produces software components exhibiting these characteristics. The methodology is intended to supplement rather than replace other sound software development methodologies. In addition to describing the reusability-oriented thought process associated with the methodology, the paper suggests new work products and validation procedures to support the methodology.

Introduction

It is commonly recognized that software reusability could provide powerful leverage for reducing future software development costs. In fact, software component reuse could very well be the most promising area for a major software development breakthrough within the next decade. Any significant advances in software reusability will include a reusability-oriented software development approach as well as reusable software component library (1). Many papers on reusability have focused on the need for reusing software, and a component library as a means for accomplishing this. Relatively little attention has been given to the issue of how software should be constructed for reuse. If the software industry cannot adequately establish software development approaches that emphasize the construction of reusable software components, then attempts to reuse software from a component library will, of course, prove to be futile.

This paper presents concepts and a candidate methodology for the development of reusable software components. The methodology is intended to be applicable to customized software product development environments as well as specialized Component Development Groups (2). It is also intended to provide a systematic thought process to be used at all stages of software decomposition to influence the production of reusable components.

Characteristics of Reusable Software

Before a reusable software component development methodology is proposed, the characteristics of reusable software components and how to build software to exhibit these characteristics must be examined. For significant reuse, software components should possess the following characteristics:

(1) Component is applicable to multiple users. If a software component is needed only in a unique application for which it is developed, the component is clearly not reusable. Conversely, potential reuse can be maximized by developing components that have a substantial domain of applicability. Note that references to component "users" in this paper are intended, in general, to apply to other software entities requiring the services of the component to function properly, not end-users of the overall software product under development.

(2) Component is usable. Component usability is a prerequisite for component reusability. If a component is not constructed to satisfy the user's needs in a highly usable manner, the component may have limited reuse even if it has a significant domain of applicability. Major factors contributing to the usability of the component are as follows:

- Specification precision
- User knowledge proximity
- Interface abstractness
- Functional cohesion

The component implementation should have a significantly smaller impact on the usability of the component than the above factors.

(1) Interfaces are completely and accurately specified. All interfaces should be explicitly defined with a formalized specification that is separate from the implementation itself. The specification should include all information which must be provided to use the component, including procedural parameters, tailoring options, and user-supplied code.

(2) Component has minimum dependency on other components. Component users should be able to utilize the component with minimum dependencies on other components are assumed to have been previously addressed. The methodology suggests a thought process for decomposing a problem solution for the Current Specification into reusable subcomponents.

The step-by-step thought process is described below:

(3) Perform Domain Analysis For The Current Specification Component. A Domain Analysis should be performed prior to the implementation of the Current Specification to identify potential users of the software and their specific needs. This Domain Analysis will influence the software implementation to accommodate these needs. Potential reusability constraints implied by the Current Specification will also be identified. Emphasis should be given to identifying commonality across the domain under consideration which will form the basis for identifying abstractions with maximum reuse potential)i.e., candidates for reusable component implementation). This analysis will also be used to extend the domain of applicability of components under consideration for implementation.

(4) Reuse Existing Software If Available. Determine if existing software can be cost-effectively reused or recovered to satisfy all or part of the Current Specification requirements. If so, use the existing software. Otherwise, proceed with new development with the goal of producing reusable software. If only a portion of the Current Specification can be cost-effectively satisfied with existing software, define a new specification that reflects the Current Specification requirements not accommodated and repeat step 1.

(5) Define Current Specification Reusable Objects. Identify Reusable Objects that are applicable to the domain under consideration. Reusable Objects are abstract classes of data that have associated reusable operations.

Examples of Reusable Objects and their associated Reusable Operations are shown below:

Reusable Object	Reusable Operation
Stack	Push
	Pop
	Make Empty
	If Empty
	If Full
Set	Union
	Intersection
	If Null
Complex Numbers	Add
	Subtract
Air Data Sensor	Determine Airspeed
	If Powered On
Symbol Cross-Referencing	Determine Next Variable
	Determine Next Statement XREF

(6) Define Current Specification Reusable Abstractions. Determine the Reusable Abstractions applicable to the problem solution. This will include both the operations (i.e., object services) for the Reusable Objects as well as functional abstractions. The Domain Analysis should exert a significant influence on the definition of the Reusable Abstractions. Whenever possible, an attempt should be made to define Layered Abstractions to increase potential reuse. Layering can be used to partition abstractions to achieve different levels of reusability potential. Maximum reuse can then be achieved through the use of Primitive Abstractions, i.e., the Reusable Abstractions that are common for all levels of layering. With the object-oriented approach, primitive operations defined for

Reusable Objects may not be directly usable by the Current Specification component and may need to be supplemented by a Layered Abstraction utilizing the primitive operation. The "Determine Next Referencing Variable" and "Determine Next Statement XREF" Reusable Operations mentioned above are examples of Primitive Reusable Abstractions for a Symbol Cross-Reference Object. These Reusable Abstractions are broadly applicable to most users of Symbol Cross-Reference information. Specialized usage of the Symbol Cross-Reference information might require Layered Abstractions. For example, if a user wants to determine if interprocess variables have been properly protected, Layered Abstractions derived from the primitive operations could be as follows (Note the different levels of reusability potential for each Layered Abstraction):

-Determine If Interprocess Variables Are Protected-

-Determine Next Interprocess Variable

-Determine Next Variable

-Determine If Interface Is Properly Protected

-Determine Next Interface (Pair of Statement XREF's)

-Determine Next Statement XREF

(7) Define Abstract Interface Specification For Reusable Abstraction. A formal specification will be defined for each Reusable Abstraction defined above. The formal specification, called an Abstract Interface Specification, should include an explicit definition of all interfaces associated with the usage of the Reusable Abstraction. The Abstract Interface Specification should attempt to satisfy the following objectives:

-Minimize the possibility of change to the interface as a result of changes in component usage or component implementation, i.e., assume the interface to be invariant.

-Accommodate an optimum number of component users in a highly usable fashion.

-Accommodate changes in component implementation, including data representation, without effecting component usage.

Applicable Existing Software. Describes existing software that may be appropriate to partially or wholly accommodate the component specification. The existing software will become a candidate for reuse during the software implementation.

Domain Analysis Summary. Includes a definition of the potential user set and their needs that can be accommodated during the implementation of the Component Specification. Potential reusability constraints implied by the component specification should be identified. Any commonality associated with the domain under consideration should be described. An initial definition of abstract data objects and their associated operations, as well as other functional abstractions, should also be provided. The Domain Analysis Summary will provide a basis for establishing Reusable Abstractions associated with the Component Specification and thus influences the Abstract Interface Specifications to achieve optimum reusability.

Abstract Interface Specifications. Provides the formal specification for the implementation of each Reusable Abstraction. All user interfaces with the associated reusable component should be explicitly defined by this specification.

Abstract Constraint Analysis Summary. Provides a definition of all remaining constraints implied by each Abstract Interface Specification, including both the Usage and Implementation Constraints when appropriate. Each constraint should be accompanied by an analysis summary documenting the rationale for the constraint.

Methodology Validation

The work products above will form the basis for both influencing and validating

proper implementation of reusability and maintainability attributes during component development. An initial review should be held prior to component implementation to influence the implementation to include reusable subcomponents. All work products for the component to be developed should be provided to support this pre-implementation review. Another review should be held after the component is implemented to validate proper reusability accommodation within the implementation. The Abstract Interface Specification and Abstract Constraint Analysis work products for any newly created subcomponents should be provided to support this post-implementation review.

At least three different points of view other than that of the component programmer (not necessarily three different people) should be represented at each review to ensure that reusability and maintainability objectives are satisfied.

Domain Analyst. Must have familiarity with the intended and potential domains of applicability of the component and its related software. By identifying areas of commonality as well as applicable existing software, he/she should exert a significant reusability influence on the component software implementation.

Software Component Engineer. Must have good understanding of software engineering practices that promote the development of reusable, maintainable components. Through application of sound software engineering practices, he/she will be able to appropriately influence the component implementation to achieve optimum reuse and reduce maintenance costs.

Component User. Must be responsible for other software that is intended to use the component. He/she should exert a significant influence towards ensuring that the components will satisfy the needs of the intended users in a highly usable manner.

Summary

In order to dramatically reduce software development costs, it is necessary that software developers learn how to reuse existing components. To accomplish this, they must first learn how to develop components to be reused. By examining the characteristics of reusable software and establishing a software development methodology that allows software to be constructed with these characteristics, an initial step has been taken to influence the production of reusable software components. The methodology presented within this paper has evolved from limited application of an initial version of the methodology (16). In order to validate that the methodology is a reasonable model for developing reusable software components, additional pilot projects must be selected and used to validate and enhance the methodology or establish reasonable alternatives. Work products, enforcement mechanisms, and support tools must then be put in place to make the resultant methodology a normal way of developing software.

References

(1) McCain, R., "Software Reusability Study Report", 1984

(2) McCain, R., "A Product Approach For Software Component Development", 1984

(3) Booch, G., Software Engineering With Ada, Benjamin/Cummings Publishing Company, 1983

(4) Booch, G., "Solve Process Control Problems With Ada's Special Capabilities", Electronic Design News, June 23, 1982, pp. 143-152

(5) Parnas, D., "On The Criteria To Be Used In Decomposing Systems Into Modules", Comm. of the ACM, 1972, Vol. 15, No. 12, pp. 1053-1058

(6) Parnas, D., "Designing Software For East Of Extension And Contraction", IEEE Transaction on Software Engineering, Vol. SE-5, No. 2, March, 1979

(7) Parnas, D., "The influence of Software Structure on Reliability", Proceedings of the 1975 International Conference on Reliable Software, pp. 358-362

(8) Parnas, D., Clements, P., Weiss, D., "Enhancing Reusability With Information Hiding", Proceedings for Workshop on Reusability in Programming, 1983

(9) Parnas, D., Clements, P., Chmura, L., Heitmeyer, C., Britton, K., Shore, J., Weiss, D., Software Engineering Principles, Naval Research Laboratory Course Notebook, 1981

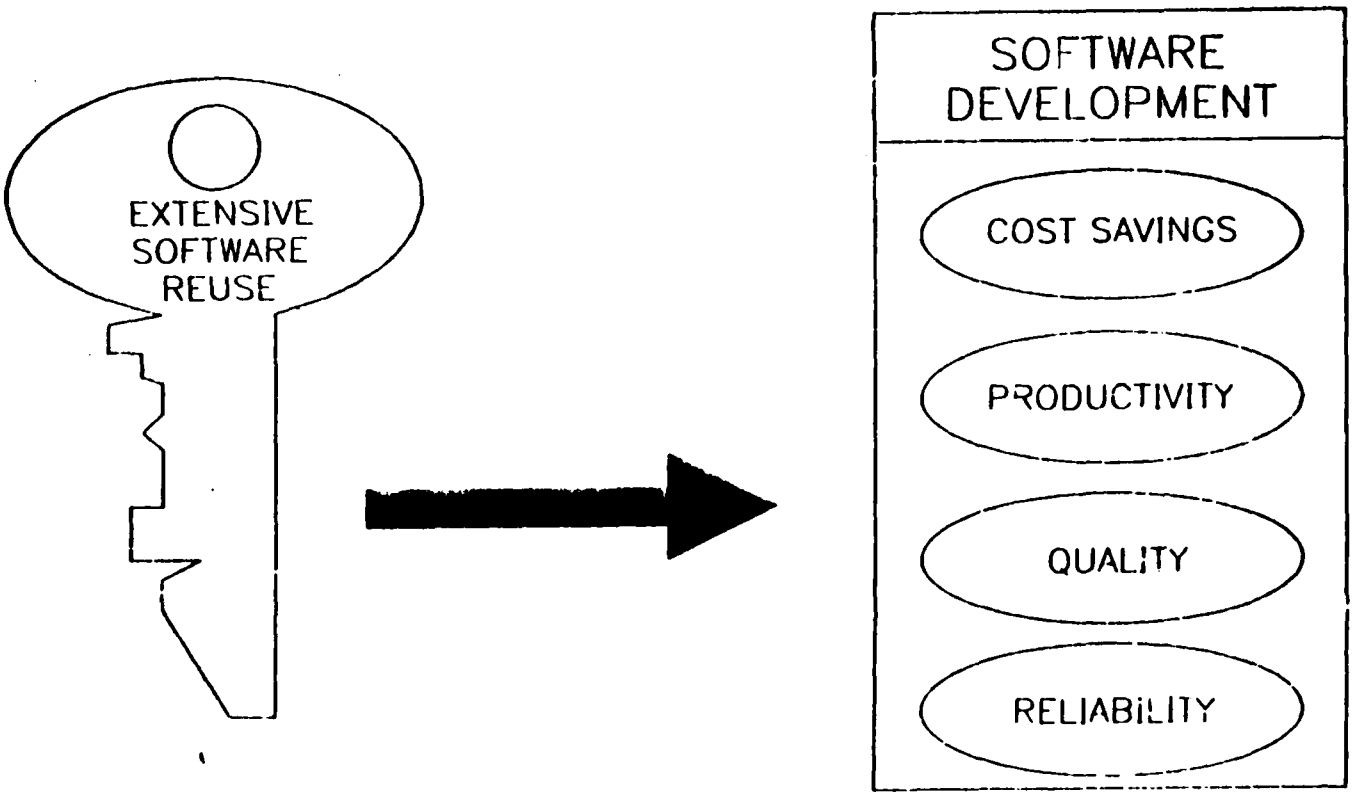
(10) Constantine, L., Myers, G., Stevens, W., "Structured Design", IBM Systems Journal, 1974

(11) Parker, A., Heninger, K., Parnas, D., Shore, J., Abstract Interface Specification For The A-7E Device Interface Module, NRL Memorandum 4385, 1980

AGENDA

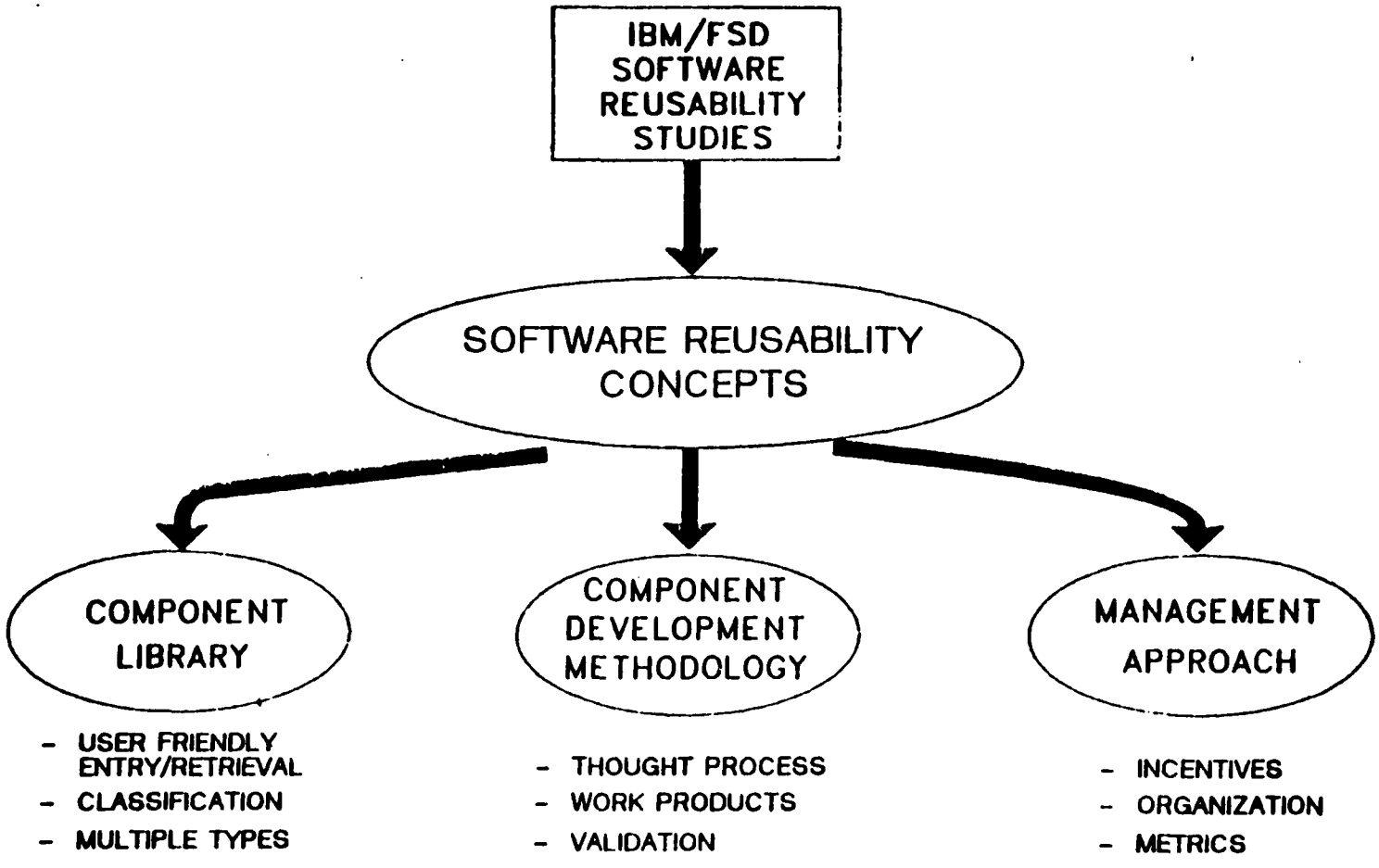
- INTRODUCTION
- CHARACTERISTICS OF REUSABLE SOFTWARE COMPONENTS
- REUSABILITY CONCEPTS
- A PARADIGM FOR REUSABLE COMPONENT DEVELOPMENT
- REUSABILITY–ORIENTED REVIEW PROCESS
- SUMMARY

IBM S 1028h



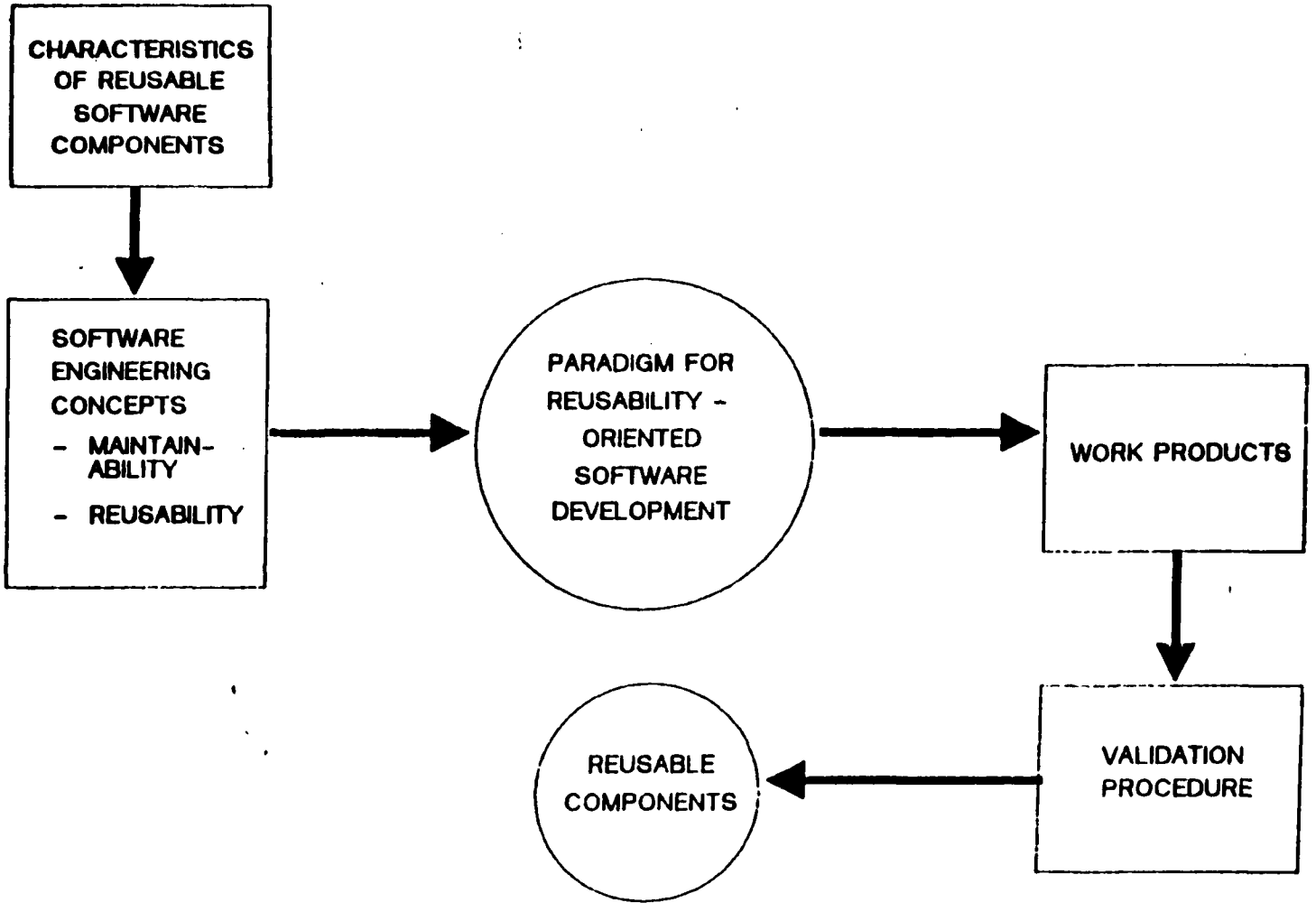
SOFTWARE REUSABILITY

KEY TO SIGNIFICANT SOFTWARE ADVANCES



SOFTWARE REUSABILITY

A MULTI-DIMENSIONAL PROBLEM



A SOFTWARE DEVELOPMENT METHODOLOGY FOR REUSABLE COMPONENTS

CHARACTERISTICS OF REUSABLE COMPONENTS

1. MULTI-USER APPLICABILITY

2. USABLE

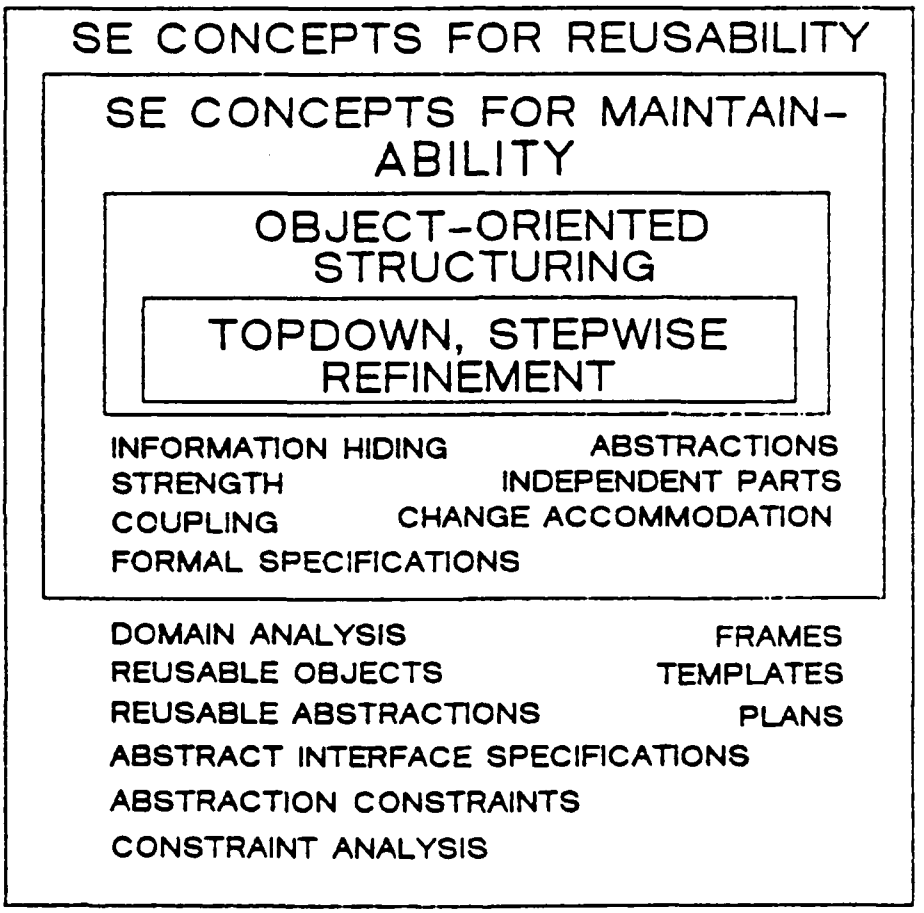
3. COMPLETE, ACCURATE INTERFACE SPECIFICATION

4. MINIMUM DEPENDENCIES

5. MINIMUM KNOWLEDGE OF IMPLEMENTATION

6. ACCOMMODATION OF CHANGE

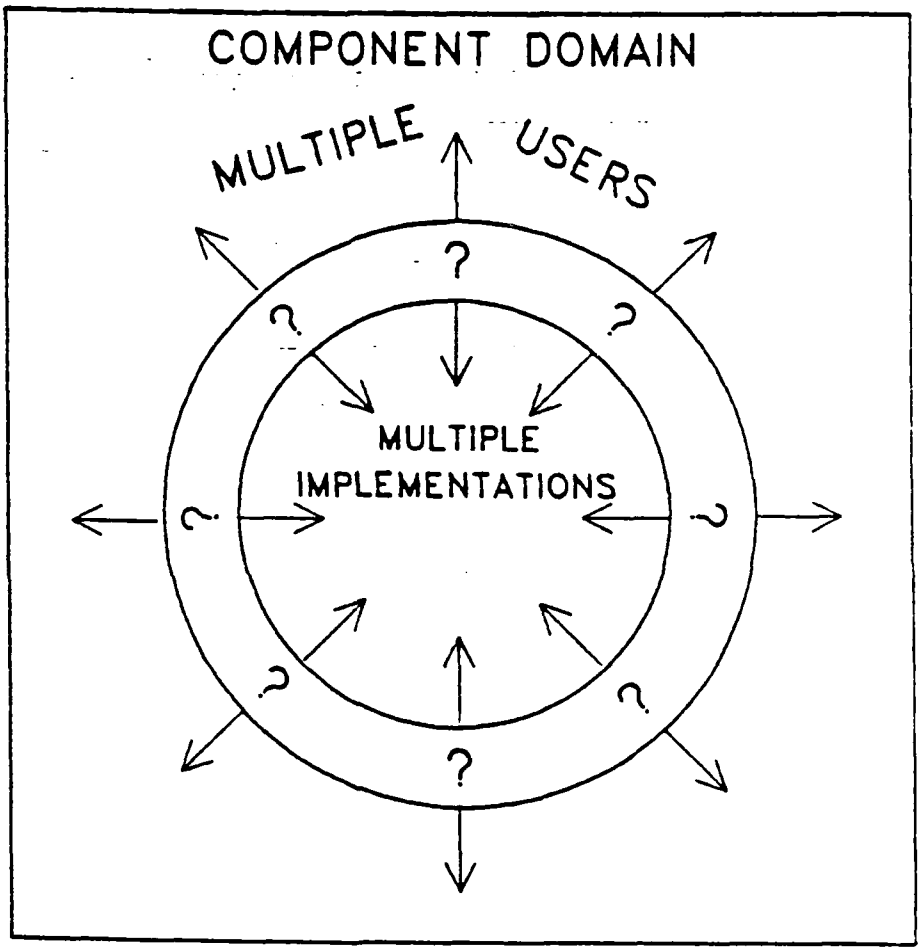
REUSABILITY CONCEPTS



IBM 1520

REUSABILITY METHODOLOGY

BASIC UNDERLYING CONCEPT



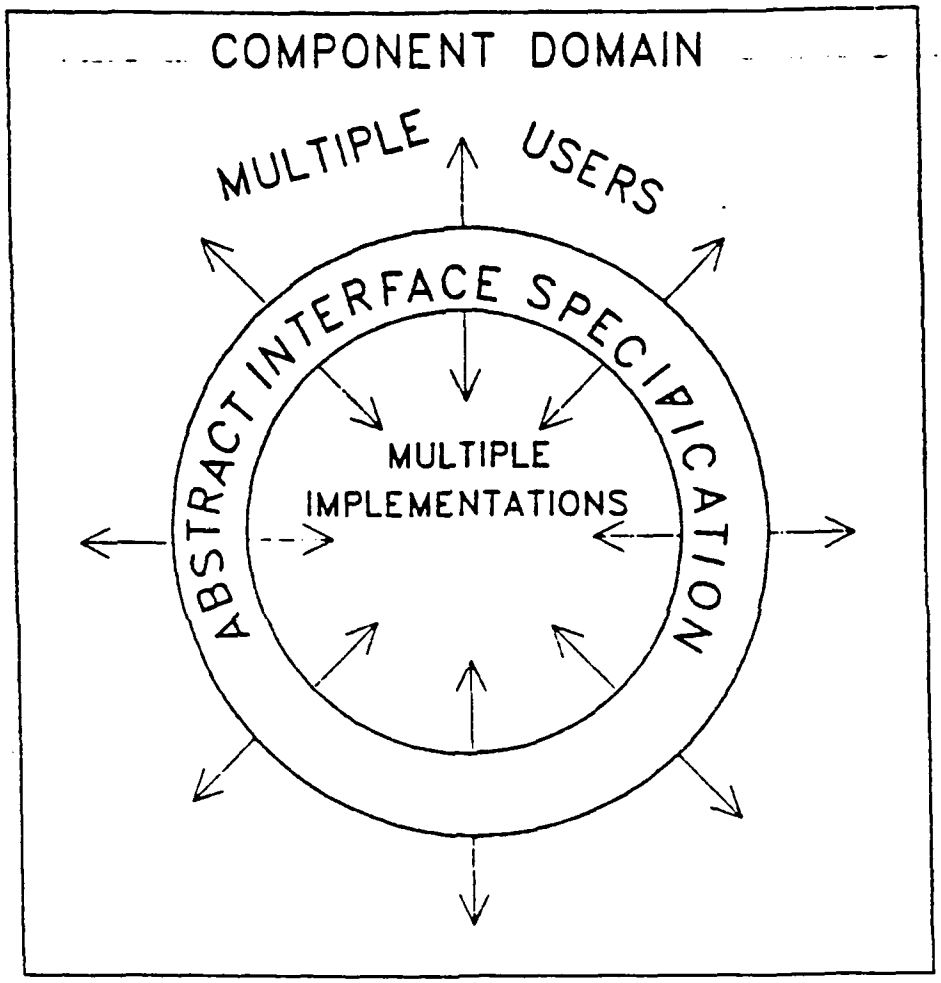
KEY QUESTION TO BE ADDRESSED BY METHODOLOGY:

HOW CAN WE OPTIMIZE NUMBER OF COMPONENT USERS WHILE RETAINING COMPONENT IMPLEMENTATION FLEXIBILITY ?

IBM 3 (12)

REUSABILITY METHODOLOGY

BASIC UNDERLYING CONCEPT

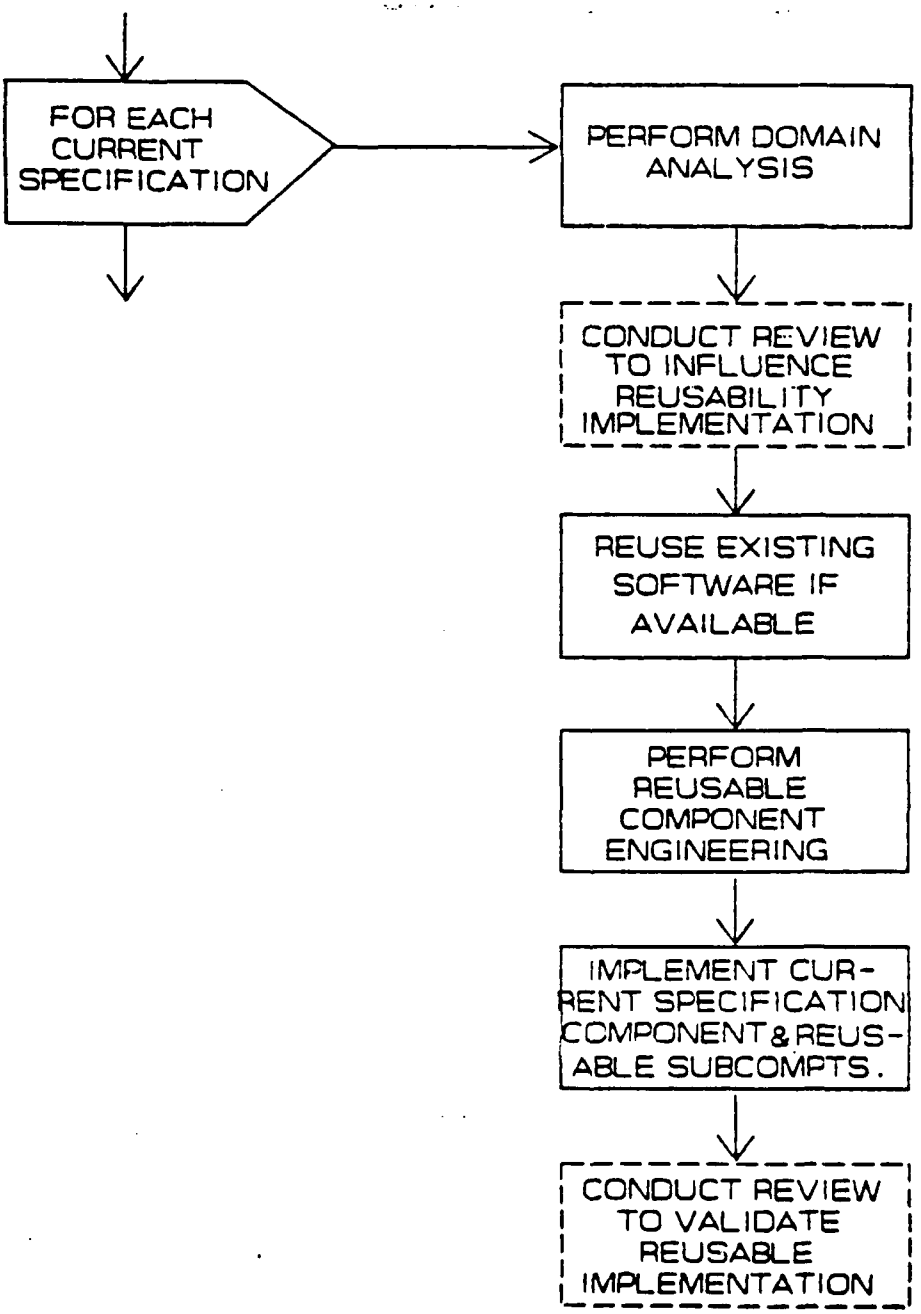


KEY QUESTION TO BE ADDRESSED BY METHODOLOGY:

HOW CAN WE OPTIMIZE NUMBER OF COMPONENT USERS WHILE RETAINING COMPONENT IMPLEMENTATION FLEXIBILITY ?

IBM 3 (02)

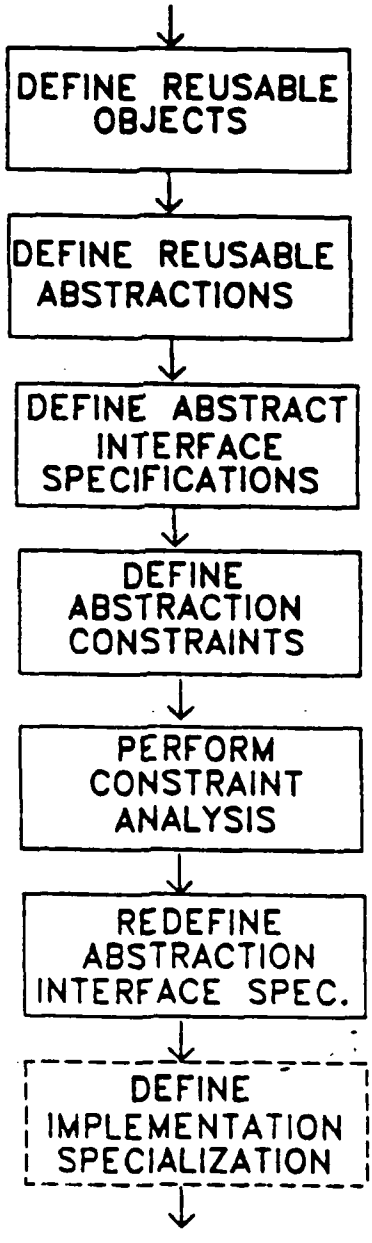
A PARADIGM FOR REUSABLE COMPONENT DEVELOPMENT



IBM 3 1024

A PARADIGM FOR REUSABLE COMPONENT DEVELOPMENT

REUSABLE COMPONENT ENGINEERING



REUSABLE OBJECTS/OPERATIONS

EXAMPLES

STACK
- PUSH
- POP
- MAKE EMPTY
- IF EMPTY
- IF FULL

SET
- UNION
- INTERSECTION
- IF NULL

COMPLEX NUMBERS
- ADD
- SUBTRACT

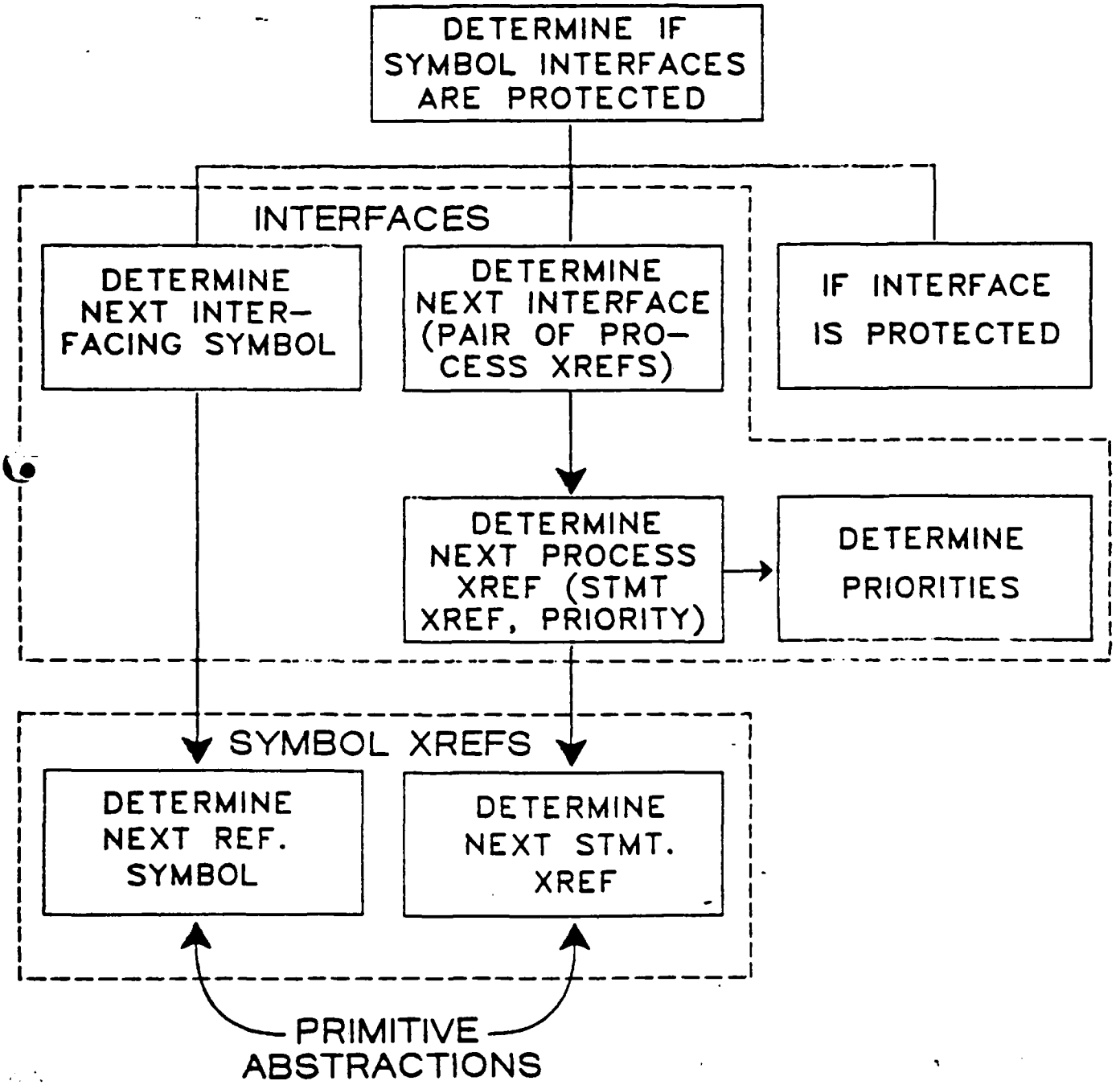
AIR DATA SENSOR
- DETERMINE AIR SPEED
- IF SENSOR POWERED ON

SYMBOL CROSS-REFERENCES
- DETERMINE NEXT REFERENCING SYMBOL
- DETERMINE NEXT STATEMENT XREF

IBM 8 (0231)

REUSABLE ABSTRACTIONS

EXAMPLE OF LAYERED REUSABLE ABSTRACTIONS



IBM 3 020 1

ABSTRACTION CONSTRAINTS

EXAMPLES

ABSTRACTION – DETERMINE NEXT STATEMENT XREF

USAGE CONSTRAINTS

- ONLY USE FOR A SPECIFIC HIGH–ORDER LANGUAGE(EX., PL/I)
- USE ONLY IF PREVIOUS EXECUTION OF INITIALIZATION OPERATION
- NON–MEANINGFUL NAME FOR ABSTRACTION (EX., DNSX)

IMPLEMENTATION CONSTRAINTS

- RESTRICTED TO SEQUENTIAL INPUT REPRESENTATION – WILL NOT ACCOMMODATE RELATIONAL DATA BASE INPUT REPRESENTATION (ALSO A USAGE CONSTRAINT)
- STATEMENT IDENTIFIED BY RELATIVE ADDRESS BUT NOT STATEMENT REFERENCE NUMBER (ALSO A USAGE CONSTRAINT)
- LINKLIST INTERNAL STORAGE ONLY OF XREF INFORMATION

IBM 3 1024

ABSTRACT INTERFACE SPECIFICATIONS

EXAMPLE

```
PACKAGE SYMBOL_INFO_PACKAGE IS
--THIS PACKAGE PROVIDES BASIC
--OPERATIONS ASSOCIATED WITH
--GLOBAL SYMBOL INFORMATION.  IT
--WILL PROVIDE SYMBOL NAMES,
--ATTRIBUTES, AND CROSS REFERENCE
--INFORMATION TO THE PACKAGE USER.
--BY ELIMINATING CONSTRAINTS,THE
--ABSTRACT DATA TYPES AND
--OPERATIONS HAVE BEEN DESIGNED TO
--MAXIMIZE REUSABILITY POTENTIAL
--AND ACCOMMODATE MANY
--IMPLEMENTATIONS.

TYPE SYMBOL_TYPE IS PRIVATE;
TYPE BLOCK_TYPE IS PRIVATE;
TYPE SYMBOL_DESCRIPTOR IS
   RECORD
      SYMBOL: SYMBOL_TYPE;
      DCL_BLOCK: BLOCK_TYPE;
   END_RECORD;
TYPE STMT_TYPE IS PRIVATE;
TYPE CATGY_TYPE IS
   (ASSIGN,REFERENCE);
TYPE XREF_TYPE IS
   RECORD
      BLOCK BLOCK_TYPE;
      STMT STMT_TYPE;
      CATGY CATGY_TYPE;
   END RECORD;
TYPE STATUS_INDICATOR IS BOOLEAN;
TYPE ATTR_TYPE IS PRIVATE;

PROCEDURE DET_NEXT_SYMBOL
   (SYMBOL_ID: OUT
      SYMBOL_DESCRIPTOR;
    ATTR: OUT ATTR_TYPE;
    END_OF_SYMBOLS: OUT
      STATUS_INDICATOR);

PROCEDURE DET_NEXT_STMT_XREF
   (XREF OUT XREF_TYPE;
    END_OF_XREFS: OUT
      STATUS_INDICATOR);
```


IBM 3 1974

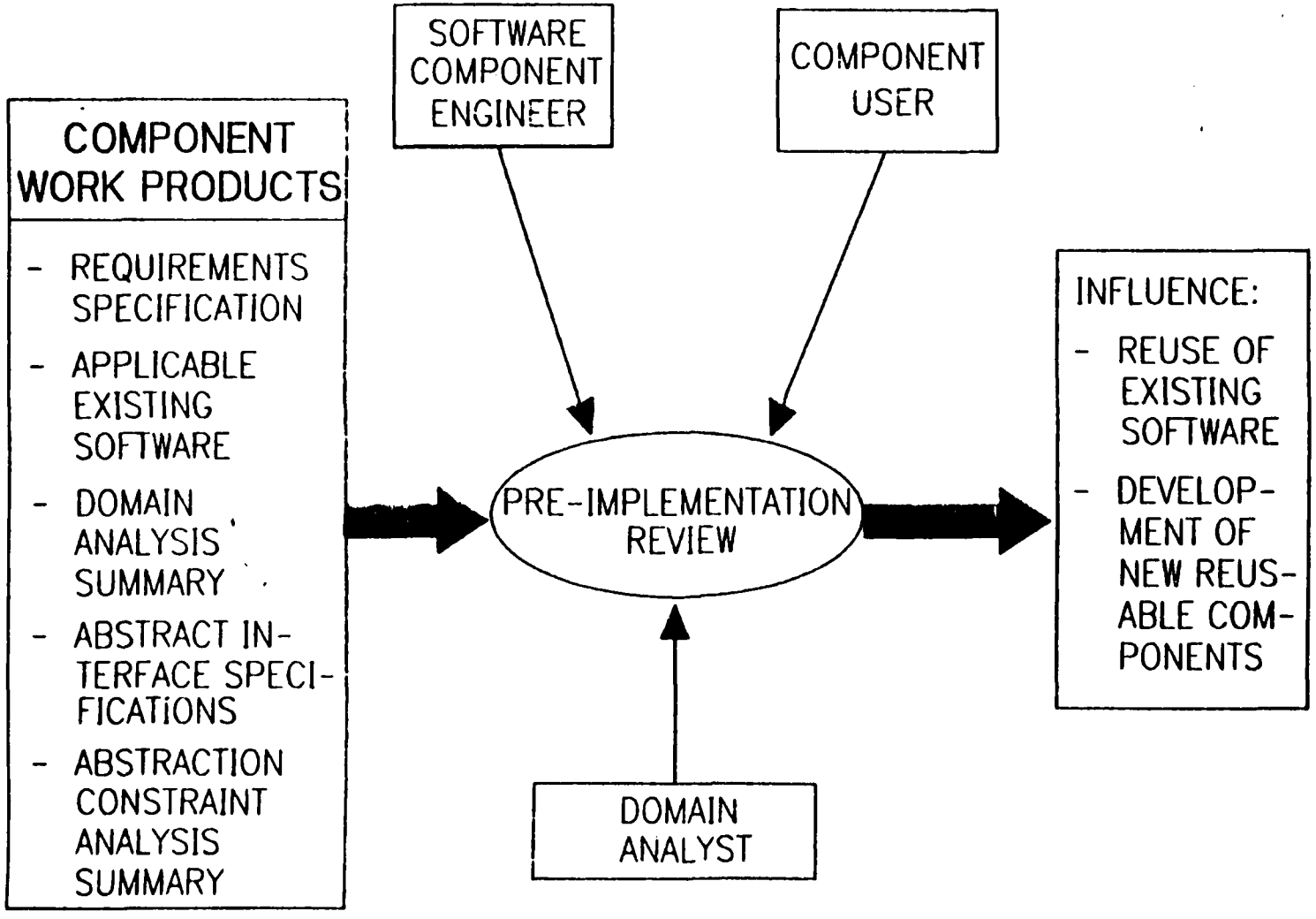
```
PRIVATE
--THE TYPE DEFINITIONS INCLUDED
--HERE ARE FOR HAL/S SYMBOLS.
--THIS PRIVATE SECTION, AS WELL AS
--THE IMPLEMENTATION OF THE
--PACKAGE OPERATIONS, WOULD HAVE
--TO BE REPLACED TO ACCOMMODATE
--SYMBOLS ASSOCIATED WITH OTHER
--PROGRAMMING LANGUAGES.

TYPE SYMBOL_TYPE IS STRING(1..64);
TYPE BLOCK_TYPE IS STRING(1..32);
TYPE STMT_TYPE IS
   RECORD
      SRN: STRING(1..6);
      ISN STRING(1..4);
   END RECORD;
TYPE ATTR_TYPE IS
   RECORD
      SYM_ATTR:  (BIT STRING,
                  SP INTEGER,
                  DP INTEGER,
                  SP SCALAR,
                  SP VECTOR,
                  SP MATRIX,
                  STRUCTURE,
                  EVENT,
                  PROGRAM,
                  PROCEDURE,
                  FUNCTION,
                  TASK,
                  COMPOOL,
                  NAME,
                  DP SCALAR,
                  DP VECTOR,
                  DP MATRIX,
                  CHARACTER);
      SYM_CAT:    (VARIABLE,
                  CONSTANT,
                  LABEL);
   END_RECORD;
END;
```


IBM 3 1021

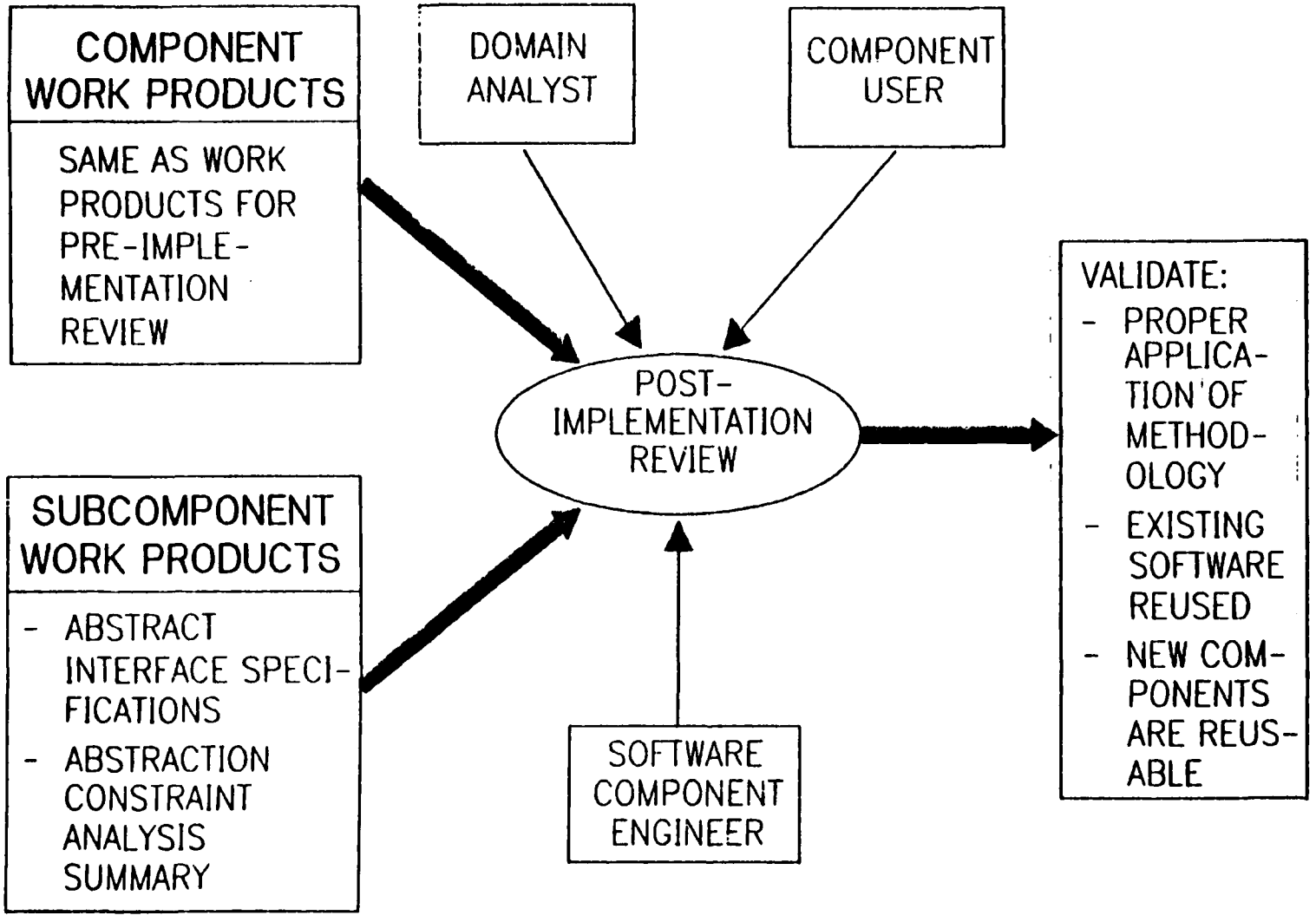
```
PACKAGE BODY
   SYMBOL_INFO_PACKAGE IS

PROCEDURE DET_NEXT_SYMBOL
   (SYMBOL_ID: OUT
      SYMBOL_DESCRIPTOR;
    ATTR: OUT ATTR_TYPE;
    END_OF_SYMBOLS: OUT
      STATUS_INDICATOR)
IS SEPARATE;

PROCEDURE DET_NEXT_STMT_XREF
   (XREF: OUT XREF_TYPE;
    END_OF XREFS: OUT
      STATUS_INDICATOR)
IS SEPARATE;

END SYMBOL_INFO_PACKAGE;
```


IBM 3:0251

THE REVIEW PROCESS



THE REVIEW PROCESS (CONT.)



SUMMARY

- SOFTWARE REUSE COULD BE THE KEY FOR DRAMATICALLY REDUCING SOFTWARE DEVELOPMENT COSTS

- SOFTWARE COMPONENT DEVELOPMENT METHODOLOGY PRESENTED
  - COMPLIANCE WITH REUSABILITY CHARACTERISTICS
  - WORK PRODUCTS
  - VALIDATION PROCEDURES

- HAS EVOLVED THROUGH LIMITED APPLICATION OF REUSABILITY CONCEPTS

- WILL CONTINUE PROOF OF CONCEPT
  - METHODOLOGY VALIDATION
  - METHODOLOGY ENHANCEMENTS
  - SUPPORT TOOLS

IBM 0 1074

CONTRACT FO 8635-84-C-0280
COMMON ADA MISSILE PACKAGES (CAMP)

PRELIMINARY TECHNICAL REPORT
VOLUME 1

STUDY RESULTS

1 February 1985

McDonnell Douglas Astronautics Co.

ABSTRACT

This report describes the work performed, the results obtained, and the conclusions reached during the first five months on the Common Ada Missile Package (CAMP) contract. This work was performed by the Computer Systems & Software Engineering Department of the McDonnell Douglas Astronautic Company - St. Louis (MDAC-STL) and was sponsored by the United States Air Force Armament Laboratory (AFATL) at Eglin Air Force Base, Florida. The MDAC-STL program manager was Dr. Daniel G. McNicholl and the AFATL program manager was Christine M. Anderson.

Dr. Daniel G. McNicholl
McDonnell Douglas Astronautics Co.
Computer System & Software Engineering Department
P.O. Box 516
St. Louis, MO 63166

Christine M. Anderson
Air Force Armament Laboratory
Aeromechanics Division
Guidance & Control Branch
Eglin Air Force Base, Florida 32542

SECTION I

OVERVIEW OF THE CAMP FEASIBILITY ANALYSIS

1. PURPOSE

The objective of the Common Ada Missile Package (CAMP) program is to determine the feasibility of (1) reusable missile software components written in Ada, and (2) an automated or semiautomated software generation system. This report is intended to describe the work performed and the results obtained from the first five months (5 September 1984 through 31 January 1985) of the CAMP program.

2. INTRODUCTION

During the past ten years, the U.S.Air Force has become increasingly sensitive both to the critical role that software plays in DoD mission critical systems and to the fact that its contractors involved in software development are facing a crisis. This crisis severely impacts the Air Force because it results in (1) rapidly escalating software development and maintenance costs, (2) delays in the deployment of new defense systems due to expanding software development schedules, (3) restrictions on the number of programs which can be concurrently developed due to a shortage of critical expertise, and (4) software reliability problems with deployed defense systems.

The basic cause of the software crisis is that the explosive growth in the size, complexity, and critical nature of modern softwasre systems has resulted in the situation where our tools are antiquated, our methods are inadequate, and our personnel are under trained and in many cases inexperienced. Obviously such a complex problem has no one solution, but concrete initiatives do exist which, if taken, will alleviate the current situation. While it is beyond the scope of this report to discuss all the initiatives which are being proposed, there is one initiative -- the reuse of software parts -- which most knowledgeable software engineers believe is essential in any solution to the software crisis. The concept of reusable software parts is the motivation behind the CAMP program.

Simply stated, reusable software parts are prebuilt software components (either code or design) which are capable of being used more than once to construct new software systems. The most obvious benefit of reusing software is that less code needs to be developed and therefore less time and money is required to be spent during the development of new software systems. However, there are a number of less obvious benefits which in some cases have an even greater payoff. If software components can be reused, then (1) less code has to be maintained, (2) fewer people are needed, and (3) a higher degree of reliability can be obtained.

Since maintenance costs (i.e., the cost to correct software errors, modify the software to a new environment, and expand the capabilities of the software) often greatly outweigh the development costs of software, a reduction in the amount of code to be maintained can result in drastically lower product life cycle costs. Assuming a significant level of software reusability, the required staffing for a software development and/or maintenance project will be descreased. Given the shortage of software engineers that exists throughout the industry, this is a major advantage. If the parts which are prebuilt include functions which typically require a high degree of application expertise (e.g., Guidance & Control), then a project can perform the same development with a lower level of such expertise. Finally, if the prebuilt parts are rigorously tested before they are cataloged for later reuse, then the reliability of the new software systems will be increased.

Given all these advantages, it is only natural to wonder why software has not been reused in the past. The answer to this question is that it has, but only to a very limited degree. Almost all software systems have incorporated certain types of prebuilt software parts. The most common type of reused part has been the mathematical part, i.e. a routine from a math library. Yet if this type of low-level reusability was all that we could hope to obtain, the benefits discussed earlier would not be achievable. The three primary reasons for our past inability to achieve a meaningful level of reusability are (1) our progrsmming languages (e.g. FORTRAN, Assembly, JOVIAL, etc.) have not had the facilities to support software reusability, (2) we have not invested the time and effort to identify the commonality in our software systems, and (3) our software developers have not been encouraged and/or required to reuse software parts.

With the advent of Ada, we now have a computer programming language which was explicitly designed with the goal of software reusability in mind. Specifically, Ada possesses facilities for (1) transporting programs across machine and operating system boundaries; (2) enforcing the design and construction of autonomous software units with clean, well-designed interfaces; and (3) developing software parts which are generic in nature and which can be tailored, using the Ada language itself, for a particular application.

One of the major barriers to an effective software reusability program is the need to conduct an in-depth domain analysis of the application area in which the software is to be reused. A domain analysis is an examination of a specific application area which seeks to identify common operations, objects, and structures. Domain analyses are not cheap to perform. They require (1) an intensive examination of existing software systems within the application area being studied, and (2) personnel skilled both in modern software development techniques and in the application area. Yet, to attempt to start a software reusability program without adequately performing this analysis is as follish as attempting to design a software system without performing an analysis of the software requirements.

One of the thorniest issues which arises in every reusable software effort and which can cause a total failure of the effort is the need to enforce the reuse of parts. Without reuse, reusable software parts become an exercise in futility and any additional cost to develop reusable parts (as opposed to one-shot code) cannot be amortized.

Bluntly stated, programmers do not like to reuse software. There are a number of reasons for this attitude. Programmers often: (1) feel that reusing parts lessens their creative role in the development of software systems; (2) have little faith in the correctness of reusable parts; (3) are not aware of the existence of reusable parts; (4) find the software parts more complex to understand and/or reuse in comparison to developing new software; and (5) feel that they can build a better part.

The key factors in overcoming the reluctance of programmers to reuse prebuilt software parts are discipline, knowledge, tools, and management commitment. A successful reusable software program must involve the imposition of a high degree of parts usage discipline within the organization. This discipline must be enforced by reviews and audits. Programmers must also have the knowledge that parts exist and that they have been validated. Just as hardware designers are expected to know which parts are available, we should also expect software engineers to know about software parts. Tools are an essential aspect of a software reusability program because they serve to relieve the software engineer of the mundane mechanical chores associated with using parts and also to increase their productivity. These tools should facilitate the retrieval of appropriate parts, the generation of new parts, the composition of software systems with existing parts, and a wide variety of other functions relating to parts usage.

The reuse of software parts offers the promise of dramatic increases in software development and maintenance productivity. Yet this promise can only be achieved if an organization is working in an application area which has a significant degree of commonality and can put in place the tools and methods needed to enforce a software parts engineering discipline.

3. SUMMARY

During the first five months of the CAMP program, the tasks performed by the CAMP teams were (a) to determine if sufficient commonality exists in missile flight software systems to justify the construction of reusable software parts written in Ada; (b) to develop a method of specifying the requirements and design of the software parts; (c) to develop a cataloging scheme for the software parts; and (d) to examine software generation technology in order to determine the feasibility of an automated software parts composition system. Sections II through V present the detailed results of our work in these areas. The following is a summary of those results.

3.1 CAMP Missile Selection

A detailed analysis of the missile software domain was performed in which ten missile software systems were examined (see Exhibit 1) including air-to-surface, ground-to-air, and ground-to-ground missiles. Paragraph 1 of Section II presents more details on these missile software systems.

(1) Flight software for the Medium Range Air to Surface Missile (AGM-109H).

(2) Flight software for the Medium Range Air to Surface Missile (AGM-109L).

(3) Strapdown inertial navigation program for the Unaided Tactical Guidance Project.

(4) Guidance and navigation program for the Midcourse Guidance demonstration.

(5) Flight software for the Tomahawk Land Attack Missile (BGM-109A).

(6) Flight software for the Tomahawk Anti Ship Missile (BGM-109B).

(7) Flight software for the Tomahawk Land Attack Missile (BGM-109C).

(8) Flight software for the Tomahawk Land Attack Missile (BGM-109G).

(9) Flight software for the Harpoon Missile (Block 1C).

(10) Safeguard Spartan missile.

EXHIBIT 1. The Missile Flight Software Set

3.2 CAMP Parts Classification

The information on the CAMP missile software systems was obtained from their software requirements and software design documents. An analysis of the requirements documents was performed to identify the domain dependent parts which could be constructed. Domain dependent parts are those which provide functions which are unique to the missile flight software area (or a highly related area such as avionics). An analysis of the software design documents was performed to identify the domain independent parts which could be constructed. Domain independent parts are those which provide functions and objects which, while highly relevant to the missile flight software applications area, also have applicability to a wide number of other areas. Exhibit 2 depicts a breakdown of the types of parts in these two areas. This taxonometric breakdown is explained in more detailed in paragraph 2.2 of Section II.

DOMAIN DEPENDENT PARTS	o DATA PACKAGES -- Data Type Packages -- Data Constant Packages o EQUIPMENT INTERFACES o MISSILE FUNCTIONS -- Primary Navigation Operations -- Ancillary Navigation Operations -- Guidance Operations -- Mission Control/Sequencing -- Warhead Control -- Telemetry
DOMAIN INDEPENDENT PARTS	o PROCESS CONTROL MECHANISMS o COMMUNICATION MECHANISMS o ABSTRACT PROCESSES o ABSTRACT DATA STRUCTURES o MATHEMATICAL FUNCTIONS -- Matrix/Vector Functions -- Geometric Functions -- Trigonometric Functions -- General Functions o GENERAL UTILITY

EXHIBIT 2. The CAMP Software Parts Taxonomy

In addition to classifying parts by the their type as just discussed, it was recognized that three levels of parts were needed, see Exhibit 3.

CAMP PARTS

Meta Parts

Simple Parts	Generic Parts	Schematic Parts
as-is parts	tailorable parts	generatable parts

EXHIBIT 3. Software Part Levels

A simple part is a software part which is capable of being reused 'as is'. In other words, these parts would correspond to Ada procedures, tasks, and packages which would be 'withed' into an Ada program without any modification. An example of this type of part would be a simple mathematical function.

Unlike simple parts, a meta-part cannot be used as it exists. Rather, it must be customized to a particular application. In later sections of this report the use of the software generation system to perform this customization will be discussed. A generic part is a template from which a number of specific parts can be obtained by means of the Ada generic facilities. These are parts in which the parameterization of the part conforms to the capabilities of an Ada generic unit. One example of this level of part would be an abstract data structure such as a generalized First-In-First-Out (FIFO) queue in which the type of the data objects to be queued would be supplied and a specific FIFO queue part would be instantiated for that situation.

A schematic part is a set of part construction rules which is used to generate a number of specific parts. Schematic parts differ from generic parts in two important aspects: (1) the generation of specific parts from a schematic part can not be achieved by means of the Ada generic facilities; and (2) there is no code to look at until a specific part is built. A relatively simple example of a schematic part would be a finite automaton which requires the association of an action with a state transition (these types of finite automata are usually referred to as Mealy machines). The requirement that actions be associated with state changes cannot be realized in Ada even with its generic facilities because Ada does not have a variable procedure data type. However, the structure of such a part is straightforward. Therefore, the schematic construction rules would be used to build an Ada unit which meets the needs of the user.

Paragraph 2.2 of Section II contains a more detailed description of the CAMP software parts classification.

3.3 CAMP Parts Identification

In the initial phase of the CAMP project the investigation of domain independent commonality has proceeded at a faster pace then that of domain dependent commonality. This is due to the fact that the investigation of the domain dependent areas (e.g., navigation, guidance, etc.) has required much more intensive, up-front analysis.

In the case of domain independent parts, many of the common operations and objects identified have had a foundation in 'classical' computer science. Abstract processes, abstract data structures, communication mechanisms and other devices of this type have been thoroughly investigated in other application areas. The major task of the CAMP team was to analyze the CAMP missile set and identify which mechanisms were needed, and which variants of the mechanisms were required.

At the current time, we are near the completion of the process of identifying domain independent parts. The next step is to begin the formal specification and design of the identified parts in these areas.

In the domain dependent areas, we have drawn upon the expertise of a large number of missile system engineers in order to identify both functional commonality and architectural commonality. At the current point in our study we are near the completion of our analysis of the primary navigation operations, and are in the midst of the analyses of the ancillary navigation and guidance operations. In all cases to date, we have been able to identify a large number of common operations at the functional level. In both the primary navigation and ancillary navigation areas we have also been able to identify common architectures. We expect this trend to continue in our investigation of the other functional areas.

The common domain dependent and domain independent parts identified up to this point are described in paragraph 2.3 of Section II.

3.4 CAMP Specification Technique

As mentioned earlier, it is important that the users of parts have a good degree of knowledge about the parts in order to establish their confidence in them. Among other items, the part user must be able to determine the requirements of the part (i.e., what the part is suppose to do and how well the part is expected to perform) and the design of the part (i.e., how the part accomplishes its requirements). The design knowledge would ideally not have to be known by the user of the part, but to overcome the programmer's reluctance to use parts, we consider this to be an essential aspect of part usage.

To this end, the CAMP team developed a method of specifying the requirements and design of the missile software parts. This method had two objectives: (1) it must be amendable to typical military documentation, and (2) it must facilitate communication. For this reason, two complementary approaches were taken. The first involved a textual specification technique which would be compatible with the new Military Standard SDS. The second involved a graphical notation which would be used to supplement the textual method and would serve as a better communication mechanism. The graphical technique which was devised was developed by extending the Ada graphical notations developed by Grady Booch and Ray Buhr.

3.5 CAMP Ada Parts Cataloging Scheme

Later in this report the use of advanced techniques for the generation and composition of software parts will be discussed, but, the fundamental first step in tooling-up for a software reusability program is to provide a software parts catalog. Such a catalog serves several purposes: (1) it ensures that an organization has an institutional memory of the parts; (2) it is an essential vehicle in disseminating knowledge of the parts to the software engineers; and (3) it is a cornerstone of any software parts composition system.

The approach taken on the CAMP program was to develop a method of describing Ada missile software parts which would facilitate the rapid access and use of the parts. Note that the CAMP software parts catalog was specifically designed for parts which were written in Ada and for the missile flight software domain. No attempt was made to make this catalog more general (e.g., handle parts written in other programming languages or for other application areas). As it turned out, it would not be difficult to generalize the resultant catalog for other application domains.

Two types of information are included in the CAMP parts catalog. Search informastion is data which is provided primarily to help catalog users find the right part (e.g., a list of keywords, the taxonometric type classification of a part, etc). Descriptive information is data that helps the user decide, once a part is found, whether that part is indeed suitable for his needs (e.g., an abstract, a list of projects using the part, etc.).

One critical design decision made during the develop of the CAMP parts catalog was that the catalog should not repeat information contained in the specification portions of Ada library units. Rather, it should contain more abstract, application oriented information.

Exhibit 4 depicts a summary of the information in the CAMP parts catalog which is described in more detail in Section IV.

3.6 CAMP Software Generation Technology Evaluation

In order to determine the feasibility of automating the generation of missile flight software systems, the CAMP team reviewed both existing software generation systems and the technologies which have potential in this area. Exhibit 5 depicts the existing systems which we examined. Descriptions of these systems are contained in Section V. Exhibit 6 depicts the technologies which we reviewed.

Part and Version IDS	The part id and version number together form a unique identifier for the part
Name	A brief meaningful name for the part
Abstract	A description of the part which describes its function, and intended usage
Category	The part's taxonometric classification
Type	Indicates whether the part is a subprogram, package, or task
Level	Indicates whether the part is a simple part, generic part, or meta-part
Class	Indicates whether the part is a specification or a body
Keywords	A list of meaning keywords
Development	The date the part was cataloged
Developer	The name of the person or organization who developed the part
Development Project	The project for which the pasrt was originally developed
Development Stage	Indicates the part's completion status
Verification Status	Indicates who verified the part
Units Withed	Delineates other parts Withed are used by a part
Withing Units	Delineates other parts which use a part
Usage	Delineates the projects using a part
Code Location	Contains the location (e.g., file name) of the part code or code constructor
Security	Indicates the security classification of both the part and its catalog entry
Others	Various information concerning its accuracy, its timing, its storage requirements, its hardware dependencies, the availability of documentation, etc.

EXHIBIT 4. Parts Catalog Attributes

DRACO	University of California, Irvine
USE.IT	Higher Order Software
DARTS	General Dynamic
PSI	Stanford University
SAFE	University of Southern California
CHI	Kestrel Institute
Programmer's Apprentice	Massachusetts Institute of Technology
KBSA	Kestrel Institute
LADDER, LIFER & DEDALUS	SRI
LUNAR	Bolt, Barenek, and Newman
RENDEZVOUS	IBM
MODEL	University of Pennsylvania
PROTRAN	IMSL
SREM	U.S. Army Ballistic Missile ATC
PSL/PSA	ISDOS, Inc.

EXHIBIT 6. Software Generation Technologies

Very High Order Languages (VHOL)	Automatic Programming
Artificial Intelligence (AI)	Expert Systems
Data Base Management Systems (DBMS)	Domain Analysis
Syntax-Directed Translation	Natural Language Interfaces
Graphic Specification Languages	Formal Specification Languages
Transformation Systems	Deductive Systems
Custom Tailoring Systems	Text Generation Systems

EXHIBIT 6. Software Generation Technologies

As an aid for evaluating the aforementioned tools and technologies, we developed a model of an ideal software generator. Such a system would provide the facilities for automating the processes depicted in Exhibit 7.

Parts Identification	The process of selecting a part, or set of parts, from a set of pre-existing parts for a specific application.
Parts Creation	The process of creating a part.
Parts Instantiation	The process of constructing an instantiation of a specific part.
Parts Generation	The process of constructing a specific part from a schematic part by means of a part construction scheme.
Parts Construction	The process of manually creating a specific software part.
Parts Composition	The process of integrating parts into a software system.

EXHIBIT 7. Facilities Provides by an Ideal Software Generator

Several key observations which were made during this evaluation process are summarized in Exhibit 8 and discussed in paragraph 2.1 of Section II.

The approach towards which the CAMP team is gravitating is one which was identified by our own work on the CAMP commonality study. As we identified various parts, we realized that the most difficult task in using the parts would be to identify what specific part was needed.

In the case of a simple part, this involves mapping the missile's requirements onto those of the part. In the case of a generic part, this involves the same activity as with a simple part plus determining the correct parameterization for the instantiation of the part. In the case of a schematic part, this involves a similar identification process but an even more complex parameterization process. For example, once the user has determined that a strapdown inertial navigation system is needed and a schematic part exists which will generate the architecture of such a system, the user will have to specify all the information to tailor the generated part for his application.

(1) The use of a formal specification language as an interfacing mechanism to a software generator will severly limit the use of the system.
(2) The concept of a universal (i.e., domain independent) software generator is not practical for the CAMP domain due to inherent inefficiencies of the code produced.
(3) Few existing software generation system have the capability of reusing parts.
(4) While there are many experimental software generation systems, many of these systems are not production quality, and will not be within the foreseeable future.

EXHIBIT 8. Observation From the Technology Evaluation

As we worked with the various missile engineers we found ourselves asking questions such as "When do you want to use direction cosine versus quaternions?" From asking these "In what situations do we want to use X?" questions, we discovered that there does exist a body of knowledge which can help guide the missile software engineer in his development of the software.

This knowledge consists of factual rules and heuristics. An example of a factual rule would be "If a data object (larger than one word) must be accessed in write mode by more than one asynchronous process, then it must have some type of mutual exclusion protection." An example of a heuristic, which is akin to a rule-of-thumb, is "If a missile software system must interface with a particular equipment peripheral, then it probably will need some type of built-in-test function for that interface."

Given that knowledge does exist about the construction of missile softwasre parts, we want to be able to formalize this knowledge. One technology which has recently emerged from the laboratory and is now in common use is that of expert systems. An expert system is a software system which emulates the manner in which humans reason about problems. It provides facilities for incorporating both factual and heuristic knowledge and for drawing inferences from this knowledge.

We are currently exploring the use of such an expert system in two complementary areas -- Automated Parts Identification and Automated Parts Constructions.

Section V contains more details on our work in this area.

4. CONCLUSIONS

We believe that our work to date has indicated that there does exist a significant amount of commonality between missile software systems and that a pragmatic method does exist for automating some of the software development tasks using these parts.

The use of the domain independent and dependent parts discussed earlier and described in more detail later in this report, would allow the DoD missile software development projects to achieve the benefits discussed in the earlier part of this report.

The use of an expert system in the role of an automated parts identifier and constructor would greatly facilitate the implementation of a parts engineering discipline.

From our discussion with MDAC-STL missile engineers and software engineers and other non-MDAC experts, we firmly believe that the systems we have identified are both feasible and of value. We have adopted an approach which balances state-of-the-art technology with hard-nosed engineering values which we believe will result in the design of a system which will be used. Although this might seem like a modest statement, one of the largest pitfalls of this type of research is the development of a system which, while technologically "fun", is too difficult to use.

RESUME

DR. DANIEL G. MCNICHOLL

Position

CAMP Program Manager; Technical Specialist, Electronics (Computer Systems and Software Engineering)

Education

PhD Computer Science, University of Missouri at Rolla, 1982.
MS Computer Science, University of Missouri at Rolla, 1980
BS Computer Science, Pratt Institute, 1972.

Experience

Since joining MDAC-STL in mid-1982, Dr. McNicholl has been involved in the development and evaluation of software engineering methodologies and tools, and in the support of ongoing projects in the areas of database design, software development, management, software cost estimating, and the use of simulation languages. He has been an active participant in the development and use of the MDAC-STL Ada Design Language (ADL). In addition to Dr. McNicholl's responsibilities as CAMP Program Manager, he heads the Software Technology Group, where he is responsible for planning and coordinating programs designed to facilitate the transition of MDAC-STL software engineers to Ada.

Dr. McNicholl is an Adjunct Assistant Professor of Computer Science at the University of Missouri - Rolla Graduate Engineering Center in St. Louis, teaching evening courses on information systems, programming languages, and operating systems.

He is a member of ACM and the IEEE Computer Society. He is an active member of the IEEE Computer Society Technical Committee on Software Taxonomy, and participates in corporate-wide Ada and related working group meetings at McDonnell Douglas.

UNCLASSIFIED

Enclosure (3)
Reference: Commerce Business Daily
Synopsis 0035

An Overview of the Common Ada Missile Packages (CAMP) Program

Presented by

Dr. Daniel G. McNicholl
CAMP Program Manager
McDonnell Douglas Astronautics Co.
P.O. Box 516, St. Louis, MO 63166

Work Sponsored by

Aeromechanics Division
Guidance & Control Branch
Air Force Armament Laboratory
Eglin Air Force Base, Florida

Government Program Manager Christine Anderson

ACM SIGADA MEETING
November 25 - 30, 1984

MCDONNELL DOUGLAS CORPORATION

UNCLASSIFIED

CAMP OBJECTIVES

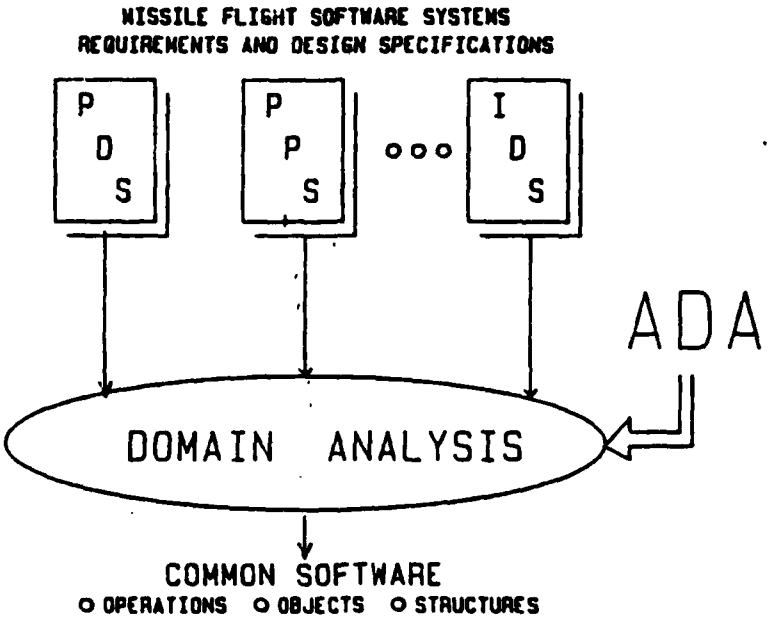
CAMP is a 12 month study designed to determine the feasibility of:

- Reusable missile flight software components written in Ada – *CAMP Commonality Study*
- An automated software generation system with facilities for parts composition – *CAMP Software Generation System Study*

ACM SIGADA MEETING
November 25 – 30, 1984

UNCLASSIFIED

CAMP DOMAIN ANALYSIS

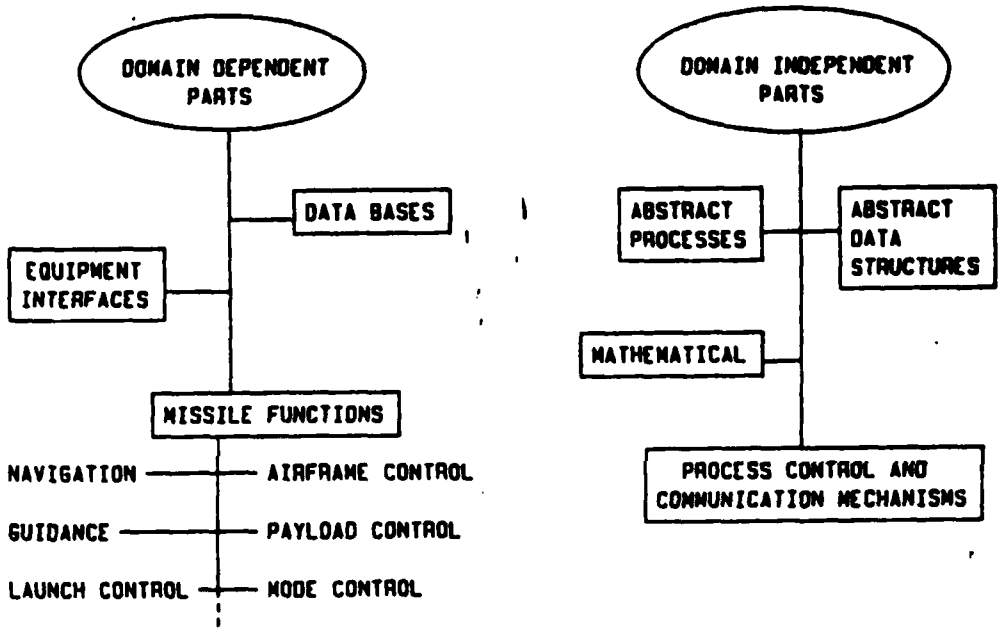


ACM SIGADA MEETING
November 25 - 30, 1984

MCDONNELL DOUGLAS CORPORATION

UNCLASSIFIED

CAMP PARTS TAXONOMY

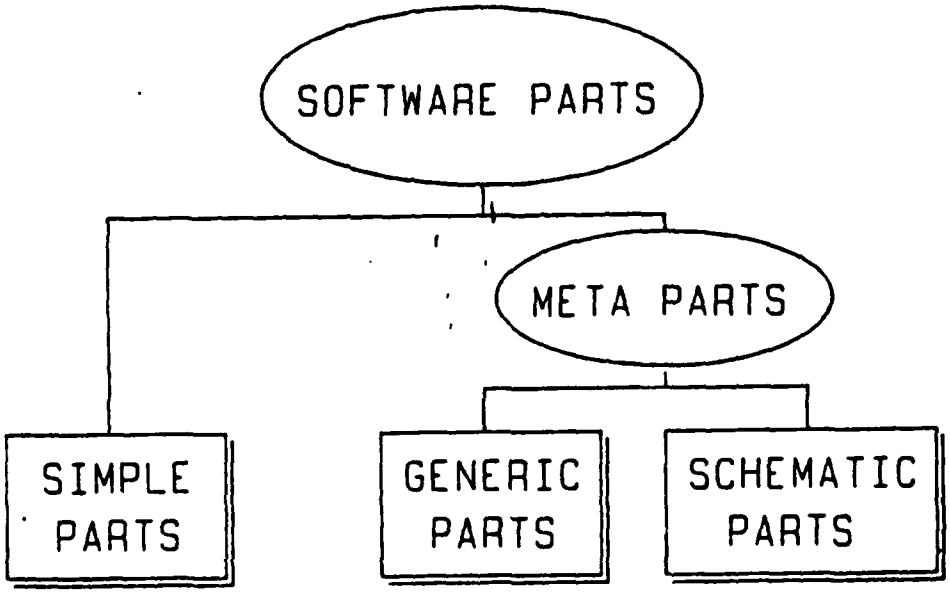


ACM SIGADA MEETING
November 25 - 30, 1984

McDONNELL DOUGLAS CORPORATION

UNCLASSIFIED

CAMP PARTS CLASSIFICATION SCHEME



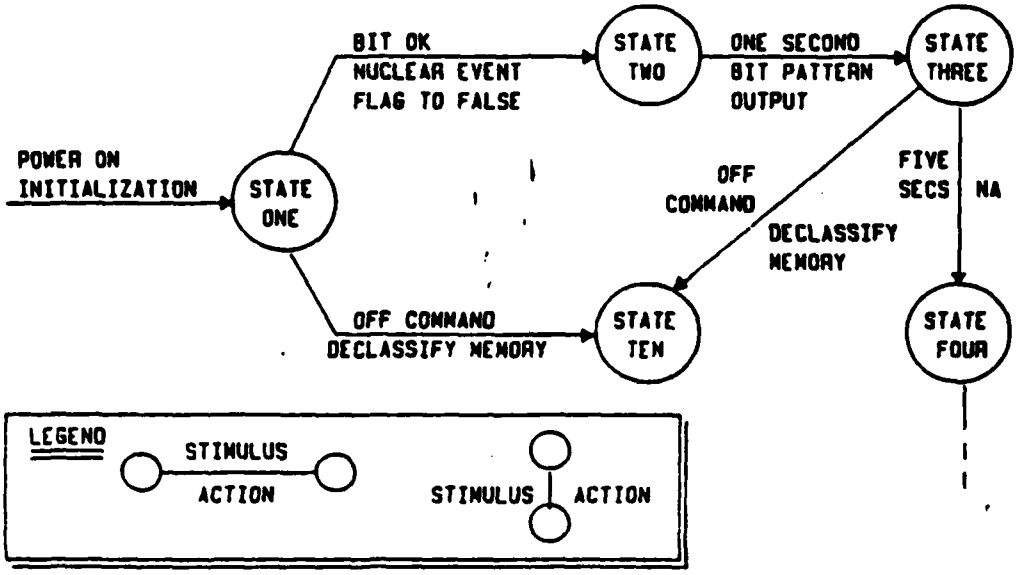
MCDONNELL DOUGLAS

ACM SIGADA MEETING
November 25 - 30, 1984

UNCLASSIFIED

AN EXAMPLE OF A MEALY MACHINE

A FINITE AUTOMATON WITH ACTIONS ON THE TRANSITIONS



ACM SIGADA MEETING
November 25 - 30, 1984

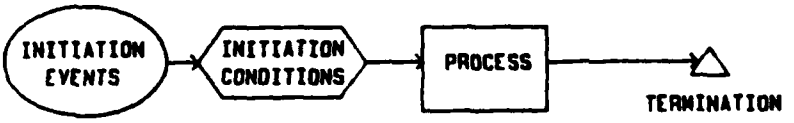
MCDONNELL DOUGLAS CORPORATION

UNCLASSIFIED

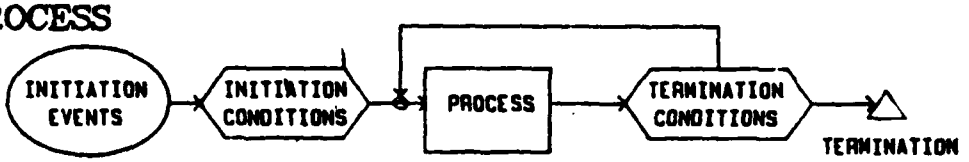
MISSILE PROCESS TYPES

GRAPHIC REPRESENTATION

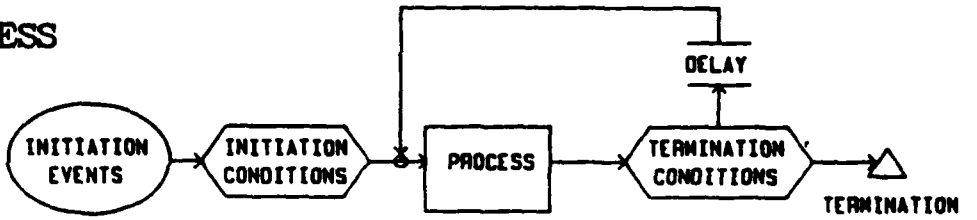
APERIODIC PROCESS



CONTINUOUS PROCESS



PERIODIC PROCESS



ACM SIGADA MEETING
November 25 - 30, 1984

MCDONNELL DOUGLAS CORPORATION

UNCLASSIFIED

MISSILE PROCESS TYPES

ADA REPRESENTATION

APERIODIC PROCESS TASK SHELL

```
task START_Process_Name;
task body START_Process_Name is
   procedure Process_Name is separate;
begin
   Process_Name;
end Process_Name;
```

CONTINUOUS PROCESS TASK SHELL

```
task START_Process_Name;
task body START_Process_Name is
   procedure Process_Name is separate;
begin
   loop
      Process_Name;
   exit when Termination_Condition;
   end loop;
end Process_Name;
```

PERIODIC PROCESS TASK SHELL

```
with CLOCK;
use CLOCK;
task START_Process_Name;
task body START_Process_Name is
   procedure Process_Name is separate;
   NEXT_TIME : TIME := CURRENT_SYSTEM_TIME;
begin
   loop
      delay DURATION (NEXT_TIME - CURRENT_SYSTEM_TIME);
      Process_Name;
   exit when Termination_Condition;
      INCREMENT (NEXT_TIME, BY => Required_Time_Interval);
   end loop;
end Process_Name;
```


MCDONNELL DOUGLAS CORPORATION

UNCLASSIFIED

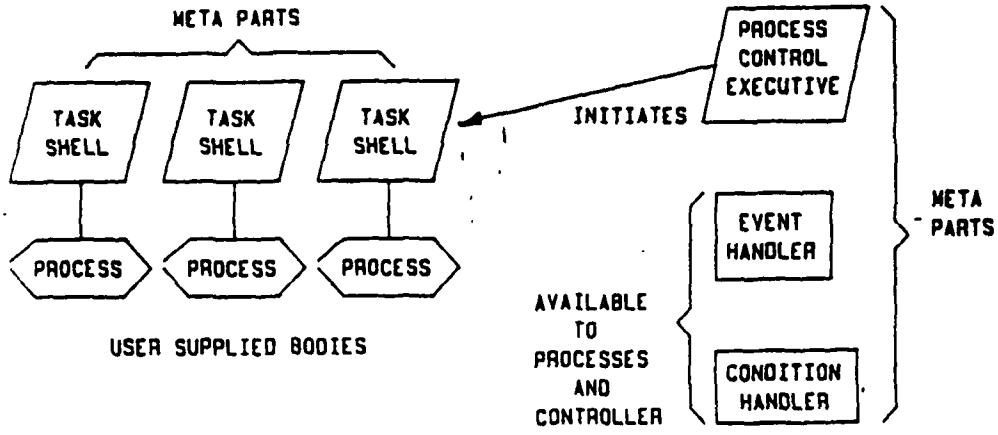
SAMPLE DYNAMIC DESCRIPTION OF A PROCESS

Kalman_Data_Processing process

initiation	as a periodic task
by	*Alignment_Initialization*
at	a *16* hertz rate
with	a priority of *12*
terminates	when *Initialization_Completed*
initiation	as an aperiodic task
by	*Data_Ready_To_Be_Processed* when *Mode = X1*
or	*Some_Other_Event*
with	a priority of *11*

UNCLASSFIED

THE PROCESS CONTROL METAPARTS



UNCLASSIFIED

OTHER CAMP COMMONALITY STUDY TASKS

- To specify the requirements for a subset of the common parts
- To develop the design for a subset of the common parts
- To develop a requirements and design specification technique for reusable Ada parts
- To use several of the new STARS measurement DID's

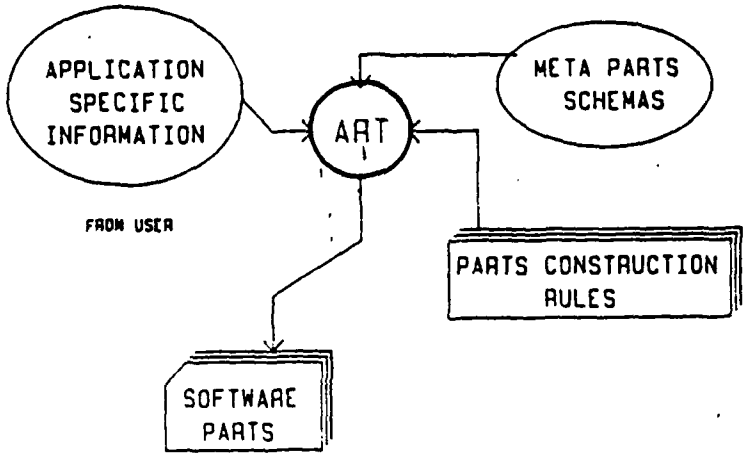
UNCLASSIFIED

THE CAMP SOFTWARE GENERATION SYSTEM STUDY TASKS

- To develop an Ada parts cataloging scheme
- To develop the design of an automated software generation system
- To develop the software requirements of the software generation system
- To examine the Japanese software reusability programs
- To evaluate the Automated Reasoning Tool (ART)

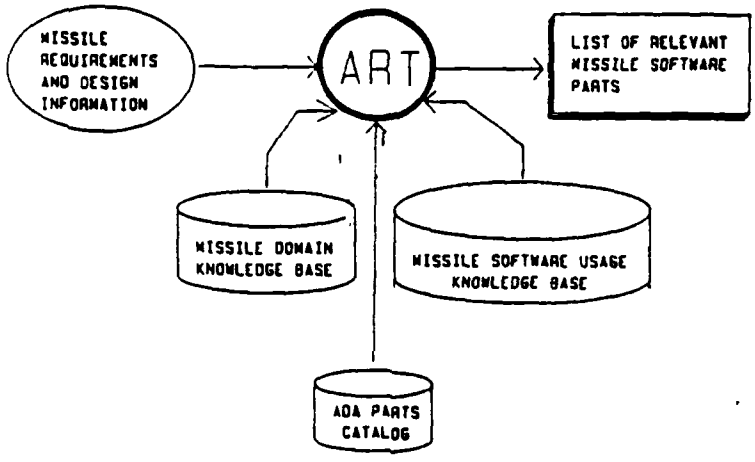
UNCLASSIFIED

MISSILE SOFTWARE PARTS CONSTRUCTION EXPERT (MSPCE)



UNCLASSIFIED

MISSILE SOFTWARE PARTS IDENTIFICATION EXPERT (MSPIE)



ACM SIGADA MEETING
November 25 - 30, 1984

McDONNELL DOUGLAS

ENCOURAGEMENT OF SOFTWARE REUSABILITY

George W. Mebus

RCA Advanced Technology Laboratories

Introduction:

The DoD has correctly identified the reusability of software as a necessary part of the software crisis solution, worth the cost and effort of achieving it. Therefore, it has been made an essential part of the Ada technology. The aims of Ada address many other pressing problems as well. Naturally, some of them affect and sometimes conflict with the goal of reusability.

This paper considers other software systems that have displayed success in fostering and maintaining reusability. Lessons learned from such success, while not all directly applicable to Ada developments, should aid our appreciation of what can be achieved under special circumstances, and guide our future actions in managing Ada-based software.

Software Reusability

If software can be truly reusable, obvious benefits accrue from not having to reinvent the wheel. Productivity will be greatly improved because of large reductions in the cost and time to develop new software. Maintainability will be enhanced. Reliability will be increased, not only because the software has been previously tested, but also because it has been pressed into service. Thus, it will have been validated by previous use to achieve some degree of maturity (the only good software is used software). Additionally, experimental knowledge of the reused software will have been established so that the printed documentation will not be the only guide to successful utilization.

To realize these benefits, the software and its supporting development systems must be easily accessed, easily understood, and easily incorporated into new developments. Any changes required for this incorporation must be easy to identify, document and execute. Ideally, the elements should be general enough to be used for a variety of related applications with no change at all.

Past Problems and Obstacles

The software community's FORTRAN experience exemplifies the problems of non-reusable software. Large monolithic programs were typically highly specialized for a single application. Where sections of code may have been useful for other applications, there was great difficulty and danger in extracting them because of obscure dependencies upon other areas of the program. The specialization of this software was largely due to the pervasive use of "hard-wired" information. Specific data values, peculiar to the intended application, were freely distributed throughout the source code.

It slowly became clear that the use of parameterized code could alleviate this kind of problem. The existence of FORTRAN libraries (e.g. the Math/Science Library) demonstrates the benefit of even this first step toward reusability. This illustration of reusability was still hampered by more deeply embedded features of the language. Among them are the requirements for using specific data types and data shapes. Such requirements still forced rewriting of potentially reusable routines to suit the specific data needs of each new application. Thus, an expansion of the parameterization idea extended to all the potentially variable features of an application should be expected to offer much greater relief for reusability pains.

Ada Features to Increase Reusability

The designers of Ada have incorporated a number of improvements. Generic units with private types allow delayed definition of many specifics until compile time and sometimes until run time. The additional provisions of attributes and delayed constraints further relax the need for specific detail in the master version of the source code, while maintaining the strong type checking desired for a compiled language. Support in the Ada library system for free access to a large body

of software also encourages software sharing. The APSE and "one Ada" (no supersetting or subsetting) decisions that have become part of the Ada culture do still more to foster reuse of software from one host to another, and the free reuse of tools as well as application programs.

Experience with Reusable Software

Other systems supporting reusable software have been in existence for some time. There is a body of experience in the use of such software development systems which should not be disregarded when considering the broad approach to reusability. Success in one area will provide valuable insight into related but necessarily different areas, especially in providing a vision of what might be achieved in the future. Particular successful software development systems supporting reusability are APL and UNIX.

The APL language has many characteristics that promote the reuse of software. These features make it convenient, even desirable, to access and use existing software and to develop reusable software as well. The following paragraphs characterize the language, the language support system, and the software that results from them.

APL is an algebraic language by its very nature. (The jargon of the APL community is also algebraic: primitive operators and user-defined programs in the language are called "functions" and the parameters to them are called "arguments".) Expressions are algebraically manipulable. It is rich in inverse functions and algebraic identities. These enable some degree of algebraic program transformation, helping efforts at formal proofs of programs. The underlying concepts of APL are few and simple. Because of its many primitive functions, there is no precedence established among them. The order of execution is determined entirely by a right-to-left execution rule and the use of parenthesis. The syntax of all functions, primitive or defined, is either prefix (for one argument) or infix (for two). Arguments, however, may be multidimensional arrays, so the restriction to two arguments is not severe. All data are rectangular arrays, so that they all have both value and shape. Although array elements can be (encapsulated) arrays, there are only two primitive data types: real, integer, and bit (Boolean) data are present, but the system determines the internal storage requirements for them. Clearly, strong typing is not a feature of APL, and some reliability is sacrificed for the benefits of generality.

Generality is one of the strong points of APL that supports reusability. The ability of any arithmetic function (primitive or defined) to accept real data, integer data, or even results of logical functions greatly extends a program's potential utility without the need for any change or respecification. Many functions are data independent in that they will work for any array. Shape transformation functions are good examples: the functions named "reverse", "rotate", "transpose", "ravel", "reshape", and others need only know the shape of the arguments independent of data types. The shape information is part of the data; it does not have to be explicitly declared or specified at any time. Thus these functions are also defined for arrays of any dimensionality and shape including scalars and empty arrays. In this respect, APL is remarkably complete and consistent. Another instance of the same quality is the two-argument arithmetic functions (add, subtract, multiply, divide, residue, power, logarithm, minimum, maximum, the six relational and ten non-trivial logical functions, and others). They are defined to accept two number arrays of the same shape, and combine their corresponding elements to produce a result array of the same shape (the "equal" and "not equal" relational functions can also accept character arrays). Also, if one argument has only a single element, its value is combined with every element of the other. This operation is remarkably consistent and dependable. It also removes the need for writing N nested loops to perform the combination of two N-dimensional arrays. The code is much more compact and readable in this respect than that of any block-structured language including Ada.

The above descriptions indicate that APL largely lacks arbitrary restrictions. Name length, dimension size, number of dimensions, expression lengths are all unrestricted. Only strict data incompatibilities limit which expressions can be arguments to which functions. As a result, the ease of combination of functions (primitive or

defined) is unusually unemcumbered. Combining functions to form short expressions that perform significant processing steps is a way of life in APL. Certain expressions are seen to recur in APL code to the extent that they are often clearly recognizable at a glance, their purpose and operation well known. These have been called "idioms" of the language. Use of idioms is another unusual instance of reusability in APL expressions.

Extremely compact code is one of the striking features of APL. The primitive functions (over 70 of them, analogous to reserved keywords in most other languages) are represented not by words but by special single-character symbols. This emphasizes the algebraic nature of the language. It also says a lot in a little space. Compactness is enhanced by the use of arrays as well. The avoidance of program loops, with the absence of counter variable initialization, incrementation and completion specifications, is a clear example. The processing of arrays as single data items rather than having to specify element-by-element processing in general reduces code size and clarifies the code, keeping it on a more abstract level. There is of course a corresponding reduction in intellectual burden and improvement in readability (providing that the reader understands the APL function symbols).

While APL code is not often considered self-documenting (although some will claim it is) the compactness of the code offers some advantages to the documentation process. One line of code often accomplishes a significant step in the problem-level description of the processing. As in other languages, comments can be placed at the ends of executable lines. One comment per line generally provides an excellent description of the processing at the problem statement level. In fact, the code lines are often developed from such statements of the problem. To the knowledgeable user reader of APL, the occurrences of idioms also provide some measure of self documentation.

Like Ada, APL is not just a language but also has an integrated programming and execution support environment. Unlike Ada, the programs (defined functions) are typically interpreted rather than compiled and all storage is dynamically allocated. Interactive debugging facilities are included in every APL system. Data and defined function objects are stored in "workspaces" special files that have analogies to Ada packages. A)COPY facility enables convenient sharing of individual data or function objects, named groups of such objects, or entire workspaces. This sharing can be done at software development time or during execution. While not enforced, there are standard workspace documentation conventions that have evolved in the APL culture and are used in nearly all APL public library systems. APL libraries from major vendors of APL services (e.g., STSC and I. P. Sharp) are extensive. The ease of software sharing and access to an enormous variety of software has been the key to the success of such vendors.

It should come as no surprise that the software resulting from the APL language and support environment is modular, general, and easily shared. The natural modularity stems in part from the syntax of defined functions being the same as for primitive functions. The valuable characteristics of primitive functions (generality, consistency, lack of restrictions, ease of combination) and their benefits in practice encourage the software designer to develop defined functions with the same characteristics in order to reap the same benefits.

UNIX is a better known quantity in the main stream of software development. It has been influential in the definition of the APSE. With UNIX there is essentially one "data type," the character string file. UNIX utilities typically have a clearly defined, simple job to do, producing files that can be easily processed by other utilities. This simplicity, modularity, and generality of UNIX utilities, along with the "pipes" facility, allow easy reuse of the tools in a great variety of combinations. As a result, fairly sophisticated process are developed quite rapidly and reliably as UNIX command procedures.

UNIX and APL have several common characteristics that promote reuse. Generality in their operations and the data they process is a strong contributor. Support for modularity in both the system facilities and in the system philosophies also has a pervasive

effect. In addition, they are both interactive with dynamic storage allocation. They handle many of the drudgerous "bookkeeping" chores, freeing the user of much detail consideration. All of these make the software development job easier, even enjoyable.

Ada Comparisons and Contrasts

An obvious major difference between the Ada system and the other described above is that Ada is a compiled language. Another is the strong typing enforced by the compiler. The number of types is not small, and is increased by the software developer as Ada encourages user-defined typing. For most user-defined types, appropriate special operations will also be defined. The main route to generality of operators is through overloading. That is, multiple definitions of the operation are developed, one for each combination of parameter types. Then the compiler determines from the context which meaning of an operator is to be used. This form of generality is gained via complexity rather than simplicity.

Ada's generic facility provides another path to generality, but again, complexity characterizes its use. It is first necessary to make all of the kinds of declarations required for a specific instantiation of a generic unit, using private types rather than defined types. Then additional information must be supplied to inform the compiler that the specific information is to be supplied to make the usable product. This is well worth the effort when several instantiations of the generic software are used, but the initial effort to develop that generic software is not small or easy.

The generality of APL's array handling scheme is not shared by Ada. Ada does support the definition of arrays of arbitrary dimensionality. However, initialization can only be done by successive vector value assignments. Catenation is defined only for vectors, rather than for arrays with the same cross-section shape. And only Boolean operations can operate on arrays (necessarily Boolean arrays) to produce array results. Identical kinds of processing for the arithmetic functions can only be achieved by defining specific operators for each, or a generic unit to be instantiated for each.

>From this complexity, there is little psychological incentive to produce the apparent simplicity, consistency, and ease of functional combination so important to reusability in the APL and UNIX systems. The burden is then on management to foster these characteristics in the produced code.

Conclusions

Programming systems that support and encourage the development of reusable software have been examined. The hallmarks of simplicity, generality, consistency, and ease of functional combination were shown to be important in such development. Ada also supports software reusability, but lacks these features because of other priorities. This lack makes the development of reusable software in Ada a much larger job requiring larger resources and strong management to achieve the benefits of reusability.

RESUME

GEORGE W. MEBUS

George W. Mebus received a B.S. degree in Electrical Engineering from the University of Pennsylvania in 1962, and MSE degree in Electrical Engineering (Computer Option) in 1964 from the University of Michigan. He completed all course requirements for a PhD degree in Computer Information Sciences at the University of Pennsylvania.

Mr. Mebus has been with RCA since 1983. He is currently a Unit Manager in the Software Engineering Laboratory of RCA's Advanced Technology Laboratories. Work under his supervision includes development of Ada risk retirement tools, Ada Program Design Language and processors, a retargetable Ada compiler for horizontally microcoded computers, and formal verification techniques and tools used in verifying the security of distributed communications systems and secure operating systems.

Prior to joining RCA, Mr. Mebus worked for twenty-four years at the Naval Air Development Center, Warminster, Pa., in digital and computer systems. From 1973, he performed software development at the Advanced Software Technology Division of the Software and Computers Directorate where he developed tools and techniques to support automated design, development, testing and maintenance of major Navy fleet software systems such as the Light Airborne Multi-Purpose System (LAMPS). He was the Division Team Leader in development of the Facility for Automated Software Production (FASP), a Navy integrated software engineering environment for the LAMPS project from 1976 to 1981. He was also a member of the Navy's Ada Evaluation Team.

Mr. Mebus is a member of the following professional associations:

IEEE, IEEE Computer Society, ACM, ACM SIGSOFT, ACM SIGAPL, Tau Beta Pi.

He has been awarded U.S. Patent No. 3,551, 664 for the design of a Bearing Angle Computer.

Mr. Mebus has a Secret clearance.

Selected Publications

"Computer Languages--A view from the top, "RCA Engineer, Vol. 29, No. 1; Jan/Feb 1984

"A Software Engineering Environment for Weapon System Software: Functional Description for the Code and Test Phase," NADC Report No. NADC-82183-50; Naval Air Development Center; 30 November 1982

"Mathematical Description of AADC (All Applications Digital Computer)" NADC Report No. NADC-75093-20; Naval Air Development Center; 12 September 1975

"Laminar Extension: An Overlooked Capability and the Search for Its Proper Home," APL Quote Quad, Vol. 9, No. 4; June, 1979

"Reducing Tips for Fat 700 Programs," Proceedings of first Wang SWAP Symposium; January, 1973.

Generality is one of the strong points of APL that supports reusability. The ability of any arithmetic function (primitive or defined) to accept real data, integer data, or even results of logical functions greatly extends a program's potential utility without the need for any change or respecification. Many functions are data independent in that they will work for any array.

Shape transformation functions are good examples: the functions named "reverse", "rotate", "transpose", "ravel", and others need only know the shape of the arguments independent of data types. The shape information is part of the data; it does not have to be explicitly declared or specified at any time. Thus these functions are also defined for arrays of any dimensionality and shape including scalars and empty arrays. In this respect, APL is remarkably complete and consistent. Another instance of the same quality if the two-argument arithmetic functions (add, subtract, multiply, divide, residue, power, logarithm, minimum, maximum, the six relational and ten non-trivial logical functions, and others). They are defined to accept two number arrays of the same shape, and combine their corresponding elements to produce a result array of the same shape (the "equal" and "not equal" relational functions can also accept character arrays). Also, if one argument has only a single element, its value is combined with every element of the other. This operation is remarkably consistent and dependable. It also removes the need for writing N nested loops to perform the combination of two N-dimensional arrays. The code is much more compact and readable in this respect than that of any block-structured language including Ada.

The above descriptions indicate that APL largely lacks arbitrary restrictions. Name length, dimension size, number of dimensions, expression lengths are all unrestricted. Only strict data incompatibilities limit which expressions can be arguments to which functions. As a result, the ease of combination of functions (primitive or defined) is unusually unencumbered. Combining functions to form short expressions that perform significant processing steps is a way of life in APL. Certain expressions are seen to recur in APL code to the extent that they are often clearly recognizable at a glance, their purpose and operation well known. These have been called "idioms" of the language. Use of idioms is another unusual instance of reusability in APL expressions.

Extremely compact code is one of the striking features of APL. The primitive functions (over 70 of them, analogous to reserved keywords in most other languages) are represented not by words but by special single-character symbols. This emphasizes the algebraic nature of the language. It also says a lot in a little space. Compactness is enhanced by the use of arrays as well. The avoidance of program loops, with the absence of counter variable initialization incremenation and completion specifications, is a clear example. The processing of arrays as single data items rather than having to specify element-by-element processing in general reduces code size and clarifies the code, keeping it on a more abstract level. There is of course a corresponding reduction in intellectual burden and improvement in readability (providing that the reader understands the APL function symbols).

While APL code is not often considered self-documenting (although some will claim it is) the compactness of the code offers some advantages to the documentation process. One line of code often accomplishes a significant step in the problem-level description of the processing. As in other languages, comments can be placed at the ends of executable lines. One comment per line generally provides an excellent description of the processing at the problem statement level. In fact, the code lines are often developed from such statements of the problem. To the knowledgeable user reader of APL, the occurrences of idioms also provide some measure of self documentation.

Like Ada, APL is not just a language but also has an integrated programming and execution support environment. Unlike Ada, the programs (defined functions) are typically interpreted rather than compiled and all storage is dynamically allocated. Interactive debugging facilities are included in every APL system. Data and defined function objects are stored in "workspaces" special files that have analogies to Ada packages. A)COPY facility enables convenient sharing of individual data or function objects, named groups of such objects, or entire workspaces. This sharing can be done at software development time or during execution. While not enforced, there are standard workspace documentation conventions that have evolved in the APL culture and are used in nearly all APL public library systems. APL libraries from major vendors of APL services (e.g., STSC and I.P. Sharp) are extensive. The

ease of software sharing and access to an enormous variety of software has been the key to the success of such vendors.

It should come as no surprise that the software resulting from the APL language and support environment is modular, general, and easily shared. The natural modularity stems in part from the syntax of defined functions being the same as for primitive functions. The valuable characteristics of primitive functions (generality, consistency, lack of restrictions, ease of combination) and their benefits in practice encourage the software designer to develop defined functions with the same characteristics in order to reap the same benefits.

UNIX is a better known quantity in the main stream of software development. It has been influential in the definition of the APSE. With UNIX there is essentially one "data type," the character string file. UNIX utilities typically have a clearly defined, simple job to do, producing files that can be easily processed by other utilities. This simplicity, modularity, and generality of UNIX utilities, along with the "pipes" facility, allow easy reuse of the tools in a great variety of combinations. As a result, fairly sophisticated process are developed quite rapidly and reliably as UNIX command procedures.

UNIX and APL have several common characteristics that promote reuse. Generality in their operations and the data they process is a strong contributor. Support for modularity in both the system facilities and in the system philosophies also has a pervasive effect. In addition, they are both interactive with dynamic storage allocation. They handle many of the drudgerous "bookkeeping" chores, freeing the user of much detail consideration. All of these make the software development job easier, even enjoyable.

Ada Comparisons and Contrasts

An obvious major difference between the Ada system and the other described above is that Ada is a compiled language. Another is the strong typing enforced by the compiler. The number of types is not small, and is increased by the software developer as Ada encourages user-defined typing. For most user-defined types, appropriate special operations will also be defined. The main route to generality of operators is through overloading. That is, multiple definitions of the operation are developed, one for each combination of parameter types. Then the compiler determines from the context which meaning of an operator is to be used. This form of generality is gained via complexity rather than simplicity.

Ada's generic facility provides another path to generality, but again, complexity characterizes its use. It is first necessary to make all of the kinds of declarations required for a specific instantiation of a generic unit, using private types rather than defined types. Then additional information must be supplied to inform the compiler that the specific information is to be supplied to make the usable product. This is well worth the effort when several instantiations of the generic software is not small or easy.

The generality of APL's array handling scheme is not shared by Ada. Ada does support the definition of arrays of arbitrary dimensionality. However, initialization can only be done by successive vector value assignments. Catenation is defined only for vectors, rather than for arrays with the same cross-section shape. And only Boolean operations can operate on arrays (necessarily Boolean arrays) to produce array results. Identical kinds of processing for the arithmetic functions can only be achieved by defining specific operators for each, or a generic unit to be instantiated for each.

>From this complexity, there is little psychological incentive to produce the apparent simplicity, consistency, and ease of functional combination so important to reusability in the APL and UNIX systems. The burden is then on management to foster these characteristics in the produced code.

Conclusions

Programming systems that support and encourage the development of reusable software have been examined. The hallmarks of simplicity generality, consistency, and ease of functional combination were shown to be important in such development. Ada also supports software reusability, but lacks these features because of other priorities. This lack makes the development of reusable software in Ada a much larger job requiring larger resources and strong management to achieve the benefits of reusability.

RCA
Advanced
Technology
Laboratories

ENCOURAGEMENT OF SOFTWARE REUSABILITY

GEORGE W. MEBUS

RCA ADVANCED TECHNOLOGY LABORATORIES

421

RCA
Advanced
Technology
Laboratories

ENCOURAGEMENT OF SOFTWARE REUSABILITY

REUSABILITY REQUIREMENTS

- EASY ACCESS, CLARITY, INCORPORATION
- EASY TO MODIFY AND VERIFY

DESIRABLE FEATURES

- GENERALITY FOR VARIETY OF USES WITH NO CHANGE

RCA Advanced Technology Laboratories

ADA FEATURES

- GENERIC UNITS WITH PRIVATE TYPES
- ATTRIBUTES AND DELAYED CONSTRAINTS
- LIBRARY SYSTEM
- "ONE ADA" -- TRANSPORTABLE APPLICATION CODE
- APSE -- TRANSPORTABLE TOOLS

RCA
Advanced
Technology
Laboratories

APL FEATURES

- GENERALITY, EASE OF GENERALIZATION
- SIMPLICITY, CONSISTENCY
- HIDES USE OF DATA ATTRIBUTES
- WORKSPACE LIBRARY SYSTEM

The Third Design Problem: Generic Set Package (Chapter 15)

```
generic
  type UNIVERSE is (<>);
package SET_PACKAGE is
  -
  type SET is private;
  NULL_SET : constant SET;
  -
  function "*"  (SET_1    : in SET      ; SET_2    : in SET)      return SET;
  function "+"  (ELEMENT  : in UNIVERSE; SET_1    : in SET)      return SET;
  function "+"  (SET_1    : in SET      ; SET_2    : in SET)      return SET;
  function "+"  (SET_1    : in SET      ; ELEMENT  : in UNIVERSE) return SET;
  function "-"  (SET_1    : in SET      ; SET_2    : in SET)      return SET;
  function "-"  (SET_1    : in SET      ; ELEMENT  : in UNIVERSE) return SET;
  function "<"  (SET_1    : in SET      ; SET_2    : in SET)      return BOOLEAN;
  function "<=" (SET_1    : in SET      ; SET_2    : in SET)      return BOOLEAN;
  -
  function IS_A_MEMBER (ELEMENT : in UNIVERSE; OF_SET : in SET)
           return BOOLEAN;
  function IS_EMPTY    (SET_1 : in SET)
           return BOOLEAN;
  subtype NUMBER is INTEGER range 0 . . (UNIVERSE'POS(UNIVERSE'LAST)-
                                         UNIVERSE'POS(UNIVERSE'FIRST) + 1);
  function NUMBER_IN (SET_1 : in SET)
           return NUMBER;

private
  type SET is array (UNIVERSE) of BOOLEAN;
  NULL_SET : constant SET := SET'(others => FALSE);
end SET_PACKAGE;

package body SET_PACKAGE is

  function "*" (SET_1 : in SET; SET_2 : in SET) return SET is
    - intersection operator
  begin
    return (SET_1 and SET_2);
  end "*";

  function "+" (ELEMENT : in UNIVERSE; SET_1 : in SET) return SET is
    - union operator
    VALUE_SET : SET := SET_1;
  begin
    VALUE_SET (ELEMENT) := TRUE;
    return VALUE_SET;
  end "+";
```

```
function "+" (SET_1 : in SET; SET_2 : in SET) return SET is
  - union operator
begin
  return (SET_1 or SET_2);
end "+";

function "+" (SET_1 : in SET; ELEMENT : in UNIVERSE) return SET is
  - union operator
  VALUE_SET : SET := SET_1;
begin
  VALUE_SET(ELEMENT) := TRUE;
  return VALUE_SET;
end "+";

function "-" (SET_1 : in SET; ELEMENT : in UNIVERSE) return SET is
  - difference operator
  VALUE_SET : SET := SET_1;
begin
  VALUE_SET(ELEMENT) := FALSE;
  return VALUE_SET;
end "-";

function "-" (SET_1 : in SET; SET_2 : in SET) return SET is
  - difference operator
begin
  return (SET_1 and ( not SET_2));
end "-";

function "<=" (SET_1 : in SET; SET_2 : in SET) return BOOLEAN is
  - subset operator
  VALUE_SET : SET := (SET_1 and SET_2);
begin
  return (VALUE_SET = SET_1);
end "<=";

function "<" (SET_1 : in SET; SET_2 : in SET) return BOOLEAN is
  - proper subset operator
  VALUE_SET : SET := (SET_1 and SET_2);
begin
  return ((VALUE_SET = SET_1) and (VALUE_SET /= SET_2));
end "<";

function IS_A_MEMBER (ELEMENT : in UNIVERSE; OF_SET : in SET)
                      return BOOLEAN is
  - membership test
begin
  return OF_SET(ELEMENT);
end IS_A_MEMBER;
```

```
    function IS_EMPTY (SET_1 : in SET)
                        return BOOLEAN is
      -- test for an empty set
    begin
      return (SET_1 = NULL_SET);
    end IS_EMPTY;

    function NUMBER_IN (SET_1 : in SET)
             return NUMBER is
      -- test for cardinality
      COUNT : INTEGER := 0;
    begin
      for INDEX in UNIVERSE
        loop
          if SET_1(INDEX) then
            COUNT := COUNT + 1;
          end if;
        end loop;
      return COUNT;
    end NUMBER_IN;
  end SET_PACKAGE;
```


The Fourth Design Problem: Process Control Chapter 18)

```
with SYSTEM;
separate (MONITOR_TEMPERATURES)
task body ALARM is
  --
  BITS     : constant := 1;
  WORDS : constant := 16 * BITS;
  --
  type LIGHT          is (OFF, ON);
  for  LIGHT'SIZE     use 1 * WORDS;
  for  LIGHT          use (OFF => 16#0000#, ON => 16#FFFF#);
  FAULT_LIGHT         : LIGHT := OFF;
  for  FAULT_LIGHT use at 16#00180#;
  --
  type LIMIT_CHECK            is array (SENSOR_NAME) of LIGHT;
  for  LIMIT_CHECK'SIZE       use (SENSOR_NAME'POS(SENSOR_NAME'LAST) + 1) * WORDS;
  OUT_OF_LIMITS_LIGHT         : LIMIT_CHECK := LIMIT_CHECK' (others => OFF);
  for  OUT_OF_LIMITS_LIGHT use at 16#00110#;
  --
```



```
begin
  loop
    select
      accept POST_FAULT_IN_SENSOR do
        FAULT_LIGHT := TRUE;
      end POST_FAULT_IN_SENSOR;
    or
      accept POST_OUT_OF_LIMITS(ON_SENSOR : in SENSOR_NAME) do
        OUT_OF_LIMITS_LIGHT(ON_SENSOR) := ON;
      end POST_OUT_OF_LIMITS;
    end select;
  end loop;
end ALARM;

with DEVICE_IO;
separate (MONITOR_TEMPERATURES)
task body RECORDING_DEVICE is
  begin
    loop
      accept LOG_THE_STATUS(OF_SENSOR  : in SENSOR_NAME;
                            WITH_VALUE : in SENSOR_VALUE;
                            WITH_STATE : in SENSOR_STATE) do
        DEVICE_IO.PUT(OF_SENSOR);
        DEVICE_IO.PUT(WITH_VALUE);
        DEVICE_IO.PUT(WITH_STATE);
      end LOG_THE_STATUS;
    end loop;
end RECORDING_DEVICE;
```

```
∇ X←SET Z
  X←,UNIVERSE∈Z ⍝ BIT VECTOR FORM
∇

∇ X←Y UNION Z
  X←Y∨Z ∇

∇ X←Y INTERSECT Z
  X←Y∧Z ∇

∇ X←Y MINUS Z
  X←Y>Z ∇

∇ X←Y EQUAL Z
  X←∧/Y=Z ∇

∇ X←Y NOTEQUAL Z
  X←∨/Y≠Z ∇

∇ X←Y SUBSET Z
  X←∧/Y≤Z ∇

∇ X←Y PROPERSUBSET Z
  X←(∧/Y≤Z)∧∨/Y<Z ∇

∇ X←EMPTY Z
  X←~∨/Z ∇

∇ X←NUMBERIN Z
  X←+/Z ∇
```

GENERIC SET PACKAGE
WITH KNOWN UNIVERSE

```
∇ X←UNIQUE Z
  X←((Z⍳Z)=⍳⍴Z)/Z ⍝ UNIQUE ELEMENTS OF VECTOR
∇

∇ X←Y UNION Z
  X←UNIQUE Y,Z ∇

∇ X←Y INTERSECT Z;Q
  Q←Y UNION Z
  X←((Q∈Y)∧Q∈Z)/Q ∇

∇ X←Y MINUS Z
  X←(~Y∈Z)/Y ∇

∇ X←Y EQUAL Z
  X←(∧/Y∈Z)∧∧/Z∈Y ∇

∇ X←Y NOTEQUAL Z
  X←~Y EQUAL Z ∇

∇ X←Y SUBSET Z
  X←∧/Y∈Z ∇

∇ X←Y PROPERSUBSET Z
  X←(∧/Y∈Z)>∧/Z∈Y ∇

∇ X←EMPTY Z
  X←0=⍴Z ∇

∇ X←NUMBERIN Z
  X←⍴Z ∇
```

GENERIC SET PACKAGE
WITH UNKNOWN UNIVERSE

RCA
Advanced
Technology
Laboratories

ENCOURAGEMENT OF SOFTWARE REUSABILITY

GENERATION OF REUSABLE SOFTWARE

- ADA
  - ACHIEVED THROUGH OVERLOADING AND COMPLEXITY
  - WON'T BE DONE WITHOUT EXTRA INCENTIVE
- APL
  - OCCURS ALMOST NATURALLY
  - USERS WANT TO DO IT

    TO HAVE SAME BENEFITS AS PRIMITIVE FUNCTIONS

COMPOSITION OF REUSABLE SOFTWARE

John R. Mellby

Texas Instruments

1.0 Introduction

This paper summarizes problems associated with the composition of systems from reused software, and demonstrates the feasibility of software composition as a development technique.

2.0 Background

The process of reusing software involves three separate functions.

(1) Locate a potential package.

(2) Determine the suitability the package's function/implementation.

(3) Adapt the package or compose packages for new systems.

The third step, manipulating packages once they have been located, is potentially the most interesting, and it is an area where few tools exist to assist in the development process.

Rather than try to cover the entire domain of software reusability, such as design methodologies for the creation of reusable software, its collection into reusable software libraries, or the identification of a package for reuse, this paper will discuss a limited topic. It will address some of the problems associated with composing a system by importing reusable software.

First we will discuss the advantages of Ada and some problems associated with reusing a single package. Then we will discuss the capability of composing a new system out of reused packages.

3.0 Software Importation

The ideal reuse of software should involve no recoding at all. This is not always possible but when packages are designed correctly, there should be only limited modification of the package necessary.

The three areas of concern in modifying the reused package are:

(1) Interfaces to the package,

(2) Internal data structures,

(3) Exported data structures, and

3.1 Interfaces

Anyone who has used standard Fortran subroutine libraries knows that the routines are written with wide, general purpose interfaces. This means there are more parameters then are used in a typical application, so in most uses of the routine one or more parameters are passed some constant value and for all practical purposes are not used. To make a routine widely reusable, the interfaces should be general. On the other than, from the reuser's point of view, the interfaces would be better if they are suited to his application. Ada gives us the means to achieve these seemingly opposite goals. From the developer's point of view the package is made general with a wide interface. As many reasonable parameters are included to make the subprogram applicable to many uses. These parameters should be given defaults to cover the most common cases.

When the user takes this subprogram he may not have to use the general interface due to Ada's named parameter association and the parameter defaults. For example, the SORT procedure below has a parameter designed to let it handle some general cases (although it is not a completely general sort). The user could specify how much of the array to sort, and whether to sort it in ascending or descending order.

```
procedure

SORT(
LIST:
IN OUT ARRAY_OF_STRINGS;

TOP_ELEMENT:
IN ARRAY_OF_STRING_RANGE
= ARRAY_OF_STRING_RANGE'LAST:

UP_OR_DOWN :
IN DIRECTION
= ASCENDING );
```

One user might only want to sort the complete array and always in ascending order. In this

case, the typical sort call would be:

```
SORT( LIST => INPUT_ARRAY );
```

and the defaults for TOP_ELEMENT and UP_OR_DOWN would be used.

In many cases the reused software does not have the proper defaults or does not have defaults at all. The user may still be able to narrow the interface to this specifications by renaming it. Renaming allows several options:

(1) A name more suited to the application,

(2) Alternate names for the parameters, and

(3) New default values for the parameters.

So to do a sort always in descending order rename:

```
procedure

SORT DOWN)
LIST:
IN OUT ARRAY_OF_STRINGS;

TOP_ELEMENT:
IN ARRAY_OF_STRING_RANGE
=> ARRAY_OF_STRING_RANGE'LAST:

UP_OR_DOWN :
IN DIRECTION
=> DESCENDING ) renames SORT;
```

Now the new sort call is:

```
SORT_DOWN( LIST => INPUT_LIST );
```

3.2 Changing The Interface/Data

The previous actions are sufficient when the proper interfaces and data structures are already built into the reused software. In some cases the needed interfaces may not exist in the reused package, or the package may exist but may perform the operation on the wrong type of data. Initially this is likely to be common since few programmers write their software to be reusable. As the use of Ada features such as generics, parameter defaults, and renaming become more common, it will be easier to reuse software. Unfortunately, the difficulty of learning Ada is going to pose the first barrier to overcome before people begin to think of reusability.

There are several general type of changes necessary for software reuse:

* Missing functions,
* Wrong data types,
* Data types which can be tailored (generics), and
* Data types exported to calling systems.

3.2.1 Missing Functions

If the reused package is missing a necessary function there is little that can be done other than writing new code.

3.2.2 Incorrect Data Types

Frequently programmers will want to perform similar functions on different data. The sort function above is written for ARRAY_OF_STRINGS but we might want to do a sort of an array of integers. To convert this software we can recode it to another type.

It may be possible to automate this conversion. A RETYPE tool to do this might be given a new type, so that it replaced ARRAY_OF_STRINGS. RETYPE would also have to check the operations on these newly typed elements so that they were still legal. RETYPE might also require certain new functions or operations be supplied to manipulate the new type.

RETYPE would support the necessary type conversion operations. In the sort example above, we might change the type to ARRAY_OF_INTEGERS. Depending on how this type was originally implemented, RETYPE may have to verify that ARRAY_OF_INTEGERS is an array, or that the array's range is integer. In addition, RETYPE would check the operations on the LIST parameter to see that the operations were legal on the new type. To make these operations legal, RETYPE might prompt for new operations such as assignments or a " " function for that type.

Obviously such an approach will not always work. To convert a sort procedure to operate on a linked list may not be possible without entirely recoding the procedure, depending on the type of sort performed.

The final situation is that conversions of data may be necessary, they sometimes may be automated, and sometimes data conversions is insufficient to reuse the software.

3.2.3 Generic Data Types

A better approach is to make the data on which the package is based generic. As a brief example, a generic sort might force the type

sorted to be an array. The elements of the array, the array range, and the " " operation would be generic parameters. A code fragment for this is below.

```
GENERIC

type INDEX is (  );
type ELEMENT is private;
type LIST is array ( INDEX ) of ELEMENT;
with function " "(LEFT, Right : in ELEMENT)
   return BOOLEAN;
procedure SORT ( TABLE : in out LIST );
```

This is directly supported by so there is little new to add here.

3.2.4 Exported Data Types

The easiest way to treat data types exported by the reused package is to consider them as private types. In this case the primary operations on object of that type are the operations provided by the reused package.

An example of that is the dynamic string package stored in the Ada Repository or ARPANET. This package exports a type DYN_STRING. In using this package, dynamic string objects can be created and then manipulated through the functions in the dynamic string package.

Like generics, this is inherently part of the Ada language, and should be familiar to anyone as an application of information hiding techniques towards reusability.

4.0 Software Composition

The eventual goal of people looking at application generators is to create software without programming. It should become easily possible to automate some software projects with a few "software composition" tools and a reasonable library of reusable routines.

We will demonstrate this by giving a scenario for development of a program with very little coding. First we describe the problem, then some packages (both existing, and planned) to be used in the system, and finally describe how the system is created.

4.1 System To Be Built

We want a simple address/phone list program which will allow names, addresses, and phone numbers to be entered and retrieved. To be brief we will ignore the problem of long-term storage and assume the list to be in memory.

4.2 Packages

4.2.1 Menu Input

TI is currently finishing a package which will accept a description of a set of data and create software to generate a menu prompting a user to enter this set of data.

The current system accepts batch or interactive input to describe the data set. It is proposed to extend the system to allow as input an Ada record definition to define the data set.

4.2.2 Record Output

A simple conversion of the above package would be a system which accepts an Ada record and creates a package to output records of that type.

4.2.3 Control Menu

A proposed modification of the Menu Input system would create a control menu. This would display a menu of actions on the terminal screen. Each input would be matched with an action to be performed. The action to be performed would be in terms of one or more subprogram calls.

4.2.4 Table Storage

A simple storage system would be simple to create. This would be generic on some data type, and would provide the facilities to store and recall items of that type.

The facilities provided might be: Create-Table, Store-Item, Locate-Item, and Delete-Item.

4.3 Creating The System

The ability to generate this system lies in the ability to create instances of each of the above packages tailored to our application. The creation of these tailored packages is either by instantiation of a generic or the package itself is generated by a program designed for that purpose.

The steps in creating the system are:

(1) Write an Ada record describing the address data structure.

(2) Write a statement to instantiate the Table package for this record.

(3) Run the Menu Input program to generate a package for inputting this record.

(4) Run the Record Output program to generate a package to output this record.

(5) Run the Control Menu program to generate a menu package offering options for Table.Create, Input (reading and storing the record), and Output (Locate and Output the record).

(6) Write a main unit to start the Control Menu.

(7) Compile the packages and link the system.

Of the seven steps, three involve writing code. The generic instantiation is basically one statement, and the main unit should be very short.

A small application has now been created with very little programming. Even more importantly, there is almost no place where an error can occur.

5.0 SUMMARY

We have tried to describe some typical packages, and some characteristics of packages which would enhance reusability. We have shown that software systems can be constructed with very little programming and with very little opportunity for error. This demonstrates the viability of software reusability as a productivity-enhancing technology.

TEXAS INSTRUMENTS
ADA TECHNOLOGY BRANCH

1.0 INTRODUCTION

This is a summary of expertise and qualifications for participation in the STARS Workshop on Reusable Components of Application Software.

Texas Instrument's Equipment Group is a 15,000 person organization based in Dallas, Texas, whose business is Government Electronics. The following briefly summarizes Texas Instruments experience in the areas of Ada and reusable software. This finishes with the resume of John Mellby, the author of these documents and TI's proposed representative to the Workshop on Reusable Components.

Enclosed separately is a position paper on the composition of reusable software.

1.1 Ada Technology Branch

TI management, recognizing the DoD's intent and policy as regards Ada, has created the Ada Technology Branch within the Advanced Computer Systems Laboratory. The Ada Technology Branch is chartered to enhance Equipment Group's competitive position with regards to Ada and is divided into the following sections:

- Compiler Development
- Real Time Issues and Methodologies
- Tools and Environments
- Training and Education

1.2 MIL-STD-1815A (Ada) LANGUAGE EXPERIENCE

Since 1977, Texas Instruments Incorporated has been engaged in numerous research and development programs involving the design, implementation, and support environments for Ada which uniquely qualify us for this program. Here are some highlights:

(1) Active participation since their inception in the Ada Implementor's Group and AdaTec/SIGAda, the ACM special interest group on Ada. TI is an institutional sponsor of SIGAda.

(2) Currently participating (by invitation) in the Navy led KAPSE Interface Team (KIT) whose goal is to evaluate the suitability of KAPSE interfaces with regard to interoperability and transportability of APSE tools. TI has a representative in the Common APSE Interface Set working group and a representative in the Guidelines and Conventions (GAC) the working group. As part of the KIT effort, TI will produce, under Naval Ocean Systems Center Contract N66001-82-C-0440, the APSE Interactive Monitor (AIM), a software tool which will provide a virtual terminal interface (multi-windowing, multi-tasking) to an APSE user. Due to our participation in this program, TI has attained significant expertise in both the ALS and AIE KAPSEs, and was selected to participate in the development of the Common APSE Interface Set (CAIS). Our representative has been responsible for defining CAIS I/O interfaces, beyond those defined by the Ada language, including:

(a) a virtual terminal interface, and

(b) a mechanism for performing Interprocess Communication

(3) Currently participating (by invitation) in the Guidelines and Conventions working group (GACWG) of the KIT. The objectives of this working group are:

(a) to develop requirements for APSE Interoperability and Transportability (IT),

(b) to develop guidelines, conventions, and standards to be used to achieve IT of APSEs,

(c) to develop APSE IT tools to be integrated into both the ALS and AIE,

(d) to monitor ALS and AIE development efforts with respect to APSE IT, and,

(e) to develop and implement procedures to determine compliance of APSE developments with APSE IT requirements, conventions, and standards.

(4) Currently participating in the APSE Evaluation and Validation Task. This is a tri-service activity with primary responsibility assumed by the Air Force (AFWAL/AAAF). Its goals are to create a validation suite for APSE conformance to the CAIS and provide a framework for the evaluation of tools for APSEs as well as evaluation of complete APSEs for potential consumers of Ada/APSE technology. Our involvement was through the submission of a position paper to the first annual E and V

Workshop, held 2-6 April 1984 at Airlie, Virginia. Our representative to that meeting is considered a member of the E and V distinguished reviewers group which will provide feedback to the E and V Team.

(5) Presently delivering an extensive in-house Ada training program, using the Data General Ada Development Environment.

(6) Presently working on a code generator for the Telesoft Ada Compiler. This will be VAX hosted and produce code for the TI9900 microprocessor family.

(7) Reviewed and commented on METHODMAN, the AJPO software methodology document.

(8) ACSL personnel are using object-oriented design (BOO83) in the APSE Interactive Monitor project. Additionally, Ada as a PDL (program design language) is also being used on selected test projects.

(9) Currently developing our own PDL.

(10) ACSL has been awarded a contract for the development of five tools to be written in Ada for the WWMCCS Information System upgrade. These tools are a Virtual Terminal, Forms Generator, Spelling Checker, Ada Style Checker and a Screen Generator.

(11) ACSL leases a Data General MV/10000 computer system running the Ada Development Environment (ADE). The validated Ada compiler which runs under the ADE is the principle Ada compiler used by ACSL personnel.

1.3 Experience In Reusability

1.3.1 Future APSE Workshop -

TI was represented at the IEEE's Future APSE Workshop at Santa Barbara, CA in September of 1984. John Mellby, the TI representative, was a member of the working group of Reusable Software. As part of this, a paper, "Issues in Software Reusability" is being prepared for publication in Ada Letters later this year. The contributing authors are: Bill Jones (NASA Ames), Herb Krasner (MCC), Steve Litvintchouk (MITRE), John Mellby (Texas Instruments), Jerry Mungle (TRW), and Herb Wilman (Raytheon).

1.3.2 Components Library -

The Advanced Computer Systems Laboratory of TI has created a components library to house software packages (not necessarily Ada) for potential reuse.

1.3.3 Ada Repository -

Richard Conn, a Texas Instruments employee, has created and is maintaining the Ada Repository. This is a facility on the SIMTEL20 machine on ARPANET which houses Ada packages and tools. All tools are to be available to the public through the ARPANET. For more information contact Rick Conn at CONN%EG@CSNET-RELAY, for ARPANET or CSNET mail.

1.3.4 Ada Tools For WWMCCS Improvement System -

The Ada Technology Branch is currently working on five tools for the WWMCCS Improvement System, as mentioned above. This includes packages specifically designed for transportability and reusability such as a package providing Virtual Terminal Interfaces, and packages which create other packages to allow menu-driven input. All these packages will eventually reside on the Ada Repository on ARPANET.

RESUME

JOHN R. MELLBY

EDUCATION

Ph.D Computer and Information Science, Ohio State University, 1980.
B.A. Mathematics and Physics, St. Olaf College, 1973.

Dr. Mellby is currently a member of the Ada Technology Branch. He is leading the real-time issues and methodologies section and guiding the Ada Experiment, an internal research effort to investigate the real-time implications of Ada through an experimental real-time system. His current project is the Ada Style Checker, a tool being developed under contract to the Naval Ocean Systems Center. He is also in the group preparing the report on "Software for Reusable Systems" for the SigAda Future APSE workshop. Dr. Mellby previously held the responsibility of work group leader for Distributed Computer Systems Support within the Languages and Tools section of the Advanced Computer Systems Lab. Dr. Mellby was also the primary software designer on the TEAMS project (Test and Evaluation Aircraft for Multi-sensor Systems), whose objective was to produce a real-time aircraft-based test bed. Prior to his employment at Texas Instruments, Dr. Mellby was an Assistant Professor of Mathematics and Computer Science at St. Cloud State University in Minnesota.

REUSABILITY EFFORTS

Reusing Minor Routines — Sorting, Stacks, Queues, etc.

Significant Packages

- Help Functions
- String Manipulations
- Word Manipulations
- Spelling

John Melby
Texas Instruments

REUSABILITY PROBLEMS

Compilers

2 Packages from Ada Repository —

One reused by rewriting data structure — Telesoft problems
Other not reusable — Embedded non-encapsulated OS calls

Engineers not writing reusable code

Generics — New feature not used at all times
Default Parameters
Constants
Attributes

Planning — for reuse not done

Encapsulation
Interfaces

Engineers not trying to Reuse Software

ADA REUSABLE CHARACTERISTICS

Packaging

- Data and Type Encapsulation

Specifications

- Separation of form/function from implementation
- Information Encapsulation
- Declaration of necessary information through constants and default values

Generics

Named and Default Parameters

NEEDS FOR REUSABILITY GOALS

After First Draft, Systems/Packages need to be redone as Reusable

Programmers need to use Ada Features supporting Reuse

Use Standard (validated) compilers

Collect Local Library of Reusable Components

Plan for Reusability during Requirements and Design

ADA REPOSITORY

HISTORY

A Public Domain Source of Ada Programs/Components

Created November, 1984

Located at SIMTEL-20 on MILNET

Maintained by Rick Conn (TI)

(214) 952-2139

ADA-SW-REQUEST@SIMTEL20

ADA REPOSITORY

CONTENTS

General Directory Structure in place

- 24 Tools and Components
- 2000 Accesses

Submissions

- NOSC/WIS Tools
- SEI
- Naval Weapons Center

Access by Anonymous FTP on ARPA/MILNET

Shortly to be an open account on ECLB for non-ARPANET people to access the Repository

On ECLB - In ADA-INFORMATION, a file describes Repository access:

ADA_REPOSITORY.HELP

MICRO ISSUES IN REUSE FROM A REAL PROJECT

Goeffrey O. Mendal
Ada?80 Technology Support Lab
Lockheed Missiles and Space Company, Inc.
Sunnyvale, CA

Abstract

The reuse of generic program units will significantly decrease the time and cost of developing software in Ada due to savings in designing, coding, and maintaining Ada software. This paper describes the design and development of a generic sorting package currently in use at Lockheed Missiles and Space Company (LMSC) in Sunnyvale, CA. This paper will focus on the user's view of the package, e.g., the package specification.

It will be shown that reuse can be accomplished in practice, during and even prior to code development, in Ada. This package demonstrates the feasibility of reducing the time and cost of building software through reuse and achieving acceptance in large aerospace projects.

The generic sorting package currently includes six well known sorting algorithms: Quicksort, Heapsort, Bubble Sort, Bubble Sort with Quick Exit, Insertion Sort, and Straight Selection Sort. Any data type can be sorted, including provisions for limited types. The generic sorting package operates on arrays and requires only the name of the array to be sorted. However, users may also sort array slices and can optionally request that instrumentation analysis results be returned along with the sorted array (or slice) to communicate performance. High standards of readability and understandability have been imposed so that this package can be used in a turn-key environment. In fact, the generic sorting package can be easily implemented as an elementary expert system and can easily be integrated with a merging package to sort data residing on external memory devices.

Reusable software needs interface modules of many different kinds. It is important to identify these interfaces early in the design phase. In fact, one can further generalize the domain of objects that can be sorted by writing a sort selector package which will automatically choose an optimal sorting package based on the type of the object. The sort selector could take the form of an expert system, which itself would be considered reusable.

For more complex applications, the user may specify the ordering relation on which to sort. Thus, the user is not limited to ascending or descending orders as in conventional sort procedures. That is, with this Ada sorting package, user-defined ordering relations are supported. A fanciful example of a user-defined ordering relation may be to sort an array of CHARACTERs based on a historical account of the temperature in New York City during the past three months. The inclusion of arbitrary ordering relations can ensure the stability of sorting algorithms (they are unstable) if they do not preserve the relative ordering of array components with equal sorting keys). It is noteworthy that the challenge of reuse led in this case to a result that is more general than the norm.

This generic sorting package has been proposed for use in a major LMSC project. It has been recognized by the upper-level project managers and by the customer that the use of generic program units such as this one will significantly decrease the time and cost of building the remaining system software. Currently, this package is being used as an example of design-for-reuse in an Ada Design Methodology course developed at LMSC. It is this project's policy that programmers must demonstrate why they are not taking advantage of this package before being allowed to write their own sorting routines.

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

1.0 OVERVIEW

Sorting is an abstraction, or more precisely, a functional abstraction. One sorts data by selecting an appropriate algorithm designed to exploit the most important attributes of the data. A sorting package that aids the user in selecting an appropriate algorithm can be viewed as an elementary expert system.

How general should a sorting package be? Should it sort many different types of objects, or only one domain of objects? Our experience in reusability at LMSC has taught us to restrict the domain of the application thus making the overall problem at hand less complicated.

Sorting is a frequently used operation in large software systems. Hence, it is important to make a reusable sorting package highly readable and understandable. Programmers should not be burdened with the details of various algorithms. Instead, they should be able to make use of the package in a *turn-key* environment.

2.0 AN INTRODUCTION TO SORTING

What is sorting, and why sort? Sorting is a process of arranging objects from one or more data sets to form a data set ordered on one or more attributes of the data. Sorting can increase the speed and reduce the complexity of algorithms that use data. Specifically, sorting aids in searching. Imagine trying to locate the telephone number of an individual in a telephone directory that is not sorted by last name. A telephone directory that is sorted by last name allows one to easily search for an individual's telephone number. However, the same directory cannot easily be used to search for all individuals who live on *Main* street and have telephone numbers that begin with *764* for area code *313*.

Although many ingenious sorting algorithms have been designed, there are still some fascinating unsolved problems. Sorting can also be used as a case study on how to attack computer problems in general. Important principles of data structure manipulation surface in sorting algorithms. Sorting techniques also portray ideas in the analysis and design of algorithms. [6].

Persons who are already familiar with sorting algorithms may wish to skip this section and continue at Section 3.0. The remainder of this section was taken from Knuth [4] and Sedgewick [6].

2.1 A Formal Definition of Sorting. The following quotations were found in Knuth [4]:

"But you can't look up all those license numbers in time," Drake objected. "We don't have to, Paul. We merely arrange a list and look for duplications."

Perry Mason (The Case of the Angry Mourner, 1951)

"Treesort" Computer—With this new 'computer-approach' to nature study you can quickly identify over 260 different trees of U.S., Alaska, and Canada, even palms, desert trees, and other exotics. To sort, you simply insert the needle.

Catalog of Edmund Scientific Company (1964)

The sorting problem can be described as follows:

You are given N records: R1, R2, ..., RN. The entire collection will be called a *file*. Each record has a key which governs the sorting process, and optional additional information (satellite information) that is not used, but is associated with each record. The *file* is held in a computer's high speed memory (RAM) or stored on an external memory device.

An ordering relation (also known as a collating sequence) "<" is specified on the keys so that for any of these values a, b, c, the following conditions are satisfied:

(i) exactly one of the possibilities $a < b$, $a = b$, $b < a$ is true
(ii) if $a < b$ and $b < c$, then $a < c$

These two properties characterize the mathematical concept of a total ordering. Any relation "<" satisfying (i) and (ii) can be sorted by any well known sorting algorithm.

The goal of sorting is to determine a permutation p(1) p(2) ... p(N) of records which puts keys in a non-decreasing order $Kp(1) <= Kp(2) <= ... <= Kp(N)$.

Sorting is *stable* if we make the further requirement that records with equal keys retain their original relative order, i.e.:

$p(i) < (j)$ whenever $Kp(i) = Kp(j)$ and $i < j$

Sorting can be classified into two different problems: internal and external sorting. Internal sorting assumes that the records are kept in RAM. External sorting assumes that there are more records than can be held in RAM simultaneously. Thus, internal sorting allows

quick, random access while external sorting requires slower access that could also be restricted to sequential order.

Internal sorting allows more flexibility, structure, and access of data. External sorting requires one to cope with rather stringent accessing constraints. For example, if you are given ten playing cards and are asked to sort them, you could do so using only your hands. You are able to sort these ten cards without an *external store* such as a desktop. If, however, you are asked to sort 1000 cards, then you will require the use of a desktop. You are no longer able to sort without an external store. As we shall see, the algorithms used to sort with and without an external store necessarily differ.

The time required to sort using a good general-purpose algorithm is roughly proportional to NlogN, i.e., we make about log*N* *passes* over the data. Thus, twice as many records will increase the sorting time roughly by a factor of two, all else being equal.

2.2 Internal Sorting. Suppose you are trying to solve the following problem.

> Memory locations M, M+1, ..., M+4 contain five numbers. You must write a program that rearranges the numbers, if necessary, so that they are placed in ascending order. Imagine that this is your first time sorting anything and you have no prior knowledge of how to proceed. (You might try writing this program before reading on.) Some of the possible solutions you might have used are as follows:
>
> — Insertion Sort: items are considered one at a time. Each new item is inserted into the appropriate position relative to the previously sorted items. This is how bridge players sort hands; they pick up one card at a time.
>
> — Exchange Sort: if two items are out of order, they are interchanged. This process continues until no more exchanges are required.
>
> — Selection Sort: the smallest/largest item is found and separated from the rest, then the next smallest/largest, and so on.
>
> — Enumeration Sort: each item is compared with each of the others; counting the number of smaller keys determines the item's final position.
>
> — Special-Purpose Sort: one that works well for sorting five elements as above, but cannot readily be generalized for a smaller/larger number of items.
>
> — New, Super Sorting Technique: one that provides a significant improvement over known methods.

As you can see, many sorting algorithms exist. Each method has its own advantages and disadvantages, so that it may outperform the others given some configuration of data and hardware. There is no known *best* way to sort, but there are many *best* algorithms, depending on what is sorted on what machine for what purpose.

2.3 External Sorting. External sorting must be performed when there are more records than the computer can hold in RAM simultaneously. The solutions are quite different from internal sorting, even though the problem is the same.

The data structures must be arranged so that slow peripheral memory devices can quickly cope with the requirements of the sorting algorithms. As a consequence, most internal sorting algorithms, by themselves, are useless for external sorting, and it is necessary to reconsider the whole question.

One very common solution is to divide the file into separate subfiles that each fit in RAM. Next, separately sort each subfile with an internal sorting algorithm. Finally, use an external merging algorithm on all subfiles. Merging algorithms only require very simple data structures (linear lists) accessed in a sequential manner; hence merging operations can be performed without difficulty on the least expensive memory devices.

Internal sorting followed by external merging is very common. To simply merge the records into longer and longer lists from the start will result in redundant read/write operations on external memory devices, and will generally be very inefficient.

2.4 Sorting Factors. The most important factor in sorting any file is its size. If the file contains less than 500 records, then it will probably be more efficient to write and use a simple sorting algorithm. If the file contains less than 50 records, a simple algorithm is always fine. Sophisticated algorithms are not justified for small files unless they must be used many times.

Files that are already sorted (or partially sorted) or files that contain many equal keys are easy to sort. In fact,

such files are often sorted faster using a simple algorithm rather than a sophisticated one.

A general rule for simple algorithms is that they take N^2 steps to sort N randomly arranged records. Using *big Oh* notation, O(...) represents the number of steps required. A general rule for sophisticated algorithms is that they take O(NlogN) steps. The *O* stands for *on the order of*.

The second most important factor in sorting is the extra memory used by the algorithm. Some algorithms sort in place and use no extra memory except for a small stack or table. Some algorithms that rely on linked lists use N extra words of memory for pointers. Other algorithms require enough extra memory to hold a copy of the file being sorted.

Most simple sorting algorithms are stable and most sophisticated algorithms are not. It is easy to take stability for granted, but the unpleasant effects of instability can cause disbelief. For example, if a teacher has an alphabetized class list, but wants to sort it based on the grades of the last exam, it is only natural to assume that students with equal grades on that exam will remain in alphabetical order as before.

Sorting algorithms generally access records in one of two ways:

— Keys are accessed for comparison.

— Entire records are accessed to be moved.

An indirect sorting algorithm does not necessarily rearrange records; rather pointers to the records are rearranged. Indirect sorting is usually more efficient than direct sorting (at the expense of memory overhead) because it is usually quicker to move pointers than large records.

2.5 An Overview of Common Internal Sorting Algorithms. This section briefly describes some of the more common internal sorting algorithms used today.

2.5.1 Straight Selection Sort. This instable algorithm first finds the smallest/largest record in the file and exchanges it with the record in the first position. Next, the second smallest/largest record is found and exchanged with the record in the second position. This process continues until the entire file is sorted.

Straight Selection Sort repeatedly selects the smallest/largest record from those not yet sorted. The running time of this algorithm is $O(N^2)$. The number of comparisons made is $(N^2)/2$ since the outer loop require N comparisons and the inner loop requires N/2 comparisons.

This algorithm is good for large records and small keys. It should only be used for files smaller than 1000 records.

2.5.2 Insertion Sort. This stable algorithm is based on the method used by bridge players to sort bridge hands. The records are considered one at a time, inserting each one in its proper place among those already considered (keeping them sorted). The record considered is inserted merely by moving larger records one position to the right, and then inserting the record considered into the vacant position.

The running time of this algorithm is $O(N^2)$. The inner loop is executed $(N^2)/2$ times. The running time depends on the number of inversions: for each record, count the number of records to its left which are greater. This is the distance that the records have to be moved when inserting into the sorted file.

2.5.3 Bubble Sort. This stable algorithm makes passes through the file, exchanging adjacent records, if necessary. A simple modification of this algorithm can be made so that when no exchanges are required, the algorithm promptly terminates. However, this improvement can make the algorithm instable.

This is generally the worst sorting algorithm for random data. Its running time is $O(N^2)$, However, it is the simplest algorithm to comprehend and hence it is often the first one learned by computer science students.

When the data are non-random, such as in the case of adding records to a sorted file, Bubble Sort with Quick Exit and Insertion Sort are O(N) (linear).

2.5.4 Quicksort. This ingenious algorithm was invented in 1960 by C.A.R. Hoare. It has been the center of much algorithm analysis and design. Since its inception, many cousins have been devised to handle various *worst cases*.

Quicksort is a good general-purpose sorting algorithm. It consumes less resources than any other sorting algorithm in many situations. Its good points include the fact that it is an in-place sorting algorithm (it uses only

a small auxiliary stack), its running time is O(NlogN), and has an extremely short inner loop. Its drawbacks are that it is highly recursive, its worst case running time is $O(N^2)$, it is instable, and it is fragile, that is, a simple mistake in its implementation might go unnoticed and could cause bad performance for some files.

Because the algorithm is so well balanced, effects of improvements in one part of the algorithm can be more than offset by effects of bad performance in other parts. Once a version has been developed, carefully tuned, and seems free of unexpected effects, it is likely to be the algorithm of choice for a library sorting utility or serious sorting application. It is important to invest extensive effort to make certain that Quicksort is not flawed.

The algorithm is based on the divide and conquer technique. Its one disturbing feature is that it runs very inefficiently given non-random data. For already sorted files, the paritions will be degenerate, the time required will be $(N^2)/2$, and the space for recursion will be N (which is unacceptable). For files with equal sorting keys, it may be difficult to decide whether to have the pointers stop on a key equal to the partition record, or to have one pointer stop and the other scan over all equal keys, or to have both pointers scan.

The best thing to do is to partition the file in half. This will make the number of comparisons satisfy the equation $C(N) = 2C(N/2) + N$ so that C(N) is approximately equal to 2NlogN. Therefore, the running time of the algorithm is O(NlogN).

Recursion can be removed by using an explicit stack to save the variables instead of having the programming environment do it implicitly through recursion. The stack size should be logN. A recursive method can abort due to no more memory (degenerate case on large files). There is no way to avoid this problem completely for large files without removing recursion. The simple use of an explicit stack will improve performance.

2.5.5 Heapsort. With this instable algorithm, the file is considered as a binary tree and the "heap" is an almost complete binary tree of N elements such that the content of each element is less than or equal to the content of its parent. The running time is O(NlogN). The algorithm is not recursive, and only requires extra space for temporary variables. During the sort, the file is used as a workspace.

This algorithm usually makes four times as many exchanges and twice as many comparisons as Quicksort. Hence, even though this algorithm is O(NlogN) and is non-recursive, its performance is still not as good as Quicksort.

2.6 Summary and History. Sorting is a process which rearranges a file of records so that the keys are in order. Orderly arrangement is useful because it brings equal keys together, allows for efficient processing of multiple files sorted on the same key, leads to efficient retrieval algorithms, and makes computer output look more authentic.

It would be nice if only a few sorting algorithms would dominate all of the others, but each has its own peculiar virtues. All algorithms deserve recognition because there are applications where each one turns out to be best. For external sorting, one must use comparatively primitive data structures, and great emphasis must be placed on minimizing input/output time.

Sorting isn't the whole story. While studying sorting algorithms, one is necessarily exposed to handling data structures, dealing with external memories, analyzing algorithms, and discovering new algorithms.

The origin for today's techniques is the 19th century, where the first sorting machine was invented. The U.S. census by 1880 was causing problems due the volume of data that required analysis. Herman Hollerith, a twenty year old employee of the Census Bureau devised an ingenious electrical tabulation machine to meet the need for better statistics gathering. This machine was first used in 1890. Hollerith's *isolating box* could sort 19071 cards in a six and one half hour working day. This was three times faster than human speed. Hollerith designed new machines for the 1900 census to handle the combinatorial population explosion. Hollerith's machine is the basis for radix sorting now used on digital computers.

There is evidence that a sorting routine was the first program ever written for a stored program computer. Designers of the EDVAC were interested in sorting because it epitomized the potential and non-numeric applications for computers. The limited memory of early computers made it necessary and natural to think about external sorting as well as internal sorting.

The history of sorting has been closely associated with many *firsts* in computing: data processing machines, stored programs, software, buffering methods, and work on analysis of algorithms and computational complexity.

3.0 REQUIREMENTS FOR A GENERIC SORTING PACKAGE

- Sort a one-dimensional array using the most efficient algorithm known (determined by an expert system).
- Process arrays of arbitrary length, including null arrays. In particular, allow array slices to be easily sorted.
- Design the package so that it can be easily integrated with a merging package in order to provide external sorting capabilities.
- Sort in any order, ascending, descending, or arbitrary. In particular, provide a mechanism to ensure the stability of any sorting algorithm.
- Sort any array component type, including provisions for limited types.
- Provide a selection of well known internal sorting algorithms.
- Provide an option for retrieval of instrumentation analysis results for each sorting algorithm.
- Allow the array index to be of any discrete type. In particular, do not require that the lower bound be zero or one.
- Provide sensible (most frequently used) defaults for the sorting algorithm and a predefined (default) ordering relation.
- Adhere to the highest defined level of standards for readability and understandability in the package specification and body to facilitate reuse.

Discussion of the Requirements. The requirements for this generic sorting package apply primarily to one-dimensional arrays. However, there are no limitations on such arrays. Thus, the array to be sorted may be of arbitrary length, indexed by any discrete type, and its values may incorporate any array component type except limited types. In Section 4.3 it will be described how one can use this package to sort limited types, and also how to integrate it with a merging package to sort external files.

This package will sort in any arbitray order. In addition, ascending and descending orders may be specified. A default order will be used if none is specified during instantiation. The flexibility of the ordering relation will become the most important requirement as we shall see below. In particular, it is this requirement that permits us to ensure the stability of every sorting algorithm.

The availability of choices among sorting algorithms is intended to cater to a wide class of problems. Some algorithms may execute faster if certain conditions can be satisfied, e.g., if the array is already partially sorted or many of they keys are equal.

As mentioned above, the components of the array can be of practically any type. The generic sorting package will sort arrays of characters, numeric types, enumeration literals, arrays, records, and access types (that designate other objects). This flexibility demonstrates the ability of the Ada generic feature to separate the algorithm from the data.

Optional instrumentation analysis results can be obtained to determine how many steps a sorting algorithm performed on a given array. These results can be used to compare the relative efficiency of various sorting algorithms.

Any programmer should be able to use the package without prior experience in the area of sorting. The user interface must be clean and simple, while also providing the flexibility demanded by complex applications.

4.0 DESIGNING THE GENERIC SORTING PACKAGE SPECIFICATION

In order to isolate the algorithm from its data types, we will make use of Ada's generic program units which allow us to pass data types and subprograms as parameters.

The generic program unit should be a **package** since we will need to export algorithm and instrumentation analysis result types. The sorting algorithms should be contained within a **procedure** since more than one result can be returned: the sorted array and the instrumentation analysis results.

The following portion of the generic program unit specification is taken from APPENDIX I. This portion is analyzed below.

```
with SYSTEM; -- predefined package SYSTEM
```

```
generic
   type Component_Type     is private;
   type Index_Type         is (<>);
   type Array_Type         is array
                           (Index_Type range
                           <>) of
                           Component_Type;

   with function "<" (
      Left, Right: in Component_Type)
      return BOOLEAN is <>;
package Sort_Pac is
   type Sort_Algorithm_Type is (Quicksort,
   Heapsort, Bubble_Sort,
   Bubble_Sort_with_Quick_Exit,
   Selection_Sort, Insertion_Sort);

   type Instrumentation_Analysis_Type is
   range -1 .. SYSTEM.MAX_INT;

   procedure SORT(
      Sort_Array           : in out
                             Array_Type;
      Number_of_ Comparisons,
      Number_of_Exchanges
                           : out
                             Instrumentation_
                             Analysis_Type;
      Sort_Algorithm       : in
                             Sort_Algorithm_Type
                           := Quicksort);

   procedure SORT (
      Sort_Array           : in out
                             Array_Type;
      Sort_Algorithm       : in
                             Sort_Algorithm_Type
                           := Quicksort);
end Sort_Pac;
```

4.1 The Generic Formal Type Parameters. We will need to know the type of the components being sorted, the type of the array index used, and the type of the array itself. In addition, we will need to know the ordering relation if the default (ascending) cannot apply.

4.1.1 The Component Type Parameter. We specify the component type as follows:

```
type Component_Type is private;
```

private specifies a component type that can implicitly support assignment, equality, and inequality operations. Note that nearly all data types in Ada do support these minimal requirements. This line of code essentially opens the floodgates to arrays of any kind of component except limited types; the problem of sorting limited types is non-trivial because not even assignment, equality, and inequality can be implicity supported. However, as shown below, limited types may be sorted if they are designated by access types. Examples of limited types are task types and file types.

4.1.2 The Index Type Parameter. We specify the array index as follows:

```
type Index_Type is (<>);
```

The symbol (<>) represents any discrete type. This is the most flexible type that Ada will allow as an array index. In Ada, the components need not be indexed by positive integers. Instead, Ada allows components to be indexed by such entities as days of the week, months in the year, galaxies in our universe, etc.

4.1.3 The Array Type Parameter. The specification of the array type is as follows:

```
type Array_Type is array (Index_Type range
<>) of Component_Type;
```

Note that this is simply a concatenation of the specifications in sections 4.1.1 and 4.1.2 above. We really are not adding any new information here. However, due to Ada's strong typing requirements, this superfluous information will be required later. Also note that this type declaration could be moved inside the package specification. Doing so, however, would require that all array objects be elaborated after the package has been instantiated. This would hinder information hiding. Also, the user would not *easily* be allowed to create his own array type name; s/he would have to use the name *Array_Type*.

What does that array type declaration say? First of all, it says that Array_Type is a template for an array. The array will have one and only one dimension denoted by Index_Type. The bounds of the array are left unspecified, as denoted by **range** <>. Thus, the length of the array is unspecified until a later time, or in Ada terms, it is unconstrained. Finally, the type of the array components is denoted by Component_Type.

4.2 The Generic Formal Subprogram Parameter. In order to conceptualize the ordering of data in a sorting routine, it is necessary to understand how ordering is implemented in Ada. Then it will be necessary to understand how one specifies an ordering relation in a generic program unit.

People do not reason in the same way as computers. When people *order* data, they do not think about it in terms of TRUE or FALSE. For example, in the following relational expression,

```
(3 + 4) < (5 + 6)
```

people do not conclude that the expression contains two possible values, namely TRUE and FALSE. Instead, we reason on a higher level by simple *knowing* that seven is less than eleven, and therefore the expression is *correct*. However, in Ada, the relational operators return results only in boolean terms. In Ada, relational operators are BOOLEAN functions.

The ordering relation in the generic sorting package is implemented by specifying a relational operator "<" as a generic formal subprogram parameter as follows:

```
with function "<" (Left, Right: in Component_
Type) return BOOLEAN is <>;
```

This is the most comprehensive line of code in the generic program unit. It is this line of code that generalizes all the tests of inequality in the Sort_Pac package, enhancing its reusability.

The with keyword in the declaration above is a notational element required to reduce a potential syntactic ambiguity. Without the **with** keyword, it would be tremendously difficult to distinguish a generic formal subprogram parameter from a generic subprogram. The **function** keyword specifies that this subprogram is, in fact, a function. The name of the function is "<". In Ada, all relational operators are BOOLEAN functions. The "<" function takes two values of type Component_Type as input and returns a BOOLEAN value (TRUE or FALSE). The remainder of this line of code, is <>;, denotes that the default function is assumed to be a function with the same name, directly visible at the point of generic instantiation. This function is usually the predefined < operator.

Thus, the < operator is the Ada implementation of ordering data. The default is ascending order. The package body associated with the Sort_Pac package specification will be responsible for implementing this default requirement. Note that the > operator is the counterpart of the < operator. Thus, > can be specified by the user at instantiation time in order to override the default ascending order, and instead implement a descending order.

Ada explicitly provides the < and > operators for discrete types and numeric types. In other words, the language has already defined functions named "<" and ">" that one can use to compare characters, numeric types, enumeration literals, etc. However, Ada does not (and conceptually cannot) provide < and > operators for composite and access types because it is impossible to say, for any set of such objects, which are *less than* or *greater than* the others. Hence, if an array of composite or access types is to be sorted, the Ada programmer must write his own ordering relation. Fortunately, as described later, the code for doing so is not complicated.

To reiterate, if Component_Type is not itself a composite or access type, then the "<" formal subprogram parameter can be left unspecified at instantiation time if an ascending order is desired, or replaced with ">" at instantiation time if a descending order is desired. Only if Component_Type itself represents a composite or access type must the Ada programmer write his own ordering relation. Of course, an Ada programmer could write his own ordering relation regardless, but this would be highly unusual if the type possessed implicit relational operators.

Assume that Component_Type is a record type. Then the Ada programmer will have to provide an ordering relation. This relation should compare a key component or components of the records to determine the ordering. For example, if we are dealing with elephants, then how are we to determine which elephant is less than (or greater than) another elephant? The answer could be that we base our decision on the length of their trunks. Thus if our elephants are represented as a record type

```
type Elephant_Type is
  record
    Trunk_Length,
    Height,
    Weight   : POSITIVE;
    Name     : STRING (1..40);
  end record;
```

then the component Trunk_Length is the key of the record type that will determine how elephants are to be sorted. We could have easily picked any of the other components, or some combination. To implement a function that will specify such an ordering, the Ada programmer would write a function such as the following:

```
function Elephant_Ordering (
  Elephant1, Elephant2 : in Elephant_Type)
  return BOOLEAN is
begin
  return Elephant1.Trunk_Length <
  Elephant2.Trunk_Length;
end Elephant_Ordering;
```

Thus, the function Elephant_Ordering when provided at *instantiation time as an actual parameter for the* generic formal subprogram parameter "<", will sort elephants in ascending order based on the length of their trunks. That is, the elephant with the shortest trunk will be first and the elephant with the largest trunk will be last. Note that their trunks are specified as positive numbers thus making the statement

```
return Elephant1.Trunk_Length <
Elephant2.Trunk_Length;
```

work. By replacing the < in the return statement above with a > or simply by interchanging the two operands, the elephants will be sorted in descending order, e.g., the elephant with the largest trunk will be first and the elephant with the smallest trunk will be last.

Not all ordering relations need be this simple. For example, the order of the elephants could be determined by a combination of record components. In another example, if we wish to alphabetically sort an array of taxpayers represented by the record type

```
type Taxpayer_Type is
  record
    Name       : STRING(1..40);
    Age        : NATURAL;
    ID_Number  : POSITIVE;
  end record;
```

we might first try using the ordering relation:

```
function Taxpayer_Ordering(
  Taxpayer1, Taxpayer2 : in Taxpayer_Type)
  return BOOLEAN is
begin
  return Taxpayer1.Name < Taxpayer2.Name;
end Taxpayer_Ordering;
```

But if we have two or more taxpayers with the same name, we might want to sort them by a secondary field, the ID_Number. Such an ordering relation might look like this:

```
function Taxpayer_Ordering(
  Taxpayer1, Taxpayer2 : in Taxpayer_Type)
  return BOOLEAN is
begin
  return (Taxpayer1.Name < Taxpayer2.Name)
          or
         ((Taxpayer1.Name = Taxpayer2.Name)
         and
         (Taxpayer1.ID_Number <
         Taxpayer2.ID_Number));
end Taxpayer_Ordering;
```

The ordering relation above uses a *collating sequence* (a succession of records ordered by attributes of the data within each record) in order to alphabetically order an array of taxpayers. If two or more taxpayers have the same name, the collating sequence dictates that they be ordered according to their identification number (ID_Number).

An ordering relation such as the one above will also assure the stability of unstable sorting algorithms. An interesting and noteworthy case of stability may be to sort alphanumeric data without respect to upper/lower case. The ordering relation can even take into account the time of day, the current date, the weather, the value of the U.S. Dollar as compared to the Japanese Yen, etc. In other words, it is possible to separate the sorting algorithm from the ordering relation, and allow the programmer to specify his own (possibly very complicated) ordering relation. Thus, the same package designed originally to sort an array of integers in ascending order can now sort an array of Taxpayer_Type records based on the weather in Ann Arbor and whether or not the University of Michigan varsity football team won their last home game. It is the decoupling and generalization of the ordering relation that make this generic sorting package reusable.

If the number of records to be sorted is large, then it may be more efficient to use an indirect sorting algorithm (one that sorts access types). In the last example, if we needed to sort 100 million taxpayer records, it would probably be more efficient to sort access types

that designate taxpayer records rather than the taxpayer records themselves. Of course, the access types do add extra memory overhead. The Taxpayer_Type above needs no changes. We will require the addition of an access type which designates those taxpayers:

```
type Taxpayer_Access_Type is access
Taxpayer_Type;
```

The Taxpayer_Ordering function above needs only one minor modification in its parameter list:

```
Taxpayer1, Taxpayer2 : in Taxpayer_Access_
Type) ...
```

The remainder stays the same. Thus, it is very easy to rewrite an ordering relation that compares data to one that compares access types which designate the data (and visa versa).

Since access types can designate limited types, it is possible to sort limited types which are designated by access types. For example, if one wants to sort a series of task types (a limited type) based on the time of day at which they were activated, one need only construct an array of pointers to those tasks and write an ordering relation that compares their activation times (presumably held in a global data structure).

4.3 **The Generic Sorting Package Definition.** All that is left is to specify the inputs and outputs for the SORT procedure. The array to be sorted is both an input and output parameter. Optionally, the instrumentation analysis results are output parameters. In our definition of the SORT procedure, the instrumentation analysis results are simply integers that count the number of comparisons and exchanges made by the sorting algorithm. The value −1 is returned if the instrumentation analysis counters overflow. This is possible when sorting extremely large arrays with $O(N^2)$ sorting algorithms. The overloading of the SORT procedure makes the instrumentation analysis results parameters optional. The default sorting algorithm is Quicksort, since it is generally going to perform better than any of the others. If the user wants to override this default, s/he must specify another input parameter that names the sorting algorithm to use. Alternatively, the user can override this default by using a renaming declaration which specifies a different default algorithm ([1] section 8.5 paragraph 8).

We do not need to have the user explicity pass the length of the array. All array attributes can be determined at runtime inside the body of the SORT procedure by using Ada's predefined array attributes. This is the way Ada solves the problem of determining the bounds of the array ahead of time. This may make the code construction of the SORT procedure slightly more difficult, but not less efficient. The potential hardship spared the user far outweighs a small effort in obtaining the array bounds. Also, explicity passing the array length would increase coupling.

One point that should be made now is that the array type definition, Array_Type, described above was needed so that the formal SORT procedure parameter, Sort_Array, could be specified. Ada requires that a type mark be specified for all subprogram formal parameters ([1] section 6.1 paragraph 2).

5.0 USING THE GENERIC SORTING PACKAGE

A user instantiates the generic sorting package in order to create an instance of it and thus make use of it. This paper does not attempt to explore the area of instantiations. However, the Ada comments provided in APPENDIX I should provide an insight into how one incorporates the generic sorting package into a normal application.

The instantiation process is not complicated. However, one must be sure that the proper data types and ordering relation (if any) are provided. An Ada compiler can usually flag most instantiation errors at compilation time.

Note that this generic sorting package is limited in that it is only intended to sort one-dimensional array objects. The design of this sorting package will not *directly* accommodate linked list structures or data that resides in backing store (disks, tapes, cards, etc.). It would be much easier to design separate generic sorting packages for different classes of objects such as linked lists and multi-dimensional arrays than it would be to incorporate all sorting routines for all objects in one unit. However, as mentioned above, this unit can be integrated with a merging package to provide external sorting capabilities.

6.0 MEASURING THE PERFORMANCE OF THE GENERIC SORTING PACKAGE

This generic sorting package has been implemented at LMSC in Sunnyvale, CA. Its performance has been measured by writing some benchmark drivers. The benchmarks have been run on the ROLM MSE-800 and

DEC VAX 11/780 machines using the ROLM and DEC compliers, respectively. The condensed results of these benchmarks can be found in Appendices II and III. An analysis of the benchmarks for each machine appears in the sections below.

These benchmarks only test performance of the algorithms for random data. No benchmarks have yet been written to test algorithm performance for partially ordered data or completely ordered data.

Three different tests for random data have been performed:

- Sort arrays of fixed point types.
- Sort arrays of records with three fields: CHARACTER, INTEGER, and FLOAT. The primary key is CHARACTER, the secondary key is INTEGER, and the third key is FLOAT. This ensures that for large arrays, most orderings will be based on the secondary key and occasionally on the third key.
- Sort arrays of access types that designate records. The records are identical in structure and composition to the second test (above). This will portray the difference in sorting records and access types which designate records.

A pseudo random number generator was used to generate the random data. The first key was always given as the integer 1357 to ensure identical data across more than one test (of the same type). The pseudo random number generator returned fixed point types. Hence, a direct comparison between the ROLM MSE-800 and DEC VAX 11/780 must be made carefully since the model numbers between the two machines were probably different. This would have resulted in the generation of different random numbers and thus different arrays across the two machines.

The benchmark drivers were designed so that the $O(N^2)$ algorithms would only be benched when the array size was less than or equal to 1000. Also, for each algorithm being tested, both ascending and descending ordering relations were benched.

In order to produce CPU timing data, the Sort_Pac package body was slightly modified. Initially, a *start CPU timer* routine call was made. Then the appropriate algorithm was performed. Finally a call to the same CPU timer routine was made, and the difference in times was calculated in the benchmark driver. Since the ROLM MSE-800 and DEC VAX 11/780 machines required different methods of accessing and computing the CPU time used by an algorithm, direct comparisons must again be scrutinized very carefully. In fact, the ROLM returned millisecond CPU timing, but the DEC VAX returned only centisecond CPU timing.

The benchmark drivers made use of package CALENDAR in order to produce overall elapsed timing.

6.1 Analysis of the ROLM MSE-800 Benchmarks. For arrays of size 25 or less, the simple algorithms were quite adequate. In fact, Insertion Sort and Selection Sort consistently performed equal to or better than Quicksort and Heapsort. However, as soon as the array was of size 50 or more, Quicksort and Heapsort began to outperform all simple algorithms.

Across the board, Bubble Sort was the worst algorithm to use on random data. Bubble Sort with Quick Exit performed better than Bubble Sort, but not as good as Insertion Sort or Selection Sort. Insertion Sort performed a little better than Selection Sort for arrays of size 250 or greater.

Quicksort consistently outperformed Heapsort for all arrays of size 25 or greater.

There appeared to be no significant difference in sorting in ascending or descending order for any algorithm. Occasionally, there was a significant difference in the number of comparisons or exchanges, but this was to be expected.

Sorting a simple type such as a fixed point number required much less CPU time than sorting a record where many collisions in the primary keys were expected. Also, sorting access types rather than the records themselves required less CPU time for Heapsort, Bubble Sort, Bubble Sort with Quick Exit, and Insertion Sort. Curiously, Quicksort and Selection Sort consistently required more CPU time to sort access types versus the time required by each to sort whole records. One possible reason for this may be that since both Quicksort and Selection Sort are so efficient in the number of interchanges of array components, the time required to dereference the access types for comparison operations became the most important factor and hence slowed down the speed of the entire sorting operation.

6.2 Analysis of the DEC VAX 11/780 Benchmarks. For arrays of size 25 or less, the simple algorithms were

quite adequate. In fact, Insertion Sort and Selection Sort consistently performed equal to or better than Quicksort and Heapsort. However, as soon as the array was of size 50 or more, Quicksort and Heapsort began to outperform all simple algorithms.

Across the board, Bubble Sort was the worst algorithm to use on random data. Bubble Sort with Quick Exit performed better than Bubble Sort, but not as good as Insertion Sort or Selection Sort. The Insertion Sort performed equal to Selection Sort for all arrays.

Quicksort consistently outperformed Heapsort for all arrays of size 25 or greater.

There appeared to be no significant difference in sorting in ascending or descending order for any algorithm. Occasionally, there was a significant difference in the number of comparisons or exchanges, but this was to be expected.

Sorting a simple type such as a fixed point number required much less CPU time than sorting a record where many collisions in the primary keys were expected. Also, sorting access types rather than the records themselves required less CPU time for all algorithms except Selection Sort when the array size was of size 10000 or less. Curiously, Quicksort and Heapsort consistently required more CPU time to sort access types versus the time required by each to sort whole records for arrays of size 25000 or more. It might be that the large memory requirements to store such arrays lowered execution time in fetching those address spaces. Also, Selection Sort always performed better on whole records than on access types. One possible reason for this may be that since Selection Sort is so efficient in the number of interchanges of array components, the time required to dereference the access types for comparison operations became the most important factor and hence slowed down the speed of the entire sorting operation.

7.0 FUTURE DIRECTIONS

Several enhancements to the generic sorting package can be made, all of which are completely upward compatible. These enhancements will improve the efficiency and flexibility of the sorting algorithms and allow one to sort other data types in Ada.

7.1 Improving Performance. Some simple additions to the generic sorting package can improve the overall efficiency of the sorting routines. They are described below.

7.1.1 pragma INLINE. Some utility subprograms are hidden in the package body. These subprograms are called repeatedly to perform mundane tasks such as bumping counters and exchanging array components. These subprograms can be specified by **pragma** INLINE, in which case the compiler will *inline* the code. This will eliminate the repetitive and possibly expensive nature of unneeded subprogram calls.

7.1.2 pragma SUPPRESS. Ada demands that an implementation provide range and constraint checking at run-time. After thoroughly testing a package, efficiency can be gained by instructing the compiler to eliminate such checks by using **pragma** SUPPRESS.

7.1.3 pragma OPTIMIZE. The speed of the sorting algorithms can be increased by using this pragma. OPTIMIZE can also be set so that the size of the code is decreased, but since the algorithms are not very complicated (most are fifty lines or less), it is better to use this pragma to optimize the speed of the code.

7.1.4 Making Quicksort Non-Recursive. Quicksort is the default algorithm for the package. Currently, it is highly recursive. Its implementation can be changed to make it non-recursive and thus more efficient.

7.2 Integrating a Merging Package to Sort External Files. Merging is a process of arranging records from two or more previously sorted data sets to form a data set ordered on the same attributes as the source data. To sort external files, one can write a merging package which when integrated with the generic sorting package above, would provide external sorting capabilities. One can view the merging operation as an *extended operation* of the sorting package; it augments the previous capability of the sorting package thus enhancing the reusability of both packages.

7.3 Internal Sorting for Other Data Structures. The design of this generic sorting package can be used to write other sorting packages that will sort different classes of data structures. Linked lists and multi-dimensional arrays are very common in large systems. The ideas incorporated in this package can be used to provide direct sorting capabilities for those data structures. Also, special-purpose sorting algorithms such as Meansort, radix sorting, and Shell Sort can be derived from the design of this generic sorting package.

7.4 Developing an Expert System to Select an Optimal Algorithm. Currently, a user of the generic sorting

package must either blindly accept the default algorithm, Quicksort, or know enough about the provided selection of algorithms to pick the best one for his or her application. A fairly simple expert system could be written that asks the user questions such as

— How much data needs to be sorted?

— Is the data already partially in order?

— In what order would you like the data sorted (ascending, descending, etc.)?

— What type of data is being sorted (large records, positive numbers, etc.)?

so that the *best* algorithm could be automatically selected. An improved version of this expert system could hide the details of generic instantiations, data locations, etc., so that users without technical backgrounds or Ada experience could make use of the generic sorting package.

Alternatively, this expert system could be integrated into the generic sorting package. By analyzing the Sort_Array attributes and optional user-specified *data type* parameters, it could automatically choose an optimal algorithm. Thus, the expert system could itself be considered a minimal applications selector.

7.5 Integrating a Generic Searching Package. It is only natural that after the data has been sorted, it will be accessed by some application. As stated above, sorting aids in searching. One could easily write a generic searching package that makes use of the fact that the data has been sorted by the generic sorting package.

8.0 THE COST IMPACT OF DEVELOPING GENERIC APPLICATIONS IN Ada

Generic program units, like the Sort_Pac package described above, can significantly decrease the cost of building software for various applications. The Sort_Pac package described above was researched, designed, implemented, tested, and documented in less than six months by one LMSC programmer.

The cost of implementing reusable software may be a function of how well the algorithms are known. Sorting algorithms, like math routines and certain stack and queue primitives, fall into the category of well known. These are easier to make reusable, because in a sense, the algorithms are already being reused.

The use of generic program units such as this one have been proposed for use in a major LMSC project. Upper-level project managers and the customer have realized that the use of such generic program units will significantly decrease the time and cost of developing software for large systems.

Reusable libraries and environments that contain generic program units are indeed becoming popular. It may not be long before software can be built largely by integrating various pieces of reusable software from a reusable library or environment.

Since generic program units such as the one described here can be designed to handle virtually any data type, users will rarely have to write their own routines, thus avoiding duplication of effort. The cost reductions in developing, testing, documenting, and maintaining these applications will significantly decrease. In fact, such generic program units can be integrated and layered as described in Sections 7.3 and 7.4 above so that users with different backgrounds can make use of the same code, each at their own level of Ada competence.

9.0 CONCLUSIONS

It has been shown how a generic program unit can provide reusability. Generic program units allow one to construct a wall between algorithms and data types. Furthermore, generic formal subprogram parameters can deliver generalizations of operations in algorithms. The conceptualization of the ordering relation in the Sort_Pac package can be applied to similar ordering relations in searching routines, schedulers, and lexical scanners. In fact, such tests of inequality are among the most frequently used operations.

The potential savings in constructing such generic program units is enormous. Consider the fact that most system sorting routines can only sort data in external files or simple data held in arrays. We are aware of no system sorting routines that can sort arbitrary arrays of records. However, as we have seen, it is possible to design and use a generic sorting package that sorts simple data types, and advanced data types as well.

In constructing a reusable software component, it is important to assume as little about the data as possible. If little or no assumptions about the data are specified, then a wider class of data can be accommodated. In the

Sort_Pac package described above, we purposely assumed only assignment, equality, and inequality about the data. There is nothing different about sorting integers and characters. Additionally, there is nothing different about sorting in ascending order, descending order, or a user-specified order.

Generic program units will typically be small, as far as lines of code are concerned. However, each generic program unit can potentially save hundreds of lines of code later in program development while simplifying the construction of future algorithms. While it is easy to appreciate the density of generic code, it is worthwhile to invest the time needed to scrutinize every line of generic code. A generic program unit should not be overly general. It is far better to provide many generic units for many different data structures than to provide only one which, due to its extreme generality, will provide less reusability.

Generic program units must also be highly readable and understandable. Reusability is dependent on the fact that others will be able to comprehend the requirements of the generic program unit to use it effectively.

REFERENCES

1. ANSI/MIL-STD-1815A, *Ada Programming Language*, AJPO, U.S.A., (22 January 1983).

2. Arkwright, T., Conversation regarding future directions of development, LMSC, Sunnyvale, (17 December 1984).

3. Kidwell, D., Conversation identifying specific micro issues in reusability, LMSC, Sunnyvale, (11 January 1985).

4. Knuth, D., *The Art of Computer Programming*, Volume 3: "Sorting and Searching," Addison-Wesley, Reading, MA, (1973), pp. 1-388.

5. Motzkin, D., *Communications of the ACM*, "Meansort," Volume 26, Number 4, (April 1983), pp. 250-251.

6. Sedgewick, R., *Algorithms*, Addison-Wesley, Reading, MA, (1983), pp. 91-167.

7. Tenebaum, A. and Augenstein, M., *Data Structures Using Pascal*, Prentice-Hall, Inc., Englewood Cliffs, NJ, (1981), pp. 367-423.

APPENDIX I
The Generic Sorting Package Specification

```
-- PROJECT: Generic sorting package

--   Author:              Geoff Mendal, LMSC
--   Date Written:        Thu 02 Aug 84
--   Last Revision:       Tue 08 Jan 85
--   Version:             3.1 (TUAN085)
--   Source file:         :UDD:GEOFF_DSS:GENERIC_SORT:SORT_PAC.ADA
--   Dependences:         package SYSTEM
--   Implementation:      ROLM Ada
--   Target machine:      MSE-800
--   History of changes:
--     FRUG244:           added Quicksort, normal Bubble Sort, Straight Selection Sort, and number of
--                        exchanges and comparisons made for each algorithm.
--     TUUG284:           "Key_Type" now named "Element_Type", better comments, more examples
--     TUCT094:           added many new features, cleaned up code, etc.
--     SACT204:           added Heapsort and Insertion Sort
--     THOV014:           now a generic package with overloaded procedures
--     TUAN085:           changes to conform to the micro reuse issues paper

--   Sort_Pac is a generic sorting package. The procedure SORT will sort a one dimensional array of
--   any component type that supports assignment, equality, and inequality (private types) indexed by
--   discrete type components. The default sort strategy is ascending order but may be overridden by
--   the user. The default sort algorithm, Quicksort, may also be overridden.

--   Note that the component type can be a record type. SORT is not restricted to simple data types. If
--   records are to be sorted, then the formal generic subprogram parameter "<" must be specified
--   with a selector function, e.g., a function provided as an actual generic subprogram parameter at
--   instantiation (see example #2 below).

--   Also note that the component type can be an access type (which can point to other objects,
--   improving sort efficiency). If access types are to be sorted, then the formal generic subprogram
--   parameter "<" must be specified with a selector function (see example #3 below). Since access
--   types can be sorted, the SORT routine below can be used to sort limited types (designated by
--   access types).

--   The number of comparisons and exchanges made to sort the array can be returned. These
--   numbers should give some indication on how much work was actually performed by the sorting
--   algorithms. These numbers can also be used to compare the relative efficiency of
--   the sorting algorithms.
```

```
-  This package can be used to sort data on external devices. The user should use this package to
-  sort a subset of the external data, then use a merge operation on all sorted subsets. For example,
-  if the system can only hold 1000 components in RAM, but you need to sort 3000 components, bring
-  in components #1-1000 and sort them using this routine, and then write them to a file. Next do the
-  same with components #1001-2000, and finally with components #2001-3000. Now merge the three
-  sorted files using a merging package.

with SYSTEM; - predefined package SYSTEM

generic
   type Component_Type is private; - type of the data components
   type Index_Type is (<>); - type of array index

-  the following generic formal type is required due to Ada's strong typing requirements. The SORT
-  procedure cannot handle anonymous array types. This type will match any unconstrained array
-  type definition (so that array slices can be sorted too—see example #3 below).

   type Array_Type is array (Index_Type range <>) of Component_Type;

-  the following formal subprogram parameter defaults to the predefined "<" operator which will
-  sort one-dimensional arrays of Component _Type in ascending order (by default). If composite or
-  access types are to be sorted, a selector function must be specified.

   with function "<" (Left,Right : in Component_Type) return BOOLEAN is <>;
package Sort_Pac is
-  users can specify the type of sorting algorithms they want by specifying an enumeration literal
-  from the type below. The default algorithm, Quicksort, generally provides the lowest number of
-  comparisons and exchanges.

-  One note about stability of the algorithms: only the Bubble Sort and Insertion Sort are stable
-  algorithms. Thus, they are the only algorithms that preserve the ordering of equal components
-  without use of a selector function. In all cases, a selector function may be specified to introduce
-  stability into the sorting algorithms (see example #3 below).

   type Sort_Algorithm_Type is (Quicksort, Bubble_Sort,
      Bubble_Sort_with_Quick_Exit, Selection_Sort, Heapsort,
      Insertion_Sort);

-  Quicksort:              Order n*log(n). Is most efficient when used with large, unsorted arrays.
-                          Recursive nature may introduce significant overhead for very large arrays.
-                          This is the default algorithm. Instable algorithm.
-  Bubble_Sort:            Order n**2. Is most efficient when used with small arrays that are almost
-                          already sorted. No recursion. Brute force. Low memory requirements. Stable
-                          algorithm.
-  Bubble_Sort_with        Order n**2. Is most efficient when used with small arrays that are almost
-  _Quick_Exit:            already sorted.  No recursion.Same as bubble sort above except brute force is
-                          limited. Instable algorithm.
-  Selection Sort:         Order n**2. Is most efficient when used with small arrays in which the
-                          Component Type is a record type. No recursion. Brute force. Number of
-                          exchanges is lower than Quicksort. Usually better than Bubble Sort. Instable
-                          algorithm.
-  Heapsort:               Order n*log(n). Is most efficient when used with large, unsorted arrays. No
```

```
-                       recursion. Very low memory requirements. Number of comparisons is usually
-                       two times greater than Quicksort. Number of exchanges is usually four times
-                       greater than Quicksort. Instable algorithm.
-   Insertion_Sort:     Order n**2. Is most efficient when used with small arrays that are almost
-                       already sorted. No recursion. Brute force. Usually better than Bubble_Sort.
-                       Stable algorithm.

-   The following type declaration should be used to specify the instrumentation analysis data that
-   can be returned by the SORT procedure below. -1 is only returned if an overflow in calculations
-   has occurred. The SORT procedure will not terminate if an overflow in instrumentation analysis
-   data calculations occurs.

    type Instrumentation_Analysis_Type is range -1 .. SYSTEM.MAX_INT;

-   the following procedure will sort a one dimensional array of components. It can sort in
-   ascending/descending order or any user-defined order. It can sort components of any type that
-   support equality, inequality, and assignment (private types). The array indices can be of any
-   discrete type. The number of comparisons and exchanges can also be returned.

      procedure SORT(
        Sort_Array                    :  in out Array_Type;
        Number_of_Comparisons,
        Number_of_Exchanges           :     out Instrumentation_Analysis_Type;
        Sort_Algorithm                :  in     Sort_Algorithm_Type := Quicksort);

-   the following overloading of procedure SORT should be specified
-   when no instrumentation analysis data are required.

      procedure SORT(                 -
        Sort_Array                    :  in out Array_Type;
        Sort_Algorithm                :  in     Sort_Algorithm_Type := Quicksort);
end Sort_Pac;

-   Example uses/instantiations:
-   EXAMPLE #1: Sorting an array of floating point numbers
-   with Sort_Pac;
-   procedure Main is
-     ...
-   subtype My_Component_Type is FLOAT range 0.0 .. 100_000.00;
-   type My_Index_Type is (Sun,Mon,Tue,Wed,Thu,Fri,Sat);
-   type My_Array_Type is array (My_Index_Type range <>) of My_Component_Type;
-     ...
-   package Ascending_Sort is new Sort_Pac(
-     Component_Type               => My_Component_Type,
-     Index_Type                   => My_Index_Type,
-     Array_Type                   => My_Array_Type);
-     ...
-   package Descending_Sort is new Sort_Pac(
-     Component_Type               => My_Component_Type,
-     Index_Type                   => My_Index_Type,
-     Array_Type                   => My_Array_Type,
-     "<"                          => ">");
```

```
-  My_Array                          : My_Array_Type (Mon .. Fri);
-  Number_of_Comparisons,
-  Number_of_Exchanges               : Descending_Sort.Instrumentation_Analysis_Type;
-   ...
-  Ascending_Sort.SORT(My_ Array);
-   ...
-  Descending_Sort.SORT(
-    Sort_Array                      => My_Array,
-    Number_of_Comparisons           => Number_of_Comparisons,
-    Number_of_Exchanges             => Number_of_Exchanges,
-    Sort_Algorithm                  => Descending_Sort.Bubble_Sort);
-   ...
-  end Main;
```

```
-  EXAMPLE #2: Sorting an array of records based on a key field
-  with Sort_Pac;
-  procedure Main is
-   ...
-  type My_Component_Type is
-    record
-      Field1 : INTEGER;
-      Field2 : FLOAT;
-      Field3 : CHARACTER;
-    end record;
-  subtype My_Index_Type is INTEGER range -10 ... 10;
-  type My_Array_Type is array (My_Index_Type range < >) of My_Component_Type;
-   ...
-  My_Array : My_Array_Type(-10 .. 10);
-   ...
-  function Ascending_Selection_on_Field1(Left,Right : in My_Component_Type)
-  return BOOLEAN is
-  begin
-    return Left.Field1 < Right.Field1;
-  end Ascending_Selection_on_Field1;
-   ...
-  function Descending_Selection_on_Field3(Left,Right : in My_Component_Type)
-  return BOOLEAN is
-  begin
-    return Left.Field3 > Right.Field3;
-  end Descending_Selection_on_Field3;
-   ...
-  package Ascending_Sort_on_Field1 is new Sort_Pac(
-    Component_Type                  => My_Component_Type,
-    Index_Type                      => My_Index_Type,
-    Array_Type                      => My_Array_Type)
-    "<"                             => Ascending_Selection_on_Field1);
-  package Descending_Sort_on_Field 3 is new Sort_Pac(
-    My_Component_Type,
-    My_Index_Type, My_Array_Type,
-    Descending_Selection_on_Field 3);
-   ...
-  Ascending_Sort_on_Field1.SORT(My_Array);
```

```
-   ...
-   Descending_Sort_on_Field3.SORT(
-     Sort_Array                    => My_Array,
-     Sort_Algorithm                => Descending_Sort_on_Field3.Selection_Sort);
-   ...
-   end Main;
```

```
-   EXAMPLE #3: Sorting an array slice of access types that point to records.
-   with Sort_Pac;
-   procedure Main is
-    ...
-   type Taxpayer_Type is
-     record
-       Name                        : STRING(1 .. 40);
-       Age                         : NATURAL;
-       ID_Number                   : POSITIVE; -- social security number
-     end record;
-   type Taxpayer_Access_Type is access Taxpayer_Type;
-   type My_Index_Type is range 1 .. 1_000_000;
-   type My_Array_Type is array(My_Index_Type range <>) of
-   Taxpayer_Access_Type;
-    ...
-   My_Array : My_Array_Type(1 .. 1_000_000);
-    ...
-   function Ascending_Taxpayers(Left,Right : in Taxpayer_Access_Type) return BOOLEAN is
-   begin
-     return   (Left.Name < Right.Name) or
-              ((Left.Name = Right.Name) and (Left.ID_Number <
-              Right.ID_Number));
-   end Ascending_Taxpayers;
-    ...
-   package Ascending_Taxpayer_Sort is new Sort_Pac(
-     Taxpayer_Access_Type,My_Index_Type,My_Array_Type,Ascending_Taxpayers);
-    ...
-   Ascending_Taxpayer_Sort.SORT(My_Array(100..1_000));
-    ...
-   end Main;
```

APPENDIX II

ROLM MSE-800 Benchmarks

NOTE: The data provided below has been averaged from a combination of benchmarks run for each of the three tests. In order to condense the amount of material, only arrays of size 1, 10, 50, 100, 500, 1000, 10000, and 100000 are provided. The results of the ascending and descending tests have also been averaged.

Only the individual algorithm statistics are being presented.

Fixed Point Types

Algorithm	Array Size	Comparisons	Exchanges	CPU Time (in ms)
Heap	1	0	0	1
Quick	1	0	0	0
Bubble	1	0	0	0
Bubble w/Q	1	0	0	0
Selection	1	0	0	1
Insertion	1	0	0	1
Heap	10	42	46	7
Quick	10	15	8	6
Bubble	10	45	23	8
Bubble w/Q	10	30	23	7
Selection	10	45	7	4
Insertion	10	30	32	4
Heap	50	427	357	58
Quick	50	221	65	47
Bubble	50	1225	613	198
Bubble w/Q	50	659	613	180
Selection	50	1225	46	80
Insertion	50	659	662	70
Heap	100	1049	814	129
Quick	100	433	158	102
Bubble	100	4950	2475	766
Bubble w/Q	100	2571	2475	700
Selection	100	4950	95	297
Insertion	100	2571	2574	257
Heap	500	7595	5266	897
Quick	500	3607	1043	690
Bubble	500	124750	62375	19312
Bubble w/Q	500	62869	62375	17113
Selection	500	124750	492	7271
Insertion	500	62869	62874	6225
Heap	1000	17203	11552	1991
Quick	1000	7336	2349	1478
Bubble	1000	499500	249747	81232
Bubble w/Q	1000	250740	249747	70277
Selection	1000	499500	994	29101
Insertion	1000	250740	250746	24827
Heap	10000	239239	149428	26960
Quick	10000	105644	30897	19188
Heap	100000	3057951	1827542	346091
Quick	100000	1344159	405503	241666

Records

Algorithm	Array Size	Comparisons	Exchanges	CPU Time (in ms)
Heap	1	0	0	1
Quick	1	0	0	1
Bubble	1	0	0	0
Bubble w/Q	1	0	0	0
Selection	1	0	0	1
Insertion	1	0	0	1
Heap	10	42	46	10
Quick	10	20	9	8
Bubble	10	45	23	11
Bubble w/Q	10	30	23	10
Selection	10	45	9	5
Insertion	10	30	32	5
Heap	50	430	358	86
Quick	50	212	60	59
Bubble	50	1225	613	269
Bubble w/Q	50	659	613	247
Selection	50	1225	45	123
Insertion	50	659	662	95
Heap	100	1056	817	216
Quick	100	544	153	141
Bubble	100	4950	2475	1057
Bubble w/Q	100	2571	2475	988
Selection	100	4950	92	528
Insertion	100	2571	2574	364
Heap	500	7610	5293	1566
Quick	500	2959	1064	866
Bubble	500	124750	62375	27215
Bubble w/Q	500	62867	62375	25141
Selection	500	124750	496	13557
Insertion	500	62867	62874	9048
Heap	1000	17260	11604	3543
Quick	1000	7254	2333	1985
Bubble	1000	499500	249747	109521
Bubble w/Q	1000	250738	249747	100408
Selection	1000	499500	994	53402
Insertion	1000	250738	250746	36232
Heap	10000	239308	149542	46919
Quick	10000	102837	31037	25913
Heap	100000	3059612	1828669	531034
Quick	100000	1197858	410224	323255

Access Types that Designate Records

Algorithm	Array Size	Comparisons	Exchanges	CPU Time (in ms)
Heap	1	0	0	1
Quick	1	0	0	1
Bubble	1	0	0	0
Bubble w/Q	1	0	0	0
Selection	1	0	0	0
Insertion	1	0	0	1
Heap	10	42	46	9
Quick	10	20	9	9
Bubble	10	45	23	10
Bubble w/Q	10	30	23	9
Selection	10	45	9	6
Insertion	10	30	32	5
Heap	50	430	358	78
Quick	50	212	60	64
Bubble	50	1225	613	246
Bubble w/Q	50	659	613	208
Selection	50	1225	45	134
Insertion	50	659	662	86
Heap	100	1056	817	190
Quick	100	544	153	151
Bubble	100	4950	2475	1062
Bubble w/Q	100	2571	2475	900
Selection	100	4950	92	533
Insertion	100	2571	2574	337
Heap	500	7610	5293	1304
Quick	500	2959	1064	913
Bubble	500	124750	62375	25545
Bubble w/Q	500	62867	62375	20572
Selection	500	124750	496	14174
Insertion	500	62867	62874	8060
Heap	1000	17260	11604	3086
Quick	1000	7254	2333	2055
Bubble	1000	499500	249747	105419
Bubble w/Q	1000	250738	249747	85103
Selection	1000	499500	994	60128
Insertion	1000	250738	250746	33311
Heap	10000	239308	149542	41163
Quick	10000	102837	31037	27708
Heap	100000	3059612	1828669	519139
Quick	100000	1197858	410224	340098

APPENDIX III

DEC VAX 11/780 Benchmarks

NOTE: The data provided below has been averaged from a combination of benchmarks run for each of the three tests. In order to condense the amount of material, only arrays of size 1, 10, 50, 100, 500, 1000, 10000, and 100000 are provided. The results of the ascending and descending tests have also been averaged.

Only the individual algorithm statistics are being presented.

Fixed Point Types

Algorithm	Array Size	Comparisons	Exchanges	CPU Time (in ms)
Heap	1	0	0	3
Quick	1	0	0	0
Bubble	1	0	0	0
Bubble w/Q	1	0	0	3
Selection	1	0	0	0
Insertion	1	0	0	3
Heap	10	42	46	5
Quick	10	15	8	8
Bubble	10	45	23	3
Bubble w/Q	10	30	23	5
Selection	10	45	7	3
Insertion	10	30	32	5
Heap	50	427	357	45
Quick	50	221	65	30
Bubble	50	1225	613	95
Bubble w/Q	50	659	613	68
Selection	50	1225	46	63
Insertion	50	659	662	63
Heap	100	1049	814	110
Quick	100	433	158	63
Bubble	100	4950	2475	368
Bubble w/Q	100	2571	2475	270
Selection	100	4950	95	238
Insertion	100	2571	2574	248
Heap	500	7595	5266	743
Quick	500	3607	1043	410
Bubble	500	124750	62375	9473
Bubble w/Q	500	62869	62375	6713
Selection	500	124750	492	5930
Insertion	500	62869	62874	5930
Heap	1000	17203	11552	1633
Quick	1000	7336	2349	858
Bubble	1000	499500	249747	37148
Bubble w/Q	1000	250740	249747	26785
Selection	1000	499500	994	23738
Insertion	1000	250740	250746	23660
Heap	10000	239239	149428	22580
Quick	10000	105644	30897	11098
Heap	100000	3057949	1827453	284660
Quick	100000	1313317	406018	134773

Records

Algorithm	Array Size	Comparisons	Exchanges	CPU Time (in ms)
Heap	1	0	0	0
Quick	1	0	0	3
Bubble	1	0	0	5
Bubble w/Q	1	0	0	0
Selection	1	0	0	0
Insertion	1	0	0	0
Heap	10	42	46	10
Quick	10	20	9	10
Bubble	10	45	23	10
Bubble w/Q	10	30	23	10
Selection	10	45	9	5
Insertion	10	30	32	8
Heap	50	430	358	85
Quick	50	212	60	68
Bubble	50	1225	613	520
Bubble w/Q	50	659	613	223
Selection	50	1225	45	133
Insertion	50	659	662	110
Heap	100	1056	817	215
Quick	100	544	153	150
Bubble	100	4950	2475	1073
Bubble w/Q	100	2571	2475	1775
Selection	100	4950	92	500
Insertion	100	2571	2574	425
Heap	500	7616	5296	1433
Quick	500	2962	1065	858
Bubble	500	124750	62375	27433
Bubble w/Q	500	62867	62375	21865
Selection	500	124750	495	12410
Insertion	500	62867	62874	10198
Heap	1000	17252	11604	3203
Quick	1000	7176	2330	1895
Bubble	1000	499500	249747	107823
Bubble w/Q	1000	250738	249747	87565
Selection	1000	499500	993	49698
Insertion	1000	250738	250746	40795
Heap	10000	239262	149484	43835
Quick	10000	100473	31129	25025
Heap	100000	3059253	1828439	561860
Quick	100000	1236385	408755	310923

Access Types that Designate Records

Algorithm	Array Size	Comparisons	Exchanges	CPU Time (in ms)
Heap	1	0	0	3
Quick	1	0	0	0
Bubble	1	0	0	3
Bubble w/Q	1	0	0	3
Selection	1	0	0	0
Insertion	1	0	0	3
Heap	10	42	46	8
Quick	10	20	9	10
Bubble	10	45	23	3
Bubble w/Q	10	30	23	5
Selection	10	45	9	8
Insertion	10	30	32	8
Heap	50	430	358	75
Quick	50	212	60	53
Bubble	50	1225	613	155
Bubble w/Q	50	659	613	113
Selection	50	1225	45	130
Insertion	50	659	662	98
Heap	100	1056	817	180
Quick	100	544	153	123
Bubble	100	4950	2475	645
Bubble w/Q	100	2571	2475	433
Selection	100	4950	92	530
Insertion	100	2571	2574	370
Heap	500	7616	5296	1245
Quick	500	2962	1065	723
Bubble	500	124750	62375	17148
Bubble w/Q	500	62867	62375	10765
Selection	500	124750	495	13108
Insertion	500	62867	62874	9103
Heap	1000	17252	11603	2810
Quick	1000	7176	2330	1640
Bubble	1000	499500	249747	68445
Bubble w/Q	1000	250744	249753	43365
Selection	1000	499500	993	52955
Insertion	1000	250738	250746	36538
Heap	10000	239295	149482	40090
Quick	10000	100473	31129	22023
Heap	100000	3059238	1828434	786968
Quick	100000	1236385	408755	345570

NRL Code 2490-0035

RESUME

Geoffrey O. Mendal

RECENT OPERATIONAL EXPERIENCE (1981 - 1985):

Lockheed Missiles and Space Company (1984-1985)

Wrote the content for an extensive Ada questions and answers data base used to test the knowledge of Ada programmers.

Researched, wrote, tested, documented, and benchmarked a production generic sorting package in Ada.

Wrote a prototype generic searching package in Ada.

Wrote a time/date conversion package in Ada.

Authored five research papers for an intensive study of Ada under contract.

University of Michigan Computing Center (1981-1984)

Designed, implemented, tested, benchmarked, and documented an on-line documentation retrieval system. This system is currently used by the University of Michigan students, faculty, and researchers as the sole source of on-line explanations for the general computer system.

University of Michigan Computer Science Department (1981-1983)

Helped to teach undergraduate core Computer Science Department courses. Led discussion sections, graded assignments, and consulted with students.

Independent Ada Research (1984-1985)

Heading a major Ada software development project outside of Lockheed Missiles and Space Company, with the goal of developing a more profound knowledge of Ada software engineering.

ACADEMIC WORK (1979-1983):

BS (University of Michigan) in Computer Science; GPA: 3.26/4.0

CONSULTING WORK (1981-1984):

University of Michigan Computing Center

Provided consulting services on the general computer system for students, faculty, and researchers.

COORDINATES:

ockheed Missiles and Space Company
1111 Lockheed Way, Department 62-J1, Building 563
Sunnyvale, CA 94088
408-743-1191

Micro Issues in Reuse From a Real Project

Geoffrey O. Mendal
Ada* Technology Support Lab
Lockheed Missiles and Space Company, Inc.
Sunnyvale, CA

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

Micro Issues in Reuse From a Real Project

- Overview
- An Introduction to Sorting
- Requirements For a Generic Sorting Package
- Designing the Generic Sorting Package Specification
- Using the Generic Sorting Package
- Measuring the Performance of the Generic Sorting Package
- Future Directions
- The Cost Impact of Developing Generic Applications in Ada
- Conclusions

Overview

- Sorting is a functional abstraction
- Sorting packages can be elementary expert systems
- The domain of a sorting package should be limited
- Sorting is a frequently used operation in large software systems

An Introduction to Sorting

- What is Sorting? Why Sort?
- A Formal *Definition of Sorting*
- Internal Sorting
- External Sorting
- Sorting Factors
- An Overview of Common Internal Sorting Algorithms
- Summary and History

What is Sorting? Why Sort?

- Sorting is a process of arranging objects from one or more data sets to form a data set ordered on one or more attributes of the data
- Sorting can increase the speed and reduce the complexity of other algorithms that use the data
- Sorting is an interesting topic:
  - ingenious algorithms
  - fascinating unsolved problems
  - a case study on how to attack computer problems in general
  - principles of data structure manipulation
  - design and analysis of algorithms

A Formal Definition of Sorting

- Quotes from Knuth, *Sorting and Searching*:

"But you can't look up all those license numbers in time," Drake objected. "We don't have to, Paul. We merely arrange a list and look for duplications."

Perry Mason (The Case of the Angry Mourner, 1951)

"Treesort" Computer — With this new 'computer-approach' to nature study you can quickly identify over 260 different trees of U.S., Alaska, and Canada, even palms, desert trees, and other exotics. To sort, you simply insert the needle.

Catalog of Edmund Scientific Company (1964)

A Formal Definition of Sorting

- Given N records: R1, R2, ..., RN
- The entire collection will be called a file
- Each record has a key which governs the sorting process
- An ordering relation "<" is specified on the keys so that for any of these values, a, b, c:
  - exactly one of the possibilities a < b, a = b, b < a is true
  - if a < b and b < c then a < c
- Goal: determine a permutation p(1) p(2) ... p(N) of records which puts the keys in non-decreasing order Kp(1) < = ... < = Kp(N)

A Formal Definition of Sorting

- Stability
  - A further requirement: records with equal keys retain their original relative order, i.e., if $p(i) < p(j)$ whenever $Kp(i) = Kp(j)$ and $i < j$
  - Examples:
    - sort without respect to upper/lower case
    - sort an alphabetized class list on the grades of the last exam, keeping students with *equal grades in alphabetical order as before*
- Time
  - Big Oh notation
  - Good general-purpose algorithm: O(NlogN)
  - Simple algorithm: $O(N^2)$

Internal Sorting

- Assumes that the records are kept in random access memory (RAM)
- Allows easy record access
- Allows more flexibility, structure, and access of data
- Sample problem: write a program which, if necessary, rearranges five numbers in memory locations, M, M + 1, ... M + 4

Internal Sorting

- Possible solutions
  - Insertion Sort
  - Exchange Sort
  - Selection Sort
  - Enumeration Sort
  - Special-Purpose Sort
  - New, Super Sorting Technique

External Sorting

- Assumes that there are more records than can be held in RAM simultaneously
- Requires sequential record access
- Usually requires one to cope with rather stringent accessing constraints
- Data structures must be arranged so that slow peripheral memory devices can quickly cope with the requirements of the sorting algorithm
- Common solution: internal sorting on subfiles followed by external merging

Sorting Factors

- Most important factor: file size
- Next most important factor: extra memory used by the algorithm
- Special case files:
  - already or partially sorted
  - many equal keys
- Stability of algorithms
- Accessing records
  - directly
  - indirectly

An Overview of Common Internal Sorting Algorithms

- Straight Selection Sort
- Insertion Sort
- Bubble Sort
- Bubble Sort with Quick Exit
- Quicksort
- Heapsort

Straight Selection Sort

- Instable algorithm
- Finds the smallest/largest record and exchanges it with the record in the first position
- Process continues until the entire file is sorted
- O(N2)
- Use for large records and small keys. Use only for files less than 1000 records

Insertion Sort

- Stable algorithm
- *Bridge Player* method
- O(N2) for random data
- O(N) for fully ordered (non-random) data

Bubble Sort

- Stable algorithm
- Exchange adjacent records, if necessary
- $O(N^2)$
- Generally the worst algorithm for random data

Bubble Sort with Quick Exit

- Stable algorithm
- Same as Bubble Sort with modification: when no exchanges are required, terminate
- $O(N^2)$ for random data
- O(N) for fully ordered (non-random) data

Quicksort

- Invented by C.A.R. Hoare in 1960
- Good points
  - in-place algorithm
  - O(NlogN)
  - extremely short inner loop
- Bad points
  - highly recursive
  - worst case is $O(N^2)$
  - instable algorithm
  - fragile
- Cousins
  - handle worst cases
  - *remove recursion*
  - names of cousins
    - Meansort
    - Concurrent Quicksort

Heapsort

- Instable algorithm
- Method: file is considered as a binary tree
- O(NlogN)
- Not recursive

Summary and History

- Sorting is a process which rearranges a file of records so that the keys are in order
- Orderly arrangement is useful because it:
  - allows for efficient processing of multiple files sorted on the same key
  - leads to efficient retrieval algorithms
  - makes computer output look more authentic
- All algorithms deserve recognition
- Sorting is not the whole story
  - handling data structures
  - dealing with external memories
  - analyzing algorithms
  - discovering new algorithms

Summary and History

- Origin: 19th century, Herman Hollerith
- Evidence of a sorting routine as the first program ever written for a stored program computer
- History of sorting closely associated with many *firsts* in computing:
  - data processing machines
  - stored programs
  - software
  - buffering methods
  - analysis of algorithms
  - computational complexity

Requirements For a Generic Sorting Package

- Sort a one-dimensional array using the most efficient algorithm known (determined by an expert system)
- Process an array of arbitrary length
- Allow capability to sort data that cannot fit in RAM
- Sort in any arbitrarily defined order
- Sort practically any array component type
- Provide a selection of well known internal sorting algorithms
- Provide an option for retrieval of instrumentation analysis results
- Allow array index to be of any discrete type
- Provide sensible (most frequently used) defaults for the algorithms and a predefined (default) ordering relation
- Adhere to the highest defined level of standards for readability and understandability in the package specification and body to facilitate reuse

Designing the Generic Sorting Package

- The Generic Formal Part
- The Generic Formal Type Parameters
  - The Component Type Parameter
  - The Index Type Parameter
  - The Array Type Parameter
- The Generic Formal Subprogram Parameter
- The Generic Sorting Package Specification

The Generic Formal Part

```
with SYSTEM;
generic
   type Component_Type is private;
   type Index_Type is (< >);
   type Array_Type is array(Index_Type range < >) of
      Component_Type;

   with function "<"(Left, Right : in Component_Type)
      return BOOLEAN is < >;
package Sort_Pac is
   .
   .
   .
end Sort_Pac;
```

The Generic Formal Subprogram Parameter

- Example: Sorting elephants

```
type Elephant_Type is
   record
      Trunk_Length,
      Height,
      Weight        : POSITIVE;
      Name          : STRING(1..40);
   end record;

function Elephant_Ordering(
   Elephant1, Elephant2 : in Elephant_Type)
      return BOOLEAN is
begin
   return Elephant1.Trunk_Length < Elephant2.Trunk Length;
end Elephant_Ordering;
```

The Generic Formal Subprogram Parameter

- Example: Sorting taxpayers, solution #1
- Direct sorting:

```
type Taxpayer_Type is
      record
         Name              : STRING(1..40);
         Age               : NATURAL;
         ID_Number         : POSITIVE;
      end record;

function Taxpayer_Ordering(
   Taxpayer1, Taxpayer2 : in Taxpayer_Type)
      return BOOLEAN is
begin
   return Taxpayer1.Name < Taxpayer2.Name;
end Taxpayer_Ordering;
```

The Generic Formal Subprogram Parameter

- Example: Sorting taxpayers, solution #2
- Direct sorting with stability requirements:

```
function Taxpayer Ordering(
   Taxpayer1, Taxpayer2 : in Taxpayer__Type)
      return BOOLEAN is
begin
   return (Taxpayer1.Name < Taxpayer2.Name) or
   ((Taxpayer1.Name = Taxpayer2.Name) and
    (Taxpayer1.ID__Number < Taxpayer2.ID__Number));
end Taxpayer__Ordering;
```

The Generic Formal Subprogram Parameter

- Example: Sorting taxpayers, solution #3
- Indirect sorting with stability requirements:

```
type Taxpayer_Access_Type is access Taxpayer_Type;

function Taxpayer_Ordering(
   Taxpayer1, Taxpayer2 : in Taxpayer_Access_Type) ...
      .
      .  -- same as before
      .
end Taxpayer_Ordering;
```

The Generic Formal Subprogram Parameter

- Use of "<" in the package body:

```
-- compare consecutive array components and exchange them
-- if necessary

If Sort_Array(I + 1) < Sort_Array(I) then
   Exchange_Array_Elements(Sort_Array(I..I + 1));
   ...
end if;
```

- The < operator above, by overload resolution, is the one provided during instantiation (or defaults)

- Complex ordering relations are essentially free, apart from their execution costs. There is no extra code required in the package body. Likewise, the ability to sort composite and access types is also essentially free

The Generic Sorting Package Specification

```
.
.
.
package Sort_Pac is
   type Sort_Algorithm_Type is (Quicksort, Heapsort,
      Bubble_Sort, Bubble_Sort_with_Quick_Exit,
      Selection_Sort, Insertion_Sort);

   type Instrumentation_Analysis_Type is range -1 ..
      SYSTEM.MAX_INT;

   procedure SORT(
     Sort_Array                  : in out  Array_Type;
     Number_of_Comparisons,
     Number_of_Exchanges         :    out  Instrumentation_Analysis_Type;
     Sort_Algorithm              : in      Sort_Algorithm_Type
                                           := Quicksort);

   procedure SORT(
     Sort_Array                  : in out  Array_Type;
     Sort_Algorithm              : in      Sort_Algorithm_Type
                                           := Quicksort);
end Sort_Pac;
```

Using the Generic Sorting Package

- Simple instantiation and use:

```
with Sort_Pac;
procedure Main is
   My_Array_Type is array(POSITIVE range < >) of CHARACTER;
   My_Array : My_Array_Type(1..100);

   package My_Sort_Pac is new Sort_Pac(
    Component_Type  = > CHARACTER,
    Index_Type      = > POSITIVE,
    Array_Type      = > My_Array_Type);
begin
   -- enter array components here
      ...
   -- sort the array in ascending order using Quicksort

   My_Sort_Pac.SORT(My_Array);
      ...
end Main;
```

Using the Generic Sorting Package

- Complex instantiation and use:

```
with Sort__Pac;
procedure Main is
   type Taxpayer__Type is
        record
           Name          : STRING(1..40);
           Age           : NATURAL
           ID__Number    : POSITIVE;
        end record;

   type Taxpayer__Access__Type is access Taxpayer__Type;

   type Taxpayer__Index__Type is range -1000 .. 1000;

   type Taxpayer__Array__Type is array
     (Taxpayer__Index__Type range < >) of Taxpayer__Access__Type;
```

Using the Generic Sorting Package (CONT'D)

```
Taxpayer_Array : Taxpayer_Array_Type(-500..250);

function Taxpayer_Ordering(
   Taxpayer1, Taxpayer2 : in Taxpayer_Access_Type)
   return BOOLEAN is
begin
   return (Taxpayer1.Name < Taxpayer2.Name) or
          ((Taxpayer1.Name = Taxpayer2.Name) and
          (Taxpayer1.ID_Number < Taxpayer2.ID_Number));
end Taxpayer_Ordering;

package Taxpayer_Sort_Pac is new Sort_Pac(
     Component_Type    => Taxpayer_Access_Type,
     Index_Type        => Taxpayer_Index_Type,
     Array_Type        => Taxpayer_Array_Type,
     "<"               => Taxpayer_Ordering);
```

Using the Generic Sorting Package (CONT'D)

```
begin  -- start of the Main procedure
   .
   .
   .
   declare
      Taxpayer_Comparisons,
      Taxpayer_Exchanges :
         Taxpayer_Sort_Pac.Instrumentation_Analysis_Type;
   begin
      -- enter array components here
         ...
      -- sort an array slice using Heapsort and request
      -- instrumentation analysis results

      Taxpayer_Sort_Pac.SORT(
           Sort_Array                 => Taxpayer_Array(-100..200),
           Number_of_Comparisons      => Taxpayer_Comparisons,
           Number_of_Exchanges        => Taxpayer_Exchanges,
           Sort_Algorithm             => Taxpayer_Sort_Pac.Heapsort);
      ...
   end;              -- local declare block statement
   .
   .
   .
end Main;
```

Using the Generic Sorting Package

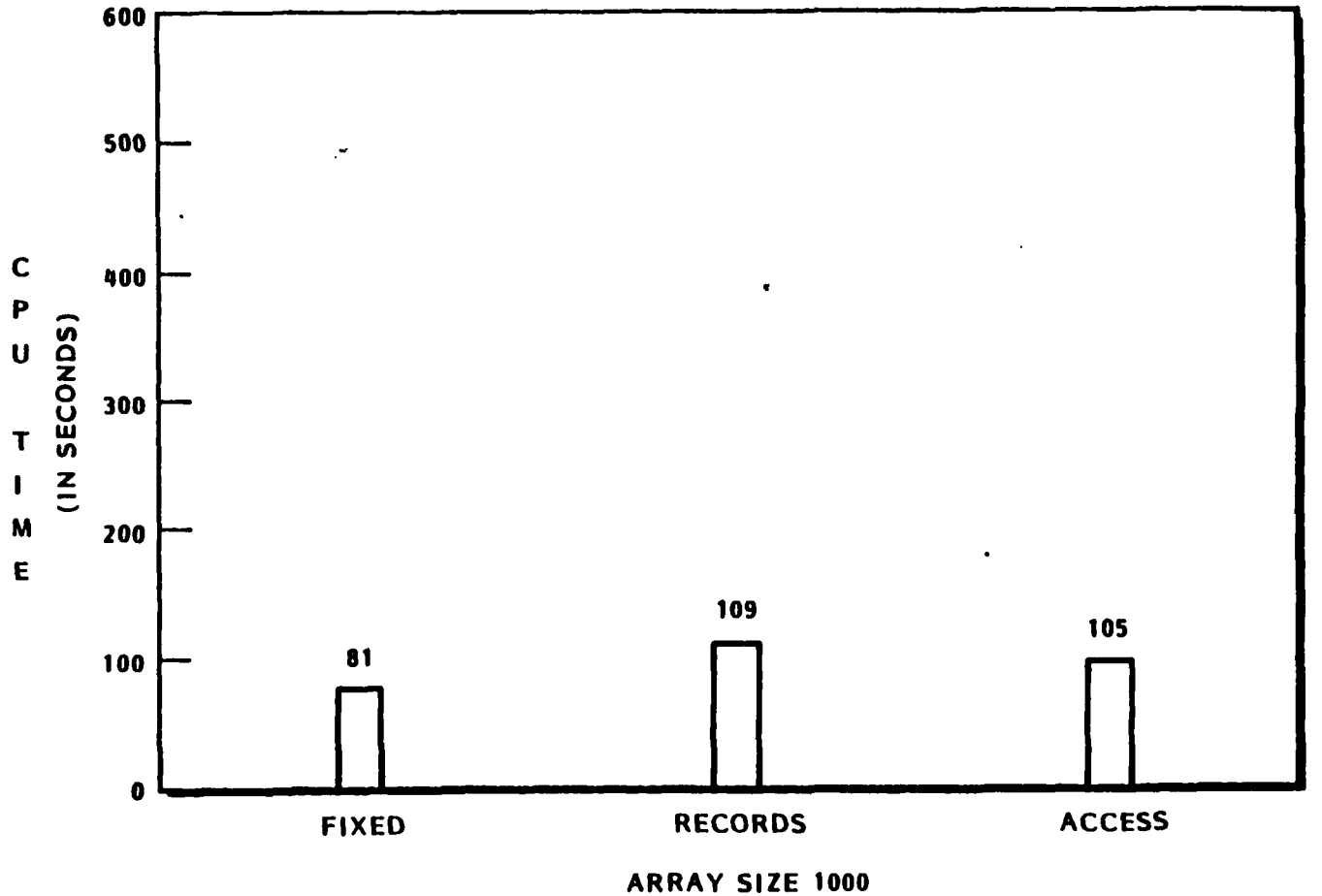
- Instantiations are easy to code
- SORT procedure calls are easy to code
- Ada compiler catches most instantiation errors at compilation time
- Complex applications are as easy to code as simple applications
- Testing is easy
  - — Nine sets of test data have been written to test the generic sorting package
  - — Three benchmark drivers have also been written and executed on two different machines:
    - ROLM MSE-800
    - DEC VAX 11/780

Measuring the Performance of the Generic Sorting Package

- Testing random data
  - — fixed point types
  - — records
  - — access types that designate records
- Analysis of the ROLM MSE-800 Benchmarks.
- Analysis of the DEC VAX 11/780 Benchmarks
- *Almost Everything You Know is Wrong*

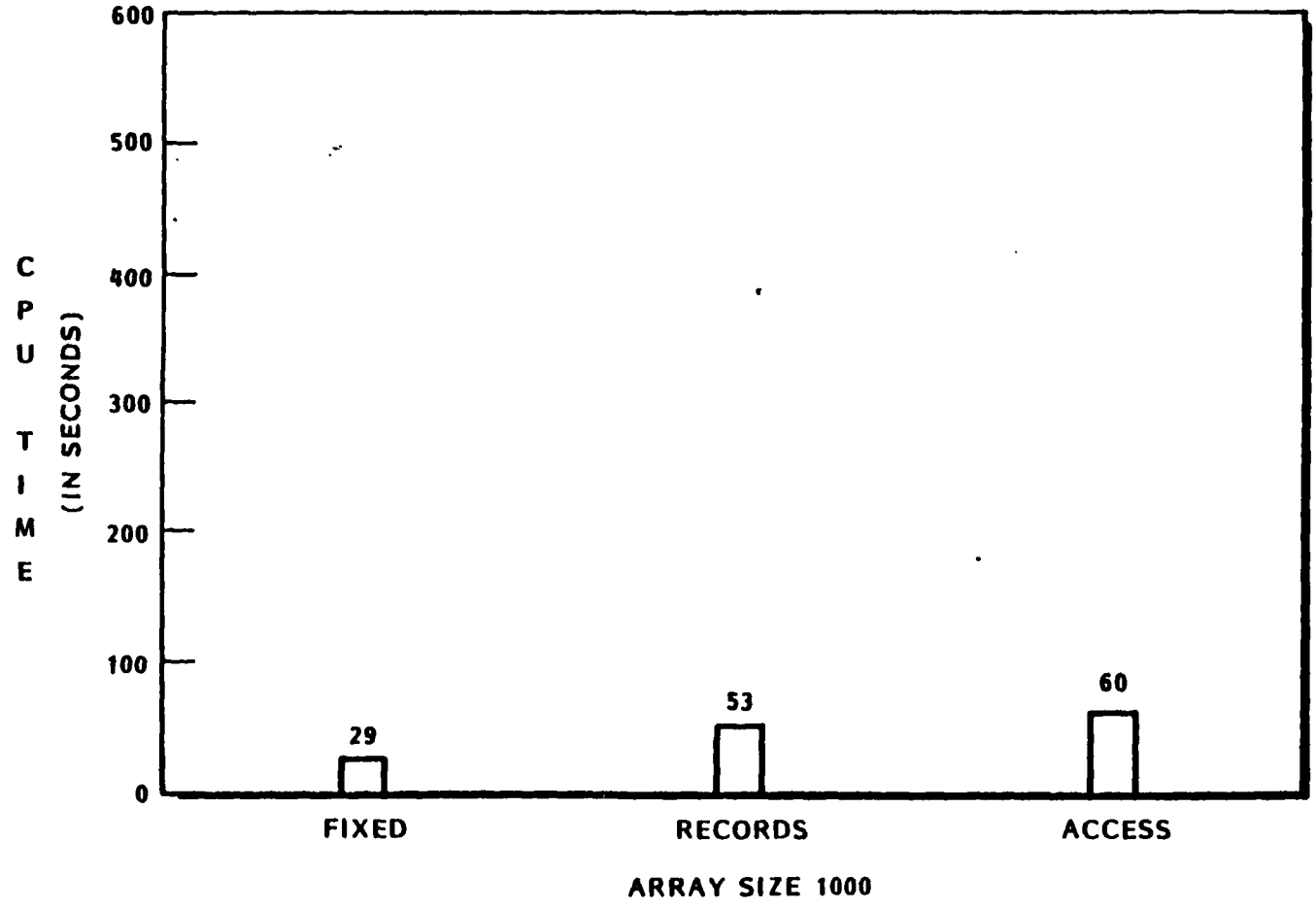
ROLM MSE-800

Bubble Sort



ROLM MSE-800

Selection Sort

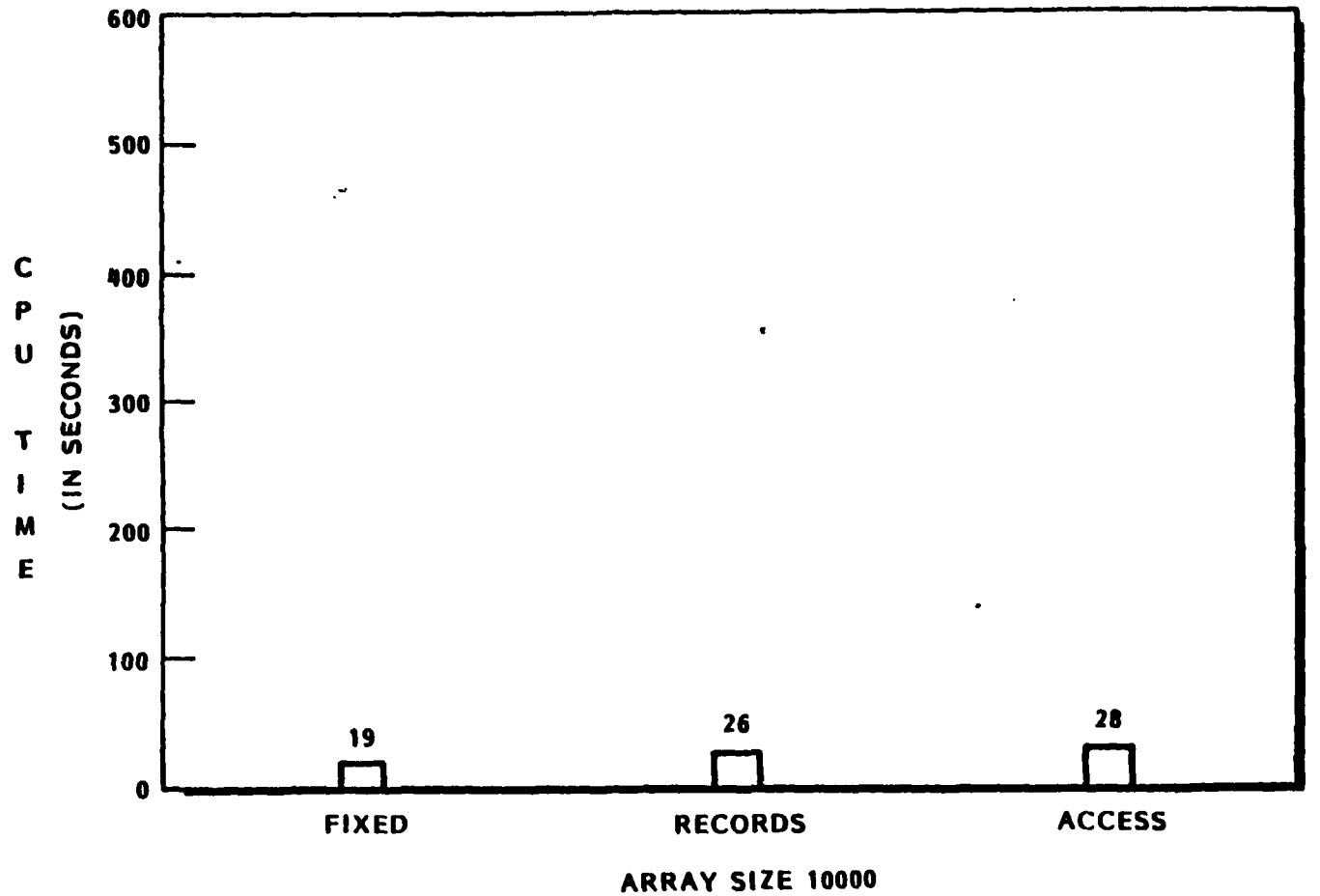


509

I-2

ROLM MSE-800

Quicksort

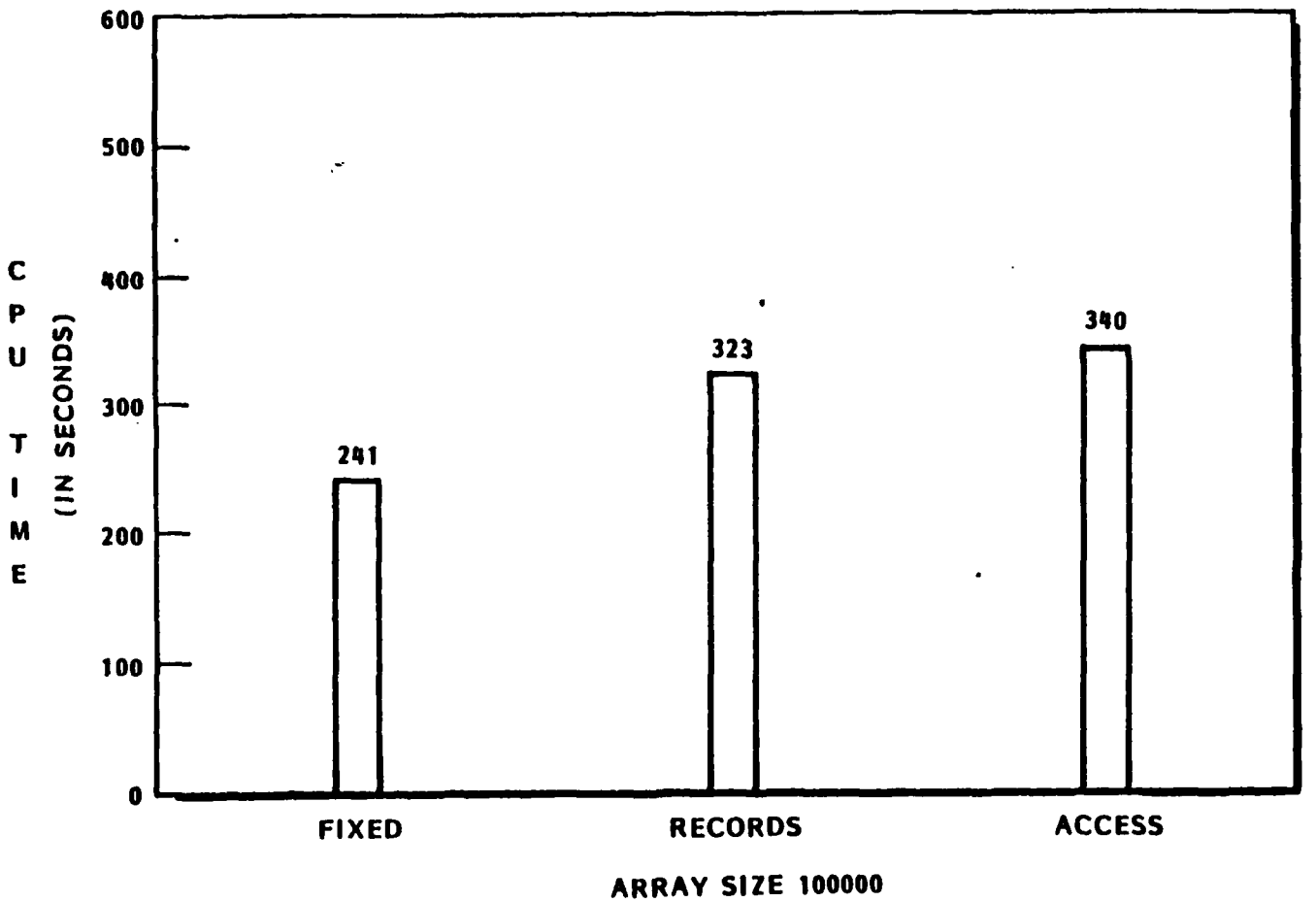


510

I-3

ROLM MSE-800

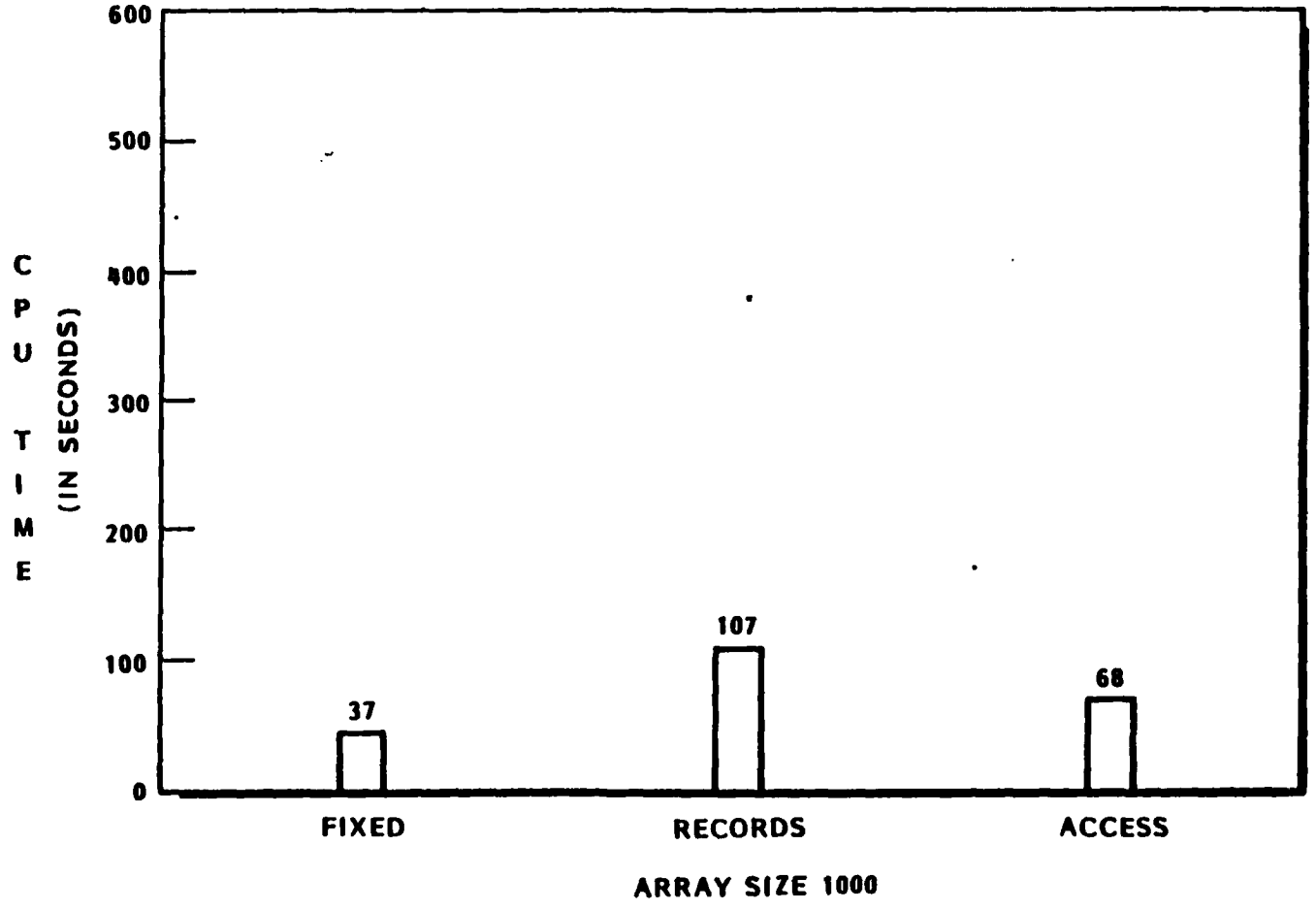
Quicksort



I-4

DEC VAX 11/780

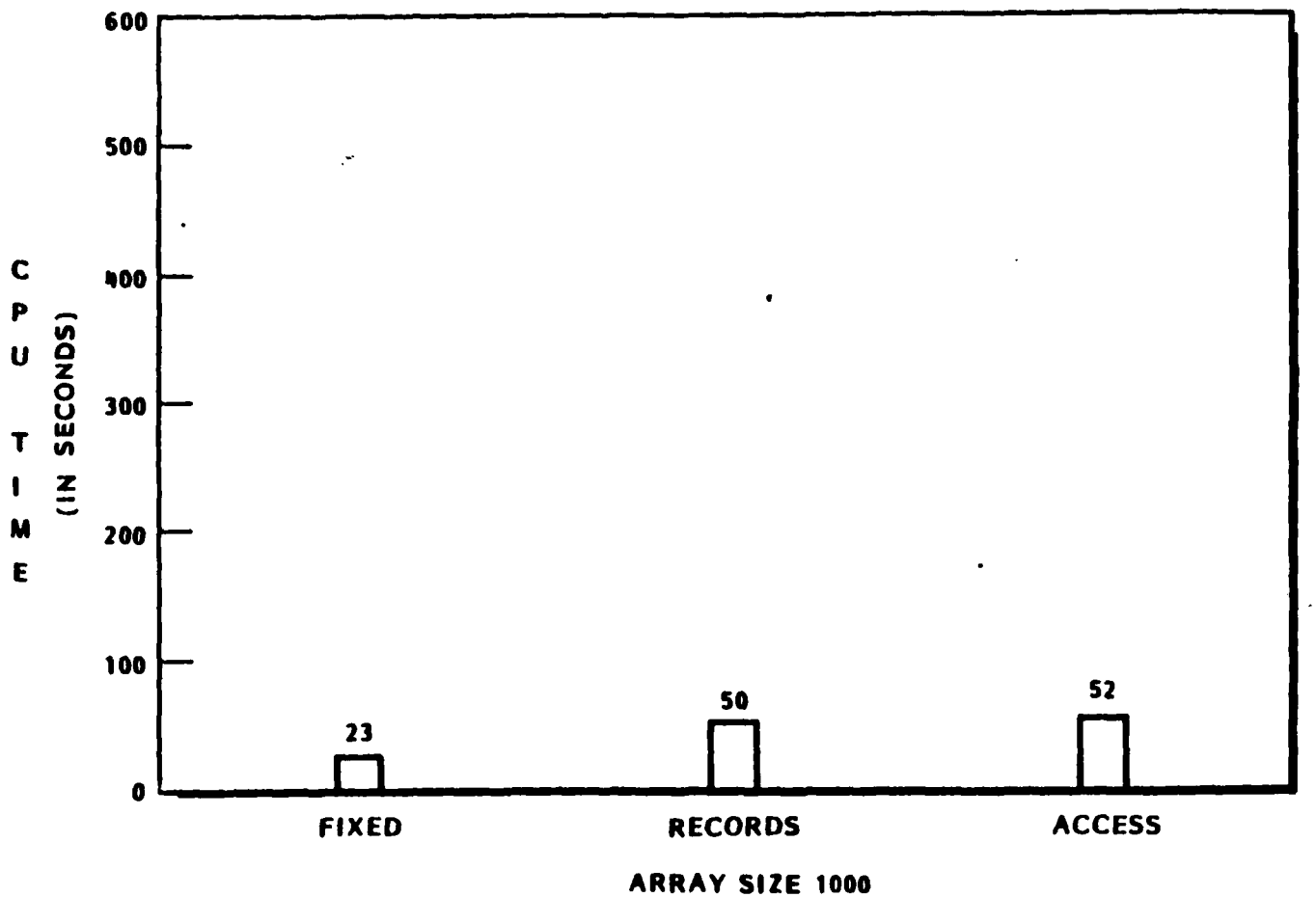
Bubble Sort



I-5

DEC VAX 11/780

Selection Sort

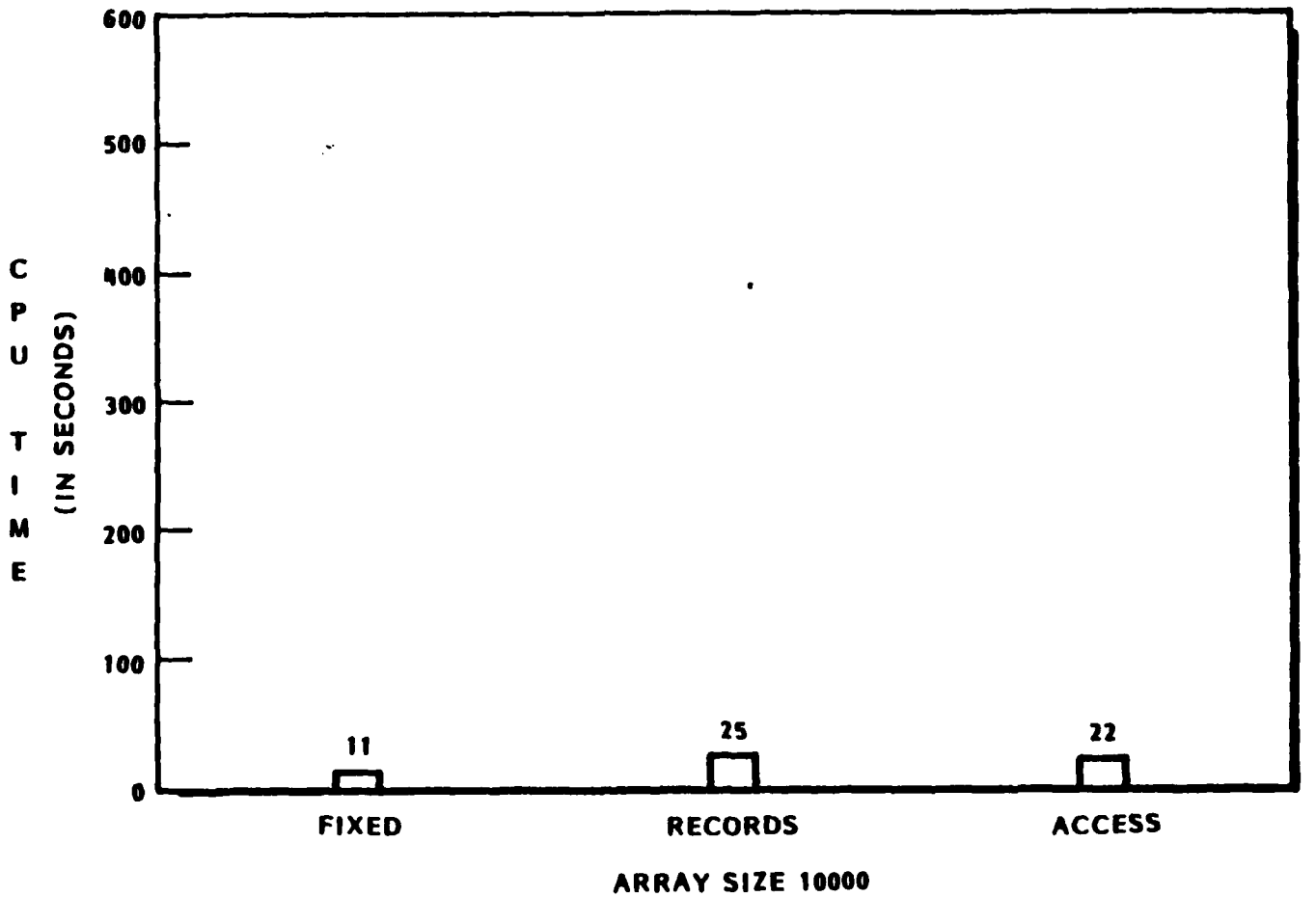


513

I-6

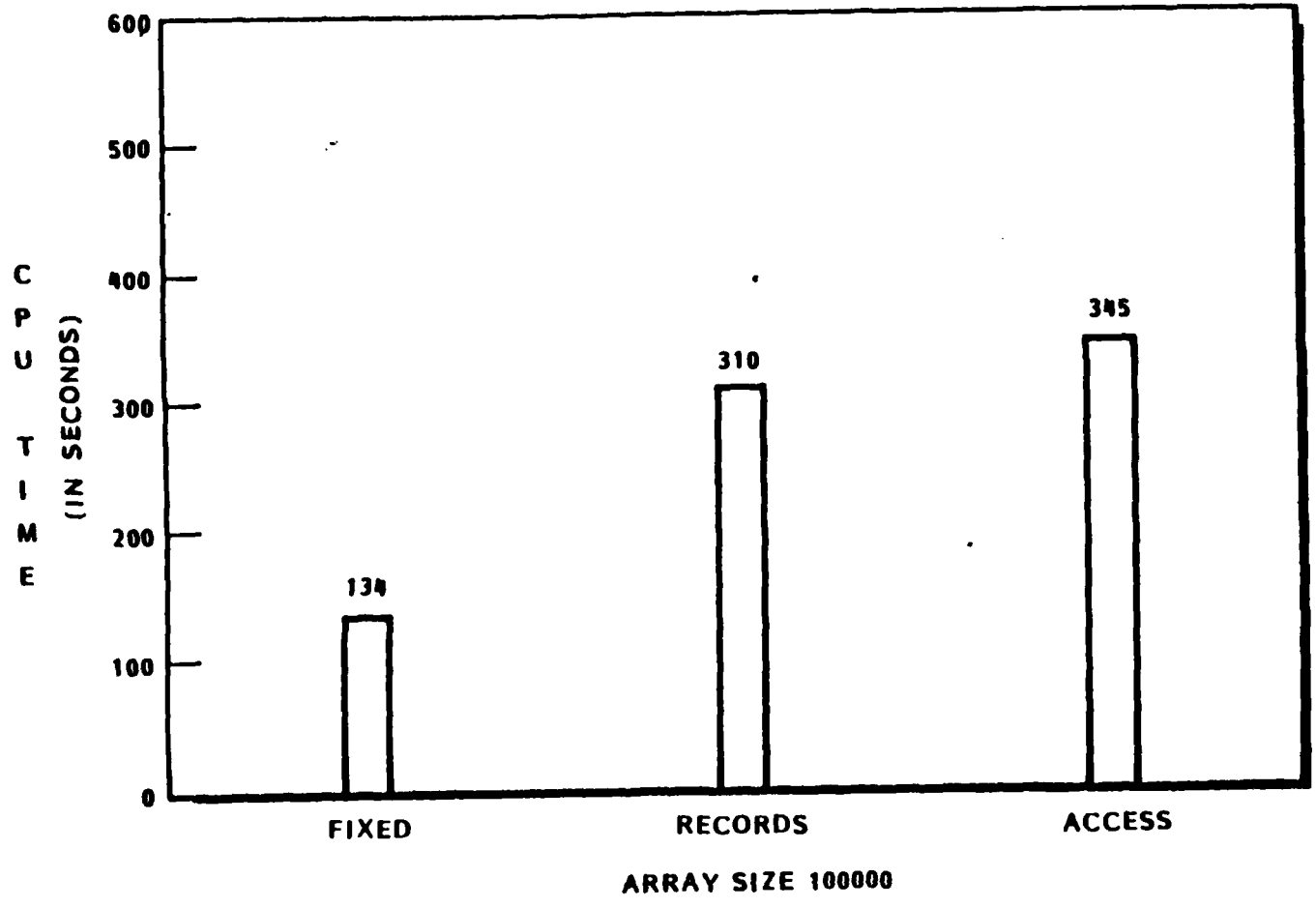
DEC VAX 11/780

Quicksort



DEC VAX 11/780

Quicksort



Almost Everything You Know is Wrong

- Sorting access types that designate records is not always faster than sorting records

- Algorithms which are too efficient in exchanging array components will generally perform better in direct sorting. The time required to dereference the access types for comparison operations becomes the most important factor and hence will cause indirect sorting to perform worse than expected

- Hence, the efficiency of some algorithms across machines may not be preserved

Future Directions

- Improving performance:
  - pragma INLINE
  - pragma SUPPRESS
  - pragma OPTIMIZE
  - Non—recursive Quicksort
- Integrating a merging package to sort external files
- Internal sorting for other data structures
- Developing an expert system to select an optimal algorithm
  - Machine-dependent attributes
  - Compiler
- Integrating a generic searching package

The Cost Impact of Developing Generic Applications in Ada

- Decrease the total cost of building software
- Sort_Pac proposed for use in a major LMSC project
- It will be a major LMSC project's policy to use Sort_Pac
  - programmers must demonstrate the need for deviations
  - Sort_Pac is used in the LMSC Ada training curriculum
  - Sort_Pac is used in as an example of Design-for-Reuse in an LMSC Ada Design Methodology course
- The cost of developing generic program units may be a function of how well the algorithms are known
- Reusable libraries
- By using Sort_Pac for all one-dimensional array sorting applications, duplication of effort in developing sorting routines for those applications is completely avoided
- Use at one's own level of Ada competence

Conclusions

- Generic program units provide reusability
- Generic program units construct a wall between data types and algorithms
- Generic formal subprograms deliver generalizations of operations
- Assumes as little about the data as possible
- Generic program units are typically small and they save many lines of code later on in development
- When developing generic program units, it is important to scrutinize every line of code because:
  - generic code is often quite dense
  - the potential applications for reuse are often never fully understood by the implementors
- Do not make generic program units overly general
- Adhere to the highest defined level of standards for readability and understandability in the package specification and body to faciliate reuse

Achieving Reusability of Ada Packages

Susan Mickel
General Electric Company
Military and Data Systems Operation
1277 Orleans Drive
Sunnyvale, CA 94086

One of the goals in introducing the Ada(*) language for software development is to increase productivity though significant increases in software reusability. The packaging feature of Ada provides the mechanism for designing reusable software components. However, regardless of how many thousands of potentially reusable packages are available in a program library, reuse will be limited by knowledge of what is available. In other words, system designers need efficient mechanisms that will quickly and easily locate packages to satisfy their requirements.

There are two facets to this problem. First, the process conventionally referred to as requdirements analysis must be conducted in such as to permit identification of reusable software at relatively gross levels. One must be able to identify entire subsystems for reuse, not just individual modules or packages. Second, information about available software must be structured and maintained so as to maximize accessibility.

This is hardly a new problem, as anyone who has used a library to write a research paper can attest. A Classical library approach could be taken, i.e. categorizing packages by function or application area. There are difficulties with schemes of this type. Categories are difficult to select. A user has to "guess" what categories may apply to a problem, usually producing a very large number of candidates to examine in detail. Users may interpret categories differently from the package author and fail to locate useful packages. As libraries expand, some categories may have to be subdivided into subcategories. A more complex categorization scheme requires a more complex path to reach a particular package and a more difficult query to be prepared by the user. Development of a classification system which can be easily extended and which allows items in the library to be classified under several categories is extremely difficult. For example, one category would relate items used in avionics systems; another category would relate control algorithms. Some components would be in both of these categories.

At the present time, reading package specifications is the usual method for locating packages for reuse. Specifications include all the information required for a designer to interface to that package from an Ada program. If meaningful identifiers have been used and the specification is well commented, a designer can determine whether the package will be useful in his application. However, this technique is feasible only when the number of available packages is quite low. To achieve reusability on a large scale, more sophisticated methods are clearly needed.

It is the author's contention that a general, long-term solution to this problem requires a much more powerful approach. One such approach is to apply expert systems technology to the problem. Consider the situation a few years hence wsusandmap as mangethe library. The designer needs a tool that will allow the defior ambiguities in the requirements provided should be noted and corrections should be allowed, again in a natural-like language. This tool should request a manageably small set of candidate packages for each defined requirement.

Currently, no successful tools of this type exist. Expert systems technology does exist, however, and has been proven in a variety of other applications. A significant amount of research will clearly be needed to make a tool of this type a reality. Without it, however, Ada cannot produce the much needed increase in productivity required to ease the software crisis.

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

GENERAL ELECTRIC COMPANY CORPORATE EXPERIENCE AND ACCOMPLISHMENTS

General Electric has a substantial ongoing research program in software engineering tools and environments, software productivity measurements, computer languages and compilers, expert systems, and related areas.

The Aerospace Business Group (ABG) employs over 2000 software development professionals engaged in software projects of all sizes ranging up to hundreds of person years. In support of this effort, ABG devotes a significant amount of its IR&D and other funds in research to develop new software engineering tools, methodologies and measurement techniques to increase productivity and quality. Among the outputs of this effort have been software development environment (SDE), a software development environment on the VAX, and software engineering and management (SEAM), a software management methodology, both of which have been widely used by ABG on government contracts.

The following paragraphs provide brief overviews of specific experience and accomplishments at specific components of the General Electric Company.

1.0 Corporate Research and Development

General Electric Corporate Research and Development (GE CRD) has a significant ongoing research program in software environment and expert system research. Some related programs are presented in the paragraphs below.

1.1 ADVANCED ADA DEVELOPMENT ENVIRONMENT

For the past two years, GE CRD has conducted a project which attempts to bring the power of modern Lisp-based environments to bear on the development of systems on Ada. Several tools have already been developed on the Lisp Machine under this project, including a smart editor.

1.2 OBLIGE

Oblige is an object-oriented language being developed for a CAD/CAM workstation.

1.3 LANGUAGE TOOL KIT

The Language Tool Kit is a compiler-compiler and other language-oriented software engineering tools developed at CRD and currently in use in ABG. The Language Tool Kit is written in Ada and produces compilers written in Ada, Pascal, or C.

1.4 REWRITE RULE LABORATORY

The Rewrite Rule Laboratory is an environment for performing research in automated reasoning that has been developed as a part of a joint NSF contract with the Massachusetts Institute of Technology. The Rewrite Rule Laboratory includes an interactive user language, called I, a user interface, and a number rewrite-rule-based theorem provers. The Rewrite Rule Laboratory is being used extensively to perform geometric reasoning for image understanding.

1.5 AFFIRM PROGRAM VERIFICATION SYSTEM

Corporate Research and Development is currently under contract from Digicomp Inc. (through Rome U.S. Air Force Development Center) to upgrade Affirm, a program verification system, for possible use in verifying Air Force software. Within GE, Affirm has been used to verify flight control software at our Aircraft Control Systems Department (ACSD).

1.6 EXPERT SYSTEMS

Corporate Research and Development has had a significant effort in expert systems for the past three years. DELTA, an expert system for locomotive maintenance, was built and transitioned to the GE Transportation Division, which plans to make it a product. A number of tools for constructing expert systems, including Delphi (a rule-based ' nguage) and Genex (a graphic system for designing expert systems) have been built and transitioned to several GE departments within ABG, which are now experimenting with their use. A proposal, "Reasoning with Incomplete and Uncertain Information," has recently been accepted by DARPA as part of the Strategic Computing Program.

1.7 KIT/KITIA

A staff member of the GE CRD Computer Science Branch is a representative on the Ada

KIT/KITIA committee, and was the original chairman of the CAIS subcommittee of KITIA.

2.0 Space Systems Division

Several Ada-related projects have been ongoing within various components of GE/ABG Space Systems Division (SSD) during the past several years. Some of these projects are described in the paragraphs which follow.

2.1 JESSE COMMITTEE

General Electric has had a participant from Data Systems Resource Management (DSRM) in Valley Forge, PA serving as a member of the Joint Services Software Engineering environment Industry Team since its origination.

2.2 ADA PROMPTER AND PROLOGUE

An Ada language construct prompter was generated at GE/SSD at Valley Forge during 1982. The prompter permits the user to insert skeletal Ada constructions into a file being edited, thus permitting the programmer to focus more attention upon the program rather than on the syntax and semantics of Ada.

2.3 DISTRIBUTED DESIGN ENVIRONMENT

A set of software tools that are specifically intended for operation in a distributed environment consisting of microprocessor workstation clusters is being constructed and prototyped by GE/DSRM in Valley Forge. This distributed environment is intended to support Ada software engineering during:

- design, using Ada as a program design language (PDL)
- Coding, using a context sensitive editor for Ada
- generation of design documentation

2.4 ADA AS A PDL

During 1983, a project at GE/DSRM in Valley Forge included an investigation of the use of Ada as a program design language (PDL). A survey of existing Ada PDLs was conducted, and a study of the associated methodology changes and relative success with each PDL was done. The results indicate that the modern software engineering methods embodied by Ada seem to be more important than PDL form (Ada subset vs. superset vs "plain" Ada), and that a given PDL should be supported by automated tools in order to obtain the full utility of the methodology.

2.5 ADA STYLE GUIDELINES

During 1983, a project at GE/DSRM in Valley Forge used Ada as both its design and implementation language. Data Systems Resource Management studied how Ada should be written, and how Ada programs should be documented. Preliminary documentation guidelinesapplied to that project's software. The guidelinsubject project and were observed to facilitate the readaiand/or updating the programs. Thbecome productive on tea

2.6 COMPUTER COMMUNICATIONS PROTOCOLS

During 1982 and 1983, a project was conducted within DSRM at Valley Forge that used Ada for the design, implementation, and demonstration of a real-time computer communications system. This project provided General Electric Company with valuable experience with the use of Ada in the design and implementation of real-time systems level computer software as well as providing a number of valuable lessons concerning the differences in methodologies and techniques required to successfully use Ada on a real project.

2.7 EFFECTS OF ADA AND OBJECT-ORIENTED DESIGN ON REQUIREMENTS SPECIFICATION

During 1983, a detailed study was performed by Data Systems Resource Management (DSRM) in Valley Forge and in Arlington, VA to determine the effects of Ada and object-oriented design techniques on the requirements specification process and the resulting documentation. As a result of this effort, an internal publication was generated, and the internal General Electric software engineering a)under review and revisis2.8 ADA EDUCATION

General Electric/Space Systems Division has used the TeleSoft on-line computer based quizzes and case studies to provide individualized instruction in Ada syntax. GE/SSD has also developed, documented, and tested a classroom course with machine problems in Ada design and programming. In the initial implementation of

the classroom course, two distinct teaching sequences were used in a controlled experiment to compare learning gains of the students.

Another Ada training research project was conducted by GE/SSD to determine the learning experiences of a programming team encountered in a controlled experiment based upon the redesign and coding in Ada of production software originally developed in FORTRAN. "TeleTraining" courses using stanstechniques at fivbeen devepresented on Ada resources and business opportunities, software productivity and software engineering.

2.9 DISTRIBUTED SYSTEMS SIMULATOR

Software to simulate and measure the performance of distributed data systems was developed by GE/SSD at Western Systems in Sunnyvale, CA. Ada was used as both the design (PDL) and implementation language on the Distributed Systems Simulator project.

3.0 Other ABG Components

3.1 ADA EDUCATION

A fifteen-week Ada course has been developed and is currently being presented at the General Electric Avionic and Electronic Systems Division (AESD) facility in Utica, NY. Particular emphasis is being given to the applicability of Ada of programmable signal processors.

3.2 FLIGHT CONTROL SOFTWARE

Currently, the GE/AESD fac

3.3 AUTO-CODE GENE

A system is being developed at GE/AS NY f generation of systblodiagr

3.4 ALS BETA-SITE

General Electric AESD Military Electronic Systems Operation in Syracuse, NY is currently a beta test site for the Ada Language System (ALS).

3.5 MICROPROCESSOR SOFTWARE CONVERSIONS

General Electric/Armament and Electrician, microprocessors into the Ada language in expert form risk assessments.

RESUME

Susan Mickel
General Electric Company
1277 Orleans Drive
Sunnyvale, CA 94086

Phone 408/734-4980

Research and Development

- Principal Investigator for building a "smart librarian" which applies expert systems technology to identify and retrieve reusable software components.
- Principal Investigator for GE software development methodologies for use with the DoD standard programming language, Ada (*). Principal Investigator for a standardized software development environment (SDE) for use in GE Space Systems Division DoD software projects.

Management Activities

- Technology Leader for software engineering: responsible for planning, managing, and providing technical leadership for technology programs in the field of software engineering.
- Past chairperson of the GE Aerospace Business Group (ABG) Software Engineering Panel; represents Western Systems on the ABG Software Subcouncil, advising senior GE software managers on Research and Development funding for software engineering.
- Represented Western Systems on blue-ribbon panels to determine GE corporate response to the DoD Software Technology for Adaptable Reliable Systems (STARS) initiative to improve software productivity and the introduction of the DoD Ada Programming Language into projects.
- Represented GE during technical exchanges with a major Japanese corporation.

Technical Activities

- Systems Engineering for a major classified real-time data system. Developed concept of operations, wrote specifications and performed analyses compliant with MIL-STD-490.
- Systems Engineering for several proposals.
- Acted as internal software technology consultant to WS including application of DoD MIL-STDs.
- Five years experience in the analysis, design, implementation and testing of large (over one million LOC) and complex aerospace application systems. Responsible for interface with customer and subcontractors.

Professional History

- General Electric Co., Space Systems Division
  Military and Data Systems Operations
  Western Systems
  Sunnyvale, CA 94086
- Technology Leader for Software Engineering
  1981 - present

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

- Math Specialist/Software Technologies
  1980 - 1981

  General Electric Co., Space Systems Division
  Military and Data Systems Operations
  Military Programs Department
  Valley Forge, PA
- Senior Programmer/Analyst
  1978 - 1980
- Applications Programmer
  1975 - 1978

Professional Societies

- Association for Computing Machinery
- SIGAda
- IEEE Computer Society

Education

- M.S. 1974 Pennsylvania State University
  Major: Computer Science
- B.S. 1973 Pennsylvania State University
  Major: Computer Science

Ada - related Activities

I. Conferences/Publications:

"Experience with an Object Oriented Method for Software Design"
Proceedings of the 3rd Ada-Europe/AdaTEC Conference
Brussels, June 1984

Papers discussing the impact of the Ada programming language on the software life cycle were presented at the following conferences in 1983:

Mission Assurance Conference
Sponsored by NSIA, AIA, NASA, USAF Space Division
Los Angeles, CA June 1983

Computers in Aerospace IV Conference
Sponsored by AIAA
Boston, MA October 1983

II. Workshops: Participated in the following DoD sponsored workshops

APSE Evaluation and Validation Workshop
Airlie, VA (AFWAL-sponsored)
April 1984

Software Initiative Workshop
Raleigh, NC
February 1983

III. Related Research:

In 1980, while CPCI leader for the GE Data Systems Modernization (DSM) project Stage II proposal, was responsible for defining the software development environment to support the development of Ada software.

In the three years following, led several IR&D projects to investigate and validate methodologies and tools to support Ada software development and to ease the transition to Ada. Object-oriented design and methodology and methods to promote software reusability are particular interests.

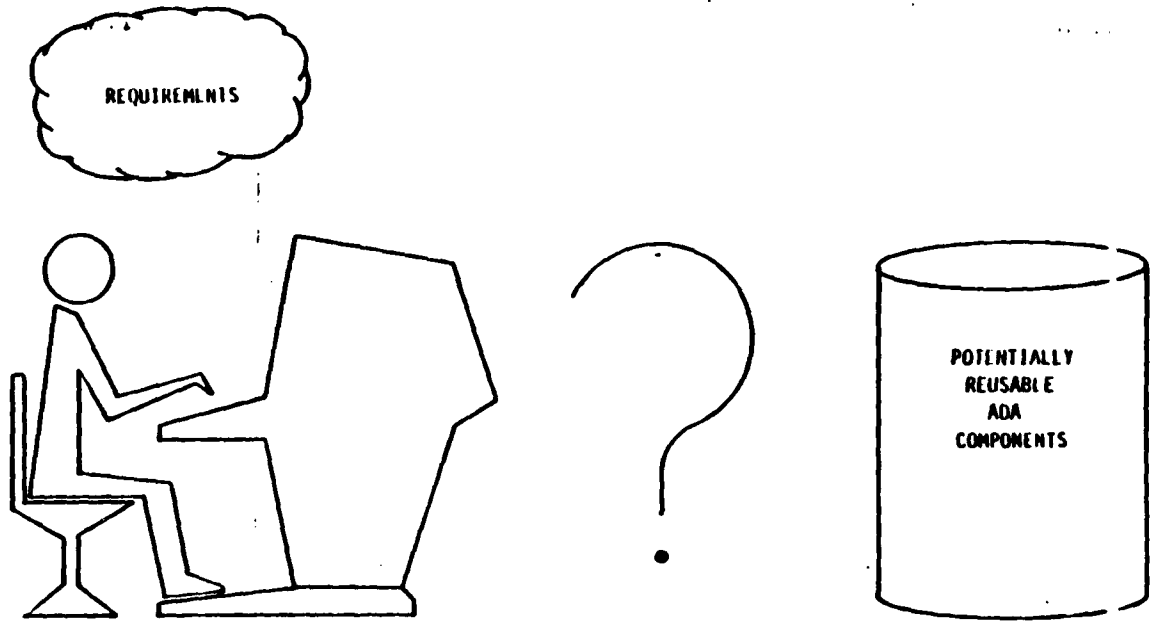
WORKSHOP ON REUSABLE COMPONENTS OF APPLICATION SOFTWARE

528

APRIL 9, 1985

SUSAN MICKEL
GENERAL ELECTRIC COMPANY
GROUND SYSTEMS PROGRAMS DEPT.
SUNNYVALE, CA

EXTENT OF REUSABILITY DEPENDS ON EASE OF REUSE



MINIMAL CAPABILITIES

I. MECHANISM FOR DEFINING REQUIREMENTS

II. MECHANISM FOR LOCATING COMPONENTS

- METHOD OF DESCRIBING COMPONENTS
- METHOD OF DETERMINING LESS THAN PERFECT MATCH
- METHOD OF REUSING DESIGN OR TECHNIQUE IF NOT THE CODE ITSELF

ADA PROVIDES HELP

- SPECIFICATIONS ARE SEPARATE FROM BODIES
- SPECIFICATIONS ARE AN ABSTRACTION, THE USER'S (I.E., ADA PROGRAM) VIEW

BUT

- LITTLE SEMANTIC INFORMATION IS AVAILABLE
- USER IS NOT AN ADA PROGRAM, BUT A SOFTWARE DESIGNER

SAMPLE RELEVANT DISCRIMINANTS

- PERFORMANCE CHARACTERISTICS
  - TIMING
  - SIZE
- PRECISION ESTIMATES
- LEVEL OF TESTING
- OWNERSHIP
- ALGORITHM DESCRIPTION
- ORDER OF INPUT RESTRICTIONS

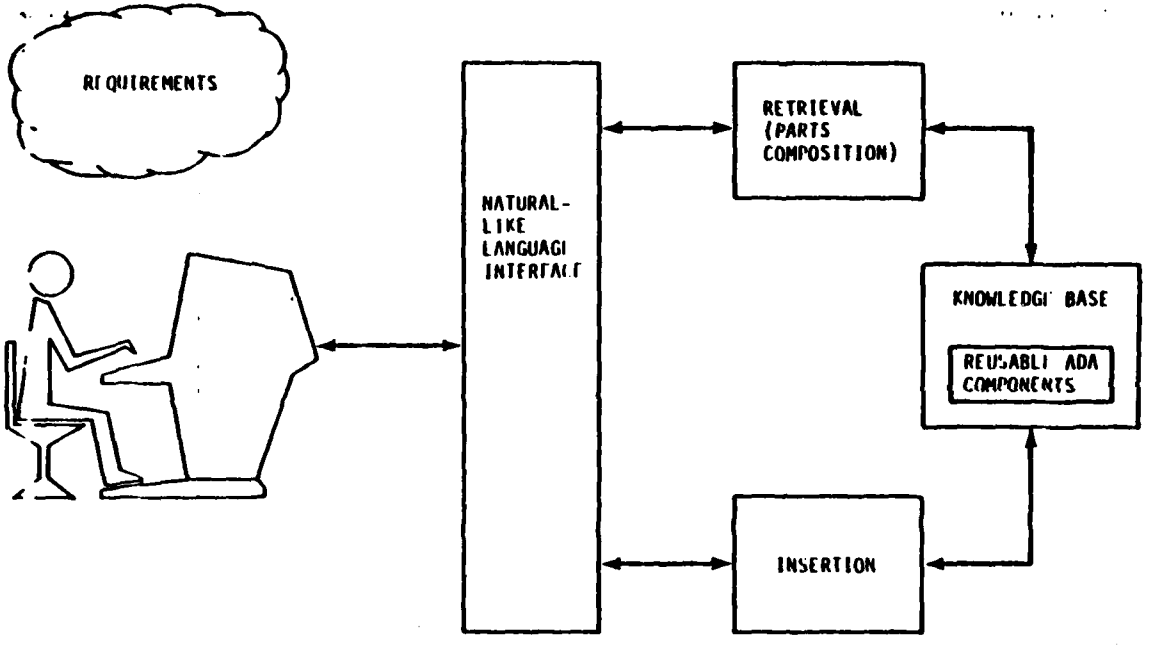
APPROACHES

1. MANUAL SCAN OF LIBRARY OF SPECIFICATIONS
2. AUTOMATED TEXT SEARCH
3. AUTOMATED TEXT SEARCH WITH SEMANTIC-ORIENTED STRUCTURED COMMENTS

THE GOAL

A LONG-TERM SOLUTION WHICH WILL SIGNIFICANTLY INCREASE PRODUCTIVITY

ONE SOLUTION



FUTURE APPLICATIONS

- INITIAL SYSTEM CONFIGURATION
- RESPONSE TO CHANGING REQUIREMENTS
- CORRECT ERRORS MADE BY DESIGNER AT EARLIER SESSIONS
- UPDATE KNOWLEDGE BASE UPON REJECTION OF SELECTED PACKAGE (E.G., INADEQUATE PERFORMANCE)
- SEMI-AUTOMATIC GROWTH OF LIBRARY
- INCREASE PRODUCTIVITY OF DESIGNER (EXPERT)
- REDUCE LEVEL OF EXPERTISE REQUIRED

CONCLUSION

- A LONG TERM SOLUTION WILL/MUST INCLUDE AN EXPERT SYSTEMS APPROACH
- RESEARCH IS NEEDED IN WAYS TO APPLY EXPERT SYSTEMS TECHNOLOGY TO THIS PROBLEM

SOFTWARE REUSABILITY

J. E. Mortison
Software Engineering
SPERRY CSD
St. Paul, Minnesota

1. INTRODUCTION

With skyrocketing software development costs, the Department of Defense is becoming increasingly interested in software reusability, and the potentials of maximizing productivity and lowering costs associated with reusability. At the request of the Deputy Assistant Secretary of the Navy, C?3OI, an Industry Study Task Group was formed to study the issues relating to software reusability and to make recommendations as to the feasibility of enhancing reusability in future Navy programs. This National Security Industrial Association (NSIA) Industry Study Task Group, ISTG 84-2, was formed in 1984. The group, of which the author was a group leader, was composed of 22 representatives from 20 defense contractor firms. The study results were presented to the Honorable Harold Kitson, Deputy Assistant Secretary of the Navy and his Staff in August 1984. The study findings were also presented to the participants of the Reusable Software Symposium at the Naval Research Laboratory in Washington D.C. in January 1985. These study findings form the baseline for the Navy's current position and plans for reusable software.

2. GENERAL PERSPECTIVE

First of all, a definition of terminology is required. What is meant by the term "software reusability"? The Task Group defined it in its broadest sense to include the full spectrum of software related items developed across the software life cycle. In this sense, software reusability can be used to refer to functional specifications, software architectures, program design language (PDL) representations, or test cases as well as the code itself. Thus, reusability is not limited to code, but may also include reusable specifications, PDL designs, documentation, test cases and test environments. (See Figure 1.)

Obviously, the motivation behind software reusability is lower cost, increased reliability and enhanced modifiability. However, the disadvantage of reusability is reduced innovation. To reuse software building blocks generated in the past means that the latest technology must sometimes be foregone. In other words, there is always a trade-off analysis that must be done between the new versus the old. However, when it is determined that reusability is to be used, handsome cost savings are potentially possible. The Japanese, for instance, are claiming productivity of 2000 lines/code/month with only .2 latent defects (6 months after delivery per 1000 lines of code). The reuse of systems, design, and code is the assumed practice in Japan. (See Figure 2.)

Currently there are many supporting DoD efforts and activities which will favorably impact the feasibility of software reusability within the next decade. Some of these activities include the DoD STARS Initiative, the Joint Service Software Engineering Environment (JSSEE), and the Navy's Reusable Software Implementation Program (RSIP) to name just a few.

3. BASELINE INFORMATION

In order to identify the issues relating to software reusability and to make recommendations concerning the feasibility of software reusability to future Navy programs, the Industry Study Task Group received Navy briefings on the following key Naval programs:

- Restructured Naval Tactical Data System (RNTDS)
- C?3O Integration Program
- Reusable Software Implementation Program (RSIP)
- Afloat Correlation System (ACS)

HIERARCHY OF REUSABILITY

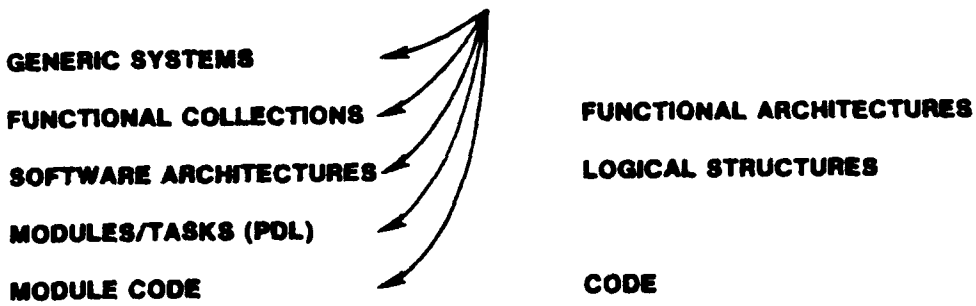


Figure 1.

JAPANESE SOFTWARE FACTORY APPROACH VS AD HOC APPROACH

	USA	JAPAN *
LINES/CODE/MONTH	200	2000
LATENT DEFECTS (6 MONTHS AFTER DELIVERY PER 1000 LINES OF CODE)	1.5	.2

* STABLE REQUIREMENTS ARE ASSUMED.
  - SYSTEMATIC, CONTROLLED PROCESS ACROSS THE LIFE CYCLE
  - PRODUCTION-ORIENTATION RATHER THAN DESIGN-ORIENTED
  - REUSE OF SYSTEMS, DESIGNS, CODE IS ASSUMED PRACTICE

Figure 2.

- Advanced Combat Direction System (ACDS)
- Joint Tactical Information Distribution System (JTIDS)
- Integrated Tactical Surveillance System (ITSS)

In addition, several field trips were made to Naval Installations and customer sites.

4. FINDINGS

The Industry Study Task Group ISTG 84-2 findings based on their study efforts and their Navy briefings included the following:

(1) Software reusability is currently technically feasible within a given broad application area such as the Navy's Restructured Naval Tactical Data System (RNTDS). However more research is needed on developing design methodologies that support reusability across different application areas.

(2) Software reusability requires a disciplined system engineering approach with the emphasis on reusability placed on the initial phases of the life cycle. Reusability requires that domain analysis techniques become part of the standard system engineering design practice.

(3) The current major hurdle to software reusability is the lack of military management commitment, not technical issues.

(4) Reusability must become part of both the military acquisition process and the military review process.

(5) Reusability requires an investment upfront to create a depository of software building blocks. Only after this investment has been made can payback benefits be derived.

(6) Sub-issues relating to reusability that must be addressed include such items as the development of a software components depository, certification, data rights, configuration management of the components, depository information dissemination, and technology transfer techniques.

(7) Trade-off studies need to be performed on contractor incentives such as royalties and other incentives.

(8) Research must be performed on how to quantify the cost savings of reusability so that the potential savings can become part of the decision-making process of the military program manager.

5. RECOMMENDATIONS

The Study Task Group recommendations to Navy management were as follows:

(1) Navy participation in the DoD Initiatives such as STARS should be strengthened.

(2) More research is needed on the software engineering procedures and design methodologies that will enhance reusability.

(3) Domain analysis must become part of the standard system engineering design practice and more research is needed on partitioning paradigms.

(4) Reusability must become an inherent part of the military acquisition and review process.

(5) Software reusability sub-issues must be addressed and recommendations made to provide solutions relating to:

- Software Depository
- Certification
- Data Rights
- Configuration Management
- Depository Information Dissemination
- Technology Transfer Mechanisms

In response to the DoD's STARS Application Task Area Plan, the Study Task Group observed that it is apparent that a large payback from the STARS Initiative is possible by maximizing software productivity through the mechanism of software reusability. It was recommended that under STARS, the technology of software reusability be advanced by funding the following activities:

- Development of software reusability design methodologies and support tools

- Development of tools and techniques to quantify reusability cost savings
- Development of recommendations regarding centralized software depositories, certification and contractor incentives
- Integration of reusability into the military acquisition and review processes
- Technology transfer of research into current industry practice

ISTG 84-2 concluded that with the increased software leveraging and software reusability brought into place by the STARS Initiative, the payback in increased productivity is going to bring dramatic changes to the benefit of both industry and DoD within the next decade.

6. REFERENCES

(1) Mortison, J.E., "System Engineering Aspects of Software Reusability", DoD STARS Conference, San Diego, May 1985.

(2) Mortison, J.E., "Software Reusability: Technical Factors and Related Issues", Naval Research Lab, Washington, D.C., January 1985.

(3) NSIA ISTG 84-2 Study Report, Washington, D.C., August 1984.

Joyce E. Mortison
Engineering Manager
Sperry Corporation

Joyce Mortison is currently the Software Engineering Resource Manager for the Underseas Applications at Sperry Computer Systems Division, in St. Paul, Minnesota. She has 24 years experience in software development and has served in various management and technical positions in software technology, development, research and VHSIC related projects.

Prior to Sperry, she was a Research Fellow on the faculty of the University of Minnesota and has held management positions at the Mid-American Solar Energy Center, General Electric/NASA, and Procter and Gamble. She is the author of 45 technical publications.

Joyce is active in the computer resources management community. She has been an active member of the NSIA Software and QRAC Committees since 1977. She served as NSIA Co-Chair of the STARTS Systems Task Area Group in 1983. During 1984, she served as a Group Leader of the NSIA Industry Task Force investigating Software Reusability. At present, she is a member of the STARS Application Industry Working Group.

Joyce has a B.A. in Economics and Mathematics from the University of Cincinnati, has completed graduate coursework in Computer Science at the University of Minnesota, and has recently completed the Government Acquisition and Contract Management MBA Program at the College of St. Thomas.

NSIA

ISTG 84-2

Software Reusability

542

STARS WORKSHOP

NAVAL RESEARCH LAB

JOYCE E. MORTISON
ENG. MGR, SPERRY
4/9/85

Study Start Up

- 4 January 1984 - NSIA Proposes Study
- 23 January 1984 - DASN (C^3I) Agrees
- 23 February 1984 - NSIA Solicits Task Group Nominees from Industry
- 23 March 1984 - Task Group Selection Completed
- 20 April 1984 - First Meeting of ISTG 84-2

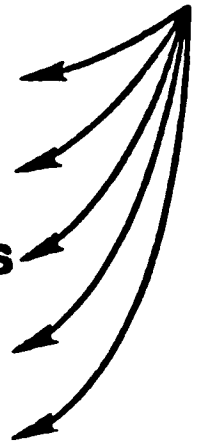
ISTG 84-2 - Membership

DAVID T. BARRY	TAC
RONALD G. CLANTON	WESTINGHOUSE
H. M. COURTER	SPERRY
RON G. DAMER	E-SYSTEMS
JACK DOZIER	ARINC
J. R. ELSTON	BOEING
OLEG GOLUBJATNIKOV	G.E.
BOBBY J. GREER	SAI
DOUGLAS S. INGRAM	PRC
PETER J. KENNEDY	AT&T
DAYLE McCLENDON	TRW
JOYCE E. MORTISON	SPERRY
TOM PARRISH	PRC
RICHARD W. SEILER	McDONNELL DOUGLAS
T. W. SHEFFIELD	HARRIS
RICHARD J. SMITH	SAI
WAYNE J. SMITH	RCA
ROBERT STOW	SINGER
TOM L. WALTERS	ROCKWELL INTERNATIONAL
WILLIAM WEDLAKE	LMSC
HOWARD YUDKIN	BOOZ-ALLEN & HAMILTON
J. BIRCHFIELD	BBN

OVERVIEW

SOFTWARE REUSABILITY — GENERAL PERSPECTIVE

Software Reusability - What Do We Mean?



GENERIC SYSTEMS

FUNCTIONAL COLLECTIONS

SOFTWARE ARCHITECTURES

MODULES/TASKS (PDL)

MODULE CODE

FUNCTIONAL ARCHITECTURES

LOGICAL STRUCTURES

CODE

REUSABILITY IS NOT LIMITED TO CODE

- REUSABLE SPECIFICATIONS
- REUSABLE DESIGN (PDL)
- REUSABLE DOCUMENTATION
- REUSABLE TEST ENVIRONMENTS
- REUSABLE CODE

MOTIVATION

- LOWER COST
- HIGHER PRODUCTIVITY
- INCREASED RELIABILITY
- ENHANCED MODIFIABILITY
- INCREASED VISIBILITY

DISADVANTAGE

- REDUCED INNOVATION

JAPANESE SOFTWARE FACTORY APPROACH VS AD HOC APPROACH

	USA	JAPAN *
LINES/CODE/MONTH	200	2000
LATENT DEFECTS (6 MONTHS AFTER DELIVERY PER 1000 LINES OF CODE)	1.5	.2

* STABLE REQUIREMENTS ARE ASSUMED.
  - SYSTEMATIC, CONTROLLED PROCESS ACROSS THE LIFE CYCLE
  - PRODUCTION-ORIENTATION RATHER THAN DESIGN-ORIENTED
  - REUSE OF SYSTEMS, DESIGNS, CODE IS ASSUMED PRACTICE

SUPPORTING EFFORTS/ACTIVITIES

- STARS INITIATIVE
- JOINT SERVICE SOFTWARE ENGINEERING ENVIRONMENT (JSSEE)
- SOFTWARE ENGINEERING INSTITUTE
- ADA PROGRAM(S)
- RSIP PROGRAM (NAVAL ELECTRONICS SYSTEMS COMMAND)
- DOD-STD-2167 (SDS)
- ACQUISITION AND DATA RIGHTS TASK FORCE
- KIT AND KITIA (KAPSE INTERFACE TEAMS)

- **WHAT CAN BE DONE IN INTERIM?**

STARTING POINT

- START WITH CURRENT PRODUCTION BASE AND LIFE CYCLE
- SHIFT EMPHASIS TO REUSABILITY BY MILITARY MANAGEMENT COMMITMENT
- PERFORM RESEARCH IN SYSTEM AND SOFTWARE DEVELOPMENT METHODOLOGIES
- ADD TRAINING, DESIGN STANDARDS, SOFTWARE REPOSITORIES, CONFIGURATION MANAGEMENT, AND ACQUISITION POLICIES TO ENSURE REUSABILITY

SIGNIFICANT FACTORS

1. ACQUISITION PROCESS
2. SYSTEM ENGINEERING TECHNOLOGY
3. SOFTWARE ENGINEERING TECHNOLOGY
4. CONFIGURATION CONTROL
5. EDUCATION/TRAINING
6. MILITARY MANAGEMENT COMMITMENT

1. ACQUISITION PROCESS

- **REUSABILITY MUST BE PART OF THE ACQUISITION AND REVIEW PROCESS**
  - **SYSTEM REQUIREMENTS**
  - **PRIME ITEM DEVELOPMENT SPEC**
  - **PROGRAM PERFORMANCE SPEC**
  - **PROGRAM DESIGN SPEC**
- **ECONOMIC INCENTIVES**

2. SYSTEM ENGINEERING TECHNOLOGY

- KEY FACTOR IS TO GET PERFORMANCE REQUIREMENTS UP FRONT TO DIFFERENTIATE BETWEEN THE UNIVERSAL AND NONUNIVERSAL ELEMENTS
- PROVIDE A LAYERED UNIFIED APPROACH
- PROVIDE NEW DESIGN PARADIGMS
- PARTITION FOR REUSABILITY

SYSTEM ENGINEERING PRINCIPLES

- IMPLEMENT PARTITIONING METHODOLOGY WHICH REFLECTS:
  - — ISOLATION OF THE VARIABLE FROM THE GENERIC FUNCTIONS
  - — HARDWARE/PROCESSING CONSTRAINTS
  - — SINGLE FUNCTION TASKS
  - — SINGLE INPUT/OUTPUT DATA SETS
  - — INDEPENDENCE FROM PRECEDING/FOLLOWING TASKS
- DESIGN FOR FUTURE REUSABILITY
  - — APPROACH FROM A "SUPERSET" VIEWPOINT
  - — IDENTIFY THE COMMON CORE OF APPLICATIONS AND SEPARATE OUT EXPECTED VARIATIONS

3. SOFTWARE ENGINEERING TECHNOLOGY

- DEVELOP AND IMPLEMENT REUSABILITY DESIGN METHODOLOGY
  - — DESIGN FOR REUSABLE ELEMENTS
  - — DETERMINE THE "PARTITIONING FOR REUSABILITY" PARADIGM
  - — DETERMINE THE SIMILARITIES AND DIFFERENCES
  - — FUNCTIONALLY SPECIFY THE "GENERIC"
  - — ENCAPSULATE THE DESIGN ELEMENTS
  - — "ADD-ON" THE DIFFERENCES
  - — REPEAT PROCESS FOR NEXT LEVEL
- DEVELOP SUPPORTING DOCUMENTATION
  - — STRUCTURAL MODELS
  - — DOMAIN ANALYSIS TECHNIQUES
  - — PARTITIONING PARADIGMS

THE REUSABLE DESIGN METHODOLOGY:

- MUST BE DEVELOPED AS AN INTEGRATED PROCESS ACROSS THE LIFE CYCLE
- THE METHODOLOGY MUST DRIVE THE TOOLS, NOT THE TOOLS THE METHODOLOGY
- THE METHODOLOGY MUST NOT BE DEVELOPED BY MERELY CONSOLIDATING INDIVIDUAL TOOLS TO FORM A "METHODOLOGY"
- *MUST BE RESULTS ORIENTED — LOOK AT WHAT IS SPECIFIED VIA A PARTITIONING DISCIPLINE, NOT MERELY HOW IT IS DOCUMENTED BY A SPECIFICATION LANGUAGE OR A PDL*

4. CONFIGURATION CONTROL

- **DEVELOP PROCESSING AND DISSEMINATION MECHANISMS**
  - CATALOGING PROCESS
  - CONFIGURATION MANAGEMENT
  - CERTIFICATION NEEDS
  - DISSEMINATION PROCESS
  - INCENTIVES

5. EDUCATION/TRAINING

- PROGRAM MANAGERS
- SYSTEM ENGINEERS
- SOFTWARE ENGINEERS
- MAINTENANCE ENGINEERS
- MILITARY MANAGEMENT
- CONTRACTOR MANAGEMENT

6. MILITARY MANAGEMENT COMMITMENT

- MOST IMPORTANT TO MAKE IT ALL HAPPEN
  - — REUSABILITY MUST BECOME PART OF THE MILITARY DECISION PROCESS
  - — REQUIRE FORMAL CONSIDERATION OF REUSABILITY AT EACH DECISION STEP IN THE ACQUISITION PROCESS
  - — ESTABLISH CONTRACTING REQUIREMENTS
  - — ESTABLISH INCENTIVES

CONCLUSIONS

AND

RECOMMENDATIONS

RNTDS IS A SIGNIFICANT FIRST STEP

- IN FORMALIZING A DISCIPLINE
- IN DEVELOPING A METHODOLOGY
- IN FOCUSING ON S/W REUSABILITY

- **THE KEY TO INCREASING SOFTWARE REUSABILITY IS MILITARY MANAGEMENT COMMITMENT**

ISTG 84-2 RECOMMENDATIONS

- NAVY PARTICIPATION IN DOD INITIATIVES, SUCH AS STARS, SHOULD BE STRENGTHENED
- RESEARCH IS NEEDED ON DESIGN METHODOLOGIES
- DOMAIN ANALYSIS MUST BECOME PART OF STANDARD DESIGN PRACTICE
- REUSABILITY MUST BECOME PART OF THE MILITARY ACQUISITION AND REVIEW PROCESS
- SUB-ISSUES MUST BE ADDRESSED SUCH AS
  - — SOFTWARE DEPOSITORY
  - — CERTIFICATION
  - — DATA RIGHTS
  - — CONFIGURATION MANAGEMENT
  - — DEPOSITORY INFORMATION DISSEMINATION
  - — TECHNOLOGY TRANSFER

STARS RECOMMENDED ACTIVITIES

1. DEVELOP REUSABILITY DESIGN METHODOLOGIES AND TOOLS
2. DEVELOP TOOLS TO QUANTIFY COST SAVINGS
3. INTEGRATE REUSABILITY INTO THE MILITARY ACQUISITION AND REVIEW PROCESS
4. EVALUATE ISSUES SUCH AS CENTRALIZED DEPOSITORIES, CERTIFICATION, DATA RIGHTS, AND CONTRACTOR INCENTIVES
5. PROVIDE TECHNOLOGY TRANSFER INTO INDUSTRY PRACTICE
   - ESTAB. PROTOTYPE S/W DEV. ENVIRONMENT VEHICLES
   - APPLY METHODOLOGIES/TOOLS ON VARIOUS SIZED PROJECTS
   - IDENTIFY METHODS/TOOLS WHICH ENHANCE PRODUCTIVITY AND REUSABILITY
   - DEV. SUPPORT TOOLS
     - — S/W DEPOSITORY DATA BASE SUPPORT TOOLS
     - — GRAPHIC DISPLAYS OF COMPONENTS MEETING SPECIFIED I/O CRITERIA
     - — AUTOMATED DOCUMENTATION GENERATORS

DOD STARS APPLICATION TASK AREA

SYSTEM ENGINEERING ASPECTS OF SOFTWARE REUSABILITY

STARS WORKSHOP

NAVAL RESEARCH LAB

JOYCE E. MORTISON
ENG. MGR, SPERRY
4/9/85

- Industry study ISTG 84–2
- General perspective
- Significant factors
- Conclusions
- Recommendations

Study Start Up

- 4 January 1984 - NSIA Proposes Study
- 23 January 1984 - DASN (C^3I) Agrees
- 23 February 1984 - NSIA Solicits Task Group Nominees from Industry
- 23 March 1984 - Task Group Selection Completed
- 20 April 1984 - First Meeting of ISTG 84-2

ISTG 84-2 - Membership

DAVID T. BARRY	TAC
RONALD G. CLANTON	WESTINGHOUSE
H. M. COURTER	SPERRY
RON G. DAMER	E-SYSTEMS
JACK DOZIER	ARINC
J. R. ELSTON	BOEING
OLEG GOLUBJATNIKOV	G.E.
BOBBY J. GREER	SAI
DOUGLAS S. INGRAM	PRC
PETER J. KENNEDY	AT&T
DAYLE McCLENDON	TRW
JOYCE E. MORTISON	SPERRY
TOM PARRISH	PRC
RICHARD W. SEILER	McDONNELL DOUGLAS
T. W. SHEFFIELD	HARRIS
RICHARD J. SMITH	SAI
WAYNE J. SMITH	RCA
ROBERT STOW	SINGER
TOM L. WALTERS	ROCKWELL INTERNATIONAL
WILLIAM WEDLAKE	LMSC
HOWARD YUDKIN	BOOZ-ALLEN & HAMILTON
J. BIRCHFIELD	BBN

Input To The Study Group

Full committee

- RNTDS
- C^3 Integration
- Reusable Software Implementation Program
- ACS
- ACDS
- JTIDS
- ITSS

Subcommittees

- Field Trips to FCDSSA San Diego, Dam Neck (RNTDS)
- C^2P/ACDS
- Interviews with OGC/CNM Acquisition Personnel

OVERVIEW

SOFTWARE REUSABILITY — GENERAL PERSPECTIVE

Software Reusability – What Do We Mean?

GENERIC SYSTEMS

FUNCTIONAL COLLECTIONS — FUNCTIONAL ARCHITECTURES

SOFTWARE ARCHITECTURES — LOGICAL STRUCTURES

MODULES/TASKS (PDL)

MODULE CODE — CODE

Reusability Is Not Limited to Code

- Reusable specifications
- Reusable design (PDL)
- Reusable documentation
- Reusable test environments
- Reusable code

Motivation

- Lower cost
- Higher productivity
- Increased reliability
- Enhanced modifiability
- Increased visibility

Disadvantage

- Reduced innovation

Japanese Software Factory Approach vs. Ad Hoc Approach

	USA	Japan*
Lines/Code/Month	200	2000
Latent defects (6 months after delivery per 1000 lines of code)	1.5	.2

*Stable requirements are assumed

- Systematic, controlled process across the life cycle
- Production–orientation rather than design–oriented
- Reuse of systems, designs, code is assumed practice

Supporting Efforts/Activities

- STARS Initiative
- Joint Service Software Engineering Environment (JSSEE)
- Software Engineering Institute (SEI)
- Ada program(s)
- RSIP program (Naval Electronics Systems Command)
- DoD–STD–2167 (SDS)
- Acquisition and Data Rights Task Force
- KIT and KITIA (KAPSE interface teams)

- **What Can Be Done In Interim?**

Starting Point

- Start with current production base and life cycle
- Shift emphasis to reusability by military management commitment
- Perform research in system and software development methodologies
- Add training, design standards, software repositories, configuration management, and acquisition policies to ensure reusability

Significant Factors

1. Acquisition process
2. System engineering technology
3. Software engineering technology
4. Configuration control
5. Education/training
6. Military management commitment

1. Acquisition Process

- Reusability must become part of the military acquisition and review process
  - Better acquisition practices, procedures, tools
  - Better proposal evaluation criteria
  - Tools for estimating reusability cost savings
  - Integrated acquisition and project management processes
- Economic incentives
  - Producers of reusable code (e.g., royalties)
  - Users of reusable code (e.g., incentive fees)

2. System Engineering Technology

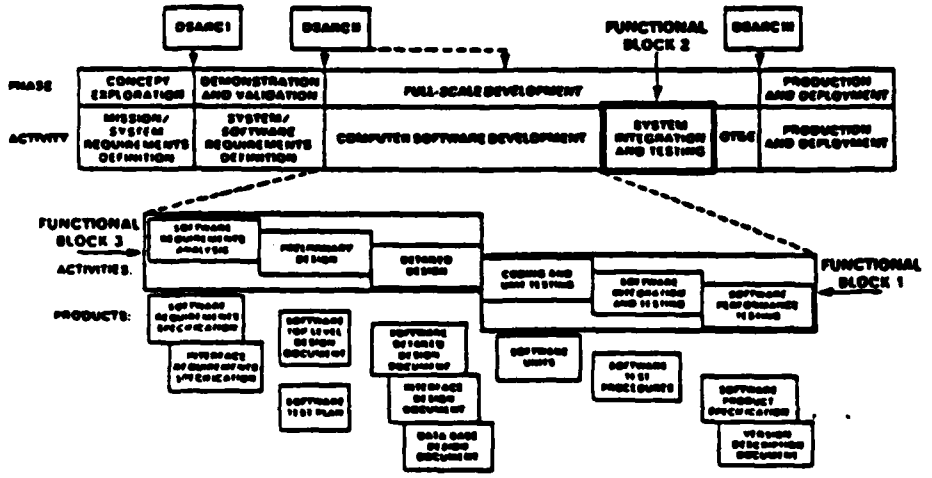
- Key factor is to get performance requirements up front to differentiate between the universal and nonuniversal elements
- Provide a layered unified approach
- Provide new design paradigms
- Partition for reusability

System Engineering Principles

- Implement partitioning methodology which reflects:
  - Isolation of the variable from the generic functions
  - Hardware/processing constraints
  - Single functions tasks
  - Single input/output data sets
  - Independence from preceding/following tasks

- Design for future reusability
  - Approach from a "superset" viewpoint
  - Identify the common core of applications and separate out expected variations

COMPUTER SOFTWARE LIFE CYCLE



3. Software Engineering Technology

- Develop and implement reusability design methodology
  - Design for reusable elements
  - Determine the "Partitioning for Reusability" paradigm
  - Determine the similarities and differences
  - Functionally specify the "generic"
  - Encapsulate the design elements
  - "Add–on" the differences
  - Repeat process for next level

- Develop supporting documentation
  - Structural models
  - Domain analysis techniques
  - Partitioning paradigms

The Reusable Design Methodology:

- Must be developed as an integrated process across the life cycle
- The methodology must drive the tools, not the tools the methodology
- The methodology must not be developed by merely consolidating individual tools to form a "methodology"
- Must be results oriented – look at what is specified via a partitioning discipline, not merely how it is documented by a specification language or a PDL

4. Configuration Control

- Develop processing and dissemination mechanisms
  - Cataloging process
  - Configuration management
  - Certification needs
  - Dissemination process
  - Incentives

5. Education/Training

- Program managers
- System engineers
- Software engineers
- Maintenance engineers
- Military management
- Contractor management

6. Military Management Commitment

- Most important to make it all happen
  - Reusability must become part of the military decision process
  - Require formal consideration of reusability at each decision step in the acquisition process
  - Establish contracting requirements
  - Establish incentives

Conclusions

and

Recommendations

ISTG 84–2 Conclusions

Software Reusability is Technically Feasible

- No technical impediment has been found to reusing if reusability has been incorporated into the design
- In combat systems there is an experience base prior to RNTDS in which reusability occurred frequently in individual cases
  - System engineering contributed to creating conditions
- Past record is that reusability seldom, if ever, crossed corporate/project boundaries
- There is an "up front" investment cost – $, time, system capacity, i.e., engineer for the "superset"
- The cost savings occur in maintenance and subsequent "variants" not in the initial development
- S/W reusability has not been institutionalized

RNTDS Is A Significant First Step

- In formalizing a discipline
- In developing a methodology
- In focusing on S/W reusability

- We found no technical reasons to preclude S/W reusability in C^3

- We found no contractual reasons to preclude S/W reusability in C^3

- **The key to increasing software reusability is military management commitment**

ISTG 84–2 Recommendations

- Navy participation in DoD initiatives, such as STARS, should be strengthened
- Research is needed on design methodologies
- Domain analysis must become part of standard design practice
- Reusability must become part of the military acquisition and review process
- Sub–issues must be addressed such as:
  - Software depository
  - Certification
  - Data rights
  - Configuration management
  - Depository information dissemination
  - Technology transfer

STARS Recommended Activities

1. Develop reusability design methodologies and tools
2. Develop tools to quantify cost savings
3. Integrate reusability into the military acquisition and review process
4. Evaluate issues such as centralized depositories, certification, data rights, and contractor incentives

STARS Recommended Activities (continued)

5. Provide technology transfer into industry practice
   - Establish prototype S/W environment vehicles
   - Apply methodologies/tools on various sized projects
   - Identify methods/tools which enhance productivity and reusability
   - Develop support tools
     - S/W depository data base support tools
     - Graphic displays of components meeting specified I/O criteria
     - Automated documentation generators

Ada* Technology Objectives and Plans (ATOP)

Norman S. Nise
Rockwell International
12214 Lakewood Blvd.
Downey, California 90241

Objective

To develop a uniform management plan that can be applied, industry wide, to the development of Ada reusable packages.

Approach

The management plan will include the following plans:

A. Reusable software plan
B. Configuration management plan
C. Design and development plan
D. Unit testing plan
E. Evaluation and validation plan (E & V)
F. Warehousing plan
G. Programming standards
H. Coordination plan

sp 0.4v

Justification

There is a need for a management plan to unify the development of Ada reusable packages for tools and applications software for space.

Reusable software represents a capital expenditure that generates reduced software development costs in the future. This savings is realized through decreased time in future software development and decreased time for personnel training. A standard management plan that is adopted industry wide is required for the development of reusable packages that themselves are accepted and used industry wide.

Current effort, mainly with DOD, is directed toward the development of tools for use with the APSE. Also, some studies are being done to determine the commonality between missile systems in order to set the bounds for future development of reusable applications software but holes exist, especially in the management of reusable applications software.

The space industry needs a total management plan that will encompass the development and use of both tools and applications software.

Objectives and Plans

In order to reduce the costs and increase the quality of software to be developed on the Rockwell Operational Software Engineering System (ROSES), Rockwell International wants to use reusable Ada packages in the development of that software.

To be accepted industry wide, the reusable software packages must be developed and warehoused under standardized management procedures that are accepted industry. Rockwell International wants to develop these standard management plans which are now detailed:

Reusable Software Plan

* Provide management approach to all plans and efforts
* Set forth the technical approach
* Set forth schedules and work breakdown

Configuration Management Plan

* Develop Ada software configuration management methods to yield a degree of control and tracking during design, development, unit testing, evaluation and verification, and upgrade phases for the reusable software package.
* The software configuration management plan should be automatic and transparent and thus a key tool for the APSE.

Design and Development Plan

* Find general areas of commonality

between software used in aerospace applications.

- Determine functional areas within the general areas that can serve serve as a basis for reusable software.
- For each functional area, determine the level of abstraction that would best serve the purpose of a reusable package.
- Determine objects and programs that would be available to the programmer using the reusable package.
- Determine what objects and associated operations should be declared private.
- Determine whether the unit of reusable software should be designed as a generic package.
- Determine approach to exception handling.
  - ... External or internal to the package
  - ... Methods to improve reusability
- Investigate various software generator systems that take abstract programs and generate code.
- Study effects of target machine dependency.
  - ... Number of bits available for number types
  - ... Available accuracy
  - ... Available character set
  - ... Definition of control characters
  - ... Differences in exceeding bounds
  - ... Differences on relying on equality of floating point numbers for conditional responses
  - ... Differences in dynamic allocation and deallocation effects and timing effects and timing (eg. trying to access code after deallocation has been issued)
  - ... Where should machine dependent statements be placed in the package and what documentation is required
  - ... Effect on handling real-time tasking timing and synchronization because of differences in instruction execution time
- Study operation order dependency
  - ... Intermediate values out of bounds because of changes in operation operation order
  - ... Functions causing side effects by modifying non-local variables that are again used in the function or another expression
  - ... Combination order of library units that use other packages
  - ... Elaboration order of compilation units that use other units
  - ... Real-time tasking problems due to dependency on order of processing jobs in a queue
  - ... Real-time tasking problems due to changes in the order in which tasks are called
- Study Implementation dependency
  - ... The effects of the program changes depending whether parameters are passed or copied
  - ... Requirements imposed by the environment tasking on program parameters
  - ... The effects of pragmas (which are mainly supported by the implementation) on reusability
  - ... The effect of interrupts upon reusability
  - ... Effect of clauses that are implementation oriented (eg. address clause)
- Other
  - ... Effects of machine code insertion or reusability
  - ... Effects of interfacing with other languages on reusability
- Develop a test plan
  - ... Describes scope, approach, resources, scheduling of test activities
  - ... Identifies test items, features to be tested, task, personnel
- Develop test design specification
  - ... Details the test approach and identifies associated tests

- Develop test case specification
  ... Specify inputs, predicted results, conditions

- Develop test procedure specification
  ... Describe sequence of actions to execute a test

- Develop test log
  ... Record of details about execution of tests

- Develop test incident report
  ... reports on any event requiring further investigation

- Develop test summary report
  ... Summarizes test activities and results

Evaluation and Validation Plan (as per Wright-Patterson or NASA JSC)

- Develop evaluation and validation requirements
  ... Methods, practices, standards, etc., that drive E&V development plans

- Develop evaluation and validation criteria
  ... Specify hurdles that reusable software packages must pass

- Develop tools
  ... Software required to carry out the E&V function

- Develop procedures
  ... Methods of E&V to ensure software meets criteria

- Develop documentation
  ... To ensure publication of methods, practices, tracking

WAREHOUSING PLAN

- Develop methods of cataloging and indexing for the library of reusable software packages

- Develop standardized supporting documentation for reusable packages that will describe the function, importable objects, exportable objects, side effects, and operating including:
  a. Allowable ranges of values for objects used by the package
  b. Expected values of objects calculated by the package
  c. Units of measure
  d. Accuracy requirements
  e. Description of algorithms
  f. Numerical conventions including significant digits rounding, truncation, etc.
  g. Exception handling
  h. Safety handling

- Develop standardized presentation of documentation
  a. Flow
  b. Nassi-Shneidermangrams
  e. Data structure before and after picture

- Develop tools for automatic reusable package selection, linking and presentation of documentation

- Develop methods to obtain and use reusable software

Programming Standards Development Plan

- Develop programming standards that will be used during development and modification of reusable packages to provide standardization for analysis, traceability, maintain ability, and quality assurance

- Develop tools for auditing reusable packages against the programming standards

- Perform audits of the code against the programming standards

- Develop documentation to describe standards and audit procedures

- Provide for information exchange within the private community

- Provide for information exchange with DoD

- Keep up to date reference library on all activity community wide dealing with reusable packages

* Promote the use of developed reusable packages

Review and Reporting

(1) Weekly reports will be available under the normal procedures followed by systems/software group.

(2) Preliminary documents describing each of the above mentioned plans will be available for review. The exact review procedure will be determined and will exist over a period of 3 months.

(3) Final documentation will be available after the review.

(4) Final documents will be updated once a year based upon experience with the development of reusable software.

(5) Monthly reports describing any suggestions or problems based upon the use of the management system will be issued after the first version of each plan has been approved.

Norman S. Nise
1444 Sunview Drive
Orange, California 92665

EXPERIENCE

SOFTWARE ENGINEER, ROCKWELL INTERNATIONAL from June 1980 to present. Implemented the Space Shuttle Backup Flight System (BFS) programming standards including the development of auditing techniques and procedures. Performed audits of the Shuttle BFS code against the programming standards.

Developed the programming standards for the Rockwell Operational Software Engineering System (ROSES).

Initiated and implemented ROSES Ada Technology Objectives and Plans (ATOP) for NASA in Reusable Software Management. Gave various presentations on the subject.

Instrumental in assisting the Software Productivity Consortium's (SPC) designing and implementing a Reusable Software project. SPC is a consortium of 11 major aerospace companies. The objective of SPC is to reduce software development costs for the aerospace industry.

ENGINEER, HUGHES AIRCRAFT CO. from June 1961 to May 1970. Performed technical feasibility studies concerning the phase stabilization of traveling wave tube phase-shifters, and the application of state-space techniques to system design.

Performed analysis in the areas of automatic detection techniques, moving target indicator digital cancelers, effect of noise in sampled-data loops, missile acceleration and data rate requirements, and miss distance.

Performed analog computer simulations of missile control loops.

Developed digital simulations of target engagements by performing subsystem modeling.

PROFESSOR, CALIFORNIA STATE POLYTECHNIC UNIVERSITY, POMONA, CA. from September 1963 to present. Full Professor since 1973. Taught courses in computer architecture, microcomputers, control systems including digital and nonlinear, network theory, communication theory, electronics, field theory, and analog computers.

Reusable Software Management

- ☐ INTRODUCTION
- ☐ SUGGESTED APPROACH
- ☐ ACCOMPLISHMENTS & PLANNING
- ☐ REFERENCES

Rockwell International

INTRODUCTION

Purpose of the Briefing

- TO STATE THE NEED FOR REUSABLE SOFTWARE
- TO DEFINE THE PROBLEM OF THE DEVELOPMENT OF REUSABLE SOFTWARE
- TO SUGGEST AVENUES OF APPROACH TO THE SOLUTION OF THE PROBLEM
- TO SUMMARIZE STEPS TAKEN BY ROCKWELL INTERNATIONAL

35SSV1S0002A

INTRODUCTION

Problem

- SOFTWARE COSTS WILL BE 5 TIMES THAT OF HARDWARE BY 1990
- REUSABLE SOFTWARE HAS NOT BEEN USED IN THE PAST TO SOLVE THE PROBLEM BECAUSE OF:
  - LACK OF A UNIVERSALLY ACCEPTED PROGRAMMING LANGUAGE THAT LENT ITSELF TO DESIGN CRITERIA OF REUSABLE MODULES
  - LACK OF ACCEPTANCE THAT THE IDEA OF REUSABLE SOFTWARE COULD WORK
  - LACK OF STANDARDIZATION — DEFINITIONS, DESIGN APPROACHES, DOCUMENTATION, LIBRARY ACCESS METHODS
  - PROPRIETARY INTERESTS RESULTING IN LACK OF COOPERATION AMONG INDUSTRY

INTRODUCTION

Impacts

- THE SINGLE LANGUAGE PROBLEM HAS NOW BEEN SOLVED WITH THE DEVELOPMENT OF Ada*
- COMMUNITY ACCEPTANCE OF THE NEED FOR REUSABLE SOFTWARE IS EVIDENCED BY:
  - SOFTWARE PRODUCTIVITY CONSORTIUM (SPC)
  - NAVY'S SYMPOSIUM ON REUSABLE SOFTWARE (REUSABLE SOFTWARE IMPLEMENTATION PROGRAM)
  - AIR FORCE COMMON Ada MISSILE PACKAGE (CAMP) PROJECT
  - ITT WORKSHOP ON REUSABILITY IN SEPTEMBER 1983
  - WORK DONE BY SUCH PEOPLE AS PARNAS TO LAY GROUNDWORK FOR DESIGN CRITERIA FOR REUSABLE MODULES

*Ada IS A REGISTERED TRADEMARK OF THE DEPARTMENT OF DEFENSE (AJPO)

INTRODUCTION

Definition of Reusable Software

- SOFTWARE PACKAGES THAT ARE REUSED IN DIFFERENT APPLICATIONS (WITH LITTLE OR NO MODIFICATIONS)
- SOFTWARE PACKAGES THAT ARE REUSED IN MODIFIED VERSIONS OF THE SAME SOFTWARE PROGRAM
- SOFTWARE PACKAGES INCLUDES SPECIFICATIONS, DESIGNS, DATA, CODE, TEST CASES, AND DOCUMENTATION
- SOFTWARE THAT CAN BE USED ON DIFFERENT MACHINES (PORTABILITY)

INTRODUCTION

Examples

1. PREDEFINED PACKAGES

- BOOLEAN OPERATIONS
- LOGIC OPERATIONS
- EQUALITIES—INEQUALITIES
- ASCII CHARACTER DEFINITIONS
- STRING OPERATIONS
- CALENDAR
- I/O OPERATIONS

Rockwell International

84SSV154120

INTRODUCTION

Examples (Cont)

2. MATH PACKAGE

- VECTOR—MATRIX OPERATIONS (MXN, M+N, INVERSION, EIGENVALUES, ETC.)
- ARITHMETIC OPERATIONS (MIDVAL, SIGN, SIGNUM, ETC.)
- ARRAY OPERATIONS (MAX(X), SUM(X), ETC.)
- CHARACTER OPERATIONS (LENGTH (C), ETC.)
- CURVE FITTING & DATA SMOOTHING
- STATISTICS
- FUNCTION APPROXIMATION & MINIMIZATION TECHNIQUES
- SOLUTION OF EQUATIONS
- TRIGONOMETRY
- CALCULUS & DIFFERENTIAL EQUATIONS

INTRODUCTION

Examples (Cont)

3. NAVIGATION & CONTROL

- COORDINATE TRANSFORMATIONS
- HEIGHT-ABOVE-REFERENCE ELLIPSOID
- GRAVITY GRADIENT
- RELATIVE VELOCITY & POSITION CALCULATIONS
- COMPUTATION OF APOGEE & PERIGEE HEIGHT
- EARTH'S CENTRAL FORCE OF ATTRACTION
- STATE VECTOR PROPAGATION
- COVARIANCE MATRIX PROPAGATION
- INTEGRATION OF STATE EQUATIONS
- GRAVITATIONAL & DRAG ACCELERATION COMPUTATIONS
- EARTH'S GRAVITATIONAL ATTRACTION MODEL
- KALMAN FILTER
- CONSTANTS (EARTH RATE, EARTH DIAMETER, ETC.)
- DIGITAL FILTERS
- TACAN RANGE & BEARING
- POSITION, VELOCITY, ACCELERATION PREDICTION

INTRODUCTION
Examples (Cont)

4. **SYSTEMS PACKAGES**
   - TIME RESPONSE
   - FREQUENCY RESPONSE
   - STABILITY
   - FEEDBACK CONTROL SYSTEM AIDS
5. **TOOLS**
   - APSE
6. **OTHER**

INTRODUCTION

Characteristics of Reusable Software

- REUSABLE SOFTWARE IS INITIALLY CAPITAL-INTENSIVE
  - LARGE CAPITAL OUTLAY TO DEVELOP FOLLOWED BY REDUCED COSTS & INCREASED PROFITS
  - REPRESENTS A TANGIBLE ASSET THAT INCREASES PRODUCTIVITY
- REUSABLE SOFTWARE AVOIDS DUPLICATION
  - WE DO NOT HAVE TO "REINVENT THE WHEEL"
- REUSABLE SOFTWARE CAN BE REUSED WITH LITTLE OR NO MODIFICATIONS
- INCREASED SOFTWARE QUALITY
- ADVANTAGES OF REUSABLE SOFTWARE DESIGN AND DEVELOPMENT
  - STANDARD APPROACHES
  - WELL UNDERSTOOD
  - MODIFICATIONS MINIMIZED
  - VERIFICATION MINIMIZED
  - ANALYSIS MINIMIZED
- DECREASED TIME FOR PERSONNEL TRAINING
  - DECREASED LEARNING CURVES

INTRODUCTION

Attributes of Reusable Software

- DESIRED LEVEL OF ABSTRACTION
- INFORMATION HIDING
- MODULARITY & LOCALIZATION
- MINIMIZATION & PROTECTION OF INTERFACES
- UNIFORMITY

Rockwell International

15SSV158693

INTRODUCTION

ADA* for Reusable Software

ADA PROVIDES THE ATTRIBUTES OF REUSABLE SOFTWARE via THE FOLLOWING RESOURCES

- PACKAGES
  - DECLARATIONS
  - RELATED PROGRAM UNITS
  - ABSTRACT DATA TYPES
  - ABSTRACT STATE MACHINES
- GENERIC PACKAGES
- PRIVATE TYPES
- STRONG TYPING
- MANAGING NAME SPACE

*ADA IS A REGISTERED TRADEMARK OF THE DEPARTMENT OF DEFENSE (AJPO)

12

Rockwell International

INTRODUCTION

Keys to Problem Solution

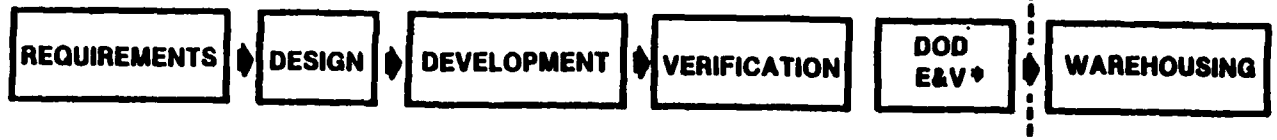
- USE A SINGLE LANGUAGE — ADA
- PROVIDE FOR DESIGN CRITERIA THAT ENSURE REUSABILITY
- PROVIDE FOR DESIGN, DEVELOPMENT & TESTING UNDER CONFIGURATION CONTROL & MANAGEMENT
- PROVIDE FOR A WAREHOUSING SCHEME FOR POOLING & DISTRIBUTING REUSABLE SOFTWARE
- PROVIDE FOR INFORMATION EXCHANGE, EDUCATION, COORDINATION

Reusable Software Management

- ☐ INTRODUCTION
- ☐ SUGGESTED APPROACH
- ☐ ACCOMPLISHMENTS & PLANNING
- ☐ REFERENCES

SUGGESTED APPROACH

Tracing the Development of Reusable Software



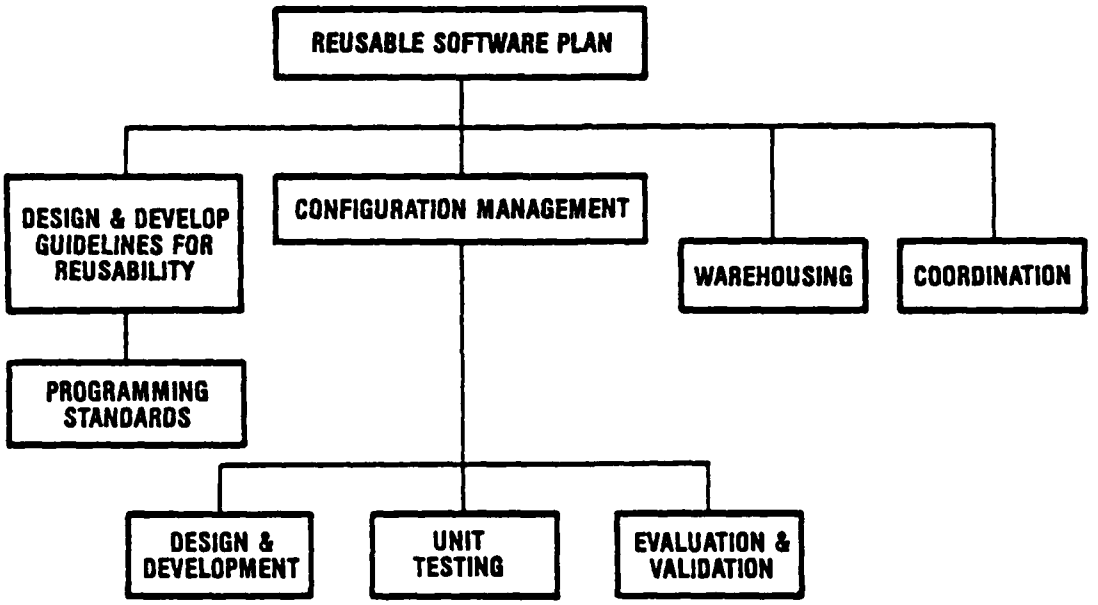
UNDER CONFIGURATION CONTROL & MANAGEMENT

DELIVERABLE

* E&V - EVALUATION & VALIDATION
WHERE VALIDATION - END-TO-END SYSTEM TEST

SUGGESTED APPROACH

Reusable Software Plan



Rockwell International

15SSV158696

SUGGESTED APPROACH

Reusable Software Plan

- PROVIDE MANAGEMENT APPROACH TO ALL PLANS & EFFORTS
- SET FORTH THE TECHNICAL APPROACH
- SET FORTH SCHEDULES & WORK BREAKDOWN

Rockwell International

17

84SSV154110

SUGGESTED APPROACH

Configuration Management Plan

- DEVELOP SOFTWARE CONFIGURATION MANAGEMENT METHODS TO YIELD A DEGREE OF CONTROL & TRACKING DURING DESIGN, DEVELOPMENT, UNIT TESTING, EVALUATION & VALIDATION, & UPGRADE PHASES FOR THE REUSABLE SOFTWARE PACKAGE

- THE SOFTWARE CONFIGURATION MANAGEMENT PLAN SHOULD BE AUTOMATIC & TRANSPARENT & THUS A KEY TOOL FOR THE APSE

35SSV1541118

SUGGESTED APPROACH

Design and Development Plan

SET GUIDELINES FOR DETERMINING:

- GENERAL AREAS OF COMMONALITY BETWEEN SOFTWARE USED IN AEROSPACE APPLICATIONS
- FUNCTIONAL AREAS WITHIN THE GENERAL AREAS THAT CAN SERVE AS A BASIS FOR REUSABLE SOFTWARE
- LEVEL OF ABSTRACTION THAT WOULD BEST SERVE THE PURPOSE OF A REUSABLE PACKAGE
- OBJECTS & PROGRAMS THAT WOULD BE AVAILABLE TO THE PROGRAMMER USING THE REUSABLE PACKAGE
- WHAT OBJECTS & ASSOCIATED OPERATIONS SHOULD BE DECLARED PRIVATE
- WHETHER THE UNIT OF REUSABLE SOFTWARE SHOULD BE DESIGNED AS A GENERIC PACKAGE
- APPROACH TO EXCEPTION HANDLING

SUGGESTED APPROACH

Design and Development Plan (Cont)

ALSO:

- INVESTIGATE VARIOUS SOFTWARE GENERATOR SYSTEMS THAT TAKE ABSTRACT PROGRAMS & GENERATE CODE
- STUDY EFFECTS OF TARGET MACHINE DEPENDENCY
- STUDY OPERATION ORDER DEPENDENCY
- STUDY IMPLEMENTATION DEPENDENCY
- OTHER

SUGGESTED APPROACH

Programming Standards Development Plan

- DEVELOP PROGRAMMING STANDARDS THAT WILL BE USED DURING DEVELOPMENT & MODIFICATION OF REUSABLE PACKAGES TO PROVIDE:
  - STANDARDIZATION
  - TRACEABILITY
  - MAINTAINABILITY
  - QUALITY
- DEVELOP TOOLS FOR AUDITING REUSABLE PACKAGES AGAINST THE PROGRAMMING STANDARDS
- PERFORM AUDITS OF THE CODE AGAINST THE PROGRAMMING STANDARDS
- DEVELOP DOCUMENTATION TO DESCRIBE STANDARDS & AUDIT PROCEDURES

SUGGESTED APPROACH

Unit Testing Plan

DEVELOP:

- A TEST PLAN
- TEST DESIGN SPECIFICATION
- TEST CASE SPECIFICATION
- TEST PROCEDURE SPECIFICATION
- TEST LOG
- TEST INCIDENT REPORT
- TEST SUMMARY REPORT

SUGGESTED APPROACH

Evaluation and Validation Plan*

DEVELOP:

- E&V REQUIREMENTS
- E&V CRITERIA
- TOOLS
- PROCEDURES
- DOCUMENTATION

*AS PER WPAFB, NASA JSC, NRL, DARPA OR STARS

Rockwell International

SUGGESTED APPROACH

Warehousing Plan

- DEVELOP METHODS & TOOLS FOR CATALOGING & INDEXING THE LIBRARY OF REUSABLE SOFTWARE PACKAGES
- DEVELOP STANDARDIZED SUPPORTING DOCUMENTATION FOR REUSABLE PACKAGES THAT WILL DESCRIBE THE FUNCTION, IMPORTABLE OBJECTS, EXPORTABLE OBJECTS, SIDE EFFECTS, & OPERATIONS
- DEVELOP STANDARDIZED PRESENTATION OF DOCUMENTATION
- DEVELOP TOOLS FOR AUTOMATIC REUSABLE PACKAGE SELECTION, LOADING, LINKING, & PRESENTATION OF DOCUMENTATION
- DEVELOP METHODS TO OBTAIN & USE REUSABLE SOFTWARE
- STUDY POSSIBILITY OF SETTING UP CLASS HIERARCHIES
- DEVELOP INSTRUCTIONAL PROGRAMS TO AID IN THE USE OF THE LIBRARY
- DEVELOP TECHNIQUES TO SET UP & ENFORCE ACCESS RIGHTS

SUGGESTED APPROACH

Coordination Plan

- PROVIDE FOR INFORMATION EXCHANGE WITHIN THE PRIVATE COMMUNITY
- PROVIDE FOR INFORMATION EXCHANGE WITH DOD, NASA, & OTHER GOVERNMENT AGENCIES
- KEEP UP TO DATE REFERENCE LIBRARY ON ALL ACTIVITY COMMUNITY-WIDE DEALING WITH REUSABLE SOFTWARE
- PROMOTE THE USE OF DEVELOPED REUSABLE PACKAGES

84SSV154118

Reusable Software Management

- ☐ INTRODUCTION
- ☐ SUGGESTED APPROACH
- ⇨ ☐ ACCOMPLISHMENTS & PLANNING
- ☐ REFERENCES

Rockwell International

3555V156691A-3

ACCOMPLISHMENTS

What We Have Accomplished

FY 1984

- INITIAL DRAFT OF SOFTWARE CONFIGURATION MANAGEMENT PLAN RELEASED FOR REVIEW
- INITIAL DRAFT OF PROGRAMMING STANDARDS RELEASED FOR REVIEW

FY 1985

- BEGAN FORMULATION OF Ada REUSABLE PROGRAMMING STANDARDS
- AWARDED REUSABLE SOFTWARE ATOP OUT OF NASA/JSC
- DISTRIBUTED STANDARDS TO OTHER ROCKWELL DIVISIONS
  - B-1B
  - AUTONETICS
  - ROCKETDYNE
  - COLLINS — SANTA ANA
  - COLLINS — CEDAR RAPIDS

3SSSV161483

ACCOMPLISHMENTS

Ada* Training Accomplishments

- DR. McKAY'S PRESENTATION TO UPPER MANAGEMENT
- SOFTECH ANALYSIS OF ROSES Ada TRAINING REQUIREMENTS
- ACADEMIC ACCREDITATION OF IN-HOUSE Ada TRAINING PROGRAM APPROVED
  - CAL POLY POMONA TO PRESENT SOFTWARE ENGINEERING IN Ada IN REAL-TIME VIDEO BY Ada EXPERTS FROM ANYWHERE IN U.S.
- TWO "INTRODUCTION TO Ada" CLASSES (OVER 20 GRADUATES)
- $100K Ada PRESENTATION COSTS (FY 1985) APPROVED
- NYU Ada IN-HOUSE
- DEC Ada FIELD TEST SITE
- THREE SOFTWARE ENGINEERS ATTEND TELESOFT Ada WORKSHOP

*Ada IS A REGISTERED TRADEMARK OF THE UNITED STATES DEPARTMENT OF DEFENSE (AJPO)

Rockwell International

3685V184066

ACCOMPLISHMENTS

Rockwell International Reusable Software Contract

- SPACE TRANSPORTATION SYSTEMS DIVISION (STSD) DEVELOPED SPACE SHUTTLE APPLICATION SOFTWARE UNDER MARTIN MARIETTA CORPORATION FOR THE DEPARTMENT OF THE AIR FORCE — HQ DIVISION (AFSD)
  - DEVELOPED 15 COMPUTER PROGRAM CONFIGURATION ITEMS TO CHECKOUT & LAUNCH SPACE SHUTTLE VEHICLES AT THE VANDENBERG LAUNCH & LANDING SITE (VLS)
  - REUSED EXISTING KENNEDY SPACE CENTER SOFTWARE TAILORED TO MEET VLS UNIQUE REQUIREMENTS & HARDWARE DIFFERENCES
    - SAMPLE PROGRAM PACKAGE INCLUDED:
      - SYSTEM MANAGER
      - OPS MANAGERS
      - SCHEDULERS
      - DISPLAYS
      - SEQUENCERS
      - TASK
      - COMPONENT

Rockwell International

35S3V161482

PLANNING

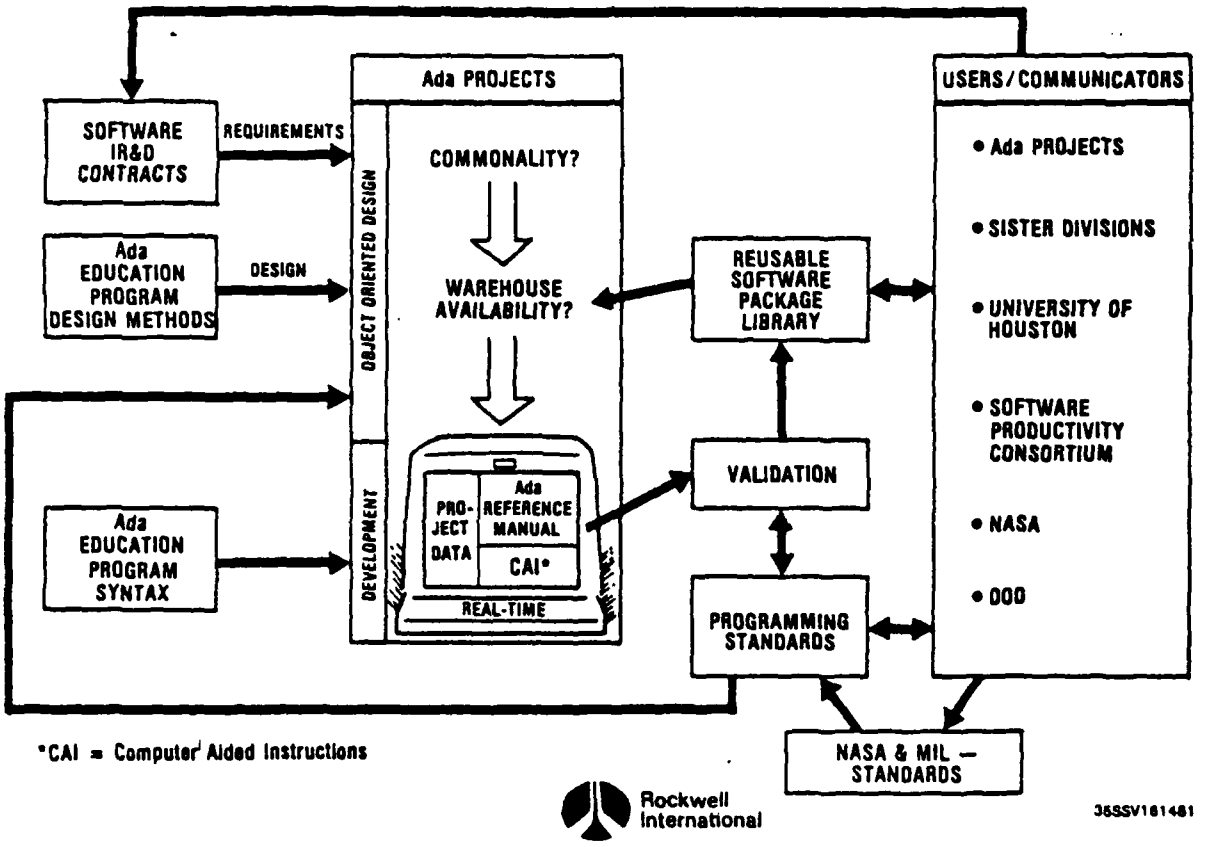
Reusable Software Projects

1. CONTINUE REVIEWING REUSABLE SOFTWARE LITERATURE
2. CONTINUE TO WRITE Ada REUSABLE STANDARDS
3. DISTRIBUTE REUSABLE STANDARDS TO OTHER ROCKWELL DIVISIONS
4. CONTINUE OBTAINING REUSABLE PACKAGES
5. TEST REUSABLE PACKAGES AGAINST STANDARDS
6. LOAD ACCEPTABLE PACKAGES INTO ROSES
7. TEACH REUSABLE SOFTWARE PRINCIPLES & METHODS
8. MEASURE REUSABLE SOFTWARE PACKAGE USE
9. CRITIQUE TECHNICAL APPROACH
   (NEED FOR DEVELOPMENT OF LIBRARY ACCESS TOOL)
   (NEED FOR PROOF OF CONCEPT APPROACH)

3535V184089A

ROSES

Ada Reusable Software Package System (Evolutionary)



*CAI = Computer Aided Instructions

Rockwell International

35SSV161481

Reusable Software Management

- ☐ INTRODUCTION
- ☐ SUGGESTED APPROACH
- ☐ ACCOMPLISHMENTS & PLANNING
- ☐ REFERENCES

Rockwell International

365SV156691A-4

References

1. P. C. Clements, R. Alan Parker, David L. Parnas & John Shore, "A Standard Organization for Specifying Abstract Interfaces," Naval Research Laboratory, June 14, 1984

2. "Reference Manual for the Ada Programming Language," MIL-STD 1815A, 22 Jan 1983

3. Booch, G., "Software Engineering With Ada," The Benjamin/Cumming Co., 1983

4. Nissen & Wallis, "Portability & Style in Ada," Cambridge University Press, 1984

5. "Reusable Software," Electrical Design News, February 3, 1983

6. "Common Ada Missile Packages (CAMP)," Rfp, Eglin AFB, March 21, 1984

7. Bruno Witte, "Checklist for Ada Math Support Priorities," ACM Ada Letters, March, April 1984

8. "Evaluation & Validation Plan," Version 1.0, Wright-Patterson AFB, 30 November 1983

9. Wegner, "Capital-Intensive Software Technology," IEEE Software, July 1984

10. "Proceedings of the Workshop on Reusability in Programming," September 7-9, 1983, ITT

11. "Strategy for a Software Initiative," Department of Defense, 1 October 1982, Appendix II

12. McDonald, Jordan, & Schaar, "Concept Paper for the Development of a DOD Ada Software Engineering Education and Training Plan," Institute for Defense Analyses, Ida Memorandum Report M-7, November, 1984

13. Freeman, P., "Reusable Software Engineering: Concepts and Research Directions," Proceedings of the Workshop on Reusability in Programming, 1983

14. Parnas, D., "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, March 1979

15. Parnas, D., "On the Criteria to Be Used in Decomposing Systems into Modules," Communications of the ACM, December, 1972

WORKSHOP ON REUSABLE COMPONENTS OF APPLICATION PROGRAMS

A. FredericK Rosene

Communication Systems Division
GTE Government Systems Corp
Needham Streets, MA 02194-9123

Abstract

The attached paper, The STEP System, is submitted to support GTE's Communication Systems Division participation in the Workshop on Reusable Components of Application Software. STEP (Structured Techniques for Engineering Projects) is an integrated software development environment based on a methodology and software design architecture that expedites the development of reusable design and code. The paper on STEP addresses many of the issues and questions included in your CBD request for participation. A cross-reference between the issues and the paper is given below.

1. Specifications/Design

STEP methodology is summarized in section 1, the STEP Analyze command enforces standards described on page 6.

2. Reusable Component Definitions

The database description on pages 3 and 4 describes how and what documentation STEP maintains. The association of documentation with components is inherent in the data structures as well as associations among components (see the Allocate command on page 5). When software decomposition is defined to STEP, the testing documentation components are created by STEP so there is always a known association between design and test documentation. As pointed out on page 6 the Configuration Management backbone of STEP allows for many revisions of a component and insures the user is aware of and does not violate the framework within which he or she is working.

3. Validation of Software Components

STEP relies heavily on the Analyze Command (page 5) and the standards it enforces (page 6) to insure structured, syntactically correct code that follows the specified high level design (pages 9 - 11). Testing time has significantly decreased using STEP (page 12) and resulting code has proved to be flexible and maintainable.

4. Library Experience

STEP is designed around a configuration backbone (pages 8 and 9). In addition all commands rights (Table 1) and subcommands rights may be granted or denied to a user. Nodes (pages 3 and 4) may be designated as archive nodes and hence provides users with on-line access to existing code that is in the form acceptable to STEP. All source and documentation of a component may be transferred from an archive to a development node by executing a single append command.

5. Automated Parts Composition

The Compile Command (page 5) selects components and builds load file releases. It insures that a consistent and complete set of components are available for compiling. The selection of component revisions which is also part of this process is automatic and controlled by predefining the selection rules to be applied (page 9).

6. Logistics of Reuse

Our experience using STEP, and reusing the generic architecture on several projects has been positive (page 12). Software personnel may be moved among projects, new personnel get up to speed quickly, and redesign of executives and creation of new untested software architectures avoided.

7. Encouraging Deposits

The Management Policy and Procedures require the use of STEP and its associated STEP Architecture. No specific incentives are in place to encourage deposit of reuse.

8. ADA Experience

Ada support on STEP is almost complete. A major concern to us is the inefficiency of Ada tasking. The generic executive is being reprogrammed in Ada.

We feel our experience in developing and using STEP (8 years) and the generic STEP architecture (12 years) on which it is based will allow us to make a valuable contribution to your workshop.

STEP is written in PASCAL and runs on DEC20-TOPS20, VAX-VMS, and IBM-MVS. Limited data rights are given to our customers who wish to maintain their source and documentation on STEP. Other arrangements are negotiable. STEP was developed both on GTE Corporate and on IR&D funds.

THE STEP SYSTEM

1.0 Introduction

In 1976 GTE's Communication Systems Division began creating a software development environment called "Phoenix". A few years later, the project obtained corporate support and the name was changed to "STEP" (Structured Techniques for Engineering Projects). Today CSD policy calls for the STEP system to be used in all major software development programs in the division. STEP is continually being improved and expanded in response to user feedback and software engineering progress. This paper describes STEP from the user's perspective and from a design perspective. In addition, it reviews our experience in using STEP for over six years.

Section 2 gives an overview of STEP. Section 3 describes the architectural framework on which the STEP environment is based. Section 4 describes the structure of the database, the core system and the tools of the STEP system. Finally, Section 5 describes some reactions to and experience in using STEP.

2.0 Overview

STEP is an automated software development environment?1O which helps produce reliable software; it is both a life-cycle methodology and an integrated system of computer programs which automate, control, support and enforce the methodology. It meets most of the requirements of an Ada environment as described in the STONEMAN document?2O.

The STEP methodology is a set of design steps that results in a controlled, structured approach to software system development. Figure 1 shows the phases into which the system life cycle is divided.

A measurable result, called a milestone, marks the end of each phase. At each milestone someone in authority must certify that the project is ready to advance to the next phase. The key features of the methodology are:

- a standardized software architecture for projects
- enforcement of standards throughout the development
- documentation that is concurrent with development
- top-down structured programming for all development
- periodic reviews for early detection and correction of errors.

The STEP system combines human procedures with software tools, a database and an interactive computer in order to structure:

- software engineering
- software quality assurance
- configuration control
- project management.

STEP is founded on the theory that a large software project is a group effort and thus information must be accessible to everyone. Progress is recognized only when it is reflected in the project's database, and

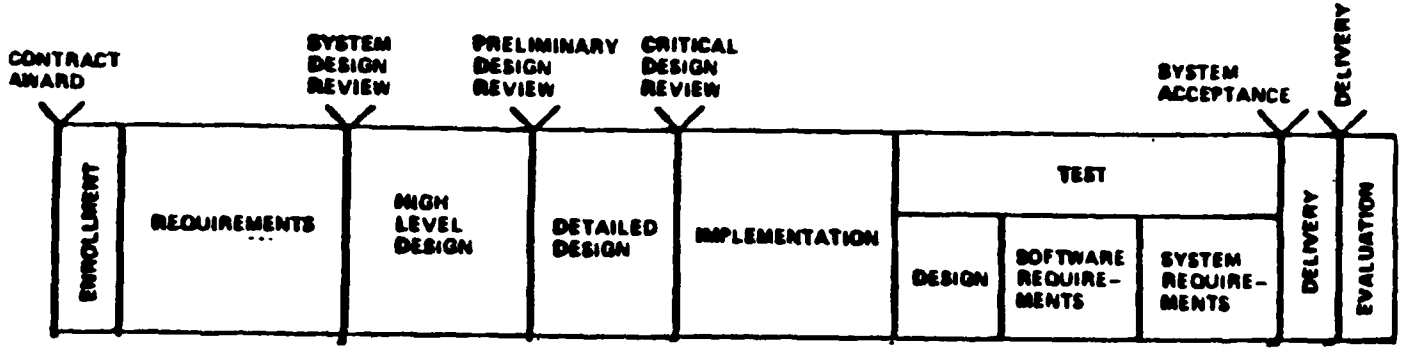


Figure 1
Software Development Phases

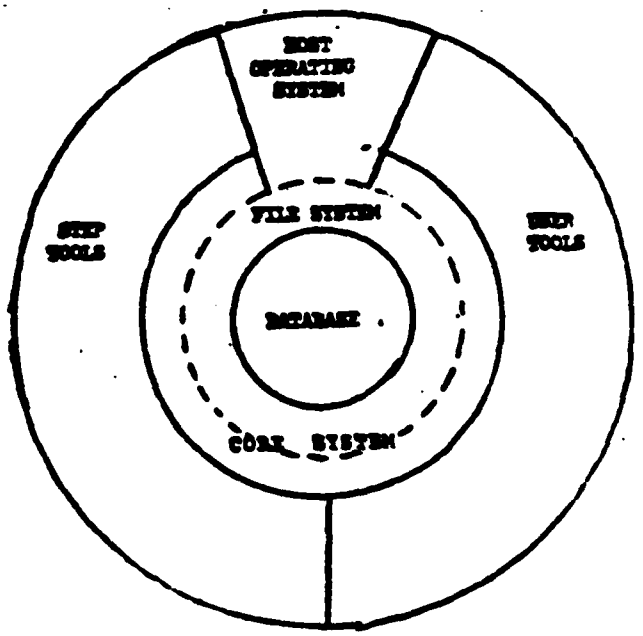


Figure 2
STEP Environment

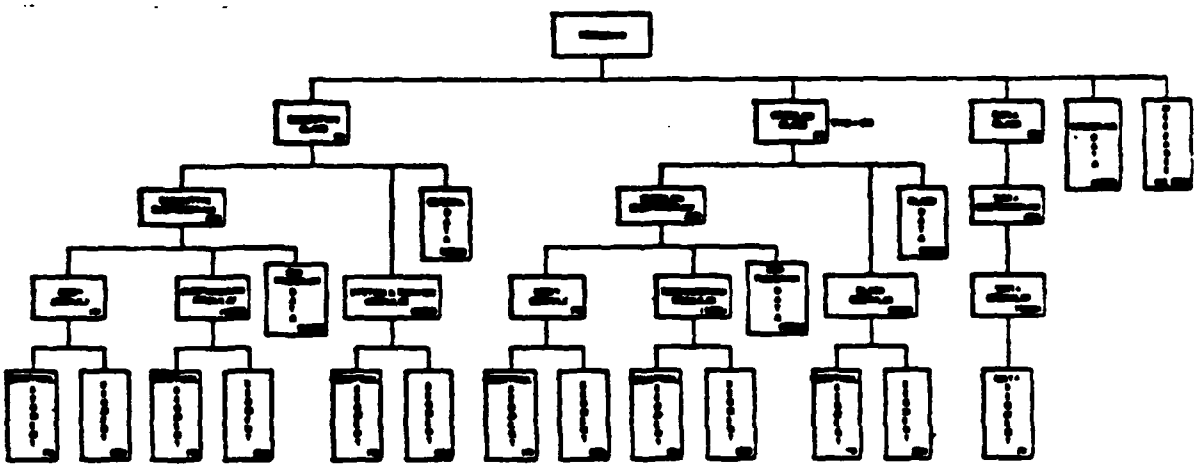


Figure 3
STEP Architecture

reviews and audits are performed only on database contents.

3.0 Software Architecture

The heart of the system is an architectural framework(3,4) which is designed on time-sharing principles. That is, the program is partitioned into subprograms, each of which has a control interface with an executive which schedules its use.

The architectural template, shown in Figure 2, is a generalization of the classical time-sharing approach, which allows tradeoffs between efficiency and isolation for specific applications. It can expand to include the most complicated real-time system or compress to meet simpler requirements. Mapping a project to this architecture is a major part of high level design.

Figure 3 shows the STEP phases divided into the three levels of development.

As the project moves to lower levels, the program is divided into smaller and smaller pieces, until at the lowest level, implementation occurs (i.e., coding in Pascal, Ada, Chill,...). Integration takes place when the project reaches the testing phases, proceeding from design to software test to system level.

Because STEP understands the hierarchical relationship between architecture elements and establishes a common architecture for all GTE software, it has the following benefits:

- Increased probability that each project is on a solid footing.
- Configuration and project management controls may be applied, as a function of architecture.
- The exact state of any architectural subdivision may be documented.
- Personnel may be moved from project to project, even location to location, with little loss of productivity.
- A new architecture is not reinvented for each new project.
- The architecture is enforced during implementation.
- Testable, maintainable, and hence reliable software.

The accumulation of metrics is a byproduct of combining a standard architecture and database. Information that assists in predicting program testability and maintainability before the detailed design is begun(4), and information that can be used to improve the development process on future projects, is easily kept in the database. Furthermore, it is possible to produce performance data down to the level of each user. The only cost is efficiency: The more data collected, the more processing required per user action.

4.0 Structure

Figure 4 shows the three principal features of the STEP system:

- the database
- the core system
- the tools - both STEP and user-supplied

Database

Database - Each project has its own database, which is the central storage for all information associated with that project throughout its life cycle. A named collection of information in the database is known as a component; for example, a module or a subsection of a document. Each component stored in a project database is related to the architectural framework and the methodology. A module's relationship determines what can call it, what it may call, and what data it may reference. A document component is related to the methodology in two ways: The point in the life cycle when it can be created, and the point at which it must be completed. Also, users may instruct the system to maintain additional relationships among components. For example, an association may be maintained between requirement document components and software architecture components.

The database for a project is a tree structure of nodes in which each node can store all components needed during the life cycle of a project. In fact, small and medium-sized projects normally require only one node. Multiple node structures are used for several reasons:

Efficiency - if a node gets too large data retrievals take too long.

Security - access is assigned by node as well as by function.

Support - database backups and audits get cumbersome if a node exceeds the capacity of one disk drive.

Resources - large projects require more computing resources than one computer can

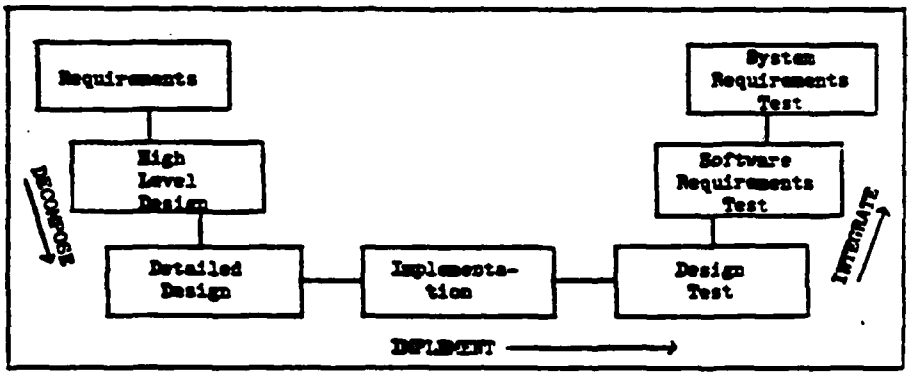


Figure 4
Development Levels of STEP Phases

provide. Nodes or subnets of of nodes may be on different computers.

Each node is divided into five volumes:

Volume 1 contains skeleton files used to initialize files in the other volumes, and instruction files that can be displayed at the user's option. Files in Volume 1 may be modified by authorized users but may not be deleted or created.

Volume 2 contains user aid files. These files are used to predefine commands and parameters to be run either on line or in batch. They also are used to store the rules by which components are selected.

Volume 3 contains all documentation associated with the software lifecycle other than test and software design documentation. This includes requirements, hardware description, user manuals, maintenance manuals, etc.

Volume 4 contains all test documentation including test plans, test procedures and test results.

Volume 5 contains high level design and detailed design documentation and source code.

The contents of Volumes 2, 3 and 5 are controlled by the user through the use of DEFINE, DATA, DELETE and EDIT commands. Components of Volume 4 track the structure created in Volume 5. The files with each component vary with volume, and within Volume 5 with function and source languages used.

Core System

The Core system consists of:

- STEP commands
- Development standards
- File system
- Configuration management

STEP Commands

The STEP commands provide a variety of capabilities associated with a project's database. After logging into the host operating system, a user runs STEP by entering the word "STEP" and a node ID. Now STEP commands and normal operating system commands may be executed. STEP commands, which in turn affect the node database, allow the user to create, modify, display, document and delete information. Questions regarding who has done what and where are easily resolved by up-to-date reports based on information in the project's database. Table 1 summarizes the commands available.

TABLE 1

STEP COMMANDS

Commands which create components

DEFINE - creates all components in volumes 2 to 5 except high level data

DATA - creates high level data components (in volume 5)

Commands which insert information

EDIT - modifies contents of any file of any component

ALTER - global edit or replacement across any or all components

ALLOCATE - inserts data that associates components

CERTIFY - marks external progress, e.g., a successful review

Commands which generate data

ANALYZE - checks syntax, enforces standards and generates reference data

COMPILE - selects components for a release, builds source for a release, exports source ready to compile and/or object for source already compiled

Commands which monitor and control a project

STATUS - estimates, stores current status and reports on status

TRACK - creates and maintains an audit trail of changes

PHASE - controls when phases of project are started

GLOSSARY - maintains a naming glossary of all mnemonics used in source code

Commands which view the state of components

BROWSE - selectively views and searches files

TYPE - displays any file of any component

PRINT - prints any file of any component

INFORMATION - displays state data for any component

Commands which create documents

DOCUMENT - selects, combines and processes STEP components

PROLOG - generates a high level - detailed design document

REPORT - provides information on the status of a node

Commands which provide file handling capabilities

IMPORT - transfers a file from host operating system to STEP

EXPORT - transfers a file from STEP to host operating system

DOWNLOAD - transfers files from STEP to a workstation

UPLOAD - transfers files from a workstation to STEP

APPEND - appends STEP files

CHANGE - changes names of STEP files

DELETE - deletes STEP files

Commands which aid users

EXECUTE - allows step to run without user input

SUBMIT - allows STEP to run in batch

CALL - allows users to interface tools with STEP

SET- allows user to set personal defaults

Development Standards

The development standards listed below are enforced by checking user actions and the results of those actions. STEP also allows each project to add standards.

Naming - All names used in source code are made up of fixed length mnemonics whose definitions are maintained in a naming glossary. Code will not be promoted to the level necessary for compiling unless all mnemonics of all its names are defined in the glossary.

Architecture - All procedure calls and data usage must agree with the architecture specified in the high level design. Code that violates the architecture will not be promoted to the compile level.

Coding - The user may define format and

design language content standards that are enforced by the system. This is done both by activating built-in checks made on source and design language documentation, and by tailoring of source code processors which analyze code.

Documentations - Use of documentation processors, such as flowcharters, result in documents that follow a specific standard. User-defined processors are also linked in using the CALL command to further tailor documentation formats. Skeletons are used to preset component formats such that the user essentially fills in the blanks.

A variety of control options are incorporated in the core system to enable a manager to tailor the system to project needs. In particular, command and subcommand rights may be tailored to each user or type of user, standard enforcement parameters may be tailored for each node, and password protection may be applied to any component of any volume.

File System - The STEP File System serves as a machine-independent foundation for the STEP command system. This greatly reduces the need for special programming for computer systems with different operating system architectures.

This portable file system uses the host computer services available in most mainframe operating systems. Each version has been optimized to perform efficiently on its host. Usually, the limiting factors are the efficiency of the host Pascal compiler, the capabilities of the host's asynchronous, direct-access disk systems, and the operating system's interprocess communication.

The STEP File System is implemented on a host computer through a direct-access data file. Disk blocks are accessed asynchronously using a low-level, host computer method which does not see any block structure. The file system operates using disk blocks of equal length organized into structure, data, and internal pointer categories. The structure blocks represent directory structure and file storage information. The data blocks store file data, and the internal pointer blocks relate combinations of various block types. Individual formats may exist within a category, according to use; unused blocks generally have no type. The types of blocks are defined below.

Block Type 0: An internal pointer block. It contains a sequential list of integer block numbers. These pointer lists chain unused blocks together.

Block Type 1: A structure block. It represents a directory and contains the appropriate series of directory entries. A subdirectory is viewed as a file in the parent directory with the reserved extension "DIRECT".

Block Type 2: Another type of structure block. This block type is used to link the pieces of large directories of large files.

Block Type 3: A data block.

Configuration Management

The configuration management system provides: (1) interlocks to ensure information in use by one person is not changed or deleted by another, (2) reports which summarize the conditions and associations of all components, (3) records of all changes made to all controlled components, (4) a framework for all user actions. This framework provides users with complete information about any components associated with a user's action. Each component defined to STEP has one or more revisions. Whenever a component name is specified in a command the following actions occur:

(1) The system displays the revisions for the named component and its associated control levels. A "?" input calls up the date of the revision, the name of the user who created it, and the reason it was created.

(2) The user selects a revision.

(3) The system checks to make sure the selected revision is at a level consistent with the command to be executed. For example, if an EDIT command is given and the revision selected is at a level that cannot be edited, the EDIT will not be allowed. However, the user has the option of creating a copy of the requested revision with a new revision number.

Revision selection logic may be prespecified so that commands with

multiple selections such as DOCUMENT and COMPILE may be run in batch or on line without user inputs. The important thing is that users always know the framework within which they are working.

A variety of reports are associated with configuration management:

Control Level Report - summarizes the levels of each component.

Release Summary Report - summarizes releases that have been created.

Release Contents Report - identifies the revision of each component a release.

Usage Report - identifies the releases containing component revisions.

Trouble and Change Reports - provide an audit trail of changes.

A variety of tools are used with STEP. They fall into two categories: tools that check user input and tools that transform user input. Checking tools include syntax analyzers for each language. Transforming tools include formatters, flowcharters, datamappers, design language processors, and text processors.

The STEP command being executed runs the tools. Users do not need to know the details of running a tool. STEP provides the necessary interface. STEP also allows users to integrate their own tools, either called directly from STEP commands or indirectly through its CALL command facility.

5.0 Use Experiencee

The STEP environment was first used on a project in 1978?5O. It has subsequently been used on a variety of real-time and non-real-time applications totalling more than 500,000 source lines and is used to maintain itself and extend its capabilities. These projects include telephone switching systems, air traffic control simulators, and communications control systems. Its first use was mandated by top management and supported by appointing one of the developers as software task manager. Initial use met with numerous complaints because the methodology demanded a lot of work and significant effort in the design phases. Yet the benefits from this extra effort are not apparent until the testing phases. However, by the time the project was completed most users were sold on the advantages of STEP and looked forward to using it on other projects. As could be expected with any large complex system, experience in using it was essential to maximizing the benefits. Our experience showed that users got more out of STEP the second time they used it.

Experience to date has shown:

(4) Testing goes much faster than expected.

(5) The resulting programs are reliable and maintainable.

(6) Projects are coming closer to budgets and schedules and have even met them.

(7) Documentation is extensive, sometimes too extensive.

(8) Documentation is consistent with code.

The following discussion deals with a variety of topics relating to the use of STEP.

Human Interface

(9) Consistency is extremely important.

(10) Upper and lower case is much easier to read than all upper case.

(11) Users resent "smart" responses. Because STEP continually tells users when they have done something wrong, we thought humorous comments might eliminate user resentment. Instead users felt they were not being treated as professionals. In fact, one user was reduced to tears after being called a "dummy" by the system.

(12) Extensive on-line help is more important than documentation. We made the mistake of providing on-line help only at the command level. The solution is to provide an expert mode for experienced users.

User Documentation

(13) The three most important things in documenting a system are examples, examples and examples.

(14) Two types of documents are needed; one that explains the theory, i.e., what's done and why, and a reference manual.

Efficiency (Workstations)

A mainframe environment will always be overloaded. If you provide more computing resources, you get more users. The only solution is to provide smart workstations which can be added as users are added. Still, every effort should be taken to make the mainframe as efficient as possible. We removed some functions because efficiency was more important than having those particular functions.

Performance

(15) Users are afraid of performance measurement. The ability to collect data on individual performance and to compare user's work was designed into STEP but never implemented because of user resistance. It was difficult enough just getting people to try a new system.

(16) Young users became efficient quickly and tended to get more out of the system than programmers who had been around for a few years. In fact, it turned out that novice programmers became contributors more quickly.

(17) Users need to feel they are part of the action. It is important to find ways to make users feel they can contribute to the evolution of the system. It is also a good idea to publicize success using STEP by giving credit to the people using it.

Portability

(18) It was possible to make an environment that was transportable among computers as different as IBM and DEC20 and still have the same user interface.

(19) It would have been a lot easier not to.

6.0 Conclusions

(20) The architectural framework was essential in the design of STEP. It made many things possible that otherwise would have been impractical.

(21) The decision to make configuration management part of the backbone of the system instead of a separate tool worked out well. It allowed users to always be aware of the context in which they were working and ensured that all actions on all data were within the prescribed boundaries.

(22) The design techniques resulted in code that stood the test of many changes and extensions.

(23) Separating the actual compiling and testing from the rest of the system worked well. That is, when the source code was exported, it was syntactically and architecturally correct and in the proper format for compiling. The test results were inserted into STEP by the CERTIFY command. This provided a flexible interface with language-dependent environments (i.e., compilers, debuggers) and target systems.

(24) Given the advances in microprocessors and user interface techniques over the last few years it is questionable whether a mainframe system is still appropriate. A distributed microprocessor based system with smart workstations, file and compute servers seems a better way to go today.

(25) The most important part of an environment is user documentation. It was the most underestimated item in the development.

References

(26) A.F. Rosene, "Phoenix Software Development System Overview." Presented at 15th Design Automation Conference, June 1978.

(27) STONEMAN, "Requirements for the Programming Environment for the Common High Order Language," DOD, February 1980.

(28) J. Roder, "Phoenix Architecture." Presented at 15th Design Automation Conference, June 1978. Published in ACM SIGDA Newsletter, Vol. 8, No. 2, Summer 1978.

(29) A.F. Rosene, J.E. Connolly, K.M. Bracy, "Software Maintainability, What

It Means and How to Achieve It." IEEE Transactions on Reliability, Vol. R-30, No. 3, August 1981.

(30) E. Erickson, and J. Roder, "A Generic Approach to Software Validation", Phoenix Conference on Computers and Communications," May 1982.

RESUME

A. FRANK ROSENE
Communication Systems Division
GTE Government Systems Corp
Needham Streets, MA 02194-9123

Program Position ENGINEERING TASK MANAGER

Years Experience 27

Education BS, MS - MIT

Clearance TOP SECRET

For the past twenty-seven years Mr. Rosene has been involved in the areas of software technology, telephone communication, system design, and radar data processing design. He has been responsible for a variety of software development in the communications and operating systems areas.

Mr. Rosene is currently manager of the Software Technology Department of the Communication Systems Division. In this role he supports the use of and extension of the division software development environment system, STEP. He has also been the Engineering Task Manager for the ALTAIR Upgrade study.

Mr. Rosene led in the development of the generic software architecture concepts which are not in use on all CSD projects. He was software manager of the International Switching Program from 1971-1976. That project was tasked to develop a new line of switching systems for the international market. Prior to that, he developed software for the car track system of the transportation and industrial division of GTE.

During the sixties he managed the software development for the first stored program switching system developed at GTE. His initial few years at GTE were involved with radar data processing program design for a variety of weapons and defense systems.

Societies and Papers

Member of the Institute of Electrical and Electronics Engineers Author of paper entitled "Phoenix Software Development System Overview" Joint author of paper entitled "Software Maintainability-What It Means and How to Achieve It" Published in the IEEE Transactions on Reliability, Vol R-30, No. 3, August 1981.

REUSABLE SOFTWARE and DESIGN

A. Frederick Rosene
Manager of Software Technology
GTE
Communication Systems Division

REQUIREMENTS for REUSABILITY

- ◇ STANDARD ARCHITECTURE
- ◇ STANDARD RUN TIME ENVIRONMENT
- ◇ ENVIRONMENT WHICH ENFORCES STANDARDS

GENERIC ARCHITECTURE

- ◇ STATIC VIEW
- ◇ DYNAMIC VIEW
- ◇ BENEFITS
  - Portability
  - Enforcement
  - Proven

STEP

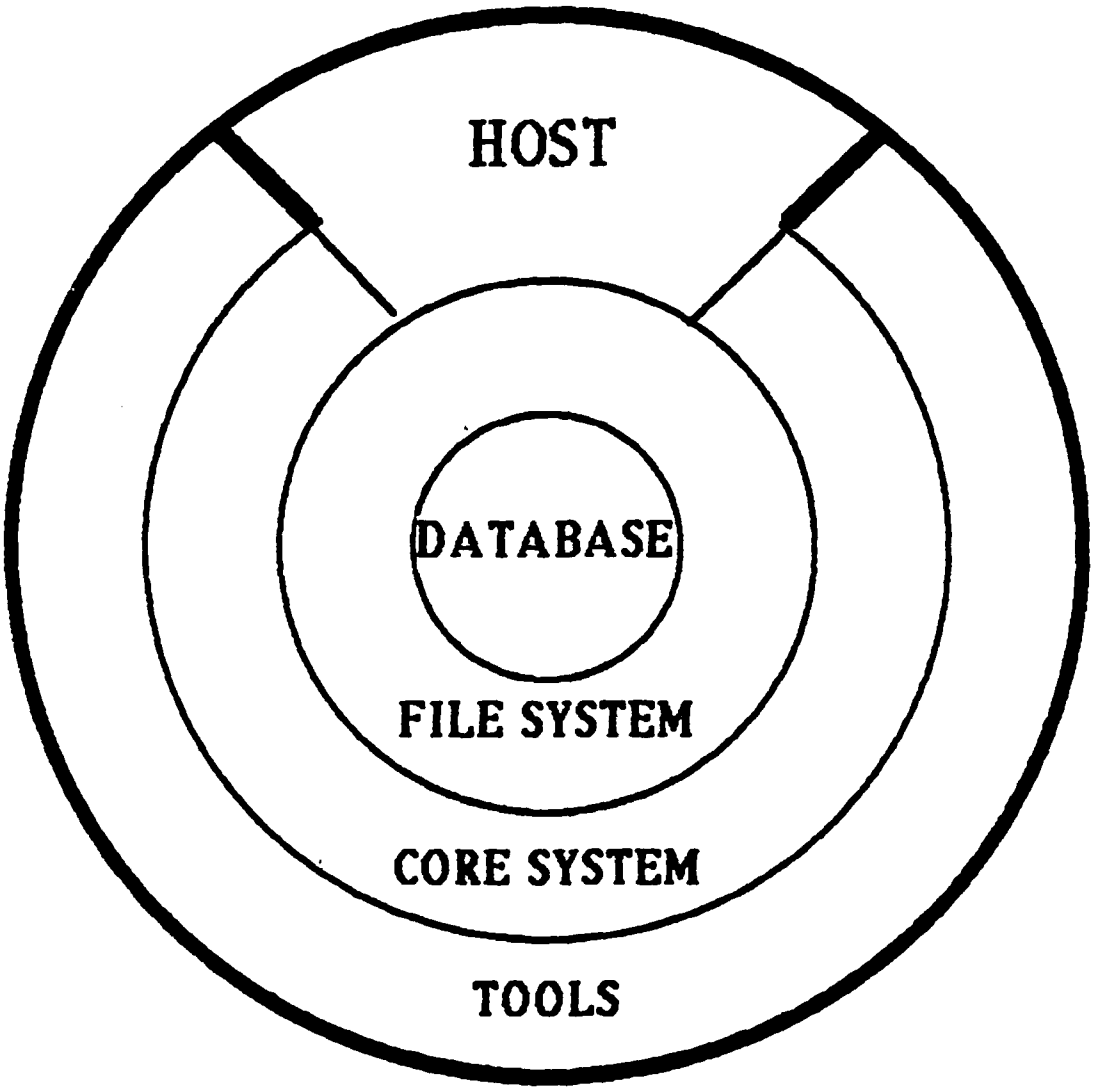
A Life Cycle Software Environment

A. Frederick Rosene
Manager of Software Technology
GTE
Communication Systems Division

INTRODUCTION

- ◇ DESIGN
- ◇ USER INTERFACE
- ◇ EXPERIENCE

STRUCTURE



DATABASE

◇ CONTENT

- REQUIREMENTS
- DESIGN DOCUMENTATION
- TEST DOCUMENTATION
- USER DOCUMENTATION
- INSTRUCTIONS & FORMS
- COMMANDS & SELECTION

◇ NETWORKING

- EFFICIENCY
- SECURITY
- SUPPORT
- RESOURCES

STANDARDS

- ◇ NAMING
- ◇ ARCHITECTURE
- ◇ CODING
- ◇ DOCUMENTATION

CONFIGURATION MANAGEMENT

- ◇ BACKBONE NOT TOOL
- ◇ REVISIONS
- ◇ REPORTS
- ◇ AUDIT TRAIL

COMMANDS

- ◇ which create components
- ◇ which insert information
- ◇ which generate data
- ◇ which monitor and control a project
- ◇ which view the state of components
- ◇ which create documents
- ◇ which provide file handling
- ◇ which aid users

EXPERIENCE

◇ PROJECTS

- COMMUNICATION SWITCHING
- AIR TRAFFIC SIMULATION
- COMMUNICATIONS CONTROL
- STEP

◇ RESULTS

- TESTING TIME REDUCED
- MAINTAINABLE PROGRAMS
- BUDGET & SCHEDULE
- DOCUMENTATION CONSISTENCY
- DOCUMENTATION TOO EXTENSIVE

HUMAN INTERFACE

- ◇ CONSISTENCY
- ◇ UPPER / LOWER CASE
- ◇ SMART RESPONSES
- ◇ ON LINE HELP

CONCLUSIONS

- ◇ ARCHITECTURAL FRAMEWORK ESSENTIAL
- ◇ CONFIGURATION MANAGEMENT BACKBONE
- ◇ MAINTAINABLE SOFTWARE A REALITY
- ◇ LANGUAGE INDEPENDENT ENVIRONMENT
- ◇ USER DOCUMENTATION UNDERESTIMATED

SPECIFICATION-BASED SOFTWARE ENGINEERING WITH TAGSOTM

G. E. Sievert
T. A. Mizell, Ph.D.

In a paper appearing in Computer in November 1983, Balzer, Cheatham, and Green asserted that the existing software paradigm (Figure 1b) had fundamental flaws that exacerbate the maintenance problem. The authors pointed out that these flaws were that there is no technology for managing the knowledge-intensive requirements analysis and design activities and that maintenance is performed on code. The authors proposed a new automation-based paradigm (Figure 1a) for which "the technology needed to support...does not exist." This article describes a software computer-aided development system that does exist and is based on the proposed automation-based paradigm.

Although the authors used the maintenance problem as the basis for asserting that the current paradigm has fundamental flaws, they could have used any portion of the software life cycle to make the same argument. For years, the industry and end-users alike have recognized that there is a software crisis. Symptoms of this crisis are visible everywhere: costs overrun; software does not meet user expectations, productivity is not greatly increasing; software specifications do not reflect the code nor are allowed to become outdated; and, the list goes on and on.

Although the symptoms are visible, the cause of the software crisis has been less apparent. As a result, response to the crisis has been to treat the symptoms by educating programmers, attempting to enforce discipline, providing software tools, and creating new coding languages. All of these approaches do help to keep the software crisis under control, but they do not treat the cause of the crisis. The cause of the software crisis is code.

Under the current paradigm, the only formal specification language is code, and the only formal specification is the program. Although modern programming practices have defined techniques to make the program more readable, the fact remains that the program is far removed from the user. Programs show only detailed logic; and from that view, it is difficult even for programmers to glean an understanding of the performance characteristics of the software. As a communication vehicle, the program listing is analogous to attempting to describe a television in terms of only schematic drawings. To the user, the program is an attempt to describe a picture using only words written in a foreign language and the price of the discipline enforced by modern coding languages is that the language has become more foreign.

Under the current paradigm, the deficiencies are addressed by requiring the production of English documents that describe the contents of the formal specification. These documents are often required during the analysis design phases of the project. In theory, the practice is good because it makes the design more visible to the user.

However, programmers tend to design in code. Program Design Language approaches act as (PDL) an alternative to English specifications and keep the programmer in the code environment by allowing specifications to be created in code-like languages. These languages do improve the software product but once again perpetuate the fundamental problems caused by the current paradigm.

As long as programmers use the current paradigm focusing on code, we will never achieve the orders of magnitude gain in productivity that are needed to end the software crisis. The anticipated gain in productivity, if

*TAGS is a trademark of Teledyne Brown Engineering, Inc.

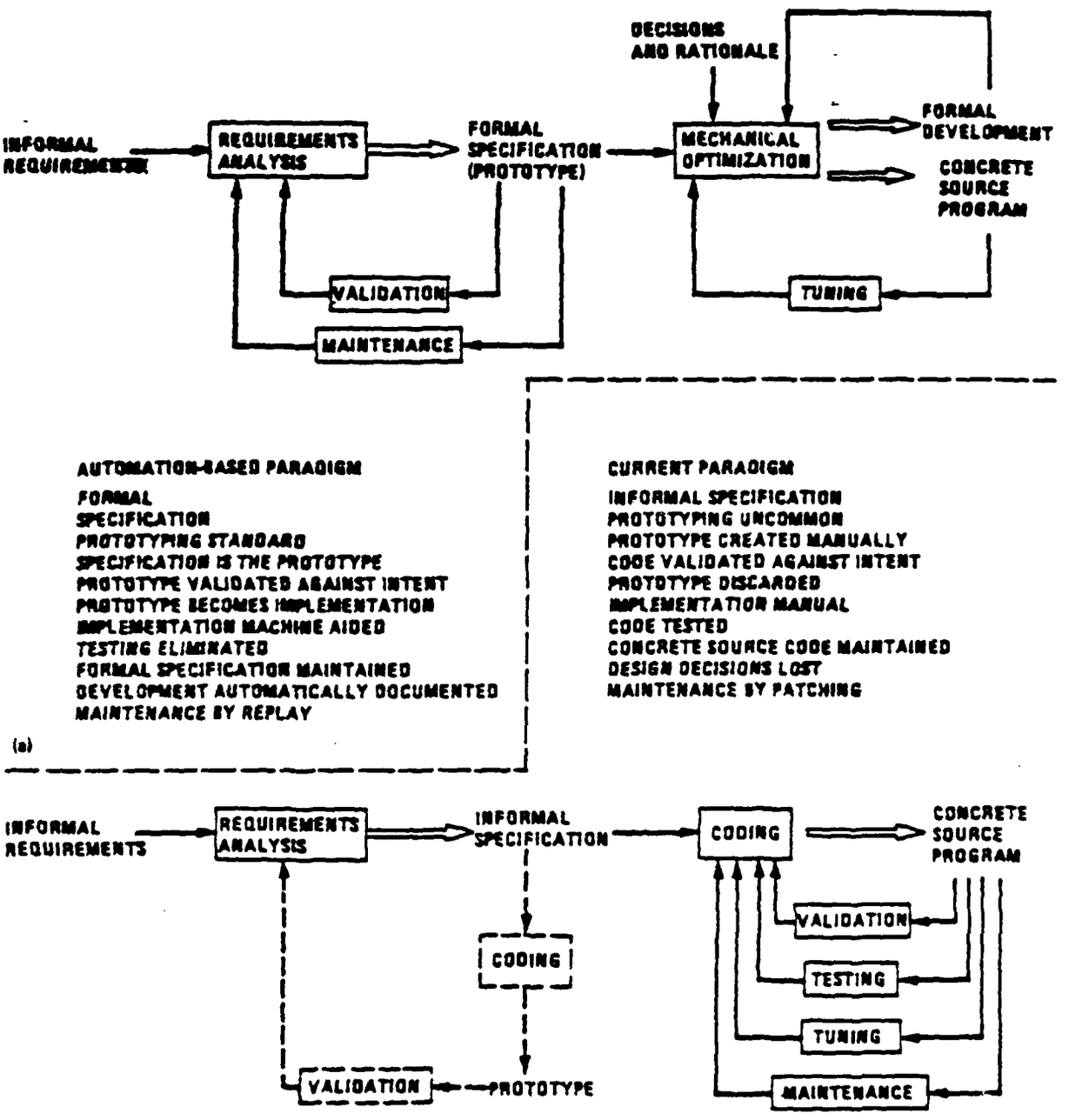


Figure 1. Paradigm comparison: (a) automation-based paradigm and (b) current paradigm

we treat all of the symptoms, is a multiplier of 4.34.?2O.

The new paradigm has been substantially implemented in an approach called Technology for the Automated Generation of Systems (TAGS). TAGS is the product of years of systems engineering and independent verification and validation experience. This experience lead the TAGS development team to the same conclusions and to a solution that is remarkably close to the solution proposed by Balzer et al.

TAGS is composed of three basic elements: the Input/Output Requirements Language (IORLOR?); a system/software computer-based tool system; and TAGS methodology. The rest of this article is devoted to describing the three basic elements.

The Language IORL

Thirteen years ago, Teledyne Brown Engineering, after experiencing difficulties in communicating system requirements, began to develop a tool that would deal with these problems. The tool was designed to deal with problems associated with system development and had to meet several requirements:

- To enforce a rigorous methodology for system development
- To be applicable to all systems, not just computer systems
- To be easy to use (and hard to misuse)
- To allow engineers to express system performance characteristics and algorithms using common mathematical notation (i.e., superscripts, subscripts, matrix notation, etc.)
- To use graphical symbols that were derived from general systems theory.

IORL is a graphics and tabular specification language that allows the designer to identify each important component during the design of any system, whether hardware, software, embedded, or for that matter, even a management system. The only constraint is that the end product of the design effort manifests the basic components of a system or group of parts, which interact via:

- Data links
- A controlling mechanism that directs how information passes among the parts of the system
- An identified hierarchy within the system.

The highest level in an IORL system is the Schematic Block Diagram, or SBD (Figure 2). Diagrammatically, the SBDs are rectangular boxes that identify all the principle system components and the data interfaces that connect them. In IORL, the designer must maintain a distinct differentiation between data flow and control flow. Here the major structures are more or less "black boxes," with the respective data flow, but control flow is not address at this level. Block diagrams have been around for many years as a way of conceptualizing the functions or components of a system in order to show the control or data connections among them. One of the advantages of the SBD is that it gives a quick synopsis or overview of the system.

In the sample top-level SBD, the unique name and number for each component, the use of comments, and the unique page identification should be noted. The system name (SYS) is SAMPLE. The document name (ID) is SAMPLE. For a top-level SBD, document ID and system name must be identical. The section name (SEC) is SBD.

The breakdown of an SBD component into a lower level SBD is continued until the resulting components can no longer be divided into independent units. Each SBD represents a different document, and the document ID must appear on all other related diagrams and tables. The second-level SBD is a decomposition of Component A on the

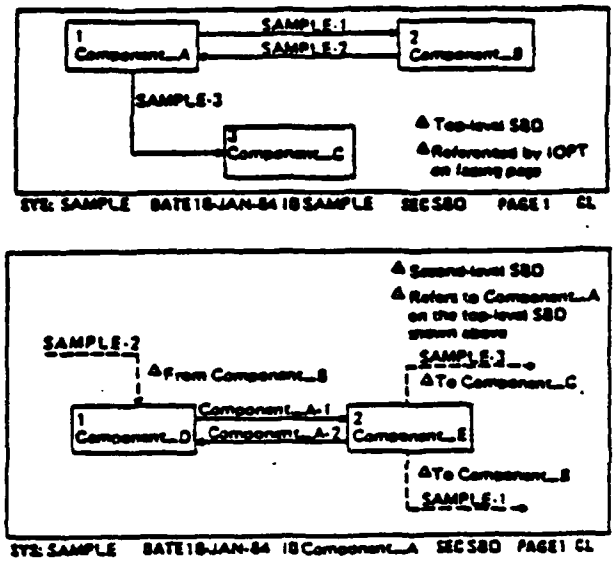


Figure 2. Schematic block diagram

top-level SBD. The document ID is now Component A. The external interfaces refer back to the higher level SBD. The label of each internal and external interface reflects its originating document. In addition, multiple identical interfaces and components can be distinguished by subscripts.

The Input/Output Relationships and Timing Diagram (IORTD) (Figure 3) shows the overall control flow for a single SBD component. The example gives a logical breakdown of Component B on the top-level SBD. This is indicated at the bottom of the diagram by the document ID SAMPLE and the number 2 in the section name IORED-2. Thus, the position of this page within the system hierachy is clearly established.

Control follows the direction indicated by the connectors from the start symbol through the Fan-InOR to the processing symbol where an assignment is made. (An assignment is an equation defining variables. Variables will be discussed in relation to Data Flow in the next section.) Process 10 is then executed and a decision is made based on the comparison statement. If the statement is true, control flows to the Fan-Out AND where both process 30 and 40 must execute. This is an example of IORL support for parallel processing.

The Predefined Process Diagram (PPD) (Figure 4) is used to depict the detailed logic flow of a single-predefined process that is referenced in an IORTD or another PPD. A PPD and an IORTD are similar in structure. However, PPDs are used to improve the readability of the specification, allow the identification of dependent components (see Methodology Section), and to permit the specification to be presented in a hierarchical manner. Consequently, a sophisticated system design that requires numerous complex processes will make extensive use of PPDs.

The sample PPD defines process 10 on IORTD-2. This is indicated at the bottom of the diagram by the document ID SAMPLE and the section name PPD-10. The PPD definition symbol in the bottom right-hand corner is required and may also include a name and brief description. Processing begins at the entry symbol. Once process 80 has been successfully performed data are produced as specified by the output symbol.

One of the features of the PPD is that the process and detailed logic may be saved on disk file for later recall by another IORL system. It is this library feature that allows the saving of enormous hours of development time as the use of the tool (described later in this article) increases. This is analo-

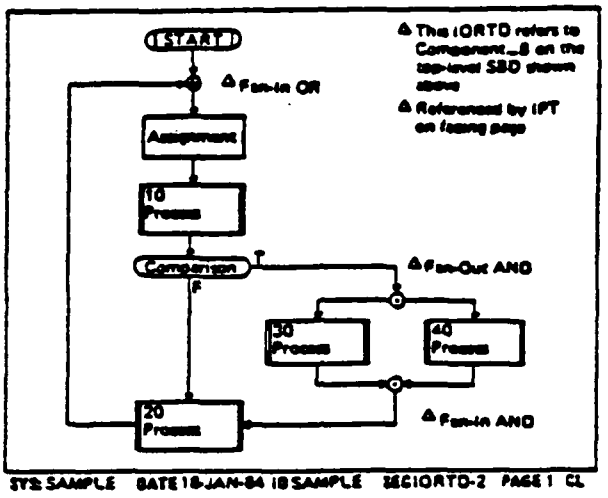


Figure 3. Input/output relationships and timing diagram

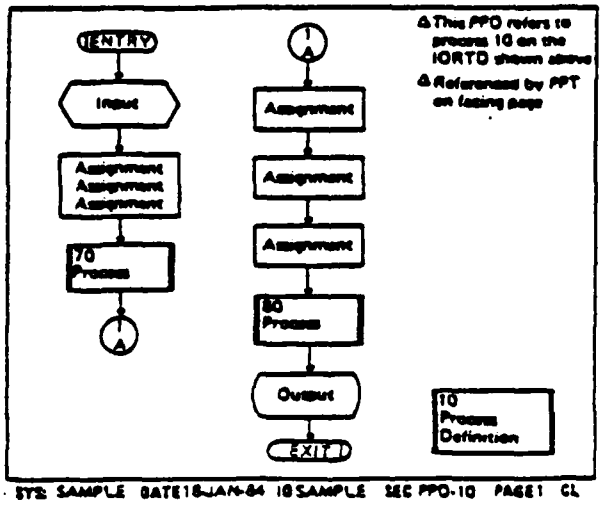


Figure 4. Predefined process diagram

gous to the custom-created library subroutines so common in every ADP center. As the number of library PPDs increases, an IORL system could involve a lengthy series of library calls without having to custom generate a routine each time another system is designed.

Data Flow

Other than a reference to data interfaces on the SBDs, no attempt was made until now to discuss the representation of data flow. Data flow in IORL is shown by the assignments made to variables in IORTDs and PPDs and by use of I/O symbols in the IORTDs and PPDs which show the timing of the flow of data between components. The basis for this approach is described in detail in the methodology section of this paper. Data in IORl is shown in tabular form using a set of tables each of which is similar in format except for the data hierarchy implied. The highest form of data definition is shown in a special form of an Internal Parameter Table (IPT-O). Data defined in this table is global in nature. The next highest form of data representation is an Input/Output Parameter Table (IOPT). This table shows data which pass over an interface between two components and variables defined in this table are defined for both components of the interface. Data which is defined for an individual component is defined in IPT-n (n>O) and finally, data which is limited in definition to an individual PPD is defined in an Internal Parameter Table (IPT). These tables all define each variable used in an IORL specification, including English description, variable name, legal or in-tolerance values (which imply type), units if applicable, and value meanings. Data structures are defined in another diagram form, Data Structure Diagrams (DSD). The following paragraphs describe the tables of IORL in more detail. DSD descriptions are omitted due to space limitations of this article.

An Input/Output Parameter Table (IOPT) (Figure 3) is associated with each interface on an SBD. It is used to identify, organize, and quantify all information that is transferred across the specified interface.

Each group is transferred as a unit in a continuous I/O sequence. A variable, or parameter, may appear in more than one group on an IOPT and on more than one IOPT.

The IOPT page shown in Figure 5 defines some of the data that are transferred across the interface SAMPLE-3 in the top-level SBD, shown in Figure 6. This is indicated at the bottom of the table by the document ID SAMPLE and the number 3 in the section name IOPT-3. Data Group 6 contains two scalar variables. The variable TIME may assume an integer value from O to 60 seconds. The variable MONTH may assume an integer value from 1 to 12 to indicate the appropriate month of the year. Data group 7 contains only one variable, named MONEY, that may indicate a sum from zero to any amount. The meaning column indicates that this amount must be measured in dollars.

An Internal Parameter Table (IPT) (Figure 6) defines variables that are internal to one IORTD and its associated PPDs. The IPT follows the same general rules as the IOPTD, except the use of groups is optional.

The IPT page shown in Figure 6 defines some of the variables internal to IORTD-2, shown on the facing page. This is indicated at the bottom of the table by the document ID SAMPLED and the number 2 in the section name IPT-2. A variety of parameters is illustrated to demonstrate the range of applications supported by IORL. The name of each variable must begin with the symbol shown to indicate the correct data type. The variable $DATA is a string of alphabetic characters varying in length from one to twenty. The variable !DATA is a matrix composed of six rows and six columns of values, all of which must belong to the set of real numbers. The variable &DATA is a logical condition.

A Predefined Process Parameter Table (PPT) (Figure 7) defines parameters that are local to one PPD. However, the PPT may include references to variables in other sections used by this PPD.

The PPT page shown in Figure 7 defines some of the variables local to PPD-

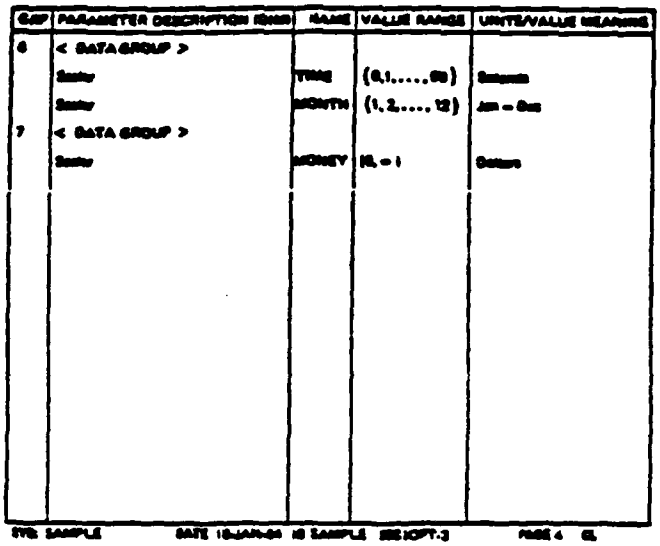


Figure 5. Input/output parameter table

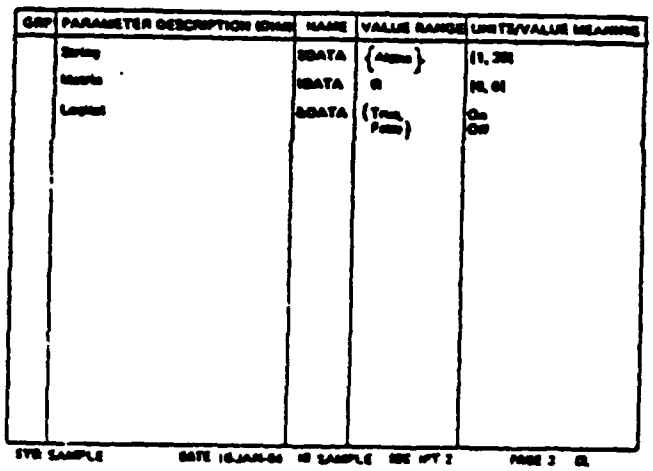


Figure 6. Internal parameter table

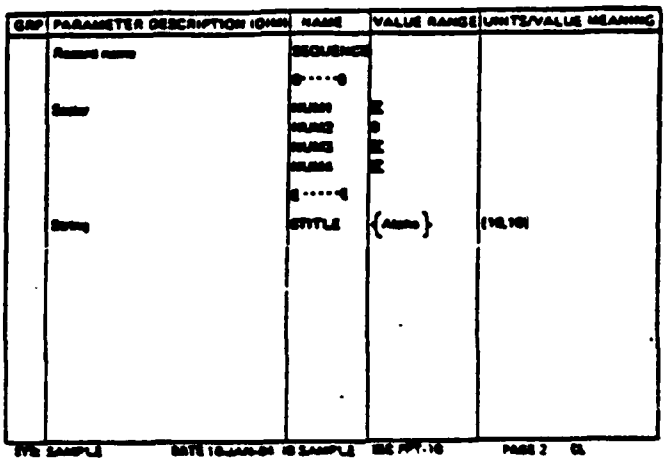


Figure 7. Predefined process parameter table

10. This is indicated by the document ID SAM;LE and the section name]PT-10. The first four parameters are arranged as a single record that is named SEQUENCE. Such a definition may appear on any parameter table and is always marked by B-----B for the beginning of the record and E-----E for the end of the record. In this case, NUM1, NUM3, and NUM4 may assume any integer value. However, NUM2 must assume the absolute fixed value of 9. Similarly, the string variable $TITLE must assume a fixed length. Since the minimum and maximum values in the meaning column are identical, $TITLE must always contain exactly ten characters.

Thus, in summary, it can be seen that IORL adequately differs between control and data flow, and at the same time is able to incorporate both into the same system design process.

The TAGS Tool

TAGS, as previously noted, is a self-contained approach to system specification consisting of the IORL language, and a graphics work station with four software tools:

- o STORAGE AND RETRIEVAL
- o DIAGNOSTIC ANALYZER
- o CONFIGURATION MANAGEMENT
- o SIMULATION COMPILER.

It is the intention of the designers to make its use as simple as possible, since the engineer should be concentrating on design effort and not be burdened by learning a great deal of work station control code. The system graphics can be created, stored, retrieved, and modified on disk without having to leave the work station. In addition to the hardcopy option, the IORL documents can be deleted and/or stored on magnetic tape. The IORL graphics software, moreover, automatically expands or contracts to ensure the best diagrammatic fit around the system, thereby saving space.

The storage and recall activities common in all on-line systems are achieved through an application package called STORAGE AND RETRIEVAL. When the system under development has been completely entered onto a disk file, the DIAGNOSTIC ANALYZER checks the IORL diagrams to ensure that the system contains no static (as opposed to dynamic) errors. Static errors typically include syntax checks, range checks, input/output problems, etc. The DIAGNOSTIC ANALYZER can find over 200 types of static errors. If no static errors were found in the system, the SIMULATION COMPILER can proceed to generate a definition of run-time parameters, simulate the system created in IORL, and

process the output data. If any dynamic errors are present, the IORL code can be corrected in STORAGE AND RETRIEVAL and recompiled, linked, and executed again using the SIMULATION COMPILER. The combination of DIAGNOSTIC ANALYZER checking for static errors and the SIMULATION COMPILER checking for dynamic errors enables the software engineer not only to develop a system but also to validate system performance and experiment with optimization. This frequently results in alternative designs that are more effective. The SIMULATION COMPILER also allows for testing specific algorithm performance, enabling the engineer to select the most efficient algorithms possible, not simply one that will work.

The process of optimization is not viewed as a distinctly different step. It is, in fact, expected that the designer be able to demonstrate that the chosen design is the most effective design possible under the constraints. This tool allows the designer to demonstrate that this expectation has been met; and, in fact, the chosen design is the best one under the existing constraints. Another strong future feature of the TAGS package is that DIAGNOSTIC ANALYZER, with the SIMULATION COMPILER, will generate an Ada * program that describes the system. The actual simulation runs are performed by compiling and executing the Ada emitted. The Ada code produced by the tool may be executed on a larger, perhaps more powerful, computer if the engineer so desires.

Another package is available for configuration management of the IORL system. IORL, like any automated tool, can produce a large volume of output, as well as a number of different versions of the designed system.

As the number of changes increases the problem of system management increases also, which after a time, becomes burdensome. CONFIGURATION MANAGEMENT enables the user to establish a set of orderly scheduled reviews of the development process. It allows management to baseline a system, modify it according to specifications, correct oversights, and implement tradeoffs. This last feature was developed from the realization that many versions of a software system may be operable at any one time, leading to the observation that management problems can grow exponentially if some mechanism of tracking these version is not available.

TAGS Methodology

When used to create system or software specifications, a unique TAGS methodology has been developed that is designed to implement the automation-based paradigm. The methodology is based on system engineering principles and is designed to utilize the potential power of the pictorial IORL. 'This methodology can be characterized by four basic activities:

- Conceptualization - User concepts and requirements are used to develop a conceptual model that is the basis for subsequent engineering.
- Definition - The model is developed and described in terms of functions and performance requirements.
- Analysis - The model, as defined, is analyzed to determine that it is complete and correct and provides an accurate description in engineering terms of the system desired. Redefinition is performed as necessary.
- Allocation - The functional and physical requirements are allocated to physical subsystems.

These activities have been incorporated into the methodology model shown in Figure 8. The development process is viewed as essentially a self-contained engineering activity that interacts with three other groups: users, management, and IV&V. Users describe initial requirements, participate in reviews, and provide clarifications. Management provides the nontechnical decisions that influence the development effort and provide the resources required. These activities include providing schedules and staffing, controlling expenses, and monitoring progress. IV&V serves as the technical representative of both the user and management.

The development process itself consists of three interrelated process: evaluation,

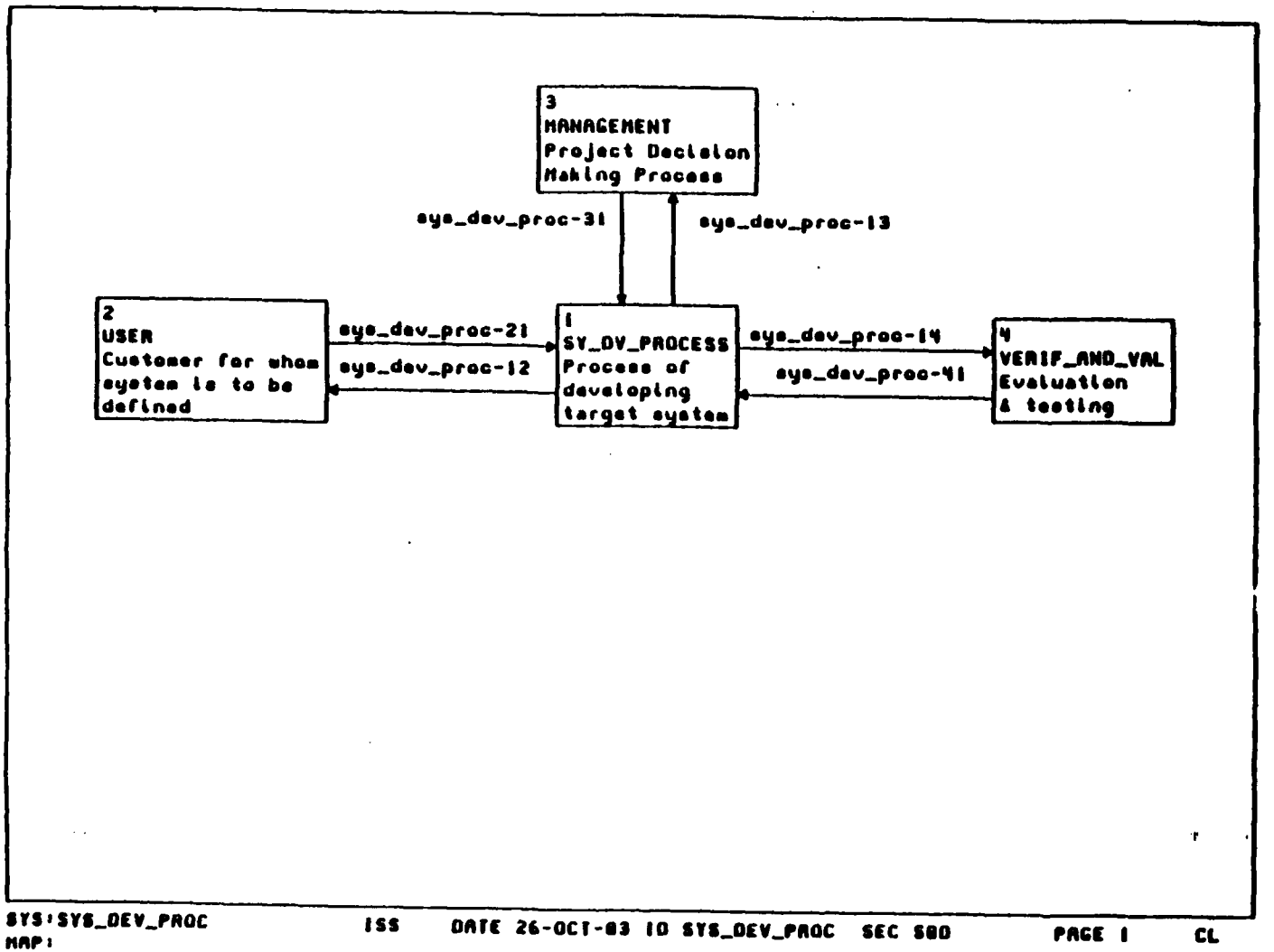


Figure 8. TAGS methodology model

specification and implementation. The process starts with the receipt of user requirements. Those requirements are evaluated to determine that they are complete, consistent, and potentially implementable. The evaluation will continue, dialoguing with the user if necessary, until the initial set of requirements are sufficiently understood to permit specification. The specification process is then performed. After the specification process is completed, the resulting specification is reviewed and evaluated. The results of this evaluation may be a decision to modify or implement the specification. The implementation is then further evaluated, and the results of that evaluation will result in a decision to modify the specification or a declaration that the system is ready for operational use. It should be noted that in terms of the methodology, there is no distinction between "system" and "software". In general, however, software specification is viewed as a special case of the overall system specification oriented TAGS methodology.

The specification process of the TAGS methodology extends the semantics of IORL by controlling the use and content of specific IORL diagram forms. To summarize the specification process, it is necessary to describe the general characteristics common to both systems and software as defined by the TAGS approach.

Data flow refers to the movement of data into, through, and out of the system. Control flow refers to the sequencing of operations performed within the system. Systems are also composed of two types of components: independent and dependent. Independent components are components that have their own well-defined control flows. For example, in a network of two computers, each computer may be defined as an independent component since each computer executes its own set of instructions in parallel with the other. Independent components can be composed of independent or dependent subcomponents. Dependent components on the other hand are components in which the execution of one component depends on the previous execution of the other. That is, dependent components are dependent in a control flow sense.

With the recognition that both independent and dependent components exist, it is possible to further characterize data flow and control flow. In a system, control flows are limited to individual independent components (by definition). Data flow through a system is not limited. Data flow from one independent component to another as it moves through the system. The transfer can only occur if the control flow of each independent component is synchronized in such a manner that one component is ready to accept the data as input when the other is ready to output it. This implies that time specification is a part of data flow specification.

In particular, time specification must be part of any input or output process between independent components.

The application of the above characteristics lead to the fundamental concept behind the TAGS methodology:

- SBDs are used to show data flow between and partition independent components. For the components represented on the SBD, data flow and associated timing dominate control flow.

- IORTDs and PPDs are used to show control flow and the relationship between dependent processing elements. Their use states that for the processing elements shown, control flow dominates. Data flow and timing are less important.

The actual specification process can be summarized in four basic steps. The four-step summary (Figure 9) is as follows:

Step 1: Build the conceptual model. The general form of an SBD that represents the system to be specified as a single component and the environment as many unique, independent components. The environment is everything with which the system must interact (i.e., other systems, operators, etc.). Interfaces between the system and the environmental components are not generally shown. As an example, Figure 9 is a conceptual model for the TAGS methodology. Note that obvious interactions between environmental elements have not been shown. This representation sets up an important thinking pattern in the engineer.

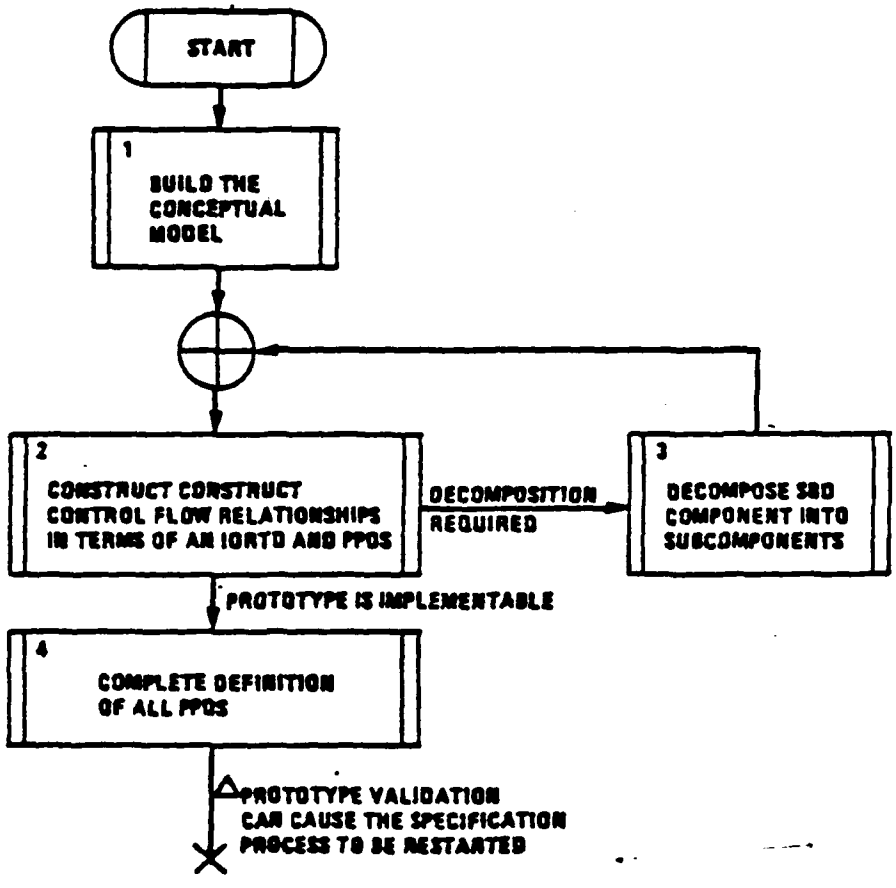


Figure 9. Summary of the specification process

Specification must be approached from the way the target system sees the environment. In TAGS, this is called "an inside, looking out" approach. In developing the details of specification, it is not good practice to consider complex interactions between environmental components in order to determine that special cases within the expected interface between the environment and the system will not happen. For example, it is not reasonable to decide that a bad message response will not happen because the operator is well trained. The system must react to all stated interface requirements even though it appears they will not happen. This approach makes the system relatively impervious to changes in the environment and later supports changes in the maintenance phase.

Finally, it is important to find the precise boundary between the system and the environment since this boundary can later be used to determine the precise input/output characteristics required by the system to interact with the environment. Often, this boundary occurs on a physical boundary but not always. For example, if the system to be specified were a software applications package to be run under an existing operating system, the components shown on the SBD might be application software, keyboard/display and supporting OS, disk files and supporting OS, etc., to signify that the interface between system and environment is a software-to-software interface.

Step 2: Build the corresponding IORTD and high-level PPDs. IORTDs are used to identify and show the control flow relationships between major components. Because IORL has an "AND" construct it is possible to show parallel control flows. The recognition that independent components exist frequently shows while developing the IORTD and high-level PPDs.

Normally, Step 2 is accomplished using engineering analysis to organize system requirements into an overall set of PPDs that are related to each other by defining their control flow relationships. Each PPD is then defined in terms of sublevel PPDs that are again related to each other in terms of control flow. Note that the basic methodology of this step is stepwise refinement or functional decomposition that may or may not go all the way to the lowest level of mathematical definition during this step. Note also that the higher levels of control flow specification are developed primarily in terms of PPDs. This approach is called PPD rich in the TAGS methodology; and since IORL's syntax permits a meaningful description as part of each PPD referenced, the approach provides a natural, user friendly way to communicate the overall functions and flow of the system which can be read even by reviewers who have limited knowledge of IORL. Although the basic methodology of this step is stepwise refinement, the overall approach should not be misinterpreted as the development of a set of purely functional requirements. Remember, that the IORTD is used to show the control flow within an SBD component and that the SBD represents a model of the system itself. The specification step is an attempt to develop a prototype of the system, a prototype in which components are allocated to an implementable architecture. If the prototype can be verified as meeting all of the system requirements, it will become the implementation.

In simple cases, the conceptual model created in Step 1 can serve as the total model required to support the prototype. However, in more complex systems or in systems where there are severe timing constraints (i.e., timing dominates control flow), a more detailed model of the system must be used as the basis for allocation and specification. This condition is recognized when parallel paths are required that also must communicate with each other or it is recognized that PPDs must execute in time-dependent fashion (i.e., periodically). Another condition in which this can occur is in large projects in which management considerations dictate partitioning of the system into manageable portions.

Other conditions can cause a decision to be made to decompose the system model (SBD) in more components (a new lower SBD). One obvious one is the need to more closely model an imposed or a prior physical architecture. Ideally, the specification should dictate the architecture. However, in practice, software engineers often start with a fixed system architecture, such as a known computer, and must fit the specification to the architecture.

It should be noted that the decision to decompose the system model into components is a significant one requiring analysis and review. SBDs enhance data flow definition but at a cost of corresponding loss of control flow visibility. The tradeoff must be studied to determine that the strategy will be beneficial to the overall system specification.

Step 3: Decompose the SBD into components. If this step is required, the system component that must be decomposed is represented as a set of subcomponents on a new lower level SBD. Step 2 must now be repeated. In this case, the step starts by allocating the PPDs defined in the previous level to the proper subcomponent. Step 2 will subsequently be performed as a minimum for each subcomponent that has PPDs allocated to it and for other subcomponents, if required, to complete the specification.

Step 4: If Step 2 has not already been done, continue to define all PPDs on the lowest level in terms of PPDs, assignments, decisions, inputs and outputs, etc., until no PPDs remain to be defined. This step states the level of detail to which the specification must be taken. This level is a complete algorithmic definition level that allows for a true validation of the specification. There is an exception to the rule of Step 4. A PPD reference may be left in the specification, if and only if, it is already defined in a TAGS library. Thus, there is a built-in advantage in the TAGS approach to reusing existing PPDs.

Life-Cycle Models

TAGS methodology has been incorporated into a unique life cycle designed to take advantage of the specification-based paradigm upon which it is based. The life cycle is shown in Figure 10. The life cycle consists of a sequence of specification phases each followed by a prototype validation phase.

Perhaps, the real advantage of specification-based paradigm lies in the fact that a prototype validation phase can be inserted into the life cycle. It is no surprise that most traditional software life cycles have no provision for design analysis. The current paradigm with its formal specification provides little basis for concrete and definitive design analyses. Although large projects employ some analysis, such as timing and sizing studies, optimization studies, etc., these studies are based on predictions of the design (remember the true design takes place during the coding period after the analyses are performed). At best, these studies can be used to verify the design. True validation that the system will perform according to user requirements must wait for coding to be complete.

For years it has been recognized that the real problem behind software cost overruns is that errors are created during the design phase that are not found until the test phase (or worse, in the maintenance phase). With TAGS methodology and tools, these errors can be found and removed during the specification phase.

Activities that can be performed during the prototype phase with the aid of the TAGS tools include:

- o A general analysis for correctness, focusing on dynamic situations and algorithm design.
- o Timing and sizing
- o Optimization and refinement
- o "What if" analysis
- o Tradeoff studies to determine the best implementation form of critical algorithms
- o Fault tolerance studies

The translation phase activity consists of translating the specification into an appropriate set of plans that includes additional implementation detail to allow the system to be built. In the case of software, this phase consists of translating the IORL to the target/implementation language. Currently, this activity is a manual operation. Experience has shown that it can be accomplished by non-degreed personnel and that the normal rate of conversion is 200 commands of target-coding language per day per translator.

Testing consists of determining that the translation is correct. With TAGS technology, this is normally a relatively short process and

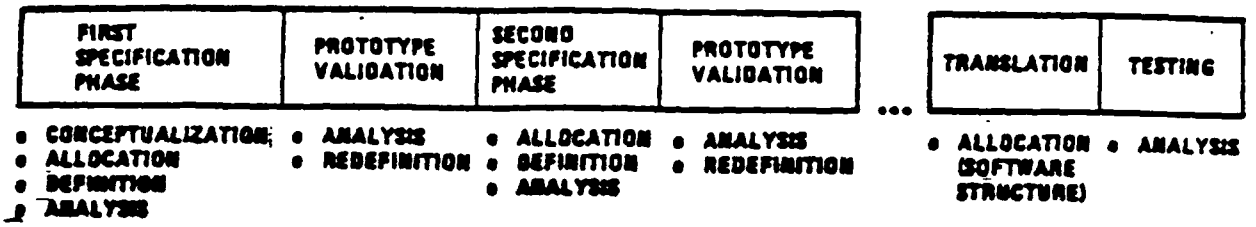


Figure 10. TAGS development life cycle showing the system engineering activities performed during each phase in order to importance

is analogous to unit and informal integration testing in the current paradigm.

The life-cycle model is designed to accommodate a large, complex system development process. In this case, multiple specification and prototype phases are required corresponding to the multiple specification cycles used by systems and software engineers. These cycles normally focus on system, subsystem, and then software levels of specification. In the TAGS methodology, more than one SBD level may be created for each of the systems/software cycles. However, the definition for each must be a complete prototype. Software specification alone seldom requires more than one specification cycle.

Putting It All Together

The three elements of TAGS work together to support the engineering process. In a typical TAGS application, software designers and system architects develop the software specification in IORL using the TAGS software engineering methodology. The IORL is entered into a central data base and edited as necessary. As the specification effort progresses, the diagnostic analyzer is used to find IORL syntax and semantics errors and statically check for design errors (i.e., a message is referenced but not defined). The specification is baselined and maintained under configuration management.

When the specification is completed, the design is analyzed in a dynamic environment using the simulation compiler. The designer interacts with the simulation compiler using menus that are structured similar to the other tools. The designer describes the type of data to be collected, the timing constraints desired for the simulation, and, optionally, the format for data input and output. When simulation is complete, the user can review any errors detected and the data generated during the simulation. Future versions of the simulation will allow user interaction during execution of the simulation.

Project Experience With TAGS

Teledyne Brown Engineering (TBE) has used IORL with a prototype version of TAGS for the last 10 years. Most of the early use of IORL was as an analysis tool supporting TBE's systems engineering and software Verification and Validation (V&V) product lines. Until 1979, the language and prototype tools were proprietary to TBE. In 1979, TBE recognized the potential of IORL as a software specification language and formed a language committee to upgrade the language to enable its use as a software specification language capable of supporting all phases of software specification. The committee's work resulted in the second version (V2) of IORL described in this article. During the same period, TBE started the development of TAGS. TAGS contains over 200,000 lines of source code and was developed using IORL, however, during the early phases of development various software development methodology experiments were conducted. In 1983, the experimentation had progressed to the point where a methodology committee was formed to formalize the methodology. The TAGS approach (IORL, methodology, and part of the tools) was subsequently used in a sequence of three pilot projects.

The projects were all flight software (aircraft and space) projects in which the

software was embedded into TBE built hardware. Two of the projects were complete projects ending with customer accepted systems. The remaining project was a competitive Concept Definition Phase activity. This latter project will continue as a larger (over 60,000 lines of code) pilot program if TBE wins the engineering development activity to follow.

All three projects used IORL, the methodology, and the Storage and Retrieval tools of TAGS in a batch environment. In this environment, IORL is recorded first on paper then entered into TAGS by a trained operator. Any benefits of the software tools are minimized but, the benefits of the use of the language and methodology can be determined. The batch environment was necessary because the projects were started before the remaining tools were complete and existing in-house workstations were in use to complete their development.

In spite of the environment, all three projects were judged to be highly successful. The reasons for this assessment are summarized below:

- o Productivity rates experienced during the two complete projects were judged to be twice what TBE would have normally bid using a traditional approach to software development. This result was determined by comparing project productivity (overall hours/line of code) with TBE's standard software estimation algorithms.

- o On both complete projects, software development was considered to be on the critical path. In the case of both projects, the software remained on the critical path through the design phase (due to the methodology) and then the hardware became the critical path item. In both programs, the software finished ahead of the hardware.

- o On the smallest project, the flight software was integrated with the target hardware in less than one week.

- o On all three projects, the customer accepted and liked IORL. It was found that IORL drawings provided an excellent vehicle for design review presentations and they were used extensively during all formal reviews.

- o Contractual documentation requirements were met by incorporating the IORL specifications into NASA documentation standards and MIL-STD 483 B level specifications thus demonstrating that IORL can be used with existing Government standards. In all cases, the full IROL specification was included in the specifications as an appendix.

- o The purpose of the pilot projects was to transfer the TAGS approach from the experimental group developing TAGS to other software development groups at TBE and the experiences described above are not intended as proof of the superiority of the TAGS approach. However, given that most of the software engineers on these projects used IORL for the first time and that the full power of the technique - interactive software workstations containing a full complement of tools - was missing, it is safe to conclude that the TAGS approach is a viable alternative to the current software paradigm.

REFERENCES

1. Robert Balzer et al., "Software Technology in the 1990's: Using a New Paradigm," Computer, Vol. 16, No. 11, No. 1983, pp. 39-45.

2. Barry W. Boehm and Thomas A. Standish, "Software Technology in the 1990's: Using an Evolutionary Paradigm," Computer, Vol. 16, No. 11, Nov 1983, pp. 30-37.

RESUME

GENE E. SIEVERT

Gene E. Sievert is the Manager of Software Development at Teledyne Brown Engineering. He is a member of the IORL Committee and Chairman of the TAGS Methodology Committee. He has over 18 years experience in systems and software engineering of command and control, real-time systems. Mr. Sievert received his B.S. from Ohio State University and his M.S. from Case-Western Reserve University.

RESUME

TERRENCE A. MIZELL

Terrence A. Mizell (Terry) is a Systems Analyst in the Software Products Department at Teledyne Brown Engineering. Before coming to TBE, Mr. Mizell served as an Operations Research Analyst with the U.S. Army Missile Command for 3 years. For 10 years previous to this, he had been a college professor at several southern universities. He is an officer in the Huntsville chapter of the IEEE and is professionally interested in design languages, operations research, and statistics.

Mr. Mizell received an A.B. and M.S. from the University of Alabama, Tuscaloosa, and a Ph.D from Emory University, Atlanta, Georgia.

RESUME

MIGUEL A. CARRIO, JR.

Miguel A. Carrio, Jr. is Manager of Advanced Programs at Teledyne Brown Engineering. His responsibilities include the coordination of life cycle definitions with developmental and IV&V procedures, that when coupled with the proper support tools enable an efficient systems development process to occur. He is also the TBE STARS Program Director.

Prior to joining TBE Mr. Carrio's Ada expertise began as a member of the Army's Software Technology Laboratory in 1979. He formulated and served as project leader on a number of unique and innovative projects that were intended to establish and transition the Ada Language Technology from DoD into industry and academia. Specifically, he was project leader on the following:

1. Development of the Ada/Ed translator interpreter, Courant Institute, New York University.

2. AN/TSQ-73 Missile Minder Ada Redesign Case Study, Control Data Corporation.

3. AN/TYC-39 Message Switch Ada Redesign Case Study, General Dynamics Corporation.

4. Ada Methodology Study, SOFTECH.

5. Army Curriculum Development Effort, SOFTECH and Jersey City State College.

RESUME

TOM WALSH

Mr. Tom Walsh is Teledyne Brown Engineering's Director of Ada Training Activities. He is currently engaged in the development of the TBE Ada courseware for dissemination throughout the company, as well as outside of it. The courseware represents an extensive set of Ada modules and texts extending from an Ada Fundamental Course to an Advanced Ada Concurrently (Tasking) course. Mr. Walsh's background in software development is extensive. He directly participated on the AN/TSQ-73 Missile Minder Ada Redesign Case Study as a member, at the time, of the Control Data Corporation team. He is currently the TBE lead representative in support of IEEE's Ada Program Design Language Working Group, and the committees endeavor to establish industry Ada PDL Guidelines.

SOFTWARE QUALITY

Raghu Singh
Space and Naval Warfare Systems Command

RESUME
RAGHU SINGH

Raghu P. Singh is a computer scientist with the Space and Naval Warfare Systems Command. As Navy Assistant STARS Program Manager, he is involved in the day-to-day management and coordination of Navy STARS program elements at the Navy STARS Program Manager's office. His main interest and work lie in the area of software engineering, software product assurance, and software metrics. His current address is: Space & Naval Warfare Systems Command, Embedded Computer Program Office, SPAWAR-03Y, Washington, D.C. 20363-5100; and can be reached at 202-692-3966 (AV 222-3966).

AGENDA

- Software Quality
  - Problem
  - Approach
- Quality Engineering
  - Overview
  - Establishing quality goals

• WHAT IS QUALITY?

— Quality means different things to different people

QAD 9/84

• QUALITY FROM PERSPECTIVE OF

- Project manager/proponent
- End-user
- Programmer/analyst

QAD 9/84

• PROJECT MANAGER/ PROPONENT

— System works as it should

— System completed within cost and schedule

QAD 9/84

- **END-USER**

— Easy to use

— Difficult to misuse

— Responsive to needs

QAD 9/84

- **PROGRAMMER/ANALYST**
  - — Code executes correctly
  - — Easy to change

069

QAD 9/84

- **FURTHER PROBLEMS**

— Quality needs vary from system to system

— Quality needs may conflict

— Lack of a quality model

QAD 9/84

• CONSEQUENCES

- — Quality needs not clearly stated and understood
- — Quality loses out in competition for resources to
  - -- Functional goals
  - -- Schedule,...
- — Systems do not meet expectations
  - -- Systems fail on quality, not on functions

QAD 9/84

AGENDA

- ☐ Software Quality
  - — Problem
  - — Approach
- ☐ Quality Engineering
  - — Overview
  - — Establishing quality goals

QAD 9/84

- **QUALITY ENGINEERING**

— A discipline designed to solve these problems
  - -- Defines what is needed
  - -- Builds what is defined
  - -- Evaluates what is built

694

QAD 9/84

- **PROVIDES A FRAMEWORK OR MODEL FOR:**
  - — Manager and proponent
    - -- Define quality needs
  - — Developer and maintainer
    - -- Build quality into system
  - — Developer and QA
    - -- Evaluate level of quality

QAD 9/84

- **FACILITATES:**

  - Manager, developer, QA interdependence
  - Communication through common medium
  - Integrated approach to quality
  - Automation of procedures

QAD 9/84

AGENDA

- ☐ Software Quality
  - — Problem
  - — Approach
- ☐ Quality Engineering
  - — Overview
  - — Establishing quality goals

• QE CONCEPT

— Considers elements of software and quality

- Software:
  - Functional requirements
  - Structured design,...
- Quality:
  - Performanee
  - Reliability
  - Complexity,...

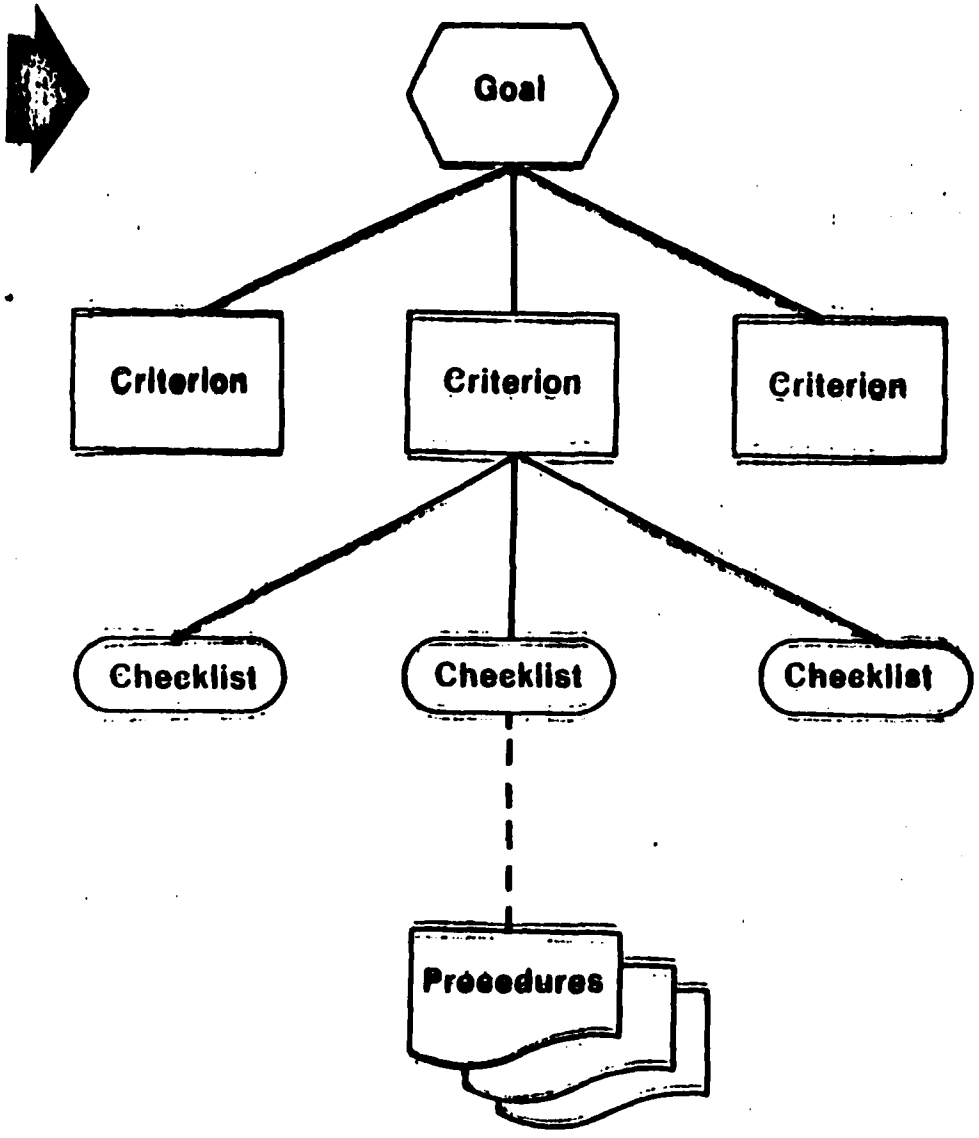
• QE CONCEPT (CONT)

— Grouped at
  - -- Management
  - -- Technical
  - -- Product/process levels

— Results in a framework (model)

J. McCall, et al
B. Boehm, et al

FRAMEWORK



Management-Oriented Views of Quality

Engineering-Oriented Concepts of Quality

Specific Items From Product & Process

Detailed, Step-by-Step Procedures to Answer Checklist Items

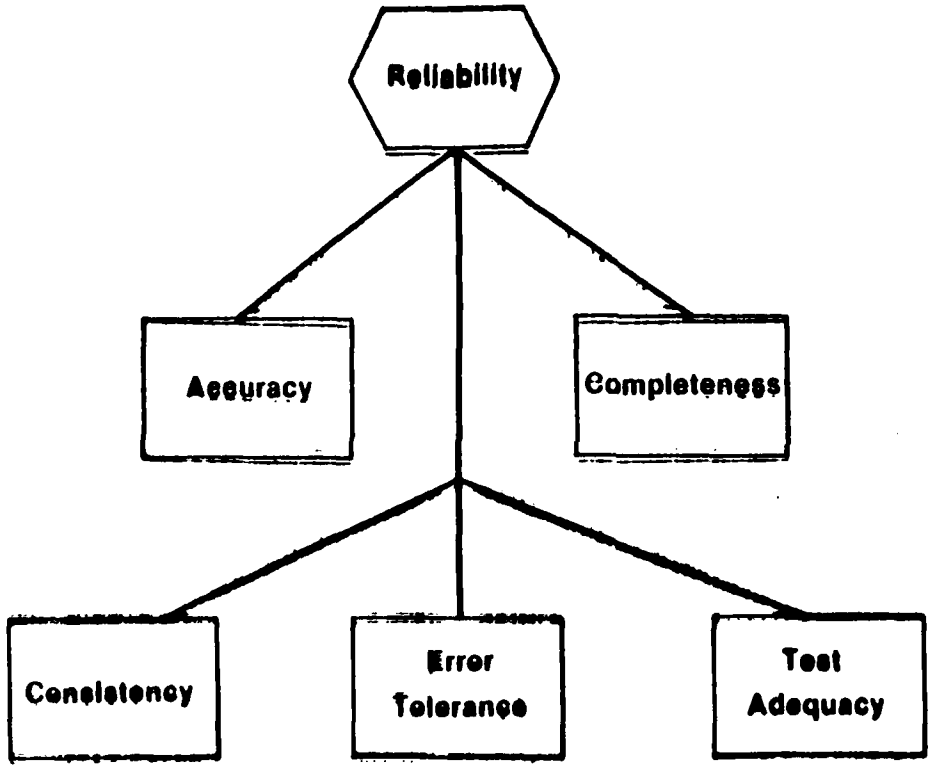
QAD 9/84

QUALITY QUESTIONS

Efficiency	— Will it run or fit on my hardware well?
Flexibility	— Can I change it?
Integrity	— Will it be secure?
Interoperability	— Can I couple it with other system?
Maintainability	— Can I fix it?
Portability	— Can I use it on another machine?
Reliability	— Will it do it accurately all the time?
Responsiveness	— Will it provide the information when needed?
Reusability	— Can I use a part of it in another system?
Testability	— Can I test it?
Usability	— Can I use it? Or not misuse it?

QAD 9/84

- **EXAMPLE:**



(Goal)

(Criteria)

(Criteria)

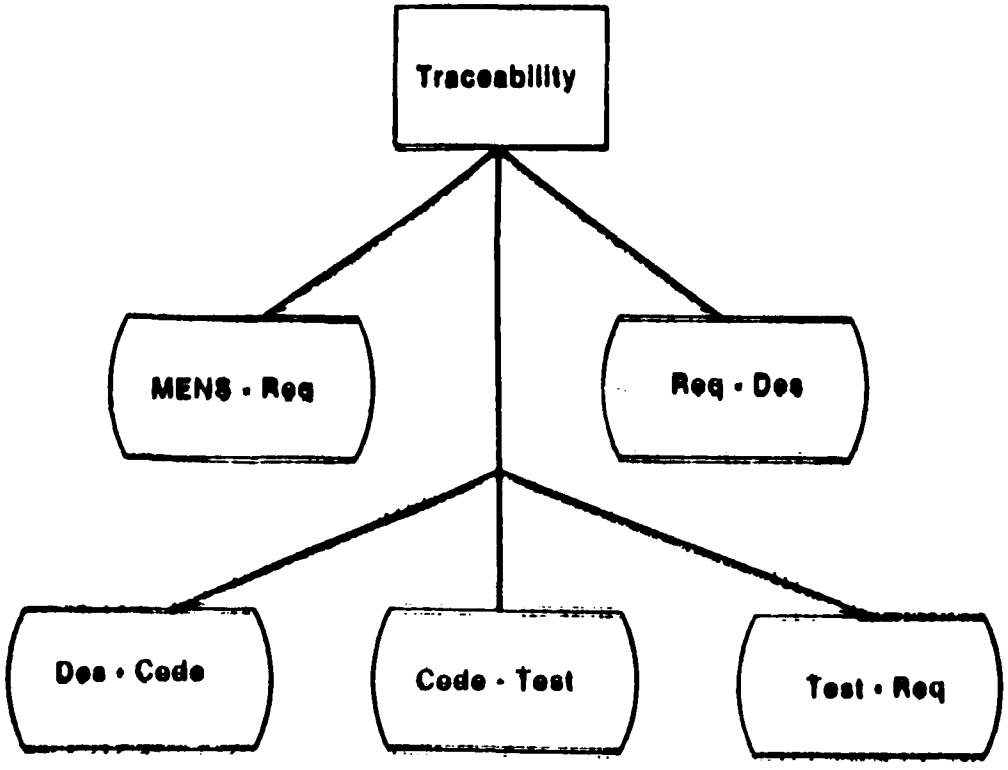
QAD 9/84

GOALS — CRITERIA RELATIONSHIPS

Quality Criteria \ Quality Goals	Efficiency	Flexibility	Integrity	Interoperability	Maintainability	Portability	Reliability	Responsiveness	Reusability	Testability	Usability
Access Audit			●								
Access Control			●								
Accuracy							●				
Communications Commonality				●							
Communicativeness											●
Completeness							●				
Consistency					●		●				
Data Commonality				●							
Documentation Adequacy		●			●					●	
Error Tolerance							●				●
Execution Efficiency	●										
Expandability											
Instrumentation										●	
Machine Independence						●			●		
Modularity		●		●	●	●			●	●	
Operability											●
Response Time Adequacy								●			
Self-Descriptiveness		●			●	●			●	●	
Storage Efficiency	●										
Structural Simplicity		●			●					●	
Test Adequacy							●				
Throughput Adequacy								●			
Traceability		●		●						●	

QAD 9/84

- **EXAMPLE:**



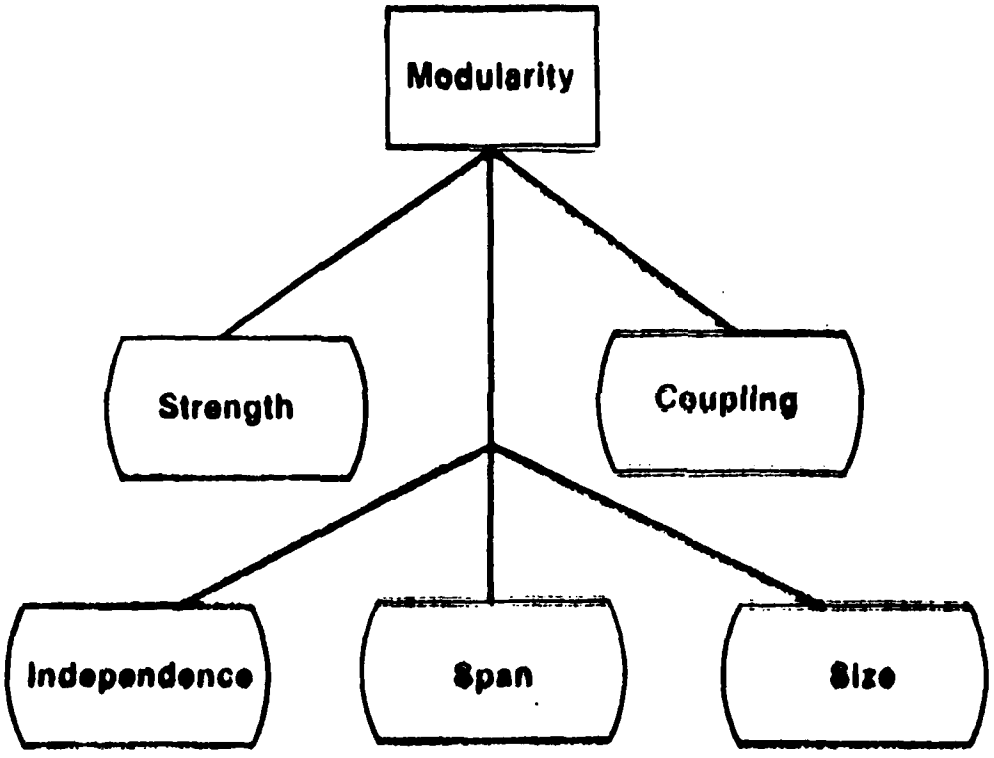
(Criterion)

(Checklist)

(Checklist)

QAD 9/84

- **EXAMPLE:**



(Criterion)

(Checklist)

(Checklist)

QAD 9/84

• QUALITY ENGINEERING — SUMMARY

Chapter	Status
Develop framework	Complete
Establish quality goals	Complete
— Automation quality program planning	= Partially developed
Build quality goals	Very limited application
— Parallel build	= To be developed
Evaluate quality goals	Only for 5 goals, 12 criteria
— Formal reviews	= Partially developed
Predict quality goals	To be developed

QAD 9/84

FUNDAMENTAL TECHNICAL ISSUES OF REUSING MISSION CRITICAL APPLICATION SOFTWARE

J.G. Snodgrass
Staff Engineer

E-SYSTEMS, GARLAND DIVISION
P.O. BOX 660023
DALLAS, TEXAS 75266-0023
(214) 272-0515

Abstract

From the list of eight issues in the CBD Special Notice on Mission Critical Application Software Reuse, it is not apparent that certain fundamental technical issues of reuse are adequately addressed. However, the issues addressed in this paper could fit into either the Specification/Design or Reusable Component Definition listed issues. This paper takes the position following:

(1) The functionality of the component of reuse is a fundamental technical issue.

(2) To attain software reuse the focus has to broaden out to the entire system including interfaces between system components and the movement of functionality among components (hardware, software, manual procedures, forms, and people).

(3) The modifiability of software systems is very important to reuse. (In fact, modifiability and reusability are considered the same by at least one researcher (Wegner 1984)).

(4) The development of a reusable software components environment requires fundamental changes in the ways of developing software and thus will take many years.

The key elements in the difficulty of reuse (Figure 1) will be referred to in supporting the four positions listed above.

Reusable Component Functionality Issue

The functionality of the component of reuse is a fundamental technical issue. Cognitive psychology research in programming has recently made a strong case for the thesis that experts mentally reuse programming plans (e.g. a RUNNING TOTAL LOOP PLAN or an ITEM SEARCH LOOP PLAN) (Soloway 1984). The reuse of the programming plans and the functionality of the plans seem to be what separates experts from novices. For mental reuse, it is clear that reuse of an Ada?RO language "case" statement block is better than accomplishing the same functionality with a series of "if...then...else" statements.

But, for reuse in the sense of finding the right "case" statement block in a library of reusable components, it is not clear that a "case" statement block possesses enough functionality to warrant the overhead of having it in a library and finding it for reuse. What size of component should we attempt to reuse? The FORTRAN subprogram has only been successful for certain types of functionality. A look at categories of functionality can help explain why FORTRAN components have been successfully reused in some situations and not others.

The widespread reuse of FORTRAN libraries has only been for physical or math models, which are the slowest to change entities modeled in software (See Figure 1, the Category of Functionality Reused column). The functionality dependent on human decisions is so rapid to change that one cannot expect to reuse without modification the software implementing the functionality. Formalized abstract data structures and processes (abstract data types) are relatively slow to change, but FORTRAN has not been adequate in implementing abstract data types. The stability of the functionality over time is a key factor in determining the difficulties of

?RO Ada is a registered trademark of the United States Government (Ada Joint Program Office)

E-SYSTEMS

FACTORS DETERMINING THE DIFFICULTY OF COMPONENT REUSE

GARLAND DIVISION

SOURCE AND TARGET OF REUSE RELATIONSHIP	CATEGORY OF FUNCTIONALITY REUSED	REPRESENTATION REUSED	AFFECTED SYSTEM COMPONENT
1. REUSE IN THE NEXT VERSION OF SAME SYSTEM	1. PHYSICAL OR MATH MODELS	1. PROBLEM	1. HARDWARE
2. REUSE IN DIFFERENT SYSTEM, BUT SAME APPLICATION DOMAIN	2. APPLICATION FUNCTIONS DEPENDENT ON PHYSICAL LAWS	2. DESIGN	2. SOFTWARE
3. REUSE ACROSS ALL APPLICATION DOMAINS	3. FORMALIZED ABSTRACT PROCESSES AND DATA STRUCTURES	3. IMPLEMENTATION	3. MANUAL PROCEDURES AND FORMS
	4. FUNCTIONS DEPENDENT ON HUMAN DECISIONS		4. PEOPLE

- THE LEVEL OF FUNCTIONALITY REUSED IS THE DOMINATING FACTOR IN BOTH DIFFICULTY AND PAYOFF OF REUSE

D5031006-140

FIGURE 1- REUSE COMPLEXITY FACTORS

reusing the software providing the functionality.

Software Reuse Is A System Issue

To attain software reuse the focus has to broaden out to the entire mission critical system including people, manual procedures and forms, hardware, and software. In the situation of reusing software in the next version of the same system, frequent changes are the movement of functionality from special electronics, manual procedures and forms, and people to software. These are system level decisions and determine the functionality to be provided by the software and thus determine reusability.

Modifiability Is A Prerequisite To Reusability

It is not reasonable to expect plug compatible software components except in slowly changing functionality situations. With the continual environment changes beyond the control of the mission critical systems and the movement of functionality from one component to another, the best attainable reuse of application specific software is with modifications proportional to the differences in the existing components functionality and the needed functionality. Thus, modifiability is a prerequisite to reusability.

Reusable Software Components Environment Development Issue

The development of a reusable software components environment is at least as difficult as developing the Software Requirements Engineering Methodology (SREM), UNIX Environment, and Smalltalk-80 Environment. It took each of these systems 18 + 3 years to pass from initial conception into widespread use (Riddle 1984). The basic research and concept formulation for each was at least 5 years, followed by approximately 4 years of development and prototyping of underlying concepts, followed by 3 to 8 years extensive exploratory use and enhancement, and finally approximately 2 years of preparation for release as a product.

Conclusion

The reuse of software is a difficult problem and a reusable software components environment would take at least twenty years to become production quality. But, near-term payoff can be realized, while moving toward the long-term reusable software components environment, by decomposing the problem into the types of changes hindering reuse. For example:

(1) The Ada technology shows strong promise of significantly improving the software to computer hardware interface (change computer hardware).

(2) The abstract interface techniques being used on the upgrade to the A-7E at NRL have promise in the software to special electronics interface (change special electronics) (Britton et al. 1981; Parker et al. 1980).

(3) The information hiding techniques may soon be engineered for general use and improve the software to software interface (Parnas 1972; Parnas 1976; Parnas 1979; Parnas et al. 1983; Parnas et al. 1984).

The person, manual procedures, and manual forms functionality movement to software are the changes which are taking place most rapidly and are receiving the least research attention.

REFERENCES

(1) Balzer, R. and Goldman, N. 1979. Principles of good software specification and their implications for Specification Languages. Proceedings Specifications of Reliable Software Conference. Cambridge, Massachusetts, pp. 58-67.

(2) Balzer, R., Cheatham, T.E. and Green, C. 1983. Software Technology in the 1990's: Using a New Paradigm. Computer Vol. 16, No. 11., pp. 39-45.

(3) Boehm, B.W. 1981. Software Engineering Economics, Prentice-Hall. Englewood Cliff, New Jersey.

(4) Boehm, B.W. and Standish, T.A. 1983. Software Technology in the 1990's. Using an Evolutionary Paradigm. Computer, Vol. 16, No. 11, pp. 30-37.

(5) Booch, G. 1983. Software Engineering With Ada. Benjamin/Cummings, Menlo Park, California.

(6) Buxton, J.N. and Druffel, L.E. 1980. Requirements for an Ada Programming Support Environment: Rationale for Stoneman. Compsac 1980. pp. 66-72.

(7) Curtis, B. 1984. Fifteen years of Psychology in Software Engineering: Individual Differences and Cognitive Science. Seventh International Conference on Software Engineering Proceedings, pp. 97-106.

(8) Davis, C.G. and Vick, C.R. 1977. The Software Development System. IEEE Transactions on Software Engineering. Vol. SE-3, No. 1, pp. 69-84.

(9) Downes, V.A. and Goldsack, S.J. 1982. Programming Embedded Systems With Ada. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

(10) Doyle, J. 1984. Expert Systems Without Computers or Theory and Trust in Artificial Intelligence. The AI Magazine, No. 5, No. 2, pp. 59-63.

(11) Fairley, R.E. 1985. Software Engineering Concepts. McGraw-Hill Book Company, New York.

(12) Fickas, S. 1982. Automating the Transformational Development of Software. Ph.D Thesis. Computer Science Department, U.C. Irvine.

(13) Goldberg, A. 1984. Smalltalk - 80: The Interactive Programming Environment. Addison-Wesley Publishing Company, Reading, Massachusetts.

(14) Green, C., Luckham, D., Balzer, R., Cheatham, T., and Rich, C., 1983. Report on a knowledge-based software assistant. RADC-TR-83-195.

(15) Heninger, K.L. 1980. Specifying Software Requirements for Complex Systems: New Techniques and Their Application. Transactions on Software Engineering SE-6:2-13.

(16) Mostow, J. 1984. Rutgers Workshop on Knowledge-Based Design. SIGART Newsletter. pp. 19-32.

(17) Parker, R.A., Heninger, K.L., Parnas, D.L. and Shore, J.E. 1980. Abstract Interface Specifications for the A-7E Device Interface Module, NRL, Washington, D.C., Memo Rep. 4385.

(18) Parnas, D.L. 1979. Designing Software for Ease of Extension and Contraction. Transactions on Software Engineering SE-5: 128-137.

(19) Rich, C. and Shrobe, H.E. 1979. Design of a Programmers Apprentice. Artificial Intelligence: an MIT Perspective, edited by P.H. Winston and R.H. Brown. The MIT Press, Cambridge, Massachusetts, pp. 137-173.

(20) Rich, C. and Waters, R.C. 1981. Abstraction, Inspection and Debugging in Programming. MIT AI memo no. 634.

(21) Shapiro, E.Y. 1983. Algorithmic Program Debugging. The MIT Press, Cambridge, Massachusetts.

(22) Soloway, E. 1984. A Cognitively-Based Methodology for Designing Languages/Environments/Methodologies. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, P. Henderson (Editor). April 23-25. pp. 193-196.

(23) Zave, P. and Yeh, R.T. 1981. Executable Requirements for Embedded Systems. Fifth International Conference on Software Engineering: 295-304.

(24) Zelkowitz, M.V., Shaw, A.C., and Gannon, J.D. 1979. Principles of Software Engineering and Design. Prentice Hall, Englewood Cliffs, New Jersey, pp. 9-11.

BIBLIOGRAPHY

(1) Britton, K.H., Parker, R.A. and Parnas, D.L. 1981. A Procedure for Designing Abstract Interfaces for Device Interface Modules. Fifth International Conference on Software Engineering, pp. 195-204.

(2) Parker, R.A., Heninger, K.L., Parnas, D.L. and Shore, J.E. 1980. Abstract Interface Specifications for the A-7E Device Interface Module, NRL, Washington, D.C., Memo Rep. 4385.

(3) Parnas, D.L. 1972. On the Criteria to be used in Decomposing Systems into Modules. Communication of the ACM 12: 1053-1058.

(4) Parnas, D.L. 1976. On the Design and Development of Program Families. Transactions on Software Engineering SE-2.

(5) Parnas, D.L. 1979. Designing Software for Ease of Extension and Contraction. Transactions on Software Engineering SE-5. pp 128-137.

(6) Parnas, D.L., Weiss, D.M., Clements, P.C. and Britton, K.H. 1983. Interface Specifications for the SCR (A-7E) Extended Computer Module, NRL, Washington, D.C., Memo Rep. 4843.

(7) Parnas, D.L., Clements, P.C. and Weiss, D.M. 1984. The Modular Structure of Complex Systems. Seventh International Conference on Software Engineering. pp. 408-417.

(8) Riddle, W.E. 1984. The Magic Number Eighteen Plus or Minus Three: A Study of Software Technology Maturation. ACM SIGSOFT Software Engineering Notes. Vol. 9, No. 2, pp. 21-37.

(9) Solway, E. 1984. A Cognitively-Based Methodology for Designing Languages/Environments/Methodologies. Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, P. Henderson (Editor). April 23-25. pp. 193-196.

(10) Wegner, P. 1984. Capital-Intensive Software Technology. IEEE Software, Vol. 1, No. 3, pp. 7-45.

RESUME

J.G. SNODGRASS

Mr. J.G. Snodgrass has a B.S. Degree in Mathematics from Arkansas State University, a M.S. Degree in Mathematics from Texas A&M University, a M.S. Degree in Computer Science from Southern Methodist University, and is currently working on a dissertation for a Ph.D in Computer Science at Southern Methodist University. Mr. Snodgrass, a member of the technical staff, is responsible for transitioning the Garland Division of E-Systems to Ada-based software engineering environments and for research projects utilizing Ada and artificial intelligence technologies to improve software quality and software engineering productivity. Also, he participates in 1) new business and proposal activities associated with Advanced Software Technology and 2) the Reusable Software and Knowledge Engineering Thrusts of the Software Productivity Consortium.

Prior to joining E-Systems in early 1983, Mr. Snodgrass was Chief of Engineering Software Development for General Dynamics at Fort Worth, Texas. In this position he managed a group of twenty software engineers and provided technical leadership and consultation on proposals and software development projects. The applications included J73 compilers, software tools, avionics algorithms, factory automation simulation, Ada methodology, and factory energy management automation. Also, he led technical volumes of proposals in areas of Tactical Campaign modeling, Modern Programming Environments, support software for F-16 aircraft subsystems, and Ada technology.

Prior to assuming management responsibility in 1980, Mr. Snodgrass held positions of technical leadership in modern programming environment projects and C?3I system IR&D projects. Prior to assuming technical leadership responsibility, Mr. Snodgrass was involved in designing, implementing, and testing software systems for C?3I, F-16 weapons system, and electronic warfare equipment evaluation applications.

E-SYSTEMS

GARLAND DIVISION

FUNDAMENTAL TECHNICAL ISSUES OF REUSING MISSION CRITICAL APPLICATION SOFTWARE

J.G. SNODGRASS

9-12 APRIL 1985

E-SYSTEMS

TECHNICAL ISSUES IMPLY REUSABLE COMPONENTS DEVELOPMENT FOCUS

GARLAND DIVISION

- FACTORS DETERMINING THE DIFFICULTY OF COMPONENT REUSE
- TOTAL SYSTEM CHARACTERISTICS
- MODIFIABILITY

- REUSABLE SOFTWARE COMPONENTS ENVIRONMENT DEVELOPMENT FOCUS

E-SYSTEMS

HIGHER LEVEL REUSABLE FORM PROVIDED BY ADA PACKAGE

GARLAND DIVISION

CHARACTER SETS

A . . . Z 0 . . . 9 " . . & ' ( ) + - * / : ; < = > _

EXPRESSIONS

```
OVEN_TEMP + 100
```

STATEMENTS

```
type TEMPERATURE is range 0 . . . 10000;
OVEN_TEMP: TEMPERATURE;
if OVEN_TEMP: > 950 TURN OFF;
```

STATEMENT BLOCKS

```
case PIXEL_COLOR is
   when RED/GREEN/BLUE =>
      INCREASE_SATURATION;
   when CYAN . . WHITE =>
      MAKE_BLACK;
   when others => null
end case;
```

SUBPROGRAMS

```
procedure ROTATE (POINTS: in out COORDINATE) is
   begin
   - - sequence of statements
   end ROTATE;

function "+" (A,B : in NUMBER) return NUMBER is
   begin
   - - sequence of statements
   end "+";
```

05060684 128

PACKAGES

```
package PASSWORD is
   type VALUE is limited private;
   function   IS_VALID(CODE : in  VALUE) return BOOLEAN;
   procedure SET      (CODE : out VALUE; AUTHORIZATION_LEVEL : in NATURAL);
private
   type VALUE is new STRING(1 . . 46);
end PASSWORD;

package COMPLEX is
   type NUMBER is record
                    REAL_PART      : FLOAT;
                    IMAGINARY_PART : FLOAT;
                  end record;
   function "+" (A,B : in NUMBER) return NUMBER;
   function "-" (A,B : in NUMBER) return NUMBER;
   function "*" (A,B : in NUMBER) return NUMBER;
end COMPLEX;

package body COMPLEX is
   function "+" (A,B : in NUMBER) return NUMBER is
      RESULT : NUMBER;
   begin
      RESULT.REAL_PART:= A.REAL_PART + B.REAL_PART;
      RESULT.IMAGINARY_PART:= A.IMAGINARY_PART + B.IMAGINARY_PART;
      return RESULT;
   end "+";
   function "-" (A,B : in NUMBER) return NUMBER is
   begin
      return NUMBER' (REAL_PART      => A.REAL_PART - B.REAL_PART,
                      IMAGINARY_PART => A.IMAGINARY_PART -
                                        B.IMAGINARY_PART);
   end "-";
end COMPLEX;
```

(BOOCH 1983)

E-SYSTEMS

SYSTEM LEVEL DECISIONS STRONGLY AFFECT SOFTWARE REUSABILITY

GARLAND DIVISION

- SOFTWARE EMBEDDED IN A LARGER SYSTEM
- UNKNOWN AND CHANGING PROBLEM
- MANY PATHS TO SOLUTION (DAVIS AND VICK 1977; SHAPIRO 1983)

- SOFTWARE CONFORMS TO THE OTHER SYSTEM COMPONENT SPECS
- SOFTWARE PICKS UP ADDITIONAL FUNCTIONALITY WHEN OTHER SYSTEM COMPONENTS FALL SHORT

(BOEHM 1981; BUXTON AND DRUFFEL 1980; DAVIS AND VICK 1977; DOWNES AND GOLDSACK 1982; FAIRLEY 1985; HENINGER 1980)

DS031005-150

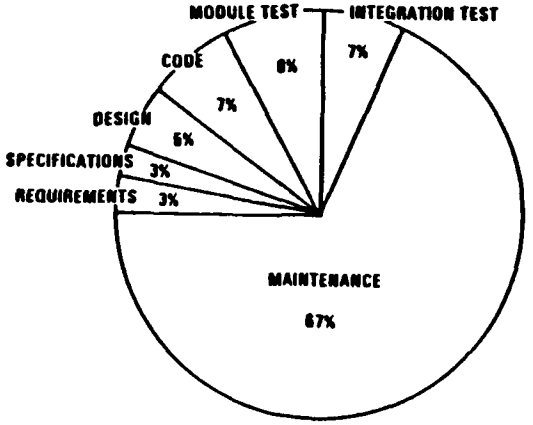
E-SYSTEMS

MODIFICATION PROMINENCE IMPLIES UNDERLYING PROCESS

GARLAND DIVISION

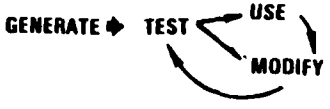
- MAINTENANCE OR MODIFICATION IS ACCOMPLISHED TO:
  - CORRECT ERRORS IN REQUIREMENTS, SPECIFICATION, DESIGN, AND CODE → IMPROVED RELIABILITY
  - ADD CAPABILITY
  - REMOVE CAPABILITY → IMPROVED REQUIREMENT SATISFACTION
  - INCREASE EXECUTION SPEED
  - REDUCE MEMORY REQUIREMENTS
  - REPLACE OUTDATED HARDWARE → IMPROVED EFFICIENCY

LIFECYCLE COST BREAKDOWN



(ZELKOWITZ ET AL. 1979)

- MODIFICATION IS FUNDAMENTAL TO THE SYSTEM DEVELOPMENT PROCESS



D5000604 124

E-SYSTEMS

REUSE FACTORS, MODIFIABILITY, AND TECHNOLOGY IMPLY APPROACH

GARLAND DIVISION

NEAR TERM & MIDTERM

- DECOMPOSE REUSE PROBLEM INTO SUBPROBLEMS BASED ON SOURCE/TARGET RELATIONSHIPS, FUNCTIONALITY CATEGORY, REPRESENTATIONS, AND AFFECTED SYSTEM COMPONENTS
- ADAPT EXISTING AND EMERGING TECHNOLOGIES TO A TOTAL SYSTEM AND MODIFIABILITY FOCUS
  - ADA LANGUAGE AND ENVIRONMENT
  - INTEGRATION OF EXISTING TECHNIQUES (BOEHM AND STANDISH 1983)
  - INFORMATION HIDING (PARNAS 1979)
  - OPERATIONAL REQUIREMENTS SPECIFICATION (BALZER AND GOLDMAN 1979; ZAVE AND YEH 1981)
  - ABSTRACT INTERFACE SPECIFICATION (PARKER ET AL 1980)
- ATTACK SUBPROBLEMS WITH ADAPTED TECHNOLOGIES AND MERGE SOLUTIONS INTO A MODEL OR METHODOLOGY

85031005-146

E-SYSTEMS

REUSE FACTORS, MODIFIABILITY, AND TECHNOLOGY IMPLY APPROACH (CONT'D)

GARLAND DIVISION

MID TERM & FAR TERM

- PERFORM BASIC RESEARCH TO ARTICULATE THE SYSTEM (NOT JUST SOFTWARE) DEVELOPMENT PROCESS (CURTIS 1984; FICKAS 1982; GOLDBERG 1984; MOSTOW 1984; RICH AND SHROBE 1979; RICH AND WATERS 1981; SOLOWAY 1984)
- ENGINEER THE ARTICULATED PROCESS FOR USE AS A MODEL OR METHODOLOGY (DOYLE 1984)
- BUILD KNOWLEDGE-BASED ASSISTANTS TO SUPPORT THE MODEL (BALZER ET AL 1983; GREEN ET AL 1983)

0S031005 148

RESPONSE TO THE TRI-SERVICE WORKING GROUP WORKSHOP REUSABLE COMPONENTS OF APPLICATION SOFTWARE

CORPORATE SOFTWARE EXPERTISE IN THE AREA OF SPECIFICATION/DESIGN

J. S. Squire
Westinghouse Electric Corp. DESC

Abstract

*The Westinghouse software capability provides a strong combination of software development experience and software support environment design. Westinghouse has a ten year history as a leader in programmable signal processors (PSP) as well as avionics systems design, development, and integration in such systems as EAR, F16, and B-1B. The PSP series (with users such as MIT/Lincoln Labs, ERIM, and Grumman) and Ada?*O support environment development (for VHSIC Phase I to support advanced multiprocessor system design) have established Westinghouse as a leader in the field of real-time systems development.*

Software Development Experience

WEC is well-versed in developing operational system software for a variety of signal processing (SP) applications in a manner consistent with top-down, structured software design and test techniques as defined in MIL-STDs 483 and 1679. Within applications of radar, sonar, EW, and image processing, we have demonstrated our expertise on such projects as APG-66, B-1B, APG-68, ALQ-131, ASPJ, and E-3A. Extensive experience in Software Engineering practices has enabled WEC to develop a consistent approach to the management of software programs. All software efforts utilize the same Software Management Guide (used in accordance with Government directives and/or contract requirements) which forms the foundation of any Computer Program Development Plan (CPDP) required for any specific contract. Our software development approach measures each phase of software development against the requirements as shown in Figure 1. This ensures Computer Program Configuration Item (CPCI) baseline control and assures a quality of excellence in the products developed.

Software Support Environment Design

Westinghouse has extensive experience in the development of support tools for production software environments based on FORTRAN, JOVIAL, and most recently, Ada. Recent JOVIAL support environment experience includes development and integration of object code file managers, relocatable loader/linkage editor, assemblers, simulators and automatic software documentation packages into a cohesive user-oriented package interfacing with host manufacturer's commonly supplied support tools (e.g., editors, file managers, etc.). Software maintenance is enhanced throughout the life cycle via extensive software module documentation, cross reference listings, and software RN generation for traceability. Software reliability is ensured through a combination of modern software engineering practices coupled with firm management disciplines, including:

- configuration identification,
- change control,
- status accounting,
- configuration control centers, and
- software control centers.

Westinghouse has adopted the modern software engineering practices (structured programming, chief programmer teams, structured walk throughs, etc.) necessary to complement efficient real-time operating systems development.

Over the past 15 years, we have developed a number of system software development environments which have been

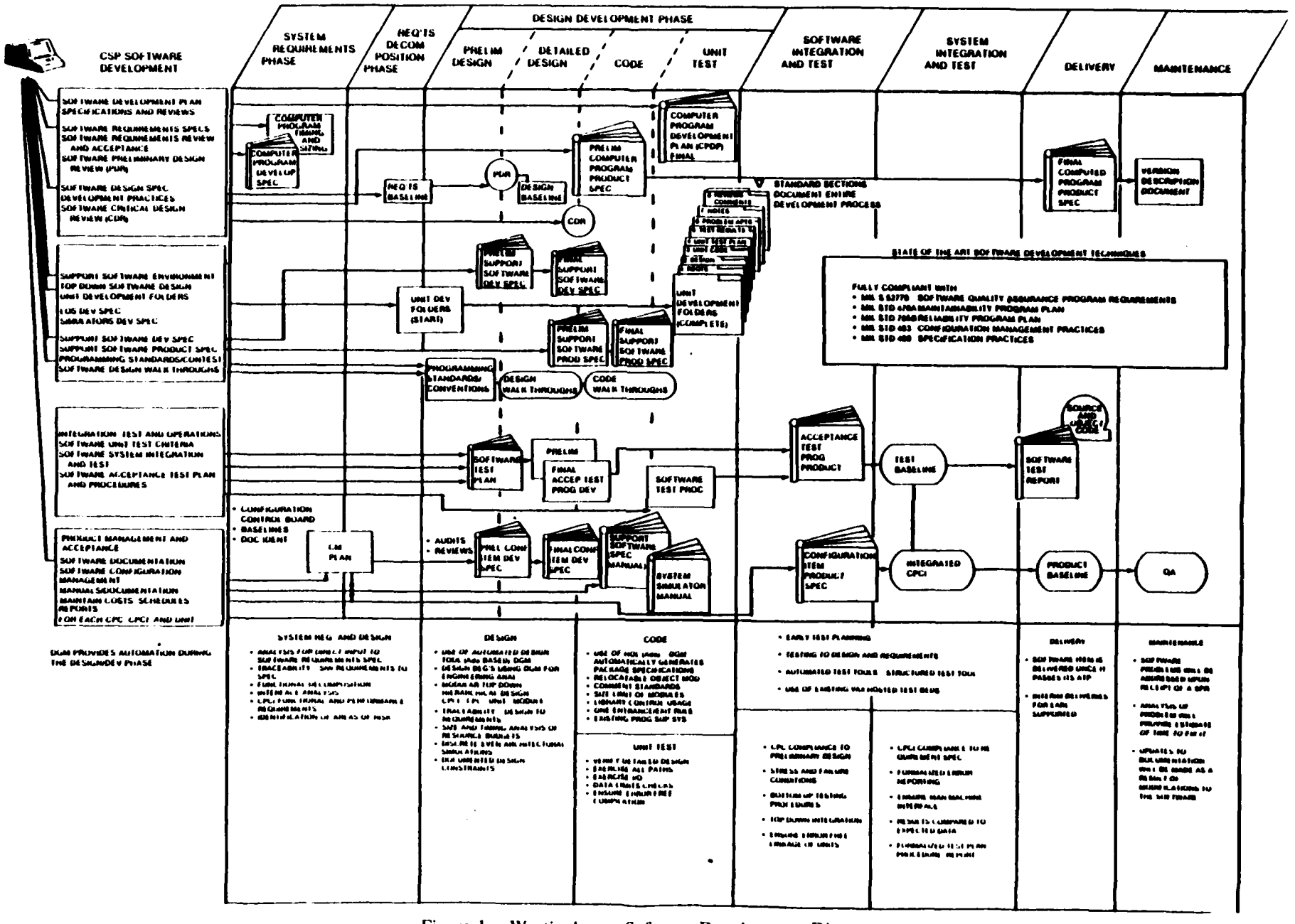


Figure 1. - Westinghouse Software Development Diagram

used to generate code for real-time operating systems applications and the maintenance thereof. Westinghouse developments in the system software area have paralleled advances in computer science technology. These include the compiler-linker-loaders for systems such as the F4 and E3A. Presently, Westinghouse is working on state-of-the-art systems which include not only compilers-linkers-loaders, but additional software development facilities for various levels of simulation, debug, and maintenance. The simulation capabilities range from register levels in object machines to system and data flow levels among machines. The program development capabilities include HOL level debugging facilities which provide for real-time interactive program monitoring. The maintenance capabilities include automatic configuration control management aids, such as the Westinghouse Automatic Revision Control System (ARCS) which controls the manner by which one may revise code and automatically updates all pertinent files and control mechanisms. Most of these facilities operate from common data bases. Many of these facilities are data base driven. An example is the Westinghouse compiler facility, wherein more than one language compiler utilizes a machine definition data base to generate code for a single object machine. At the same time, a single compiler utilizes the data base which contains many object machine descriptions to generate code for various machines.

Many of these techniques are currently being utilized in the development of an Ada environment associated with the ongoing DOD sponsored VHSIC program. Westinghouse has been actively involved in Ada since the formation of the High Order Language Working Group (HOLWG). At this facility, there are currently 6 Ada compilers (including the latest release of the DEC Ada compiler) in use on commercial computers; and, Westinghouse will continue to acquire these compilers as they become available. Since 1979, Westinghouse has been actively involved in the development of an Ada compiler (MIL-STD-1815A) for the MIL-STD-1750A Instruction Set Architecture (ISA). This VAX/VMS hosted Ada compiler is designed and implemented to provide good software engineering productivity for the 1750A target computer as well as the Westinghouse VHSIC Signal Processors (VSPs). It is an integral part of the Ada support environment within the Integrated Design System (IDS) that is presented in this package.

As a result of the IDS development on Phase I of the VHSIC program and associated software technology research Westinghouse has embarked on a program addressing not only the Ada program support environment, but also the total mission support environment as related to large, diverse, sensor-based systems. This Westinghouse concept, the Mission Design Support System (MIDSS), consists of a number of state-of-the-art software programs and proposed concepts developed in an Ada environment and integrated to form a system design automation package that is usable by operations analysts, system architects, Ada programmers, and more. The software ranges from the top-level Mission Design Language through the basic hardware interface or signal processing procedures. The Directed Graph Methodology (DGM) is the basis for the Mission Design Language concept which allows a user to interface to this multi-level environment at any convenient design point and hierarchically traverse to each successive level of detail. DGM and its tool-set is the philosophy/methodology that is presented in this package.

SPECIFICATION/DESIGN
THE WESTINGHOUSE INTEGRATED DESIGN SYSTEM

In support of the VHSIC Phase I program, WEC developed an Integrated Design System (IDS) that assists the systems analyst and software engineer in developing a Signal Processing (SP) application from the requirements level to the actual coding of the algorithm. WEC has installed and effectively used the tools of the IDS within its software development environment. The IDS is pictorially represented in Figure 2 in which the Directed Graph Methodology (DGM) is essentially the "focal point" of the IDS. DGM's integrated tool-set is pictured in Blocks 1 and 3 of Figure 2 and includes the following: (1) DGM Editor, (2) DGM Library Manager, (3) DGM Plot Package, (4) DGM Translator, and (5) DGM Simulators. The association of DGM with the remaining tools of the IDS is pictured in Blocks 2, 4, and 5 (not presented in this package).

DGM supports top-down structured design techniques, is used on-line, provides automatic documentation, and generates code (Ada package specifications) during the Design/Development Phase of the Software Development Process (as shown in Figure 1). A description of how DGM supports the development of this software follows.

Design Philosophy - Westinghouse Integrated Design System

WEC has developed an Integrated Design System (IDS) that assists the systems analysts and software engineers in transforming an application to Ada/SP source and in documenting it. This source can then be compiled by the Ada compiler. The IDS provides the necessary tools for designing architectural configurations and allows the analyst to evaluate these architectures against real-time SP requirements. This design philosophy has been applied to radar mode(s) and benchmarks. In implementing this philosophy and designing/developing application software, 4 steps are generally followed:

(1) Selection of an algorithm

(2) Representation of the algorithm by a data flow graph--consisting of nodes and arcs,

(3) Adaptation of the algorithm to DGM, and

(4) Coding of the nodes in Ada

These steps are shown in Figure 3 and demonstrate how DGM is used within WEC's IDS.

(1) Selection of an Algorithm

The system designer begins by defining system requirements for a SP processing mode(s) he is attempting to analyze as pictured in Block 1 of Figure 3. The algorithm(s) necessary for the execution of the mode are established. Next, a top level block diagram of that algorithm is produced to some level of refinement (detail). It is at this point that the algorithm can begin to be represented as a data flow graph.

(2) Representation of the Algorithm by a Data Flow Graph

After the selection of an algorithm, the next step in the development of the software is to group logically related functions and subroutines into entities that will be coded in Ada as packages. Block 2 of Figure 3 represents this step of the design/development process. After identifying the entities (which are called nodes in DGM and packages in Ada), the data flow graph is drawn.

(3) Adaptation of the Algorithm to directed Graph Methodology

DGM is a means of specifying a SP application in a distributed system environment. Its menu-driven interactive tools

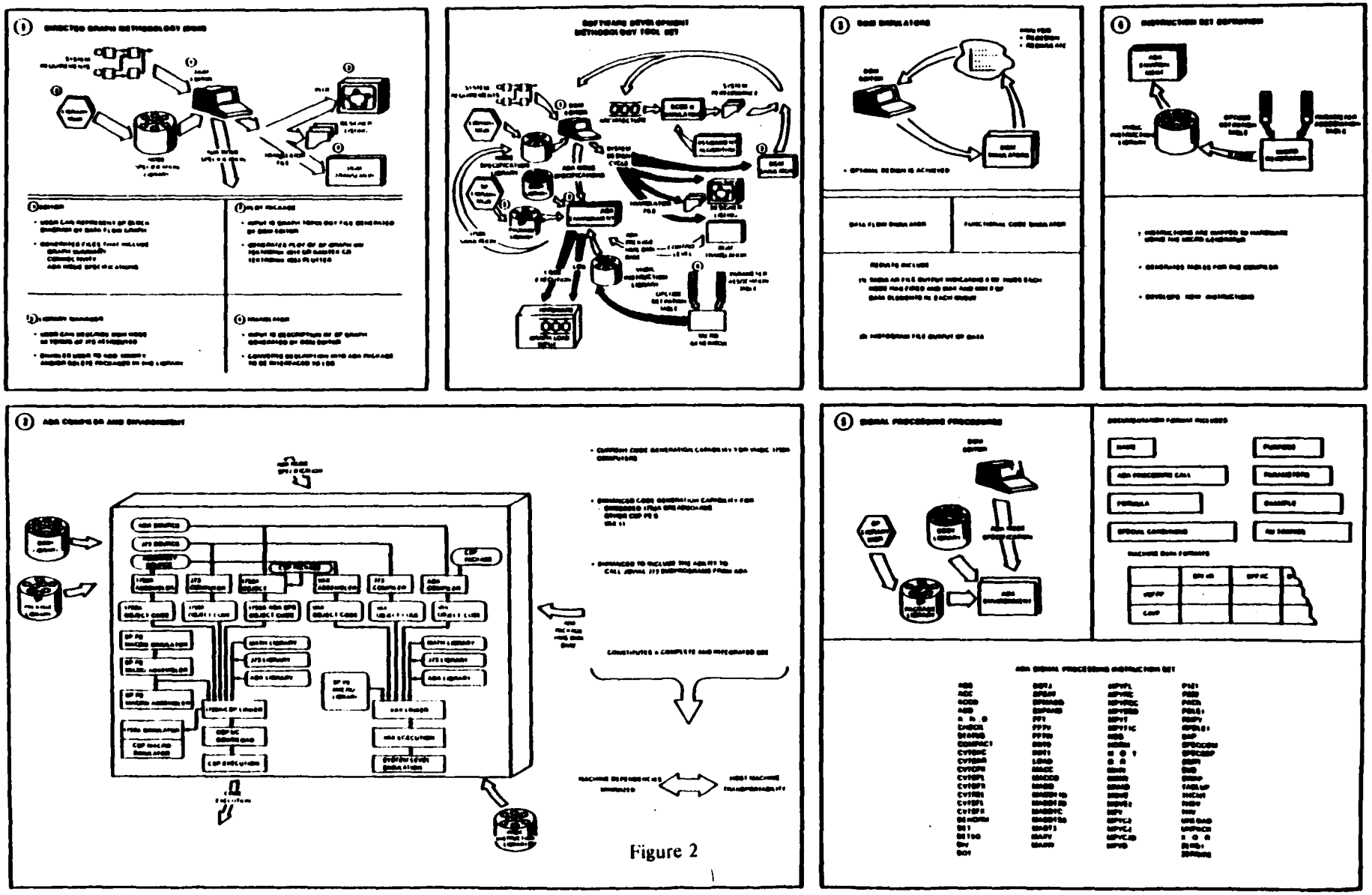


Figure 2

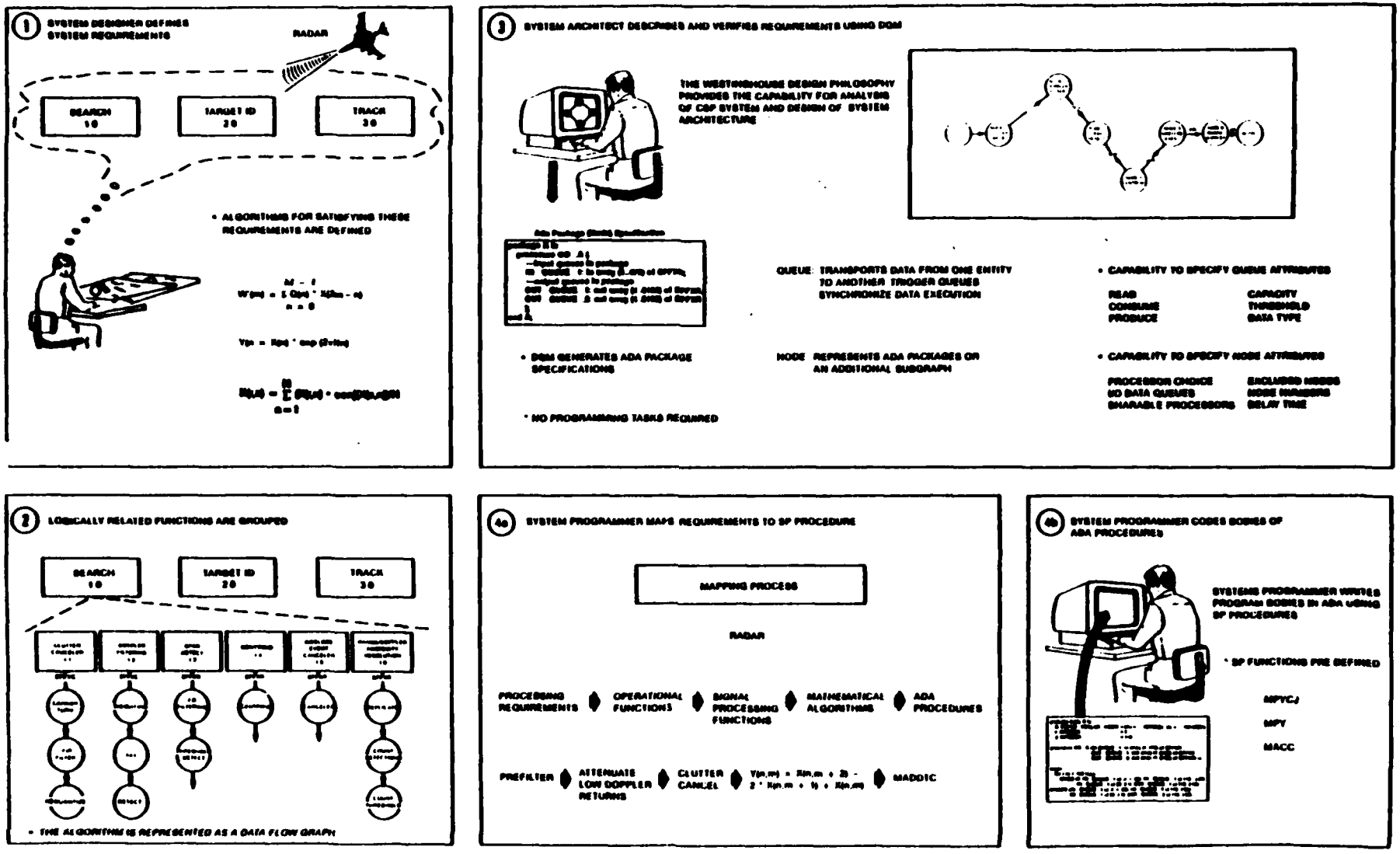


Figure 3. - Implementation of DGM Within the Integrated Design System

enable a user to enter textual data on the VAX describing the requirements of the graph (node and queue attributes, etc.). Once the information is entered and edited, it is then transferred into a translator whose output is a collection of Ada routines and transport tables. Block 3 of Figure 3 shows how the system architect describes his requirements using DGM and emphasizes the point that no programming tasks are required at this stage.

(4) Coding of the Nodes in Ada

Nodes are then coded as Ada packages, as diagrammed in Blocks 4a and 4b of Figure 3. Prior to the coding of each node, the systems programmer maps the SP requirements to the SP function (which can be broken down into an instruction or collection of instructions (Ada procedures). These instructions are defined in Ada as procedures and can be invoked by a procedure call statement. At present, 80 generic instructions have been defined in Ada as procedures (see Vol I of the enclosed manual) and can be used for radar, sonar, EW, and other SP applications.

The actual implementation of the SP instruction occurs when the microcode associated with the instruction is executed. The support package specifies the type of processor to be used for processing the node, defines the data types used, and lists all the functions and procedure calls from the SP library that will be used in the implementation package.

Implementation

DGM clarifies system requirements by constructing data flow graphs out of SP block diagrams. Nodes, queues, graph variables and optional mode control programs define a pictorial representation of an application. The vertices of the directed graph are called nodes and the edges are called graph queues. Each node on a graph represents a processing element of the graph and can be defined as an Ada package or subgraph.A graph queue transports either data or synchronization pulses between two nodes. Through the input and output queues, the node communicates with other nodes in the graph. Graph variables enable communication of information between nodes, the graph's control program, and the outside world. Several data files are created by the DGM Editor. These files summarize the attributes of the graph, its connectivity, data flow, queue and node requirements, and provide output files for the DGM Translator, ECSS II System Level Simulator, Ada compiler, and Plot Package. The automatic generation of these files is a cost-saving feature of DGM; and the data and code that is created by these files assist in the documentation, translation, and specification of SP applications. Thus, DGM reduces programming costs.

Throughout the System Design/ Development Process, simulation plays a key role in the verification/validation of the top level design down through the gate level abstract design. At the functional simulation level, there exists two (2) simulators for the DGM Graph--the Data Flow Simulator and the Functional Code Simulator, as shown in Block 3 of Figure 2. Both simulators allow the user to simulate the data flow of the SP graph which has been defined via the DGM Editor. Both verify the queue attributes and firing rates and check for deadlock conditions. The primary outputs are statistical results of the execution of the algorithm.

In summary, DGM provides us with a means of specifying the entire system even before the coding of the modules starts; it supports the development of highly modular systems. In most software development methodologies, integration and maintenance of the programs are both difficult and costly. With DGM techniques and separately compiled units in Ada, software integration is greatly facilitated. Several software engineers can write code with a minimal amount of interaction; they can also compile small portions of the program separately. Once all the packages for a given mode are coded, DGM will generate the necessary tables and specifications to connect all the entities together and ensure synchronized execution. The impact on software maintenance is also favorable because if changes are made to any of the packages, the rest of the system compilation units will not be affected.

Table 1 - IDS Tool-Set Summary Sheet

SOFTWARE ITEM	TOOL	FUNCTION	DEVELOPER	% COMPLETED	HOST LANGUAGE	HOST COMPUTER	
						VAX	OTHER
SIMULATORS	EXTENDABLE COMPUTER SYSTEM SIMULATOR (ECSS II)	SYSTEM LEVEL SIMULATOR (SLS)	FEDSIM	20%	SIMSCRIPT	✓	CDC UNIVAC IBM
	1750A SIMULATOR	MACRO-SIMULATOR	WESTINGHOUSE	100%	FORTRAN	✓	–
	N mPc META MICRO ASSEMBLER MODEL ISP' MICRO SIMULATOR MODEL	MICRO-SIMULATOR	CASE WESTERN UNIVERSITY	100%	'C'	✓	–
PROGRAM PREPARATION SUPPORT SOFTWARE	ADA COMPILER ADA/1750A (WADA)	CODE GENERATION FOR 1750A COMPUTERS, ADM, PE'S	WESTINGHOUSE	60%	FORTRAN	✓	UNIVAC
	ADA/VAX	CODE GENERATION FOR VAX II HOST COMPUTER	TELESOFT	100%	PASCAL	✓	–
	JOVIAL J73 COMPILER J73/1750A J73/VAX	CODE GENERATION FOR 1750A COMPUTERS, ADM, PE'S CODE GENERATION FOR VAX II HOST COMPUTER	SEA SEA	100% 100%	JOVIAL JOVIAL	✓ ✓	
	META MICRO ASSEMBLER	MICRO-ASSEMBLER	CASE WESTERN UNIVERSITY	100%	'C'	✓	–
	1750A ASSEMBLER	MACRO-ASSEMBLER	WESTINGHOUSE	100%	FORTRAN	✓	
	LINKER/LOADER	LINKER/LOADER	WESTINGHOUSE	60%	FORTRAN	✓	
	SIGNAL PROCESSING PROCEDURES	MACRO LIBRARY	WESTINGHOUSE	100%	ADA	✓	–
	DIRECTED GRAPH METHODOLOGY TRAN	LANGUAGE TRANSLATOR	WESTINGHOUSE	50%	PASCAL	✓	–
	DGMED LIBMGR	DGM EDITOR/GENERATES ADA PACKAGE SPECS DGM LIBRARY MANAGER	WESTINGHOUSE	100%	ADA	✓	–
	DGM SIMULATORS DFSIM FCSIM	DGM DATA FLOW SIMULATOR DGM FUNCTIONAL CODE SIMULATOR	WESTINGHOUSE	100% 100%	ADA ADA	✓ ✓	–
LOCAL OPERATING SYSTEM	CLUSTER MASTER EXECUTIVE NODE MANAGEMENT EXECUTIVE BULK MEMORY EXECUTIVE	LOCAL OPERATING SYSTEM	WESTINGHOUSE	10%	ADA		1750A CVP GLOBL MEM

Table 1 provides a summary of the tools within the IDS by item name, description, function, developer, % completed, host language, and host computer. Many of the tools (in their current state) have been delivered to the Government under the VHSIC Phase I contract. These tools include:

- 1750A macro simulator
- Signal processing procedures (macro library)
- Directed Graph Methodology (Editor and Library Manager), and
- DGM Simulators (Data Flow and Functional Code).

At this time the cost and/or licensing of the the ECSS II Simulator from FEDSIM) is still being reviewed internally at Westinghouse.

RESUME

Jon Stuart Squire
Manager

EDUCATION

BS, Electrical Engineering Univ. of Michigan, 1960

MS, Electrical Engineering Univ. of Michigan, 1962

MS, Mathematics Univ. of Michigan, 1963

Computer Science Studies The Johns Hopkins Univ., 1971, 1972; Univ. of Maryland 1979,1980

EXPERIENCE

1979-Present

Westinghouse Electric Corporation Defense and Electronic Systems Center Baltimore, MD, Manager, Software Engrg.

1970-Present

Manager, Software Engineering responsible for DESC software standards and procedures, compiler, assemblers, automatic documentation, computer-aided system and related software areas. This included compilers targeted to the MIL-STD-1750 Instruction Set Architecture for Ada, Jovial and Fortran Languages.

1969-1970

Application and Operational Software, Systems Development Division; Supervisory Engineer. Responsible for coordination of computer-aided design effort for this division. Responsible for training, staffing and line supervision of operational software developments such as B57G avionics computer software, F-15 radar director software, SAN-DRAM weapon delivery software.

1966-1969

Advanced Control Data Systems. Fellow Engineer. Group leader responsible for advanced computer organization development and system software programming for Aerospace computers. Specified the organization of a general-purpose satellite-borne computer being built for NASA. Program Manager for NASA ON-BOARD PROCESSOR software including assembler, loader.

1963-1966

Westinghouse Electric Corporation. Computer and Data Systems. Senior Engineer. Preliminary design of SOLOMON software. Research on numerical techniques for advanced computers. Program manager for development of NELIAC compilers for WEC 2402 computer and IBM 360 computer. Director of software development for Aerospace WIP and ABC computers.

1961-1963
University of Michigan Research Institute, Assistant Research Engineer. Developed a translation algorithm for use in producing code for the Multiple Processor Computer. Developed machine organization of a Multiple Processor computer.

1961-1963
University of Michigan, Instructor, Department of Mathematics.

1960-1962
University of Michigan Research Institute, Programmer-Analyst. Development of a Powerful Problem Oriented Language and Translator.

SOCIETIES

Institute of Electrical and Electronics Engineers

IEEE Electronic Computer Group

Association for Computing Machinery

PUBLICATIONS

(1) "Iterative Circuit Computers," Coauthor, Computer Organization, Spartan Books, Inc., 1963.

(2) "Programming and Design Considerations of a Highly Parallel Computer," Coauthor, AFIPS Proceedings, SJCC, 1963.

(3) "A Translation Algorithm for a Multiple Processor Computer," ACM National Conference, Denver, Colorado, 1963.

(4) "An 11 Cryotron Full Added," IEEE-Transactions. Electronic Computers Correspondence, 1962.

(5) "Techniques for Developing Compilers for IBM System 360," Association for Computing Machinery, Symposium, Baltimore, Maryland, 1965.

(6) "Advanced Concepts of Computer Organization," IFIP Congress, New York, 1965.

(7) "New Techniques to Obtain Ultra Reliable Digital Systems," Westinghouse Technical Report, 1967.

(8) "A General-Purpose Onboard Satellite Computer," Coauthor, Westinghouse ENGINEER, Vol. 29, 1, 1969.

PATENT DISCLOSURES

Optically Alterable Optical Storage Device, 1964.

MOS_FET Permanent Repair Physical Device, 1967.

Digital Logic Simulation, 1969.

Computer-Aided String List Generation, 1970.

MOS CELL Layout, 1969.

BOEING MILITARY AIRPLANE COMPANY (BMAX)

Earl T. Startzman
Ada Information Management System

BOEING MILITARY AIRPLANE COMPANY (BMAC)

RELATED CORPORATE EXPERTISE AND CONTRACTS

BMAC HAS BEEN INTIMATELY INVOLVED IN THE DESIGN AND DEVELOPMENT OF MAJOR WEAPONS SYSTEMS EMPLOYING REAL-TIME BUS-ORIENTED DISTRIBUTED MULTIPROCESSOR ENVIRONMENTS UTILIZING HIGH ORDER LANGUAGE (JOVIAL) APPLICATIONS FOR THE LAST 10 YEARS.

- BMAC has specified, procured, validated, and maintained JOVIAL compilers and maintained a comprehensive software support system for the development and maintenance of large-scale, real-time avionics on the IBM 370. BMAC has also installed and maintained a JOVIAL J73/I Compiler on a DEC-10 under TOPS-10 and has written a code generator for it.

- BMAC has specified, procured, validated, and used JOVIAL compilers with multiple code generators for major military contracts including: B-52 Offensive Avionics System (J3B), General Support Rocket Systems (J73/I), Inertial Upper Stage (J73/I), B-1B Avionics (J3B), and AWACS (J3).

- BMAC has developed a comprehensive integrated support software system which provides facilities for development, test documentation, and maintenance of software for embedded computer systems.

- BMAC personnel participated in the Ada language definition cycle as members of the Phase II Air Force Evaluation Team. Boeing also participated in the Phase II Ada test and evaluation, submitted several language issue reports and presented four papers at the Ada Test and Evaluation conference.

- BMAC is the prime contractor for the Ada Integrated Management System (AIMS).

- BMAC has over three years of hands-on experience with actual MIL-STD-1750 hardware, including flight testing of actual avionics systems. Additionally, BMAC is a charter member of the 1750 Users Group and participated in the original definition of the MIL-STD-1750 ISA.

- BMAC has developed a suite of benchmark tests for Ada which evaluate MIL-STD-1750A code generating effectiveness of Ada compilers.

- BMAC, through its IR&D, is in the process of developing a family of MIL-STD-1553 A/B-based terminal controllers for the MIL-DTD-1553 A/B avionics bus.

The following BMAC contracts include the development of significant quantities of Mission Critical Applications Software, support software and test and maintenance software.

Advanced System Integration Demonstration (Pave Pillar) Contract No. F33615-82-C-1902)

The Advance System Integration Demonstration (ASID) program, designated Pave Pillar, consists of the following projects: the Advanced System Avionics (ASA) project, which will define, design, integrate, test, and evaluate advanced avionic system concepts in a test bed facility; the Integrated Flight Demonstrator (IFD) project, which will integrate and evaluate configurations of these advanced concepts in a flight test environment; and the integrated Communications/Navigation/Identification Avionics (ICNIA) project, which will integrate communication, navigation, and identification functions across the 2MHz to 2GHz spectrum. The ASID program goal is to define, develop, and evaluate new approaches to integrated avionic system technology that will achieve improvements in using ground-based evaluation facilities and an airborne test bed in performing test and evaluation of advanced avionics system concepts. Concepts to be defined, developed, and evaluated include the integration of information from multiple sensors and subsystems toward the end of functional automation, manageable pilot workload more effective and survivable weapon systems through the application of advance technology.

B-1 Avionics (Contract No. F33657-72-C-0600)

In April 1972, Boeing won the Air Force contract to design, develop, integrate, and test the avionics system for the B-1 bomber. Initially, the B-1 avionics system included navigation and weapons delivery, mission and traffic control, controls and displays, the Avionics Control Units Complex, and Stores Management System. This responsibility was later expanded to include the design, development, integration, and test of the defensive management subsystem, and additionally, responsibility for the total avionics integration and installed performance of the radio frequency surveillance and electronic countermeasures subsystem. During this period, Boeing performed a primary role in establishing and coordinating the complex interface activities between associate contractors.

B-1B Avionics (Contract No. F33657-81-C-0212)

In October 1981, Boeing was awarded a contract to develop offensive and defensive management avionics for the B-1B, an natural extension of Boeing efforts on the B-1 avionics program.

The latest technology equipment being incorporated into the B-1B, such as state-of-the-art synthetic aperture radar, capitalizes upon the work now being performed under the B-52 OAS contract. With two production readiness reviews, successfully completed, the first flight date of 31 December 1984 is expected to occur as scheduled.

B-52 Offensive Avionics System (OAS) (Contract No. F33657-78-C-0500)

In the B-52 Offensive Avionics System (OAS), Boeing is replacing the low reliability analog bombing and navigation system originally installed on B-52G/H aircraft with high reliability solid state digital equipment that provides the benefits of greater accuracy, smaller size, reduced weight, and decreased maintenance costs.

The OAS consists of a set of integrated weapon control and delivery and navigation subsystems linked to three digital avionics control units (ACUs) by a MIL-STD-1553A data bus. In addition to improving reliability, it significantly improves performance and provides a nuclear hardened system, including the weapons release train. The ACUs perform all navigation and weapons delivery computations and control communications between the various OAS equipment. Boeing has developed and delivered the operational software ground maintenance software, simulation software and mission data preparation software. This program is on schedule and has met all scheduled milestones.

Advanced Technology Cruise Missile Study (ATCM) (Contract No. N00019-78-C-0195)

The objective of the ATCM study was to take postulated threat and concept data and, utilizing forecasted technology, devise cruise missile concepts. These concepts were tested in single engagement analysis and those that performed well were evaluated in a total force effectiveness analysis. Technologies which contributed to the effectiveness of each of the designs were then prioritized and a technology plan was developed for each.

Air-to-Surface Technology and Integration Study (ATS) (Contract No. F33657-76-C-3150)

Boeing conducted the ATS study that has the objective of assessing the value of advanced

technologies to mission capabilities and overall system effectiveness of future air-to-surface, all-weather, manned strike fighters. The study included the integrated system components of weapons, avionics, airframe, and their interactions.

RESUME

EARL T. STARTZMAN

Senior Software Engineer

EDUCATION

BS, Mathematics and Physics Friends University, 1968

EXPERIENCE

Since 1981, Mr. Startzman has been the Ada focal point for Boeing Military Airplane Company, Wichita, Kansas. During the past year and a half, he has participated in the formulation of BMAC high order language strategy. He has developed specifications for a J73 compiler code generator targeted for MIL-STD-1750A as part of the B-1B Standards program. He has currently advised other BMAC project organizations in the utilization of Ada for their particular application. Currently he is principle engineer of the Ada Information Management System Contract (No. F33615-83-C-1052) sponsored by AFWAL/AAAF. In this capacity he has supervised the procurement of an Ada compiler targeted for the MIL-STD-1750A processor.

From 1971 to 1981, Mr. Startzman was employed by NCR Corporation in Wichita where he has involved in the following types of activities:

- Design and implementation of operating systems for mini- and micro-based commercial data processing equipment.
- Design and implementation of a command language for a microprocessor-based operating system.
- Design and implementation of device drivers for synchronous and asynchronous devices.
- Devised architecture for dynamic resource allocation in connection with a distributed processing environment.
- Chairman of Corporate Committee for introduction of extensions to the BASIC language.
- Development of corporate-wide standards for programming languages and definition of a common logical interface for all random-access media used within NCR systems.

Tse-Wayne Mah
Software Development Engineer

Education:

BS, Computer Science, Wichita State University, 1981

Experience:

Mr. Mah has three years experience at BMAC where he has designed and implemented software for the Center of Gravity/Fuel Level Advisory System for the B-52. He has worked on firmware monitors for MIL-STD-1750A and Zilog Z8000 processors and has developed software to operate a Bus Interface Unit as a terminal/controller on a MIL-STD-1553A/B data bus. Worked on implementing the Single Processor Synchronous Executive on a MIL-STD-1750A processor system. He has supported development of an interface for an emulation of an AP101C on a Nanodata QM-1 with a MIL-STD-1553A/B data bus. He has completed the Language Control Facility's JOVIAL (J73) programming language course and produced code in JOVIAL (J73) targeted for the MIL-STD-1750A processor for the B-1B benchmarking effort.

Reviewed the design of the Ada compiler targeted to the MIL-STD-1750A being produced for the AIMS project. Coordinated and attended the first Ada language course at Boeing. Worked on implementing an Ada code generator for the VHSIC 1750A. Currently working on using Ada in a lab model of an avionics system.

Prior to his employment at Boeing, Mr. Mah held various software development positions with NCR Corporation, Sedgwick County Department of Public Works, and United Computing Systems while he was obtaining his computer science degree from Wichita State University.

Projects he worked on prior to graduation include:

- Maintenance of an operating system for a small business computer.
- Design, implementation, and maintenance of a variety of software on a Wang 2200 series minicomputer. Also, day-to-day supervision of use and operation of the system.
- Assisted system analysts in maintenance and fixing problems in an operating system running on an IBM 360/365.

Honor:

Wallace Scholar, Wichita State University, College of Engineering, 1977

Third place in Wichita State University, College of Engineering, Scholastic Tournament and Design Competition in the area of Computer Science, 1977.

Member first place team in ACM programming contest for high school students, 1977.

Other:

Programming languages used extensively include Ada, various assembly languages, JOVIAL, FORTRAN, BASIC, and PL/1.

Member SAE AE-9, Committee on Aerospace, Avionics Equipment & Integration.

Donald W. Higgins
Senior Software Engineer

Education:

BS, Computer Science, Troy State University , 1977

Experience:

Mr. Higgins joined the Boeing Military Airplane Company in May 1982 and has been involved in the design and implementation of digital autopilot systems and avionics ground support systems.

Mr. Higgins has been following the evolution of Ada for the last two years and was a member ofthe Ada Information Management System (AIMS, contract sponsored by AFWAL-AAAF, No. F33615-83-C-1052) research team which investigated the feasibility of Ada for the implementation of real-time, embedded systems.

Mr. Higgins has completed the classroom requirements for a masters degree in Computer Science at Wichita State University and is currently working on his master's practicum. His practicum work is in the area of the automated retargeting of compilers.

Mr. Higgins is currently a Senior Specialist Engineer in the Computational Systems and Software group of the Avionics Technical Staff.

From July 1977 through April 1982, Mr. Higgins was a systems programmer for NCR Corporation in Wichita, Kansas. While at NCR, he was involved in the design and implementation of several significant enhancements to an NCR interactive, multi-programmable operating system. Mr. Higgins' area of emphasis was file management with particular application to techniques for maintainig data integrity and recovery of data files from a hardware or software failure.

From 1972 through 1977, Mr. Higgins was in the United States Air Force working as a computer programmer.

Thomas C. Leavitt
Senior Software Engineer

Education:

BS, Engineering, University of California, Los Angeles, 1970

Experience:

Mr. Leavitt joined the Boeing Military Airplane Company in December 1981, and has been involved in vendor selection and review for a digital autopilot system, a source selection effort for a JOVIAL (J73)-to-1750A compiler for the B-1B benchmarking effort, and the development of an Ada performance evaluation test suite which evaluates Ada compilers in the area of efficiency of generated code for various language features and combinations of language features.

Mr. Leavitt has worked on the Ada Information Management System (AIMS) project and was involved in the specification of a VAX-hosted, MIL-STD-1750A targeted Ada compiler for a BMAC Request For Proposal. Subsequently, he participated in the technical evaluation of the resultant proposals and the management of the contract that was awarded.

Mr. Leavitt is currently a Specialist Engineer in the Computational Systems & Software group of the Avionics Technical Staff.

From 1969 to 1973, Mr. Leavitt worked for the UCLA Engineering Department where he designed and coded a portion of the META-7 translator writing system used to generate compilers and translators for various target systems. Additionally, at UCLA, he provided design support for the run-time support package used on a digital logic simulation system. From 1973 to 1974, he designed and implemented a compiler for a Xerox-peculiar high-level language (CTL) for process control of a copier/duplicator system at Xerox in El Segundo, California. In 1975, he started work with NCR in Wichita, Kansas, where he served in the language products group supporting COBOL and BASIC interpreter design and implementation. He also proposed, designed, and implemented a performance monitoring package which provided execution time and frequency histograms for COBOL applications. He also worked in NCR's Corporate Data Base Management committee.

Mr. is a member of Tau Beta Pi and the Association for Computation Machinery.

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS		WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

Ada SOFTWARE REUSE ISSUES

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	REUSABILITY	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

- REUSABILITY - THE USE OF SOFTWARE ON DIFFERENT PROJECTS.
- PORTABILITY - THE USE OF SOFTWARE ON DIFFERENT TARGET MACHINES (SUBSET OF REUSABILITY).

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	PROBLEM	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

COMPLEXITY OF Ada DATA STRUCTURE

USE OF COMPLEX Ada DATA STRUCTURES LIMITS REUSABILITY.

- PRIMITIVE DATA TYPES SIMPLIFY THE PROBLEM.
- THE SIMPLER THE INTERFACE THE LIKELIER THE SUCCESS.

CONCLUSION

- THE BEST CHANCE FOR PRODUCING REUSABLE CODE IS IN THOSE APPLICATIONS THAT HAVE WELL-DEFINED DATA STRUCTURES

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	**PROBLEM**	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

EXAMPLES OF PROGRESSIVELY MORE DIFFICULT PROBLEMS

EASY - A MATH ROUTINE WHICH REQUIRES *FLOAT* IMPUTS AND OUTPUTS *FLOAT*.

MORE COMPLEX - GRAPHICS PACKAGE WITH WELL-DEFINED DATA STRUCTURES BUT WHICH MUST MANIPULATE THE DATA INTERNALLY.

DIFFICULT - A FULL NAVIGATION SYSTEM. DIFFERING TYPES OF DATA ARE REQUIRED WHICH ARE INPUT AT DIFFERENT RATES.

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	PROBLEM	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

Ada STRONG TYPING

Ada STRONG TYPING DEMANDS SAME NAMES FOR ANY AND ALL (REUSABLE) PACKAGES USED.

- TYPE NAMES FOR ALL PACKAGES AUSED MUST AGREE OR STEPS (POSSIBLY EXTENSIVE) MUST BE TAKEN TO RESOLVE THE INCOMPATIBLE TYPES.

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	REUSABILITY ANALYSIS	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

GIVEN - A SUITABLE CANDIDATE APPLICATION FOR A REUSABLE PACKAGE.

QUESTION - DOES THE APPLICATION ALSO HAVE TO BE PORTABLE ?

IF NOT, THE PROBLEM IS SIMPLIFIED.

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	REUSABILITY CONSIDERATIONS	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

- IF ONLY ONE PACKAGE IS BEING REUSED BY THE APPLICATION, TYPE COMPATIBILITY RESOLUTION IS STRAIGHT FORWARD (THERE WILL BE NO CONFLICT).

- IF MORE THAN ONE PACKAGE WILL BE REUSED BY THE PACKAGE, THE MULTIPLE PACKAGES MAY HAVE TYPE INCOMPATIBILITIES.

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	**APPROACHES TO INSURE COMPATIBILITY**	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

- DEFINE A PACKAGE CONTAINING ALL TYPE DEFINITIONS.

  LIMITATIONS

  - ASSUMES THE AUTHOR OF THE TYPE DEF PACKAGE KNEW ABOUT ALL OTHER PACKAGES TO BE REUSED BY THE APPLICATION.
  - LEAVES THE PROBLEM OF INTEGRATING A PACKAGE THAT WAS NOT ACCOMODATED BY THE GLOBAL TYPE DEF PACKAGE.

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	**APPROACHES TO INSURE COMPATIBILITY (CONT'D)**	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

- MAKE ALL TYPES OPERATED ON GENERIC.

  LIMITATIONS

  - MAKES DEVELOPMENT OF THE CODE MORE DIFFICULT.
    - NON-PRIMITIVE DATA IS DIFFICULT
    - PERFORMANCE CAN SUFFER SIGNIFICANTLY
  - PERFORMANCE OF THE CODE IS OBSCURED.
    - THE PROGRAMMER WILL NOT BE AWARE OF THE PERFORMANCE COST OF HIS APPROACH.

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	**PORTABILITY CONSIDERATIONS**	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

- ACCURACY OF MATHEMATICS
- COMPUTATIONAL CONSISTENCY

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	ACHIEVING DESIRED ACCURACY	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

- Ada MODEL NUMBER APPROACH PRODUCES MACHINE INDEPENDENT ERROR BOUNDS.

- MORE ACCURATE RESULTS THAN THE MODEL IMPLIES CAN BE ACHIEVED.
  - GIVEN KNOWLEDGE OF THE HARDWARE, TRICKS CAN PRODUCE THE DESIRED ACCURACY.
  - THE "TRICKY" CODE NOT LIKELY TO BE PORTABLE.

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	COMPUTATIONAL CONSISTENCY	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

- FROM MACHINE TO MACHINE, OVERFLOW, UNDERFLOW, GUARD DIGITS, ETC. ARE TREATED DIFFERENTLY.
- RESULTS OF COMPUTATIONS WILL VARY ACROSS MACHINES.

  EXAMPLE - TO PRESERVE MAXIMUM PRECISION, ON SOME MACHINES, IT IS BETTER TO COMPUTE

  1.0 - X as (0.5 - X) + 0.5.

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	REUSABILITY QUOTIENT	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

- TOTAL REUSABILITY MAY NOT BE ATTAINABLE.
- IF ONLY THE DESIGN IS REUSED, CONSIDERABLE BENEFIT IS ATTAINED.

  EXAMPLE:

  A SECOND ORDER FILTER IS NEEDED BUT THE CURRENT FILTER IS ONLY FIRST ORDER. THE EXISTING FILTER CAN SERVE AS A VALUABLE MODEL IMPLEMENTING THE SECOND ORDER FILTER.

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	REUSABILITY LIMITS	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

- TOTAL REUSABILITY SHOULD BE CONSIDERED.
- A DEGREE OF REUSABILITY MAY BE THE PRACTICAL APPROACH.
- TOTAL REUSABILITY MAY BE PROHIBITIVELY COSTLY.
- LIMITING THE EXTENT OF REUSE CAN MAKE IT FEASIBLE TO IMPLEMENT THE COMPONENT.

MARCH 1985

BOEING MILITARY AIRPLANE COMPANY WICHITA, KS	TESTING REUSABLE SOFTWARE	WORKSHOP ON REUSABLE APPLICATION SOFTWARE COMPONENTS

- "WHEN IS TESTING DONE?" TAKES ON ADDED MEANING.
- PORTABLE CODE CAN NOT BE DEMONSTRATED UNTIL TESTED ON ALL TARGETS.
- INCREASED EMPHASIS WILL BE PUT ON "PROOF OF CORRECTNESS" TESTING METHODS.

MARCH 1985

A DISCUSSION OF PROTOTYPING IN THE SOFTWARE DEVELOPMENT CYCLE

M. K. Thomson
General Dynamics

February 1, 1985

Rapid Prototyping

This paper addresses factors which lend to the success of rapid prototyping. Several case histories will be presented which support the importance of these factors and demonstrate the potential benefits.

There are three major factors involved in a rapid prototyping: problem definition, prototype "craftsman" uses to create the model.

Problem definition is prerequisite to prototyping. The completness of the definition may have some bearing on the nature of the prototype. A complete definition indicates a prototype whose primary purpose is to demonstrate a well thought out design. A sketchy defintion suggests a protype whose initial purpose is to refine and develop the problem. In the latter case, prototyping is particularly important.

In rapid prototyping, it is the experience, resourcefulness, and creativity of the programmer which is going to have the most affect on the outcome. Assign an indifferent programmer to a prototyping effort and give that programmer the sophisticated programming tools and you will get, at best, an adequate and indifferent prototype. This, of course, disturbs managers who are often forced to plan only for the mediocre in human resources. It is this reliance on less than average programmers which tends to prevent rapid prototyping from becoming a recognized discipline.

The proceding is not meant to suggest that prototyping tools are not important. Given the right tools, a programmer can work miracles in rapid prototyping. We will discuss the two broad categories of hardware and software tools separately.

In the last decade, several trends in computer hardware have emerged which will have a profound impact on rapid prototyping:

- The availability of single user workstations.
- The availability of economical graphics hardware.
- Availability of new data entry devices, such as graphics frame-grabbers, digitizers, electrical mouses, lightpens, and so forth.
- Hardware assistance in high level language support. In particular, the availability of dedicated LISP machines make it feasible to use this CPU intensive interactive language in prototyping.

While not critical to the prototyping effort, these hardware advances have helped to not only make the effort easier, but also to deliver a professional and realistic prototype. In paraticular, it has been our experience that the sophistication of the prototype output has a great bearing on the success of the prototype.

A prototyping effort is often an intensive one. The availability of dedicated computer resources can be a major factor in preventing a bottle neck in the development process. Also, it is more effective to bring the prototype to the customer rather than vice versa. These two factors make the availability of portable workstations attractive in rapid prototyping.

Software tools provide the rapid prototyping leverage to the programmer. Their impact is difficult to measure since their effectiveness is closely tied to the quality of the programmer using them. Software tools might be categorized as follows:

- Programming languages
- Language preprocessors
- Program libraries
- Text or syntax-directed editors
- The operating system environment
- Debugging aids

Few question the advantages of high-level languages over assembly languages. It would be difficult to determine which high-level languages are "best" for rapid prototyping. The following are what we believe to be important attributes of successful rapid prototyping languages:

- Interactiveness. The edit/compile/link/run cycle of noninteractive languages discourages experimentation and testing because of the time it takes to make changes. Interactive languages allow changes to be made on-the-fly and generally have superior debugging features. Examples of interactive languages include LISP, AXE*, FORTH, APL, and SNOBOL.

- Powerful I/O and graphics support. As mentioned earlier, the visual aspects of a rapid prototype tend to be important: it is coding these which typically dominates prototyping.

- Symbolic processing. This is an ambiguous term which primarily refers to the ability to handle amorphous data dynamically. List processing languages, such as LISP are particularly strong in this area.

- Extensibility. Language extensibility makes it possible to build new primatives resulting in an even higher-level language. It can be argued that effective macro processors and library managers also provide language extensibility.

A powerful macro processor can transform a weak language into a powerful language. Given a large macro library, many assembly languages could become candidates for rapid prototyping. Some languages, such as LISP and C, have built-in macro processors; the AXE language has not only built-in macros, but also knowledge base driven symbol substitution and code expansion.

Program libraries provide the facilities to save and reuse generic code. This can be particularly useful in developing prototyping tools for "cut-and-paste" prototypes. Libraries are the most common form of language extension for compiled languages, such as Pascal and Ada.

Typically, programmers spend a majority of their time editing. This fact underlines the importance of the editor in the prototype effort. A powerful editor can significantly speed up the development process. In some cases, editors have features which make up for the lack of a macro processor.

Syntax-directed editors are language specific. They catch syntax errors at edit time and typically have special features which reduce the number of keystrokes needed for code entry. It follows that a syntax-directed editor can assist in getting a prototype done quickly.

An operating system which is unmanageable, difficult to get around to, and unfriendly is an impediment to software development. Operating systems, such as UNIX, were designed with the programmer in mind and provide additional tools to aid in the development cycle. Recently, integrated multiwindow environments, such as Flavors and Smalltalk, have made it possible to perform multiple tasks concurrently. This provides a short-cut to program development.

Software prototypes are prone to error (if not more so) than more "standard" software efforts. Since rapid prototypes are often developed in days or weeks, as opposed to months or years for "standard" software efforts, the sophistication of the debugging aids is important. As mentioned earlier, interpretive languages are strong in this area.

LISP is noted for its ability to trace, stop midexecution and modify, and continue execution.

A prototype may be undertaken for a number of reasons, which we will classify into two groups: evaluation and marketing. For example, a prototupe might be developed as a means of evaluating an already existing design, to find flaws, awkward interaction, or to choose one of several possible designs. Prototypes can be used in the design process as a means of trying possibilities and identifying additional requirements.

The term "marketing",as mentioned above, is being used metaphorically. Sometimes a concept or a design has to be sold to one's peers or management before it is sold to an outside customer. Software designs presented on paper or through viewgraphs can be difficult to conceptualize. A working model, even if it is incomplete and slow, can do a great deal in communicating the intent of the design. There is another subtle advantage to prototype demonstrations. They show progress before the work has even begun and can establish credibility with management or customer.

General Dynamics has developed rapid prototypes for a number of diverse problems. We will briefly discuss several of these which we feel represent our general experience.

General Dynamics Fort Worth Division was awarded an Army contract, to develop a message communications system in Ada. When the contract was awarded, the requirements were vague. Furthermore, some requirements were based on the antiquated approach in current use. General Dynamics wished to test alternate approaches and to present the customer with a working model at the project kick- off meeting.

A multi-terminal prototype was developed in one week at San Diego using the AXE language, a proprietary symboilic processing language. This prototype was brought to Fort Worth and a number of enhancements were made in the day preceding the kick-off meeting.

The prototype was then demonstrated to the customer. Some of the proposed design enhancements were rejected, several generated a great deal of interest and discussion, amd several design misunderstandings were cleard up. The customer was pleased with the prototype and requested that it be demonstrated to potential users for their evaluation. Thus, the prototype was useful not only in evaluating the design, but also in improving communication with the customer and in generating a certain confidence in our abilities to handle the contact.

Aspects of the AXE language which lent to the success of the prototype were:

- Support of a multi-terminal/multi-task environment.
- Being an interpretive and extensible language, which made it eary to get a kernal up and running quickly and then to add refinements.
- Being a list processing language with an integrated knowledge base making it easy to simulate a message network.

General Dynamics Electronics Division was in design review for an important new product when it was discovered that the customer was unable to comprehend the proposed operator interface. This caused a crisis when the customer recommended creating a working group to define a new approach. General Dynamics asked if it could quickly create a model of the proposed operator interface.

Based on a sketchy description of the problem, a first prototype was created in a few days by two programmers using AXE and shown to the customer. The prototype revealed that still more information was required from the customer.

A better description was provided by the customer, and in two days, a second prototype was developed from scratch. This was demonstrated and received enthusiastically by the Electonics Division management, and then demonstrated to the customer, with dramatic results. They were able to understand the

proposed operator interface, and it was approved.

This prototype later went through several refinements, and was used to evaluate some of the details. Consequently, the prototype was successful both as an evaluation and as a marketing prototype.

The features of AXE which contributed to the success of this prototype include:

- o A powerful CRT programming language extension used to create a large number of interactive menus.
- o The interactiveness of the AXE language, which made it easier to test and integrate a large number of modules.

As a result of the Strategic Defense Initiative, General Dynamics Convair Division has had to develop and evaluate space battle management strategies. A strategy was proposed which combined artifical intelligence and distributed processing. A prototype of a simulation model was developed on a Symbolics 3670 computer for a ten minute video presentation. The prototype makes extensive use of ZetaLisp features running on the Flavors environment to create a real-time color graphics simulation.

The prototype was developed on ten hours by a Symbolics expert. It is a goodexample of a "marketing" prototype. Its purpose is to sell the viewer on the viability of the proposed approach.

Factors which contributed to the success of the prototype are:

- o The powerful programming environment offered on the Symbolics LISP computer.
- o The ease of creating multiple tasks in ZetaLisp in the Flavors environment and for them to interact.
- o The powerful color graphics features incorporated into the system.
- o The programmer was an expert in ZetaLisp and the Flavors object-oriented environment.

Since prototyping has been so successfully used within General Dynamics, we are taking the following steps to increase our capabilities:

- o A complete evaluation of available environments, including languages suitable to prototyping and the computers to support them.
- o The acquistion of workstations particularly suited to rapid prototyping. Accompanying this must be the thorough training of programmers to effectively use the workstations.
- o Some effort to develop a rapid prototyping methodology. This may include a history of what approaches succeeded or failed, what problem areas are good or bad candidates for prototyping, and perhaps the creation of prototyping libraries to assist in future efforts.

Rapid prototyping offers tangible benefits, both in the design and marketing of software. At present, rapid prototyping is an art dependent on the programmer and tools available. Although this is not likely to change significantly, increased awareness and coordination could improve the changes for future efforts.

Ada is registered trademark of the U.S. Government, Ada Joint Program Office.

The following are trademarks of General Dynamics Corporation AXE, BOLT, BOLT JR, DARTS Technology.

RESUME

M.K. Thomson
-Senior Software Engineer, DSD

EDUCATION

B.S. Mathematics Cal Poly San Luis Obispo, 1977

EXPERIENCE

Mr Thomson is currently the project technical leader for DARTS, a sophisticated artificial intelligence environment and methodology developed at General Dynamics. He has done extensive work in the development of expert systems, translators, and rapid prototypes.

In 1980 Mr. Thomson became a founding member for the darts project. In this capacity he helped design and implement major sections of the AXE language compiler and run-time environment. AXE is a state-of-the-art AI language incorporating features of LISP, SNOBOL, FORTH, and APL.

Mr. Thomson has also supported a real-time missile guidance system and was a member of the launch team at Vandenberg AFB. At that time he designed and implemented a stand-alone computer/radar interface diagnostic package used to determine pre-launch readiness.

PROTOTYPING

in the software development cycle

762

M. KEMER THOMSON

General Dynamics, Data Systems Division

FACTORS IN PROTOTYPING

* Problem Definition
  - Concept Demonstration
  - Concept Refinement

* Prototype Craftsman

* Prototype Tools
  - Hardware
  - Software

PROTOTYPE HARDWARE

* Single User Workstations
* Economical Graphics Hardware
* Data Entry Devices
* Hardware-Assisted Language

PROTOTYPE SOFTWARE

* Programming Languages
* Language Preprocessors
* Program Libraries
* Editors
* Environment
* Debugging Aids

PROTOTYPING LANGUAGES

* Interactiveness

* Powerful I/O and Graphics

* Symbolic Processing

* Extensibility

FACTORS IN PROTOTYPING

* Problem Definition
  - Concept Demonstration
  - Concept Refinement

* Prototype Craftsman

* Prototype Tools
  - Hardware
  - Software

FUNCTIONS OF PROTOTYPES

* Product Evaluation

* Product Marketing

CASE #1
MESSAGE SWITCHING SYSTEM PROTOTYPE

* Developed Using Axe in One Man-Week

* Language Features Contributing to Prototype Success

  - Multi-terminal/multi-task environment
  - Language interactiveness
  - List processing
  - Integrated knowledge base

* Prototype Benefits

  - Helped establish project credibility
  - Helped test design alternatives

CASE #2
OPERATOR INTERFACE PROTOTYPE

* Developed Using Axe in Two Man-Weeks

* Language Features Contributing to Prototype Success

  - Powerful CRT programming language
  - Language interactiveness

* Prototype Benefits

  - Cleared customer confusion
  - Helped refine design details

CASE #3
BATTLE MANAGEMENT PROTOTYPE

* Developed by General Dynamics and Symbolics using Zetalisp in two days

* Features Contributing to Prototype Success

  - Powerful, integrated tool set
  - Multi-tasking environment
  - Sophisticated color graphics
  - Programmer expertise in the environment

* Prototype Benefits

  - Demonstration of concept
  - Attractive packaging results in greater audience interest

CONCLUSIONS

* Prototyping has offered tangible benefits at General Dynamics

* Currently a combination of powerful tools and better-than-average programmers offer the greatest chance for prototype success

* Advances in reusable software techniques will directly benefit prototyping efforts

A DISCUSSION OF METHODOLOGIES FOR THE DEVELOPMENT OF REUSABLE SOFTWARE

M. K. Thomson

General Dynamics

1. Introduction

In 1984, an IRAD study was started to produce a reconfigurable ATLAS compiler written in Ada. This compiler must be reconfigurable in three areas: syntax, test station operating systems, and form of intermediate language and test equipment. In addition, the compiler must run as fast as possible on 64K logical address machines. Presently, the design phase is complete and software implementation has begun. The software consists of ADA code and tools which will reconfigure the compiler. The software which reconfigures the compiler was developed using a system called DARTS TM, which was developed internally by General Dynamics and is expressly designed to develop reusable software.

This paper shares the experiences obtained from creating software, which from its inception, is designed to generate reusable software. The scope of this paper consists of three parts: the design methodology to create reusable software; to relate the methodologies to our experience in developing the retargetable ATLAS compiler; to give an overview of our future methodologies and the areas in where they will be applied.

2. Design Methodologies

The methodologies presented are automated techniques where software is generated from high level design concepts as opposed to the conventional library approach which concentrates on reusing source code. The automated techniques strive to reuse software designs rather than to reuse actual software. The automated approaches offer the greatest impact for the reuse of software. Presently, there are two major automated methodologies for generating reusable software. One embeds higher order structures in a programming language. This methodology is a surface model because it has little knowledge of the program structure. This model has existed in the past in many forms. The first use was probably macro expansion in assembly language. The C language which features macro expansion (1) and the generic feature of ADA (2), is an example. Surface models are attractive in that they can use the information in the software in which they are embedded. The model only deals with the changes that augment the original software.

One problem with the surface model is that it is useful only over a narrow domain range, because it has little knowledge of the program structure and no knowledge of the design concepts. As a result, its range of usefulness is limited to programs which share the same physical program structure. However, a changing design requirement can totally change the physical layout of the program. This is true for systems requiring high speed execution which characterize embedded systems and one of the major requirements of this project.

A second methodology creates and manipulates symbolic models of the software to obtain reusable software. This methodology is a deep model because it has, to some degree, knowledge of the software design as well as some of the software requirements. Deep models will usually have software structures similar to those found in AI programs. An example of a deep model is given by Neighbors (3). As will be pointed out later in this paper, deep models are more powerful than surface models. However, for the software reuse process to be cost effective, deep models sometimes must be integrated with surface models. In the survey article by Horowitz, Kemper and Narasimhan (4), the trend in Application Generations is to embed the Application Generator into a conventional high level language. An Application Generator would be considered as a deep model which generates data base programs. However, in order to put in general features to support more flexibility, the designers were faced with either turning the Application Generator into a programming language or embedding the Application

Ada 'RO Is a Registered trademark of the U.S. Government, Ada Joint Program Office The following are trademarks of General Dynamics Corporation: AXE, BOLT, BOLT JR, DARTS Technology

Generator into a conventional language. The embedding choice was simpler and offered the greatest flexibility.

In using these methodologies to design software, some basic problems are encountered. The first problem, pointed out by Boehm (5), is that reusing software is a tricky undertaking. Projection models sometimes predict that the integration cost of reused software can exceed the cost of developing totally new software. In an article by Keringham (6), the C modules which were reused the most were the ones which were initially designed for reuse. Past experience indicates that for reusable software to be effective, the software must be initially designed with the reuse as a design criteria.

However, designing reusable software contradicts normal software design practices in that, with a conventional top down design practice, the requirements should be known before the software is designed. As a result, there needs to be a new type of methodology for developing reusable software. The methodology, which has proven successful, is the logical models which represent the behavior of the software. The logical models were then simplified and modularized as much as possible. The success of the models depends directly upon how much the problem domain can be modularized. Then, tools were developed to transform the logical models into physical code. Surface models consist of conventional software embedded with special information and are an effective means of doing the transformation.

This type of design criteria has theoretical appeal, but the criteria must also solve practical problems. A question exists as to whether it is better to create special tools to generate reusable software or to simply recreate the software manually. Using the special tools extracts resources from the project, so there must be a simple rationale for their use.

There are two scenarios for using automated tools to generate reusable software. In the first scenario, software is used in a single application. The tools generate reusable software effectively if the following inequality holds:

$$Y + Y \qquad Z$$

where:

X = lines of software needed with the automated approach

Y = other information needed by the automated approach

Z = lines of software needed by conventional approach

The inequality states that the "equivalent" amount of lines of code for the automated process must be much less than the conventional approach. The term "equivalent" amount of lines of code is used because data or information are often needed by the software models and in essence count as lines of code.

In the second scenario, software is used for more than one application. Then the following inequality must hold if the automated analysis is to be effective:

$$X1 + Y1 \qquad Z1$$

Where the "1" terms represent additional effort to transform a solution from one application to another.

In developing the logical models for the reconfigurable ATLAS compiler, these inequalities are kept in mind. The inequalities in essence determine areas where automated help is most needed and areas where the tool may not be effective. Also, leverage can be obtained in documentation as well as in lines of code. Experience indicates that simple models give 7 to 1 leverage in lines of code and usually another 7 to 1 in the documentation over manual means. It is not unusual to see models coupled together and leverages multiply, much in the same manner as the mechanical efficiencies of pulleys multiply when coupled together in a block and tackle. Designs of automated systems which have extremely large leverage (greater than 50) usually consist of many models with much smaller leverage factors. The problem of generating models with high leverages is similar to designing mechanical tools with high mechanical advantages.

3. Practical Experience

This section presents the practical experiences obtained from developing a reconfigurable ATLAS compiler. Addressed are our experiences with the tools used to develop reusable software, and the experience gained in developing logical models.

The tool used for building the reconfigurable ATLAS compiler is DARTS, which was developed internally by General

Dynamics, and was designed to be used as a software manufacturing tool. The tool is general purpose and has been used to build software translators and expert systems. DARTS was used on this project because of its effectiveness in producing reusable software. DARTS has list processing which is needed for symbolic processing. LISP and many other AI languages also have these features; however, one of the main features needed in reusable software development is string manipulation. Included in DARTS is a pattern matcher which operates like the pattern matcher in the SNOWBAL language. The pattern matcher and string manipulation are critical in automatically generating software since there is extensive text manipulation. Most LISP based languages do not have a flexible string representation to be competitive with DARTS. DARTS also has a data base which allows one to easily create a frame type AI system, a feature which was used extensively in this project. The system which generates a reconfigurable ATLAS compiler is shown in a top level view in Figure 1. The system is layered as shown. The top level is a configuration management system which manages the configurations which are generated by the rest of the system. Either the next layer or the inner layers but not next inner layers are the front ends which allow users to enter data into the system. The front ends vary from ones that allow detailed information to be entered by design specialists, to front ends that assist unskilled users in building the desired tailored compilers. The inner most level is the system which builds a compiler to desired specifications. This is the main part of the system around which the rest of the system evolves.

Figure 2 shows a block diagram of the inner most system which builds the ATLAS compiler to the requested design specifications. The figure shows that the system which builds the ATLAS compiler consists of many specific models. There are models for processing syntax features, generating ATLAS intermediate code and generating the data structures and the routines to manipulate the data structures needed by the compiler. There are two classes of modules. The first class builds the components of the ATLAS compiler. The second class links the components together to form a working ATLAS compiler. The modules which link programs together play an important role in extending the range of the products build by this system. Also, the linking modules have the responsibility of constructing the compiler for optimal speed execution. The target architecture for the ATLAS compiler is a 64K logical address space and speed performance will be largely determined by the number of disk accesses needed. The disk accesses will be minimized, and as shown in Figure 3, by being pipelined with other processing to further speed up processing. Performance data will be placed in the linking modules so they can arrange the software as to maximize speed.

In developing reusable ATLAS software, we found that our tools are compatible with ADA methodology and, in fact, we believe that the tools enhance the ADA methodology. In particular, the package concept was useful in the design approach. The results of this project show that the package concept in ADA could be extended through surface models by embedding them in the package itself and that packages could be generated through the use of logical models working with surface models.

4. Directions for Future Research

Currently, General Dynamics is investigating the use of DARTS in the area of embedded systems, particularly the distributive processor environment. Our direction is to use data flow techniques integrated with our automated software development to support multiprocessor software development.

The data flow techniques which have been developed are similar to the techniques outlined by BABB (7). There is believed to be considerable potential in these techniques when they are combined with automated software techniques. Currently, a prototype system is being developed which will take a program generated for a single CPU system and distribute the software in a multiple CPU system. The system will have the ability to analyze performance and spot potential bottle necks in real time systems.

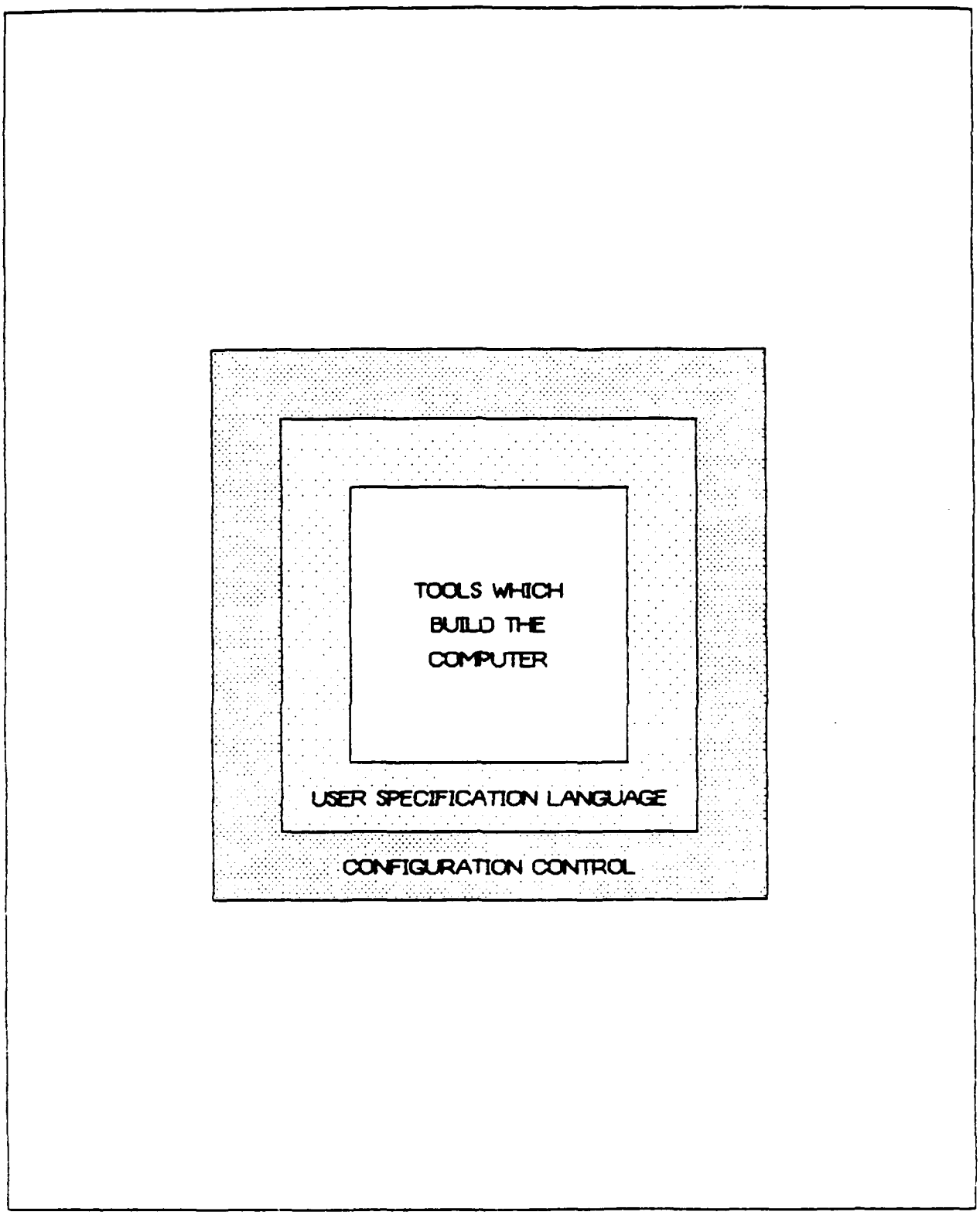


FIGURE 1. AUTOMATED ENVIRONMENT

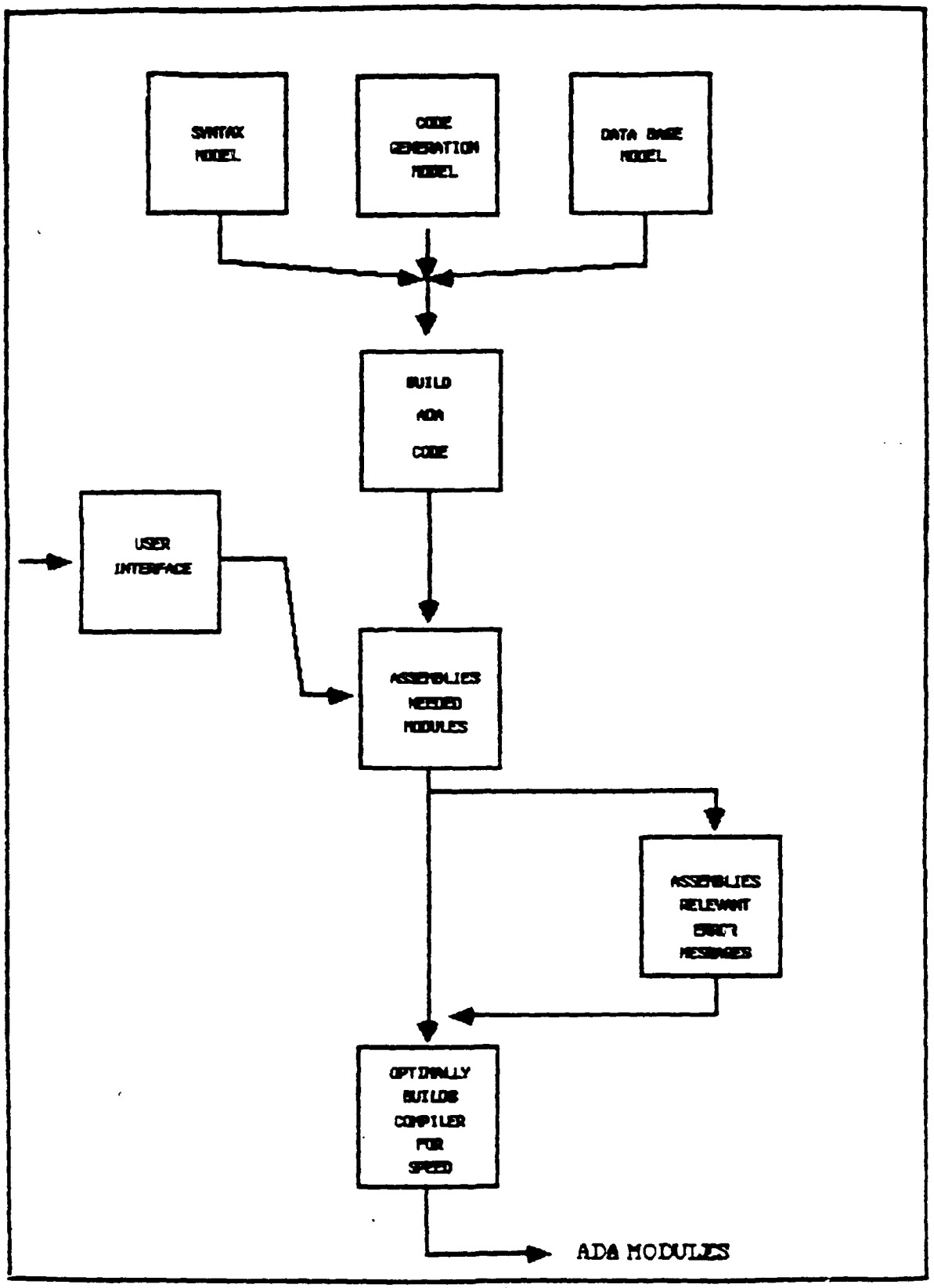


FIGURE #2 TOP LEVEL DIAGRAM

REFERENCES

(1) Kernigan and Ritchie, the C Programming Language, Prentice-Hall, 1978

(2) G. Booch, ADA Book, "Software Engineering with Ada, Benjamin Cummings, 1983

(3) J.M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components", IEEE Transaction on Software Engineering, Vol. SE-10, Number 5, pp. 564-573

(4) Horowitz, Kemper and Narasimhan, "A Survey of Application Generators", IEEE Software Magazine, January 1985, pp. 40-54

(5) Boehm, Software Engineering Economics, Prentice-Hall, 1981

(6) B.W. Kernighan, "The UNIX System and Software Reusability", IEEE Transaction on Software Engineering Vol. SE-10, Number 5, pp. 513-518

(7) Babb, "Parallel Processing with Large Grain Data Flow Techniques", IEEE Computer, pp. 55-61

RESUME

M.K. THOMSON

Senior Software Engineer, DSD

b. "EDUCATION:"

B.S. Mathematics, Cal Poly San Luis Obispo, 1977

b. "EXPERIENCE:"

Mr. Thomson is currently the project technical leader for DARTS, a sophisticated Artificial Intelligence environment and methodology developed at General Dynamics. He has done extensive work in the development of expert systems, translators, and rapid prototypes.

In 1980, Mr. Thomson became a founding member for the DARTS project. In this capacity he helped design and implement major sections of the AXE language compiler and run-time environment. AXE is a state-of-the-art AI language incorporating features of LISP, SNOBOL, FORTH, and APL.

Mr. Thomson has also supported a real-time missile guidance system and was a member of the launch team at Vandenberg AFB. At that time he designed and implemented a stand-alone computer/radar interface diagnostic package used to determine pre-launch readiness.

RESUME

C.A. HANSEN

Software Chief (Acting)
Advanced Technology

EDUCATION

B.S. Mathematics, Rutgers University, 1969
M.S. Computer Science, Fairleigh Dickinson University, 1971

EXPERIENCE

Mr. Hansen has over 15 years experience in the area of software design for real time systems. He has been a manager in software support organizations for 7 years.

Mr. Hansen is currently responsible for the management of the Advanced Technology department within DSD, Western Center. His department is responsible for investigating new technologies in software development, artificial intelligence, and expert systems.

RESUME

G. EDGAR

Software Design Specialist, DSD

EDUCATION

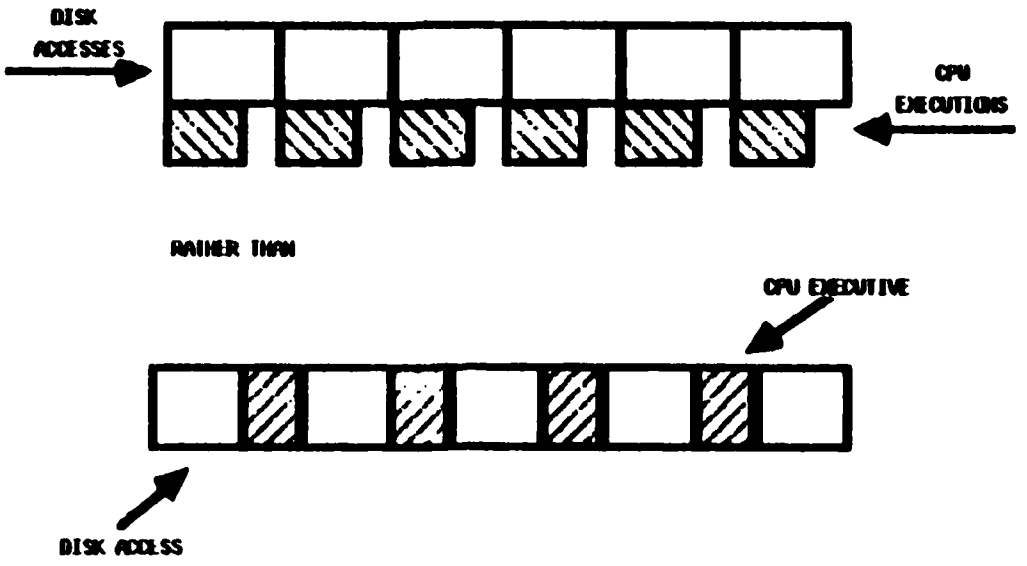
B.S. Engineering Physics, Ohio State University, 1975

b. "EXPERIENCE"

Mr. Edgar has over 9 years experience in software design, particularly using expert system techniques. He is currently developing a fast, reconfigurable ATLAS compiler, written in Ada, using DSD's DARTS technology. This project blends Artificial Intelligence, automated software techniques, and compiler technology.

Mr. Edgar has also functioned as the chief architect for a group specializing in microprocessor based test equipment. In this role, he directed the development of an expert system for small, portable test sets. This system interfaced to a larger expert system which automated test software as well as maintaining the software during test validations.

* THE STRATEGY OF OVERLAY PROCESS IS TO MINIMIZE THE DISK ACCESSES AND TO PIPELINE THE DISK ACCESSING PROCESS AS SHOWN BELOW.



* THE LINKING MODULE WILL HAVE THIS PERFORMANCE DATA SUCH THAT THIS TYPE OF ANALYSIS CAN BE MADE.

FIGURE 3. AUTOMATED TOOLS TO IMPROVE PERFORMANCE

METHODOLOGIES FOR THE DEVELOPMENT OF REUSABLE SOFTWARE

M. KEMER THOMSON

INTRODUCTION

- GENERAL DYNAMICS IS DEVELOPING A RECONFIGURABLE ATLAS COMPILER
- SOFTWARE REUSE IS ACHIEVED THROUGH THE DEVELOPMENT OF A COMPILER "DEEP MODEL"
- DARTS™ TOOLS ARE BEING USED TO IMPLEMENT THE MODEL

METHODOLOGIES FOR THE DEVELOPMENT OF REUSABLE SOFTWARE

- INTRODUCTION
- DESIGN METHODOLOGIES
- RECONFIGURABLE ATLAS COMPILER EXPERIENCE
- FUTURE DIRECTIONS

TWO APPROACHES TO REUSABLE SOFTWARE

- **LIBRARIES**
  - **REUSE ACTUAL SOFTWARE**
- **AUTOMATED REUSE**
  - **REUSE SOFTWARE *DESIGN***

AUTOMATED REUSE MODELS

- **SURFACE MODEL**
  - — EMBED HIGH-ORDER STRUCTURES IN A PROGRAMMING LANGUAGE
- **DEEP MODEL**
  - — CREATE AND MANIPULATE SYMBOLIC MODELS OF THE SOFTWARE

SURFACE MODEL

- HAS LITTLE KNOWLEDGE OF THE PROGRAM STRUCTURE
- IS USEFUL OVER A VERY NARROW DOMAIN
- EXEMPLIFIED BY MACROPROCESSORS AND ADA GENERICS

DEEP MODEL

- HAS KNOWLEDGE OF THE PROGRAM STRUCTURE
- CREATES AND ORDERS CODE
- MAY COMBINE SURFACE MODELS
- NECESSARY FOR TIME-CRITICAL AND/OR MEMORY-BOUND APPLICATIONS

A METHODOLOGY FOR REUSABLE SOFTWARE

- DEVELOP LOGICAL MODELS OF SOFTWARE BEHAVIOR
- SIMPLIFY AND MODULARIZE LOGICAL MODELS
- DEVELOP TOOLS TO TRANSFORM LOGICAL MODELS INTO PHYSICAL CODE

FOR THIS METHODOLOGY TO BE VIABLE

THE AMOUNT OF SOFTWARE NEEDED TO IMPLEMENT THE MODELS

+

OTHER INFORMATION NEEDED TO TRANSFORM THE MODELS INTO PHYSICAL CODE

$\ll$

AMOUNT OF SOFTWARE NEEDED BY CONVENTIONAL APPROACH

COMPLETE MODELS ARE COMPLEX

- MODULAR DESIGN IS ESSENTIAL
- A DEEP MODEL WILL COMBINE MULTIPLE MODELS ENHANCING THEIR LEVERAGE
- MODELS AND THEIR TOOLS SHOULD BE DESIGNED LIKE PARTS FOR AN ENGINE

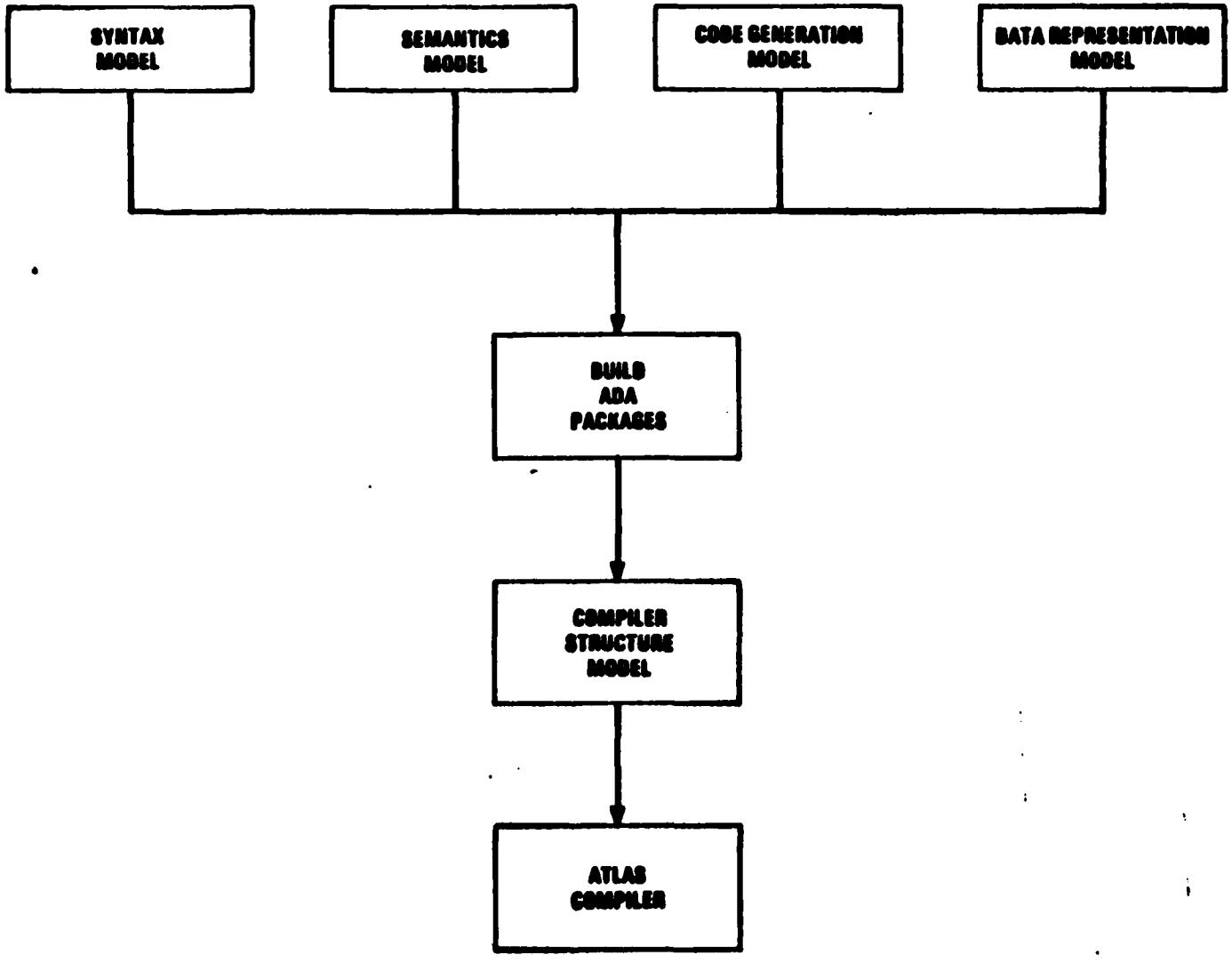
RECONFIGURABLE ATLAS COMPILER

- RECONFIGURATION BASED ON
  - SYNTAX
  - TEST STATION OPERATING SYSTEM
  - INTERMEDIATE LANGUAGE
  - TEST EQUIPMENT

ATLAS COMPILER MODELS

- SYNTAX ANALYSIS
- SEMANTIC ANALYSIS
- SYNTAX DIRECTED CODE GENERATION
- DATA REPRESENTATION
- COMPILER STRUCTURE

RECONFIGURABLE ATLAS COMPILER



ADA EXPERIENCE

- PACKAGES PROVIDE POWERFUL MEANS TO GENERATE AND ASSEMBLE CODE AUTOMATICALLY
- ADA EXCEPTION HANDLING IS A KEY COMPONENT TO OUR CONTROL STRATEGY
- ADA PROVIDES GREATER PORTABILITY AND THUS GREATER REUSABILITY

FUTURE DIRECTIONS

- CUSTOMIZED COMPILERS AND RUNTIME SYSTEMS
- RECONFIGURABLE AVIONICS SYSTEMS
- LARGE-GRAIN DATA FLOW MODELS

INFOMATION PACKAGE FOR WORKSHOP ON REUSABLE COMPONENTS OF APPLICATIONS SOFTWARE

Dr. Gregg Van Volkenburgh
ALLIED CANADA, INC.
201 City Centre Drive
Mississauga, Ontario
L5B 3A3
(416) 276-1044

and

Mr. Paul Bassett
NETRON, INC.
99 St. Regis Crescent North
Downsview, Ontario
M3J 1Y9
(416) 636-8333

JANUARY 31, 1985

Section 1: Introduction

Section 2: The Reusable Code Problem (Topic 2)

2.1 External Subroutines (Topic 2)

2.2 Code Generators (Topic 2)

2.3 The Frame Methodology (Topic 2)

2.4 Software as a Manufacturing Enterprise (Topic 5)

Section 3: Software Tools that Exploit Reusability (CAP*) (Topic 5)

3.1 The Common Structure of CAP* Tools (Topic 5)

3.1.1 The Optional Rapid Prototyping Phase (Topic 5)

3.1.2 The Fine-Tuning Phase (Topic 5)

3.1.3 The Program Production Phase (Topic 5)

3.2 CAP*'s Impact on Maintenance (Topic 5)

3.3 CAP*'s Impact on Quality of Code (Topic 5)

3.4 CAP*'s Impact on Documentation (Topic 5)

3.5 CAP*'s Impact on the Software Development Environment (Topic 5)

3.6 CAP*'s Support for Portability, Retro-fits and Upgrading non-CAP* Code (Topic 5)

3.7 CAP*'s Support for Security (Topic 2)

3.8 User Experience with CAP* (Topic 1)

3.9 Our CAP/Ada* Reusability Experience (Topic 8)

---

* CAP, CAPenvironment, CAPinput, CAPscreen, CAPreport are trademarks of NETRO
Ada is a trademark of the U.S. Department of Defense.

Section 1: Introduction

This submission, made by Allied/Netron, discusses several of the topics raised in the Special Notice regarding "Workshop on Reusable Components of Application Software" in the December 21, 1984 issue of Commerce Business Daily. Using its own funds Netron has already developed and implemented, and is selling software products which enable developers to increase productivity by a factor of 10 (at least) while simultaneously employing a methodology which:

(1) enables rapid prototyping to be carried out,

(2) guarantees reusability of code by defining code components,

(3) automatically processes reusable code libraries,

(4) promotes program designs which make extensive use of reusable libraries.

The success of this product (CAP*) for COBOL has prompted Allied to underwrite the development of an expanded version for Ada*. This new product is named CAP/Ada* and became functional as of December 15, 1984. It is available for demonstration at the present time. As a self-contained system, CAP/Ada* operates on any DEC VAX machines supporting the VMS operating system. Detailed information regarding use is available from Allied, at the address given on the cover page.

Since Allied/Netron are already applying the design, production, use and maintenance of Ada* code via CAP/Ada*, we discuss several of the workshop topics from a product viewpoint, rather than from a problem analysis viewpoint. Our submission most closely falls under topic 5 (automated part composition), although in discussing the capabilities and benefits of CAP/Ada*, topics 1 (specification and design), 2 (reusable definition), and 8 (Ada* experience) addressed. To aspects of our submission relate to the most appropriate topic, we have cross indexed the topics in our headings.

Section 2: The Reusable Code Problem (Topic 2)

In the software industry's current cottage industry style, it is common practice to build new programs by "cutting and splicing" pieces of old programs together. This approach demonstrates that

(1) there is a great deal of potentially reusable code available, and

(2) it is worth the effort to adapt it rather than starting from scratch.

Unfortunately,

(1) the programmer does not have any systematic way of isolating just what portions of programs are relevant;

(2) the customization process is time-consuming, tedious, and prone to error;

(3) once the process is finished, both old and new programs must be maintained as if each is completely unique, despite the considerable common functionality. Maintenance effort should be proportional to the novelty in the system, not the number of source statements.

The central thesis of this submission is that a good solution to the reusable code problem turns out to provide a solid technical basis from which to understand and deal with the production, quality, and maintenance issues currently besieging the software industry.

2.1 External Subroutines (Topic 2)

It is still widely believed that external subroutines form a satisfactory repository of reusable code. Separately compiled and linked subroutines are obviously useful, but they are limited because there is no graceful or systematic means of effecting:

(a) local customization of an external subroutine to fit each calling program's particular context of use, and

(b) global evolution of a subroutine when it must change to benefit all future callers of that subroutine without victimizing current callers.

The fundamental problem is that a subroutine is a representation for a single function which is not adaptable at source-program (function) construction time. It may

have considerable run-time flexibility, but at the time of actually molding the subroutine into the program that must use it, an external subroutine by its very nature has no flexibility at all.

2.2 Code Generators (Topic 2)

Code generators have been around for years (e.g. RPG) and although they are usually very succinct and expressive, they have never enjoyed widespread use. The simplest kind of code generators are those that generate "raw" source code. The problem with those generators is that they are basically "one-shot" tools. Because each generator is an expert at only a part of the overall problem, programmers must supplement and modify the generated source code to suit their own needs. Having adapted the code, they have no means of reusing the generator without destroying all of their manual modifications. To be more useful, a code generator must allow some follow-on mechanism which can adapt the generated source code automatically, thus allowing reuse of the generator without the loss of the customizations.

More sophisticated code generators typically supply "user exits" for handling this problem. These provide linkage to separately compiled, external subroutines which can usually be written in a variety of general purpose languages. The trouble is that:

(a) this is always an additive technique; there is no way to change or remove generated functionality;

(b) predefined interfaces often omit information that is essential in the customization (the "black box" effect);

(c) all non-procedural parts of the generated code, such as data declarations, are simply unavailable for customization.

A proper solution requires generators to provide for automatic customization of generated code (not just run-time communication with generated modules).

2.3 The Frame Methodology (Topic 2)

We have developed a frame methodology (CAP*) to address the reusable code problem from the perspectives of both programmers and code generators. A frame is a machine-processable representation of an abstract data type, with "abstract" meaning functional. Because the data operators are functionals, not functions, frames can accommodate both local customization into an individual program, and global evolution to benefit all future programs. Frames are implemented xtu(e.g. Ada) and (prord frtext.

There are just four macro commands whose essential role is to automate the "cutting and splicing" of programs:

> COPY-INSERT allows a frame hierarchy to be copied into a program (by naming the frame at the root of the hierarchy), and causes customizing frame text to be INSERTed anywhere into that hierarchy.
>
> BREAK-DEFAULT defines a named "breakpoint". Breakpoints mark arbitrary places in a frame where custom frame text can be INSERTed to supplement and/or replace DEFAULT frame functionality.
>
> REPLACE systematically substitutes a specific code string for a generic one (throughout a frame hierarchy). For example, field names, picture clause elements, etc. are generic if they tend to vary from program to program.
>
> SELECT incorporates into a program one frame text module from a set of modules in the frame. SELECTs are like CASE statements (with arbitrary nesting) which operate at text construction time. An important use of SELECT is to automate version control (global evolution).

Frames are written by both analysts and tools. Having code generators produce frames solves the problem of destroying subsequent modifications by automating the "cutting and splicing" of the customizing frame text into the generated frame text.

All customizing frame text for one program is localized into a SPECIFICATION or SPC frame. An SPC governs the entire process of building the compilable source program from its frame components. As will be seen, a methodology incorporating frames at its heart offers a potential for:

(a) fill-in-the-blanks program specifications (rapid prototyping),

(b) automation of the process of reusing previously built, high quality software (both human and machine written),

(c) automatic customization in context,

(d) maintenance of only what is unique in a program,

(e) evolution without obsolescence (elimination of unnecessary retrofits),

(f) painless enforcement of good programming technique (standards).

2.4 Software as a Manufacturing Enterprise (Topic 5)

In the next section a frame based design for software manufacturing tools is presented, in which standard frames are the standard sub-assemblies, various code generation steps are the processing operations on basic components (raw materials) to produce fabricated parts, and the frame processor operating on the SPC frame is the process of final assembly with any custom options.

Section 3: Software Tools That Exploit Reusability (CAP*) (Topic 5)

The discussion and analysis of Section 2 implies that the software engineering discipline of automatic adaptability can be realized as a coherent yet open ended set of software manufacturing and maintenance tools. Netron affiliate has developed and successfully marketed such tools for the past two years under the name "Computer Automated Programming" and "CAP*". CAP*'s initial focus was in the COBOL software market. More recently, Allied has sponsored the extension of the CAP* methodology by Netron to automate Ada* programming, in anticipation of the needs of the D.O.D. and others.

While D.O.D.'s primary interest is in embedded systems applications, it must be stressed that it is in the nature of reusable software techniques to be based on highly generic underlying formalisms, independent of particular languages and environments.

3.1 The Common Structure of CAP Tools (Topic 5)

At the common core of all CAP* tools is the library of model reusable frames (see Figure 1). These are assumed to be available prior to the building of the programs that reuse them. The library is open-ended in two senses: new frames can be added at any time; and old ones can be upgraded at any time without forcing retrofits. The frames are designed by a combination of systems analysis and usage experience in the application domains to which they are applied. The frame library in turn supports a three phase manufacturing/maintenance process.

3.1.1 The (Optional) Rapid Protyping Phase (Topic 5)

At this phase end-users and analysts alike can define a compilable program in minutes. The key here is that a heuristic designer module extracts a few "broad-brush" requirements from the user, then automatically designs a complete set of specifications for the desired program.

Details of this process are as follows. First, a special purpose editor engages in a dialogue of questions and answers, and actively prescribes the major choices to be made, depending on answers to previous questions. The code generator then uses heuristic logic (i) to specify all the minor choices, creating one or more declarative specifications, and (ii) to create specific frame specifications (in an SPC frame) containing any code that may be necessary to properly combine the functions being expressed in (i).

Clearly, using a rapid prototyping facility by itself automates 100% of the programming. However, for this approach to be viable, the result should be a good first approximation. The prototyping designer is optional because the complete specification of a program can always be done from scratch in phase 2. If the heuristics are inappropriate to the needs of the current program then simply bypass them.

3.1.2 The Fine-Tuning Phase (Topic 5)

At this phase any number of definitional components may be specified which collectively define what and how the executable module (program) is to be written. The one essential component is the Specification frame (or SPC; cf Section 2.3) which, if phase 1 of the tool was used, will already have been prototyped for the application analyst/programmer. Otherwise, he starts by making a copy of a prototype SPC, called a Template. A Template simply contains, in one linear list, all of the Break-Defaults, all of the Replace symbols and all of the Select options that are available throughout the entire frame hierarchy. The Template also contains the version code which insulate the new program from all further

CAP[tm] TOOL STRUCTURE

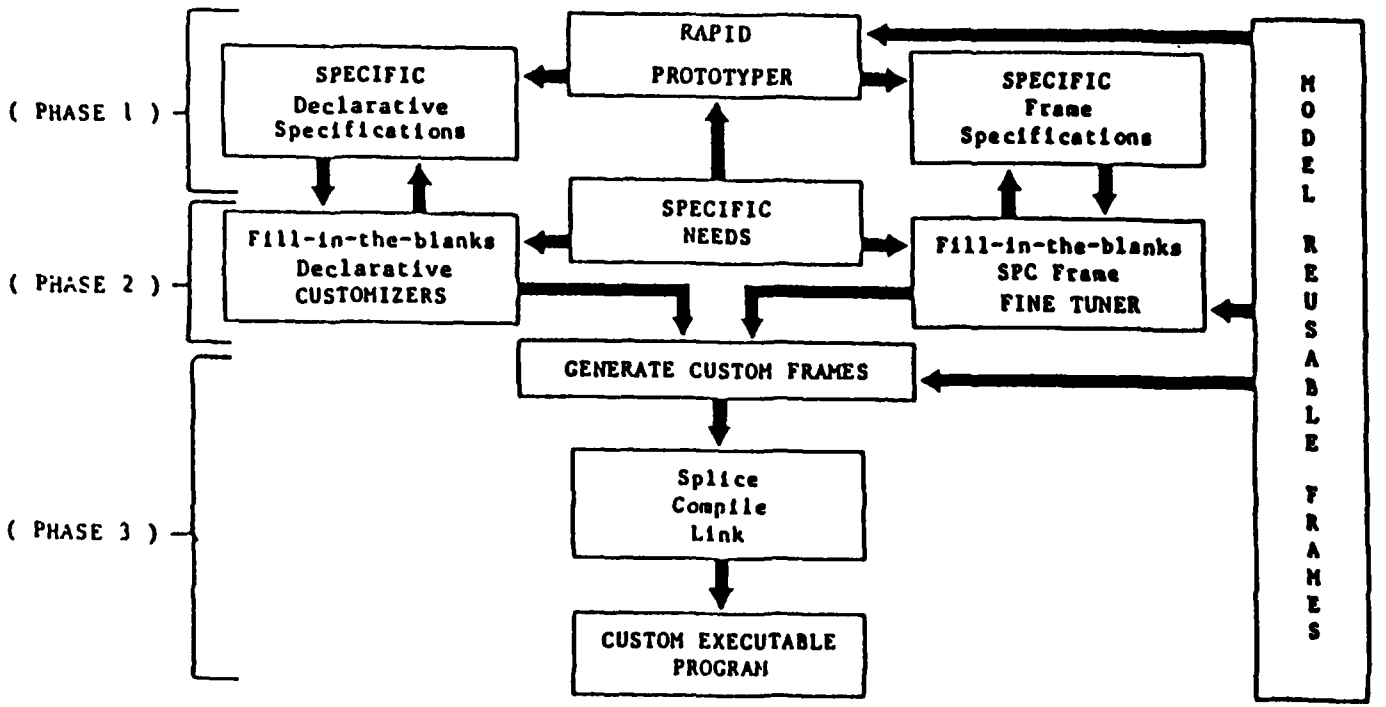


FIGURE 1

changes to the underlying frames.

The analyst really doesn't know and doesn't care where in the hierarchy these particular customization options take place. For he has a fill-in-the-blanks list that he walks through and he is systematically reminded of all the places the frames allow him to make customizations. (The word "allow" is used deliberately, because a well structured, software engineering discipline is being enforced by the CAP* system here.)

The process of filling in the Template with customizing details creates the architectural blueprint for the entire program. The analyst thus transforms the Template into an SPC, which forms the definition of that program. It collects in one spot, for easy maintenance access, just the options which are important. In other words, the SPC is what makes this program distinct from all other programs in the system.

In addition to predefined templates, the library of frames is accessible through an on-line frame taxonomy. The developer uses it by matching his informal problem description to the most general problem categories at the top of the taxonomy. At each lower level the user refines his choices until a specific frame is selected. At this point, a template is created automatically from the chosen frame and merged with the SPC (if any) for further customization.

In addition to the SPC, zero or more declarative definitions of specialized program components are created/refined in phase 2. As with the SPC special purpose editors are used to create and customize these definitions. But at no time must a user treat the resulting code as a "black box". The editors permit him to look at specific details of the source code whenever this is needed for fine-tuning purposes. As for the fine-tuning itself, arbitrary amounts of custom code can be written to supplement and/or replace modules of the pre-written and generated frames. (In our experience with COBOL applications, less than 10% of the final code is custom. cf Section 3.8)

3.1.3 The Program Production Phase (Topic 5)

CAP*'s use of the SPC to operate on frames is fully automatic. It generates frames that stem from the declarative definitional components; it pulls pre-written frames from the library; it cuts and splices them all with any customizing code (whenever it belongs throughout the source); it compiles and links the resulting (Ada*) source.

The result is an executable program. CAP* treats the source as a transient file that is actually deleted after the compiler has processed it. There are two reasons for this. First, it is completely redundant, capable of being reconstructed at a moment's notice. Second, because it is well structured and easy to read, it poses too great a temptation to be modified. Of course, tinkering at the source level also destroys one's ability to maintain that program from the much higher specification level. The listing is always available so that one can understand what's really going on at run-time for further fine-tuning purposes.

3.2 CAP*'s Impact on Maintenance (Topic 5)

Typically a two or three page SPC spans a 30 or 40 page source program. And that's all that needs to be maintained throughout the life cycle of the program. This represents an order of magnitude reduction in being proportional to the numf l* pmeans to factor systems into unique pieces, each independently.

Frames as previously stated have an evolvability property: they improve with age. And improvements can be easily made without forcing retrofits. It is the Select mechanism which enables a frame to incorporate arbitrary amounts of change while retaining all of the previous versions in a manner transparent to all programs.

3.3 CAP*'s Impact on Quality of Code (Topic 5)

In a hand coded programming environment, a typical programmer works very hard to implement the specifications within his deadlines. He often lacks the time to "go the extra mile" to ensure the full efficiency and robustness of his code. Furthermore, he's not particularly inclined to be that "fussy", because he doesn't get a sufficient return on his mental investment. The next time he writes a similar program, he has to reinvent those fussy details all over again.

On the other hand with frames, a software engineer is naturally encouraged to be fussy because he invests that skill and effort once and only once. From then on that quality is available "for free" every time that frame is

reused. It also provides a means of reducing the disruption that occurs when skilled analysts change jobs. Their frames form a reusable legacy for all those who remain.

Frames constitute a means to formalize and enforce standards with a degree of rigor and precision exceeding the usual pseudo-English standards manual approach, which depends on the highly questionable understanding, concurrence, and self-discipline of the programmers. Consistent external interfaces across applications become simStructured programming style standards also find their greatest scope when used to code standard frames, because the break-points tend to attach themselves to the structured units of code. While the SPC frames may suffer from non-standard codelets, these are a small fraction of the total program, and the standard frames tend to teach-by-example how to code using the standards.

Frames exhibit much higher mean-time-to-failure rates than ordinary code. The reason is that standard frames and frame generators are highly seasoned components in the course of whose evolution many improvements and optimizations have been made. Frames also tend to be "extrenmum tested" very early in their lives. Whereas ordinary code is tested only in the context of its current use, frames are tested as well for their reusability, in quite dissimilar environments.

Programs handwritten from scratch have no chance to acquire the quality and thoroughness that is the hallmark of a good frame. This is why one thinks of frames as capital assets. Just as with a conventional manufacturer, the capital assets lie precisely in the standard sub-assemblies which undserpin all the different flavours of the products he produces, and in the tools for assembling and maintaining them.

Frames play exactly the same role as standard sub-assemblies. Notice that generated frames avoid the problems with traditional code generators by simply arranging that they emit frames rather than raw source code. CAP* can keep the customization process entirely factored from generated code and automates the customizing as a final assembly step.

3.4 CAP*'s Impact on Documentation (Topic 5)

CAP* benefits both system and user documentation at several levels:

- o There is less of it since the reusable frames need to be documented once, not each time a version of the code appears in a program.
- o The meta-information about the purpose and scope of the code is formalized to a much greater degree than with ordinary code, and the specifics are separated from the generic so that it is plain to see how each use differs from the others.
- o Before building any code or at prototyping time, all the declarative specifications of various system and program components can be printed to automatically provide the bulk of a formal specification of the system, in a "what you see is what you get" form. This significantly collapses the time to produce formal specifications. Of course, this same mechanism is available at any time a hard copy definition of the functioning system is required. But with the specifications already available on-line, the need for hard copy is reduced.
- o The CAP* approach facilitates and supports interactive, "help key" documentation, about the tools, the frames, and the software produced with them.
- o Frame techniques permit "documentation frames" to be reused and customized in just the same automated fashion as software frames, with all the attendant benefits.
- o CAP* provides utilities that permit tutorial user documentation to be created as a by-product of running the actual system. On-line documentation produced in this fashion is called a CAP* "Movie". Much tedious manual writing is replaced by a "hands-on" simulation that can be edited with additional commentary. Further, as the application software evolves, the movie system provides a relatively painless way to quarantee that the documentation continues to match the software.

3.5 CAP's Impact on the Software Development Environment (Topic 5)

Currently CAP* uses conventional alphanumeric terminals to support a development environment schema (called CAP* environment) designed especially for software engineers. Each engineer can create and evolve a network of his own customized development environments. Each such environment can store up to 45 names of various files, programs under development, test utilities, and so on. By placing a cursor under any name and pressing a function key, he can cause any of the entire arsenal of CAP* tools, standard system functions, and his own special tools to operate on the selected file/module.

Typical operators activated by single keystrokes will convert specifications to executable form, edit and list files, run programs, search libraries using pattern masks, access documentation, copy, display, or scratch files, preserve changes to the environment itself, and link to other environments. By saving environments which contain related files, procedures, and utilities, along with appropriate comments, work is automatically focussed onto a single screen. As a work organizer and practical means to manage project development, CAP environment* endows CAP* with its vital ease-of-use properties.

It is envisioned that a bit-mapped graphics development terminal will replace the current alphanumeric ones, with an embedded, powerful, micro-based CPU in which to package CAP*. Rehosting CAP* can then be replaced by the simpler problem of re-interfacing a standard CAP* development terminal.

3.6 CAP's Support for Portability, Retro-fits and Upgrading non-CAP Code (Topic 5)"

Portability of code is another objective of CAP*. This is a natural consequence of the reusability properties of frames. Thus, it is possible to maintain one canonical definition of an application system together with variations, then produce executable versions of the application for a wide variety of deployment environments (machines, languages and operating systems).

CAP* now supports portability across six machine-language environments. Frame techniques are also used to assemble custom "ship-sets" for every CAP* customer (a ship-set is the particular combination of compatible components of a product that is consistent with the customer's shipping order).

There are many schools of thought about retro-fits. The use of evolvable frames permits a new degree of freedom in the formation of retro-fit policies. As new or improved contexts of use are discovered for a frame it is possible to change it in arbitrary ways without forcing any existing software to be retro-fit or rebuilt. Of course, it may be imperative for other reasons to do a partial or complete retro-fit. But that is a decision that can now be made using criteria unrelated to the maintainability of the software.

Often in hand coding environments, retrofits are unavoidable even for the simplest of changes. Because of a lack of consistency in the manual reuse of code, each affected program must be treated as a unique case and subjected to exhaustive retesting. In other-words no advantage can be taken of the considerable functional overlap that otherwise used to exist among the programs. Retro-fitting is a painful, unproductive, error-prone task, producing "burn-out" among talented people.

Should retro-fitting be desired in a CAP* environment, frames offer major advantages. First, the custom code is permanently separated from the reused code. This means that the places in the code needing retro-fitting are localized for easy access. Second, the formal parameters that customize a frame guarantee that, once the nature of the retro-fit is worked out, it can be applied with absolute consistency across all the users of that frame. This means that less work is necessary and fewer errors are likely as a result of the retro-fit.

Often the question is asked: "How can the man-centuries of existing software be upgraded to take advantage of frames?" First of all, CAP* happily coexists with all non-CAP* software because the operating system cannot disgthem. Should a rewrite be desired (and often what is needed is a redesign, not a rewrite), several semi-automated utilities can provide assistance.

For example, utilities exist that can observe the external run-time behavior of a program and induce declarative CAP* definitions of those external aspects of the

program. Often this reduces to a small fraction, the work required to duplicate the precise functionality of I/O intensive programs. Another utility can be provided that converts existing source programs into frames as first approximations to a unified definition of parts of related programs. There is still ample work to be done when rewriting to CAP* but much of the error prone tedium can be avoided.

3.7 CAP's Support for Security (Topic 2)

Security control is provided by assigning individual user access rights on a program-by-program basis, or on a user-by-user basis, or both. Automatic monitoring is also provided. A log is kept of all attempts to access programs covered by security. This log can be printed as a report, allowing system usage analysis, and detection of unauthorized access attempts.

The system works with a User List and a Program List and, in effect, builds a matrix containing the access information. Each time a protected program or procedure is run, the Security System automatically checks this matrix, denying or granting access accordingly.

Users are given indirect access to the data files through an inheritance process. Main menus are given Read and / or Write access to the appropriate data files. Programs (including secondary menus) inherit Read and / or Write access to data files from the menus from which they are run.

Incorporating security into a new program or adding a straightforward. Access rights are set up from either User Mode or Program Mode, by user or by program. Specifying access information for a new user or program is facilitated by the "Copy Access Information" feature, which allows one to copy the access rights for an existing user to a new user, or to copy the access rights for an existing program to a new program. Once copied, it is a simple matter to tailor the access information for the particular user or program.

3.8 User Experience with CAP* (Topic 1)

CAP* has been in continuous use in COBOL, Wang VS environments for over four years. It has been a marketed product since April 1982 and is in use at over 50 sites in the U.S. and around the world. There are numerous industry and academic experts with independant knowledge of CAP* (for example Dr. Cordel Greene, Director of the Kestrel Institute and consultant to the STARS project). Several refereed papers have been published. Names and phone numbers of expert reviewers and customers can be supplied upon request.

The following is a summary from a detailed case study which analyzes the actual usage of CAP*.

CANADIAN OUTDOOR PRODUCTS INC. is a subsidiary of NOMA INDUSTRIES LTD. In March 1983, Canadian decided to create a computerized Requisition system to replace their manual Requisition system.

The system was created using CAP* and is run on a WANG VS computer using interactive terminals. The system allows requisitions to be created, maintained, displayed, searched, authorized, ordered, recorded and reported upon.

The Requisition system was built by a student analyst during his first work term leave from the University of Waterloo. After the first week, enough of the system had been prototyped that Canadian users recognized serious design problems. The system was redesigned and put into production by the end of the third week.

Sixteen programs were created using CAP* tools, to create and control the interaction of the 22 screens and 3 reports though which the Requisition system is operated. CAP* tools enabled the author to create the Requisition system by writing less than 10% of the total COBOL lines needed.

One method of judging COBOL program production with an without CAP* tools is to compare the total number of lines of submitted source code in the entire Requisition system with the number of hand-written lines. Purely comment lines were discarded.

The results show more than a 10:1 productivity gain by this measure. There were 34,000 lines of submitted code contained in the 16 programs of the installed Requisition system. This represents a net productivity of over 2,000 lines of fine-tuned COBOL per man-day. Only 3,000 lines were written by hand, but even by this measure there is a ten-fold gain over normal code production rates.

The following table shows, for each of the 16 programs forming the Requisition system, the number of lines (i) hand written in the SPC frame, (ii) in the generated frames, (iii) in standard frames, and (iv) in the total submitted to the COBOL compiler.

Program SPC Name	Main CAP Generated Tool Frame	Total Source Frames	Standard Frames		
PREQ1	CAPinput	2979	56	1731	1192
PREQ2	"	2130	71	1264	795
PREQ3	"	2318	78	1013	1227
PREQ4	"	1721	62	869	790
PREQ5	"	3440	421	1904	1115
PREQ6	"	2776	157	1766	853
PREQ7	"	1510	40	673	797
PREQ8	"	3018	206	1806	1006
PREQ9	"	3238	281	1910	1047
PREQA	"	3659	436	2223	1000
PREQI	"	3399	436	1916	1047
PREQF	Frame Lib.	274	187	0	87
PREQG	"	223	136	0	87
PREQR	CAPreport	954	140	198	616
PREQS	"	1086	226	216	644
PREQT	"	1152	179	290	683
TOTALS		33,877	3,112	17,779	12,986

Figure 2
Number of Code Lines

3.9 Our CAP/Ada* Reusability Experience (Topic 8)

Recently CAP* technology has been re-engineered to produce structured Ada* programs. In the process of building and testing the CAP/Ada* reusability tools, a number of properties of the Ada* language have been evaluated regarding Ada*'s support for reusability. These are:

(1) Packages

(2) Operator overloading

(3) Generic subprograms

(4) The ability, in a subprogram, to specify default values for parameters whose values are not supplied by the caller

(5) The ability of a calling program to refer to parameters symbolically as well as positionally

(6) Predefined attributes of variables and types

Reviewing these briefly, a package is a collection of subprograms and associated programming resources whose external interface is formally defined; operator overloading means that Ada* can distinguish between identically-named subprograms on the basis of differences in the number and/or type of their parameters. Generic subprograms are subprograms which accept one or more type-names (or subprogram names) as arguments, so that a single sour versioof a subprogram can serve as a template for multiple instaniations, within narrow limits. Predefined attributes are such things as lengths of strings, array dimensions, machine addresses, offsets of elements within agther f-explanThe general effect of these features is to improve the suitability of subprograms as repositories of reusable code. Some of them (3,6) allow greater generality in subprograms than most programming languages do. Other features (4,5) help address the problem of retrofitting changed or extended subprograms to existing programs. Operator overloading helps in the management of subprogram variations, while the package concept is meant to facilitate the implementation of abstract data types, which are attractive as basic units of reusable code. Nevertheless, in comparison with a reusable-code methodology which operates at source-program construction time, even the Ada* approach suffers from severe limitations.

First, the generic subprogram facility is extremely limited and does not begin to approach the power of a true meta-language. It is not possible, for example, to select among code paths according to argument type: the only variability possible is by direct substitution of type-name or subprogram-name parameters in contexts where they are syntactically permitted. Nor is it possible to perform arbitrary text substitution, nor to alter, override, or insert program code to meet specific requirements.

These are not shortcomings of Ada* implementation in particular, but are inherent in the very concept of the subprogram as discussed in Section 2. To allow a programmer such degrees of freedom at run-time would be extremely dangerous from the point of view of program reliability. The difference between run-time and program construction time in this respect is that the consequences of errors at program construction time are explicit in the constructed program text, whereas run-time errors must be inferred from program behavior.

Second, the power of generalization available to the designer of a reusable code module is necessarily much less at run-time than at program construction time, despite the availability of certain attributes of variables and so on, because whatever run-time facilities are available must be less than those provided by a compiler, unless the run-time program incorporates the full power of a compiler. (If we are dealing with an interpreted language, of course, the distinction between program-construction time and run-time is blurred; nevertheless, even interpreters do not, for reasons at least of efficiency, make their full power of generalization available to the programmer.) In designing reusable code modules at source-construction time, advantage can be taken of the typing, symbolic reference, diagnostic, expression-evaluation, and other features provided by a compiler, and thus the designer's ability to generalize is much greater than anything that could practicably be provided at run-time.

Third, in attempting to achieve generality at run-time rather than at program construction time, issues of run-time efficiency and program size are raised which are by no means secondary in the context of a language whose primary area of application is seen to be in the development of mission-critical embedded systems. The usual answer to efficiency concerns, that they can be addressed by "adding more horsepower", is inadequate when by "adding horsepower" we are adding to the number of parts that can fail, not to mention to manufacturing costs.

When CAP* technology is coupled to Ada*, the result is synergistic. Not only do frames supply an escape from the above limitations, but the programmer is also freed from the tedious and error-phone coding redundancies implied by Ada*'s strong typing and structuring requirements. CAP* also allows the programmer to group functionally related program and data structures which Ada* forces to be scattered across a package. Finally, CAP* permits the expressive power of Ada* to be amplified by embedded special purpose (declarative) notations.

Allied/Netron have implemented CAP/Ada* on the DEC VAX family of computers under the VMS operating system. System dependencies are relatively minor and well-isolated, so that porting the implementation to other computers and/or operating systems is not a major problem. The implementation is designed to work with any full Ada* compiler, and can be fitted to most existing partial compilers. Space and capacity requirements are not onerous, and the system can certainly operate without causing response degradation or excessive disk-space usage on any machine capable of running an Ada* compiler. As an illustration, portions of the system were developed on an IBM PC/XT computer using a very restricted subset compiler, without encountering capacity or performance problems.

The CAP/Ada* package consists of the following components:

CAP frame processor, frame library, screen editor/generator, report
generator, programmer's workbench and file-maintenance program generator

User Manual

Training

System component documentation (proprietary)

Use of the package requires a DEC VAX or compatible computer, the VMS operating system, and any reasonably complete Ada* compiler.

Further details regarding CAP/Ada and/or a demonstration of its capabilities can be obtained by contacting Allied at the address shown on the cover page.

RESUME

Dr. Van Volkenburgh

Profile

Dr. Van Volkenburgh began his career as a project scientist and project manager in the aerospace industry. He broadened this experience by similar assignments in geophysics/geochemistry, the environmental sciences, and advanced manufacturing. Throughout this period, and earlier while in college and graduate school, he was closely involved in developing a wide variety of software systems, ranging from university administration to large scientific modeling systems, and more recently, real time multipoint remote data acquisition and multi-point alert systems.

For the past five years, Dr. Van Volkenburgh has occupied senior management positions in government, the venture capital community, and high-tech industry. He is now employed by Allied Canada, Inc., in Mississauga, as Manager, External Research in the corporate Research and Development division. Allied Canada, Inc. is a wholly owned subsidiary of Allied Corporation, Morristown, New Jersey. He is currently developing several business areas for Allied in technologies such as advanced materials, biotechnology, manufacturing processes, and artificial intelligence.

A large portion of his time is being spent on assignment to the Allied/Netron joint venture company to coordinate, develop, and exploit business opportunities in the fields of AI, Ada software, and reusable Ada code.

Dr. Van Volkenburgh has authored scholarly papers in a variety of disciplines and is the recipient of several awards for academic excellence and distinguished public service. He is the owner of a small business and serves on the Board of Directors of several companies.

PII Redacted

Education

Bachelor of Arts, Chemistry and Mathematics (cum laude):
University of California
Irvine, California
June, 1969

Master of Science, Space Sciences:
York University
Downsview, Ontario
November, 1970

Doctor of Philosophy, Space Sciences:
York University
Downsview, Ontario
November, 1973

Prior Work Experience

Senior Management and Business Development

1983-84: Idea Corporation, Toronto. Position: Director, Research and Development. Responsible for locating and evaluating commercially promising technology across Ontario Universities, research institutes, and hospitals. Also responsible for formulating and closing commercial investments in these technologies, managing the investments and marketing them when mature.

1981-83: Ontario Ministry of Environment, Toronto. Position: Director, Air Resources Branch. Management of a shop containing 125 professionals from a variety of disciplines and having a budget of $9 million/year. Line responsibility for all research and special projects (e.g. acid rain, destruction of toxic chemicals), for all regulation development connected with pollution control, and for protection of public health during abnormal pollution conditions and/or emergencies. Frequent presentation of position papers and/or expert testimony to the media, public forums, or legislative hearings.

Systems Analysis and Programming

1981-83: Ontario Ministry of Environment, Toronto. Position: Director, Air Resources Branch. Management authority over new system for province-wide air pollution and meteorological real time data collection, and real time analysis and Air Pollution Index calculation and public alert notification.

1978-80: Ontario Ministry of Environment, Toronto. Position: Program Manager, NEMP: New system developed to collect air pollution data using field microprocessors, transmit it upline, and perform analysis and reporting on Data General Eclipse S120.

1970-73: York University, Toronto. Position: Graduate Student. Developed rocket payload data transmission and analysis capability on PDP 11/70.

1968-70: University of California, Irvine. Position: Research Assistant. Numerical reaction kinetics simulations on IBM/360 (20 hour/week position).

1966-68: IBM, Irvine, California. Position: Program Analyst. Programmed and implemented timeshared computer-aided student records/enrollment systems, developed on IBM/360 (20 hour/week position).

Project Staffing and Management

1978-80: Ontario Ministry of Environment, Toronto. Positions: Supervisor, Technology Development and Appraisal Section (1979-80); Program Manager, NEMP (1978-79). The latter position involved managing a 5-party field study of a large new industrial complex near Lake Erie. The former position covered supervision of all special projects carried out for air pollution research and measurement.

1976-78: Barringer Research Limited, Toronto. Position: Head, Projects Administration (1977), and Assistant to the Vice-President of

Research and Commercial Projects (1976). This company has been a world leader for 30 years in developing new geophysical/geochemical instrumentation, remote sensing systems, and analytical capability. The positions involved management of instrument development, manufacturing, and sales, as well as trouble shooting for contract R&D work and coordinating manpower and business proposals across the entire country.

1975-76: National Oceanic and Atmospheric Administration, Boulder, Colorado. Position: Project Engineer, Fluourocarbons Measurements. This U.S. government agency conducts advanced research and monitors the environment. The particular project cited was a joint field investigation with the Space Sciences Centre at York University (Toronto) utilizing high altitude balloons with subsequent trace chemical analysis of payload samples.

1973-75: Intra-Space International, Toronto. Position: Research Director. This company was started as an outgrowth of university aerospace research. The company quickly grew to a staff of 12 people and successfully developed and delivered electro-optical hardware to NASA and international industrial clients.

Career Accomplishments

Idea Corporation (1983-84)

* Identified key structural issues involving technology transfer from R&D institutions
* Formulated several novel programs to promote technology transfer, the development of entepreneurs, and the creation of wealth via business development in Ontario universities
* Located over 170 investment opportunities in these institutions
* Negotiated and managed 6 investments involving $5.1 million covering the licensing of research and the start-up of 4 "high-tech" companies
* Negotiated with several large companies to buy 3 of the above investments. Buy-out of 2 investments has been completed.

Ontario Ministry of Environment (1978-83)

* Protection of public health at the Mississauga chlorine/petrochemical derailment (1979), and also during several air pollution alerts (1981-1983)
* Successful completion of the first comprehensive provincial government/industry environmental impact assessment program (NEMP, 1977-1983)
* Key contributions to the development of Canadian policy regarding the control of acid rain, and to an influencing of U.S. policy in this area
* Negotiation of Ontario/Michigan Pollution Control Agreement and Member, IJC Board

* Successful "cradle to grave" management of leading-edge research projects in air projects in air pollution, including design, installation, and operation of 2 real time air pollution measurement, analysis, and control systems

* Organized and implemented a tripartite joint-venture $4 million acid rain research program with the Federal Republic of Germany and Environment Canada involving advanced numerical modeling of the troposphere on a super-computer

* Procured funds for, and managed design of several new multi-million dollar air pollution initiatives, including 3 van-mounted trace contaminate analysis systems involving GC, GC/MS, and other instruments

* Timely delivery of a complete, world-class air pollution management program in a period of severe constraints.

Barringer Research Limited (1976-78

* Managed the transition of several commercial products in environmental, geophysical and geochemical fields from prototype to production and sales stage

* Brought in over $3 million of commercial products business through direct effort

* Rationalized the operation of a 100 person team of professionals involved in lab services, research and commercial products/services, and significantly boosted company productivity

* Successfully marketed high-technology research services internationally.

National Oceanic & Atmospheric Administration (1975-76)

* Engineered key components of a stratospheric balloon payload, from concept to hardware

* Constructed, tested, and operated associated ground preparations systems

* Participated in laboratory measurements program of recovered payloads.

Intra-Space International (1973-75)

* Turned a university laboratory prototype device into fully qualified NASA hardware, meeting manned space flight standards, through hands-on effort and management of a project team

* Built a high-tech company from the ground up

* Managed all aspects of a small company under the executive direction of the President

* Marketed specialized aerospace hardware in Canada, the U.S., and Europe

* Successfully resolved severe conflicts involving labor and contracts.

Awards and Distinctions

* 1981-1983: Member, International Joint Commission, Michigan/Ontario Air Pollution Board

* 1979: "Order of the Tank" award (Ontario Ministry of Environment) for Distinguished on-site Service during the Mississauga derailment

* 1973-1975: National Research Council (Canada) Industrial Postdoctrol Fellow

* 1970-1973: National Research Council (Canada) Postgraduate Fellow

Publications

(1) Don L. Bunker and G. Van Volkenburgh; "A Trajectory Study of Phosphorus-32 Recoil in Sodium Phosphate", J Phys Chem, 74, 2193(1970)

(2) W. Braun, C. Carlone, T. Carrington, G. Van Volkenburgh and R.A. Young; "Collisional Deactivation of H(2?2OP) Fluourescence", J Chem Phys, 53, 4244(1970)

(3) R.A. Young and G. Van Volkenburgh; "Collisional Deactivation of CO(a?3O) J Chem Phys, 55, 2990(1971)

(4) G. Van Volkenburgh and T. Carrington; "Resonance Fluourescence Transfer and Radiationless Deactivation of Systems of Intermediate Optical Depth", J Quant Spect Rad Trans, 11, 1181(1971)

(5) G. Van Volkenburgh, T. Carrington and R.A. Young, "Electronic Energy Transfer in Isotopic Variants of the H*(2?2OP - 2?2OS) + HO2? System", J Chem Phys, 59, 6035(1973)

(6) A. Schemltekopf, G. Van Volkenburgh, et al; "Stratospheric Balloon Measurements of Chlorofluorocarbons" NOAA Technical Report, 1976

RESUME

Mr. Bassett

Profile

Mr. Bassett is considered to be an expert in both theoretical computer science and applied data processing. He has published several technical articles and papers, taught as an assistant professor of computer science, founded and operated Sigmatics Computer Corporation, and is currently Vice President of Netron Inc.

Education

Honors B.Sc., Mathematics Physics and Chemistry (MPC), 1967
University of Toronto

M.Sc., Computer Science, 1970
University of Toronto and was accepted into the doctrinal program

CPIM, (Certification in Production and Inventory Management), 1982
American Production and Inventory Control Society (APICS)

Certified by the Ontario Mortgage Brokers Association, 1978

Two Society of Actuaries Fellowship examinations

Career Chronology

March 1981 - present Vice President and part owner of Netron Inc., a division of Noma Industries

June 1976 - June 1982 Founder and President, Sigmatics Computer Corporation

July 1973 - June 1980 Assistant Professor of Computer Science, York University - September 1969 - 1977

Systems Consultant: Hospital for Sick Children, Ramtek Corporation Norpak Limited (others on request)

July 1974 - 1980 Partner in Carol Electronics; a firm specializing in marketing computer graphics equipment

Sept 1968 - 1969 Project Leader (Real Time Systems) Hospital for Sick Children

May 1966 - Sept 1968 Systems Engineer/Programmer, IBM

Professional Career

Mr. Bassett is the Vice President of Netron Inc., the software products division of Noma Industries, Ltd. Netron develops and markets Computer Aided Programming (CAP?tmO) products, designs and installs turnkey manufacturing systems (MRP-II), and provides software conversions from various machines to WANG-VS systems.

CAP is based on a proprietary software engineering methodology (Frames) for manufacturing and maintaining custom application software invented by Mr. Bassett. Launched in March/83, CAP has generated intense interest in both academia and industry. CAP is being sold world-wide. The company is currently undergoing rapid expansion. The department of Industry Trade and Commerce has awarded Netron a $650,000 R&D grant for the further development of CAP products.

Mr. Bassett developed his Frame concepts at Sigmatics Computer Corporation as a means of improving the production, quality, and maintainability of their customized computer systems for small businesses. Sigmatics continues in the turnkey business today, while Netron's primary focus is development and sale of CAP tools.

A few of the many systems designed and implemented by or under Mr. Bassett's direction are:

- A complete tax administration, general ledger, payroll, accounts receivable and accounts payable system for a municpal government.
- Accounting and financial systems for a newspaper, security systems manufacturer, a paper products distributor.
- Work-in-process control systems for a printing company and a containers manufacturer.
- Inventory control systems for a warehouse operation.
- A real-time clinical support system for patient monitoring.
- A run-time support package for raster graphics display system.
- A Bill-of-materials system for a containers manufacturer.

A a consultant, Mr. Bassett has been of service to many companies on topics ranging from artificial intelligence, through the design of new computer graphics hardware products, to management consulting tasks regarding organizational structure and information flow analysis. (c.f. Technical Manuals and Reports).

In the area of medical computing, Mr. Bassett is credited with the following technical innovations:

- Development and implementation of "open ended design" techniques for real-time (critical care) clinical support systems.
- A complete software package to implement a conversational time-shared terminal system with alphanumeric and graphic displays; new data reduction algorithms for handling redundancies arising in the storage and retrieval of beat-to-beat ECG information; a medical record data structure implemented with pure hash coding techniques; the introduction and adoption of structured Programming in a Scientific (Fortran) programming shop.

Directorships and Other Administrative Experience

Mr. Bassett is or has been a Director of the following corporations:

Holdco Corporation of Toronto, Ontario (Owens Netron jointly with Noma)
Caleq Corporation in Montreal, Quebec
Carol Electronics, Ottawa, Ontario
Datron Systems Ltd., Vancouver, B.C.
Sigmatics Corporation, Barrie, Ontario, as well as being president of this company
Skildata Corporation Ltd., Barrie, Ontario
Stroud Curling Club Ltd., Stroud, Ontario

Mr. Bassett is a member of the Curriculum Advisory Committee of the Business Division of Georgian College of Applied Arts and Technology in Barrie, Ontario.

Membership in Professional Societies

Association for Computing Machinery

IEEE Computer Society

Academic Career

Teaching Experience

Mr. Bassett was an Assistant Professor in the department of computer science at York University. Over his seven years at York he has been the course director for the following course types:

Data Structures
Machine Structures
Introduction to Data Processing
Introduction to Artificial Intelligence
Real-Time Systems
Advanced Projects

In November 1983 Mr. Bassett helped develop a course for managers to teach them how to apply techniques and technology to improve their productivity. This was sponsored by the Ontario Ministry of Colleges and Universities' Management Productivity Improvement Project.

At Sigmatics Mr. Bassett has designed and given several introductory "hands-on" computer programming courses in conjunction with Georgian College for training groups of Canadian Armed Forces from Base Borden.

Referred Publications, Articles and Invited Talks

(1) Bassett, Paul: "Design Principles for Software Manufacturing Tools" Presented at the Symposium on Application and Assessment of Automated Tools for Software Development, November 1, 1983, San Francisco.

(2) Bassett, Paul, and Giblon, Jay: "Computer Aided Programming: Part I" in proceedings of ACM/IEEE/National Bureau of Standards Soft-Fair Conference, Washington D.C., July 27, 1983.

(3) Bassett, Paul, and Rankine, Scott: "The Maintenance Challenge" in Computerworld (In Depth), May 16, 1983.

(4) "Computer Aided Programming" presented at:

DPMA Conference, Baltimore, October 31, 1983
ACM-83 Conference, New York, October 24, 1983
IMPACT'83 (WANG User Conference*), Boston, September 12, 1983
20'th Design Automation Conference* (ACM, IEEE sponsored), Miami, June 27, 1983
CIPS Conference'83, Ottawa, May 16, 1983
University of Windsor, Departments of Computer Science and Physics, Windsor, May 5, 1983
Federal DP Expo Conference, Washington D.C., April 14, 1983
University of Guelph, Dept. of Computer Science, Guelph, March 30, 1983
SOFTWARE/Expo-East, New York, March 15, 1983

*also in proceedings

(5) "Computer Aided Programming in an Artificial Intelligence Context" presented to CIPS special interest group, Toronto, May 25, 1983

(6) "Fourth Generation Languages" Panel Discussion, CIPS Conference, Toronto, May 18, 1983

(7) Bassett, Paul: "Computers are the Key to Advances in Software", Computing Canada Vol. 8, No. 19 (September 16, 1982)

(8) "Towards Assembly Line Techniques for Manufacturing and Maintenance of Commercial Application Software", presented to a York University Computer Science seminar, March 1980

(9) Bassett, P.G., Wong, J.W., Aspin, N., "An Interactive Computer System for Studying Human Mucociliary Clearance", Computers in Biology and Medicine, Vol. 9, July 1979

(10) "Towards a Clockwork Intellect: Recent Advances in Models for Epistemology and Teleodology", February 14, 1975, presented to York University Department of Philosophy Graduate Seminar

(11) "Programming Methodology for Delivery Real-Time Computer Services to Diverse Clinical Environments", September 6, 1974, Montreal, presented to 5th Canadian Medical and Biological Engineering Conference

(12) "Semi-Automatic Procedure Synthesis", two seminars to Psychology Graduate Seminar (Ph.D. thesis report), July 1974, York University Appeared on CBC Network television program called "Tomarrow Now" to discuss Artificial Intelligence, May 1974

(13) Horny, G.T., Bassett, P.G., Shepley, D.J., "Design of an Open Ended Clinical Support System", 25th Annual Conference for Engineering in Medicine and Biology, Hal Harbour, Florida, 1972

(14) Horny, G.T., Bassett, P.G., Shepley, D.J., "On-line, Real-Time Computer Support in an Intensive Care Environment", 4th C.M.B.E.S. Conference, Winnipeg, 1972

(15) Bassett, P.G., "A Combinatorial Theorem",. Canadian Math Bulletin, Vol. 9, No. 4, 1967

Technical Manuals and Reports

(1) "RPG 400, 500, Programmers' Reference Manual" Published by NORPAK Corporation, Ottawa, Ontario, July 1977

(2) "IGP Systems Reference Manual" Published by NORPAK Corporation, Ottawa, Ontario, May 1977

(3) "RPG 3000, 4000, 5000 Programmers' Reference Manual" Published by NORPAK Corporation, Ottawa, Ontario, December 1976

(4) "The Hospital Graphics Language (HAGL) Programmers' Reference" (Manual) Hospital for Sick Children unpublished report, December 2, 1974

(5) "The Hospital Graphics Language (HAGL) Run-Time Specifications" (Manual) Hospital for Sick Children unpublished report, December 2, 1974

(6) "FS-2000 Programmers' Guide" (Manual) Published by RAMTAK Corporation, Sunnyvale, California, October 1974

(7) Preliminary Specifications for the FIDDLE Macro-Assembly Language" York University Department of Computer Science - Internal Memo, October 1974

(8) "HAGL" Compiler Maintenance Manual" Hospital for Sick Children unpublished report, May 1974

REUSABLE SOFTWARE IMPLEMENTATION TECHNOLOGY REVIEWS

P. Grabow
W. Noble
C. Huang
J. Winchester

Hughes Aircraft Company
Ground Systems Group

Summary

Hughes recently completed a Navy study contract "Reusable Software Implementation Technology Reviews". The objective of this study was to review and evaluate available software development methodologies with respect to reusable software for Navy embedded computer systems. The concept of software reuse was also discussed in order to understand the wide varieties of reusability that are possible, determine what kinds of software reuse is suitable for a given environment, and uncover problems in achieving reusability.

Nineteen software development methodologies, representing current research, commercial and industrial sectors, were selected for review and evaluation. The framework for the methodology evaluation is based on five technology areas (language, structuring methodology, design environment, performance evaluation, and maintenance) and one non-technical area (market factors). Within each area, a number of attributes were derived to highlight its most important aspects. When appropriate, a range of possible values was defined for an attribute to provide insight into the nature of that attribute. These attributes were applied, in turn, to each methodology under review.

Some of the more significant observations and conclusions from the methodology reviews are: (1) No methodology provides code-level reuse between dissimilar application areas; (2) No methodology for large-scale development provides a reliable storage and retrieval mechanism for a code-level library; (3) Reuse of personnel is prime means of reusing software in industry; (4) Language technology, by itself, is only a component of the software reuse solution; (5) The larger the breadth of component reuse, the greater the need for formal specifications; (6) The level of specification required for reusable components is more formal than is currently used in large-scale developments; (7) Information explosion counterbalances the desire for formal specifications; and (8) Good specification language is very important for software reuse.

Based on the evaluation of the methodologies and the current software development practices, some aspects of software development methodologies could be implemented immediately and improve the efficiency of the software development process through reuse or by facilitating reuse. These recommended aspects include: (1) Provide consistent software development environments across multiple projects; (2) Reuse field-support software; (3) Provide automated support for documentation; (4) Reuse requirements, specifications, and designs; and (5) Design for reusability within restricted application domains. A procedure for selecting appropriate methodologies that support reuse can be derived from the results of the study along with determining what future enhancements are required.

The last part of this paper provides an example of a successful software reuse program that has been carried out at Hughes Aircraft Company. This discussion provides quantitative information on the productivity improvement through reusing software across

similar projects.

Section I. Hughes RSIP Phase I Report Highlights

1. Reusable Software Problems and Objective of the Report.

- o Problems - Cost, reliability and timeliness of software for Navy embedded computer systems.
- o Objective - To evaluate available software development methodologies.
- o Study Report Organization -

  SECTION 1 - Introduction and Summary

  SECTION 2 - Concept of Reusable Software

  SECTION 3 - Software Development Methodologies

  SECTION 4 - Software Development Methodology Reviews

  APPENDIX A - Advanced Combat Direction System

The incentives in software reuse lie in its increased reliability, improved development time, reduced cost, and more efficient use of manpower.

The objective of the report was to review and evaluate available software development methodologies with respect to Navy embedded computer systems, and to determine which methodologies are suitable for producing reusable software.

The report, as delivered, contains four major sections. The first section describes the purpose of the report, how the study was conducted, conclusions drawn from the reviews, and recommendations for incorporating software reuse into the software development process. Section two discusses the concept of reusable software and the factors that affect the feasibility and value of software reuse. In section three, attributes for a software development methodology are defined with respect to six technology areas. Section four reviews 19 software development methodologies with respect to these attributes. Appendix A contains a description of a real-time, embedded system currently under development for the Navy.

An overview of the report can be obtained by reading sections one and two. Serious comparisons of the methodologies, however, should be made by also reading sections three and four.

2. Analysis Applied to Reusable Software Implementation Technology Reviews

Selection of 19 Development Methodologies Through Research, Interviews, Etc.

- Prog. Apprentice
- SARA
- SCR
- PAISLey
- RNTDS
- Gypsy
- Etc

ESTABLISH TECHNICAL AREAS OUTLINED IN STATEMENT OF WORK

- Language Technology
- Structuring Methodology
- Design Environment
- Performance Evaluation
- Maintenance
- Market Factors

ESTABLISH KEY ATTRIBUTES OF TECHNICAL AREAS

- Paradigm
- Breadth of Problem Domain
- Life-Cycle Phase Product
- Size of Product
- Etc.
- Mgmnt Supp Tools
- Tool Integration
- Complete, Automd
- Etc.

ESTABLISH POSSIBLE RANGE OF VALUES FOR ATTRIBUTES (WHERE APPROPRIATE)

APPLY ESTABLISHED CRITERIA TO 19 METHODOLOGIES SELECTED

- Single Statement
- Procedure
- Subsystem
- System

- None Considered
- Minimal Reports
- High, Formal Reports
- Complete, Automated
- Reports

Nineteen software methodologies were selected for review. Those chosen represent approaches that are well documented in the literature or have been used by the industrial sector. For each of these development methodologies five technical areas and one marketing area were addressed (see figure). Within these areas, four to 14 attributes were examined. Each attribute is provided (where appropriate) with a range of values that the attribute might assume. These values are meant to provide insight into the nature of the attribute. They are not, in general, used as quantitative metrics for the comparison of the methodologies in this report. However, in some of the methodology reviews attribute values have been used to describe particular methodology attributes.

An example attribute is Size of Product, associated with the Structuring Methodology technology area. The possible range of values assigned to the attribute includes Single Statement, Procedure, Subsystem, and System. For the SCR methodology, this attribute was found to have the value "procedure, subsystem, or system".

3. Subjects of Software Development Methodology Reviews

- o Research
  - Programmer's Apprentice (PA)
  - Harvard Program Development System (PDS)
  - System Architect Apprentice (SARA)
  - Draco
  - PAISLey
- o Commercial
  - Software Cost Reduction (SCR)
  - Restructured Naval Tactical Data System (RNTDS)
  - Formal Development Methodology (FDM)
  - Gypsy
  - Information System Design Optimization System (ISDOS)
  - Software Requirements Engineering Methodology (SREM)
  - Higher Order Software (HOS)
  - Structured Analysis and Design Technique (SADT)
  - Ada*-Oriented Methodologies
  - Smalltalk
  - Raytheon
- o Industrial
  - Hughes
  - Grumman
  - Boeing

To conduct the methodology review, a sampling of different software development methodologies was chosen, 19 of which were reviewed. The data for these methodologies were gathered by reviewing the relevant literature and, where feasible, interviewing knowledgeable personnel. The report contains detailed reviews of each of these methodologies based on a set of attributes (defined in Section 3 of the report).

The software development approaches reviewed in Section 4 of the report, which represents a cross section of current methodologies from three categories: research, commercial, and industrial. The first category relates to approaches which are still in a laboratory environment (usually within a university). The second refers to methodologies available for purchase on the commercial market, in the public domain, or as

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

government-furnished equipment. The last refers to approaches in use within industry and not generally available outside a particular company.

4. Technology Areas Addressed

- o Language Technology
- o Structuring Methodology
- o Design Environment
- o Performance Evaluation
- o Maintenance
- o Market Factors

This report reviews software methodooogy in terms of five technology areas delineated in the statement of work. In addition, Market Factors has been added to include non-technical aspects of the analysis.

Language technology includes languages used for describing a system at each stage in the software development process, including specification languages, design languages, implementation languages, and test languages. This is important to reuse since reusable components require precise definition.

Structuring methodology is the methodology used for synthesizing a new system from reusable software. Such a methodology attempts to classify software for later reuse, and provides the necessary tools and procedures to help the designer define, store, retrieve, and modify reusable components.

Embedded software system design environment technology includes the tools, procedures, and facilities used in the development of a system. This is important to software reuse because the environment provides continuity among the various application systems that reuse software.

Performance evaluation includes the technology for specifying and evaluating performance characteristics of a system, including timing, accuracy, and adherence to specified functional characteristics. Testing tools to monitor and analyze system execution are included in this area. Performance is important in deciding whether an existing component will satisfy the requirements of the system being built.

Maintenance includes the technology for correcting errors or modifying reusable software. Configuration control tools and procedures, testing tools, and system monitoring tools are in this area. This support is essential to software reuse, since the reusable components used in new systems come from libraries that are in the maintenance phase of the software life cycle.

Market factors are the viability attributes that influence the decision to adopt a methodology, including whether an implementation of the methodology exists, the cost of acquiring the methodology, and if the methodology is available. Even if a methodology satisfies all technical demands, market factors will determine if it is viable.

5. Conclusions from Reviews

- o No methodology provides code-level reuse between dissimilar application areas.
- o No methodology for large-scale development provides a reliable storage and retrieval mechanism for a code-level library.
- o Reuse of personnel is prime means of reusing software in industry.
- o Language technology, by itself, is only a component of the software reuse solution.
- o The larger the breadth of component reuse, the greater the need for formal specifications.

- The level of specification required for reusable components is more formal than is currently used in large-scale developments.
- Information explosion counterbalances the desire for formal specifications.
- Specification language is more important than the implementation language.
- Only methodologies surveyed using formal, semantic descriptions were Gypsy and HDM.
- Research methodologies concentrate on small well-defined systems.
- Industrial methodologies address large-scale systems that are incompletely defined.
- A methodology should be defined before it is implemented.

The following conclusions can be made on the basis of the methodology review:

Between Dissimilar Applications- examined which purported to provide the reuse of source code between dissimilar application areas. In fact, where source-code reuse occurs at all, it happens within narrow application areas.

Source-Code Library- development efforts provide a reliable way of storing and retrieving items from a code-level library. Some methodologies were able to implement libraries, but the retrieval of the correct item from the library was manual process. (Which was often so difficult that it was easier to code a new item than to look for one to reuse.)

Importance of Personnel- reusing software products is via the reuse of the personnel who created the products. Much of the knowledge which advanced methodologies attempt to capture is already resident within these knowledgeable personnel.

Language Technology- component of the solution to the software reuse problem. A language which allows the creation of portable and re-linkable components can facilitate reuse of low-level primitives. But the inherent variability of higher-level functions all but precludes their incorporation into a language as reusable elements.

Breadth of Reuse- greater the need for formal specifications. For example, when reuse crosses organizational boundaries, the need for precise component specifications increases substantially.

Degree of Formalism- reusable components is more formal than is currently used for large-scale system development. Specification techniques for large systems usually address syntax and inadequately handle semantics. The description of a reusable component must include a semantic description as well as the syntax definition of the component's interface.

Information Explosion- specifications is counterbalanced by our inability to deal with large volumes of information. As a specification becomes more formal, the amount of information in the specification can easily become overwhelming.

Importance of Specification Language- language used to describe a component is more important than the programming language used to implement the component. A specification is the foundation on which a component is built, tested, maintained, and reused.

Formal Semantics- formal, semantic specification techniques were those that are used to produce formally verified software (e.g., Gypsy and FDM). However, they have not been applied to large-scale software development.

Research Methodologies- small-scale systems that are well-defined.

Industrial Methodologies- deal with large-scale systems that are initially incompletely defined. Incomplete definitions are

not necessarily anyone's "fault", since the development process is highly iterative.

Methodology Implementation- it is implemented. This may appear to be obvious. However, the number of "software tools" lacking an underlying methodology says otherwise.

6. A Procedure for Choosing a Methodology

1. Determine size of system you intend to build.
2. Determine breadth of problem domain.
3. Determine organizational distance for reuse.
4. Choose life-cycle phase for software reuse.
5. Determine level of semantic description required.
6. Choose methodology compatible with the above.

The following procedure can be used to select a suitable methodology:

First determine the size of the system that is to be built and the breadth of the problem domain that it will address. Eliminate those methodologies that cannot address systems of your size and problem domain.

Next, determine the longest organizational distance between the people who will produce the reusable software and those who will reuse it. Based on this distance, choose the life-cycle phase in which the software will be reused. Unless this distance is small, the level of reuse should be above the code level (e.g., requirements, specifications, or design).

Determine the level of semantic description required for a reusable component based on the organizational distance and the life-cycle phase product chosen. The longer the distance and the closer the life-cycle phase product is to the code level, the greater the need for formal semantic descriptions. Eliminate those methodologies that cannot provide the level of semantic description required.

Finally, determine the starting point for system development in your organization (e.g., requirements). Eliminate those methodologies that do not address the software life-cycle phases from your starting point to the phase in which software reuse will occur.

Given this procedure, let us choose a methodology for a large, real-time, embedded communications system for ships, planes, and ground stations. Assume that the individual subsystems will be built by different companies using as much reusable software as possible and that the starting point for any of these systems is a set of requirements. (To carry out the procedure, the information contained in Figure A of Section 1 and the detailed methodology reviews in Section 4 of the delivered report should be used.)

On the basis of size, SCR, RNTDS, ISDOS, SREM, SADT, Hughes, Grumman, and Boeing are possible candidates among the 19 methodologies reviewed. However, when problem domain is considered, this list is reduced to SCR, RNTDS, Hughes, Grumman, and Boeing.

Since the organizational distance will be large, reuse should occur above the code level (e.g., requirements, specifications, or design) and a high degree of semantic description will be required. Of the methodologies in the reduced list, only SCR provides more than adequately covers the entire software life-cycle.

Another factor that has not been mentioned (but occurs within the set of methodology attributes) is the level of automated support. Unfortunately, the SCR methodology lacks the necessary automated support for large-scale system development. The organization choosing a particular development methodology must decide how much

automated support is necessary to make the approach practical.

7. Recommendations

- o Provide standard software development environment.
- o Reuse software development environment.
- o Reuse field-support software.
- o Provide automated support for documentation.
- o Reuse requirements, specifications, and designs.
- o Design for reusability.
- o Restrict application area.

Based on the evaluation of the various methodologies, and the review of the pertinent literature and current practices, there are several recommendations that appear to be obvious. Many of these could implemented without risk, and could improve the efficiency of software reuse development.

A standard software development environment should be provided that encompasses the entire software life-cycle. The quality and ease of use of the software development environment can make a significant impact on the ease of developing software and the quality of the software which results.

There are numerous instances of the reuse of the software development environments. However, there are still cases where the support tools and procedures needed to create a body of software are created largely from scratch for a particular project.

Over and above the need for development software, many projects require substantial amounts of software to support the project in the field. This includes maintenance and diagnostic packages, data reduction and analysis packages, simulators, and exercise generators for training. The combined size of field-support software often exceeds the size of the application software.

Automated support for the generation of formal documentation should be provided. The reuse of software requires the generation of documentation that is more formal to ensure that the software to be reused matches the requirements of the new system. The preparation of documentation for this process is time-consuming, tedious, and error-prone. Automated tools can help the user work more efficiently, and enforce the level of formality needed.

It is apparent from the review of the methodologies that reuse of source or object code is generally beyond the scope of the approaches examined, unless the problem domain is highly constrained. Therefore, higher-level software products should be reused with the relative de-emphasis of source-code reuse.

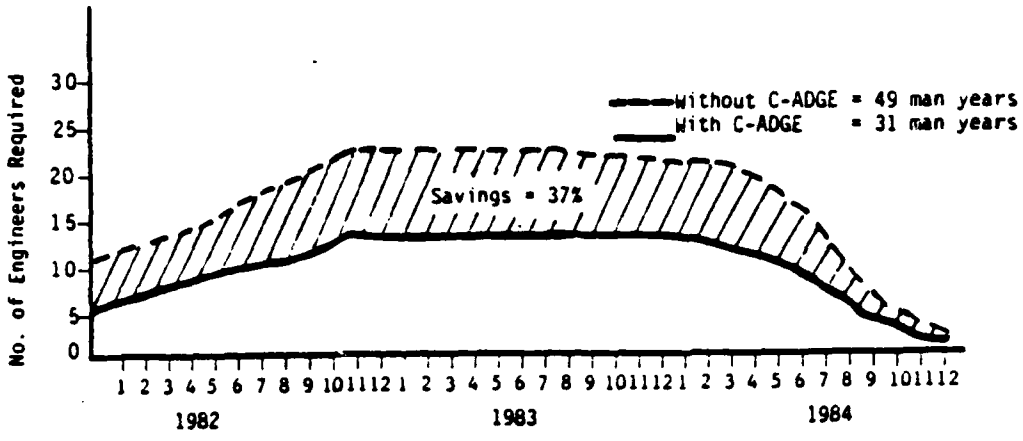
Software that is to be reused must be designed for reuse. Attempting to reuse components that were not designed for reuse will probably fail.

As the application area becomes larger and less-understood, the ability to reuse software rapidly diminishes. Therefore, the application area in which software is to be reused should be restricted to well-understood problems.

Section II. An Example Successful Software Reuse

8. Productivity Improvement Using C-ADGE Software at Hughes

Quantitative benefits associated with software reuse are difficult to capture. At Hughes, a Common ADGE (C-ADGE) project has been established to allow the reuse of a segment of air defense ground environment (ADGE) software across multiple projects.



Productivity improvement is measured by comparing the number of engineers required if each project using C-ADGE packages were to build that software themselves versus the number of C-ADGE engineers providing the packages.

Cost savings history has also been maintained during a portion of this effort.

The similarity of ADGE system requirements makes ADGE software a fertile area for increasing productivity, by increasing software reusability. C-ADGE is a formal effort which has as its objective increasing software reusability by generating product line software for the operating system, support utilities, and diagnostics to be developed and maintained by a functional organization for use by all ADGE programs. Product-line requirements specifications and baseline modifications to the standard existing packages were completed. These packages were then implemented.

The cost saving was based upon the reduced labor requirements when two actual projects used the same software package. Each project did not have to acquire duplicate staff to develop the functionally identical package, as has occurred in the past. Additional savings were obtained through reduced demands for the Software Development System computers and consumables.

An Overview of the Software Design Library

J. Hearne
J. Winchester
Hughes Aircraft Company

1. INTRODUCTION

The potential for application of computer has grown rapidly in recent years, due primarily to the decreasing cost and size of computer hardware components capable of performing at increasingly high rates of speed. As hardware technology advances, more and more complex systems become technologically feasible - provided the software needed to control and integrate such systems can be produced. Unfortunately, software technology has not advanced at the same rate as hardware technology. More powerful,higher order languages (e.g., Pascal, Ada) and associated development tools and techniques (e.g., structured design and coding methodologies) are making significant contributions. Nevertheless, these efforts alone do not appear adequate to cope with the crisis arising from increasing system complexity and spiraling production/maintenance costs.

The dramatic decrease in the hardware cost/performance ratioachieved over the past decade stems, in large measure, from an extensive use of "building block" techniques. More and more complex functions are being embedded in single, relatively general purpose components, which can be readily and quickly combined to produce even more complex hardware systems and subsystems.

To a very limited degree, by making use of standard subroutine libraries, particularly for performing common mathematical functions (e.g., sine, cosine), software developers have employed building block components in a similar way. But such subroutines are generally applicable only at the lowest design levels and are normally used at the discretion of the implementor, where they provide little or no benefit to the software designer. The ready availability of more powerful software building blocks, combinable in more general ways, would greatly increase the potential for reuse of existing software modules.

For example, the Hughs Air Defense Radar (HADR) is a system utilizing an embedded computer for controlling a single radar antenna, processing radar returns, and forwarding target information back to a central site. Successive installations of the HADR system typically include substantial reuse of software developed for previous installations, but also typically require new features and modifications of earlier capabilities. Thus, new software subsystems must be developed to augment or replace previous ones, and these new subsystems are often very similar to these they replace. For example, the human interface in one HADR installation had to be replaced by a new version when the originate display station was replaced. Yet, essentially the same information was to displayed, and essentially the same kind of operator actions were to be implemented. If the human interface subsystem had been composed of readily manipulable software components, the amount of subsystem rework required would have been substantially less.

This paper reports on a system, called the Software Design Library (SDL), that provides computer assisted design techniques to facilitate reuse of previously built software modules in designing and building new systems. Section 2 reviews related work and discusses how it compares with the SDL. Section 3 presents an overview of the SDL itself, while section 4 describes its application to a large command and control system product line. Section 5 draws some conclusions based on this application experience, and section 6 discusses future directions in refining the system and applying it to other application domains.

2. RELATED WORK

As noted above, software reuse has not been applied effectively to alleviate extensive rework of applications software. Most reuse has been accomplished by providing libraries of general purpose, low-level functions, such as mathematics and statistics libraries. These kinds of routines frequently dictate the use of particular data layouts, which must be rigidly adhered to by any calling program, and thus impose potentially unacceptable constraints on data representation. Furthermore, the reuse of software components need not be restricted to computer programming language statements (source code) or their translated, machine language form (object code). Horowitz and Munson (1) discuss a wide range of potential approaches to software reuse, encompassing requirements and design as well as code. These approaches include program generation, formalized expression and interpretation of requirements, and the reuse of previously generated designs as well as code. Freeman (2) proposes research toward the reuse of software products at all stages of the software development cycle. Neighbors (3) has developed a software tool based on Freeman's approach. Work continues on that project in the form of classification and application domain analysis.

In spite of these advances, however, there is still a need to demonstrate the practicality of these approaches in large scale application. As yet, software reuse has only been employed successfully in isolated, well-understood problem areas such as report generators and compiler-compilers. The problem of reusing complex, real-time application software, from one project to another within a product line, has not been effectively resolved.

3. SYSTEM OVERVIEW AND RATIONALE

Our objective in developing the Software Design Library was to support effective reuse of software components in large project environments. Two impediments stood out clearly as having major importance. (1) the difficulty of forming a deep understanding of existing software operation at a level sufficient to adapt it to new requirements, and (2) the heavy impact that performance properties of software components have on the design of real-time embedded computer systems. After surveying other approaches for dealing with these issues, a method of approach was developed. The SDL views software as realizing a set of programming goals which can be described in relatively high-level terms, terms normally suitably to requirements documents. Such goals can be realized by different strategies, or implementations. The goal identified for a given system frequently arises in future systems in the same product line. The SDL thus presupposes that the high-level description of software goals can be made to correspond to pre-exiting software components. To support this approach, we developed a prototype Software Design Library system. The SDL facilitates the construction and application of catalogs of malleable components that are based on the semantics of an application domain, in a hierarchical way. The SDL also facilitates their configuration for reuse without requiring the deep understanding of each component's structure. Two roles were defined. The cataloger supports the design activity for the first generation system of a product line. He has expertise in the subtle design issues, tradeoffs, and functional requirements that influence the use of software components. The designer of a future system in the same product line uses software cataloged for previous systems. He has expertise as the general application area and in specific functional requirements for a new project.

Thus, for the cataloger, an interface was provided for constructing catalogs of components and the semantic models which describe their application. For the designer, we provided an interface suitable for configuring application systems solely from requirements expressed in terms of the application's specific vocabulary. We believe that separating the roles of cataloger and designer encourages higher generality on the part of program developers, and simplifies the exercise of deeper practical knowledge about the application area on the part of designers.

3.1 General Characteristic of the SDL

As noted above, the SDL supports the logical naming of goals as groups of procedures and data structures, allowing any number of versions (strategies) within the same logically named goal group. By describing the subtle semantic differences among various versions within a group, the designer has greater flexibility in configuring a specific goal, yet need not consciously select the individual version to be used. Each of the versions represents one possible refinement of the goal, but is closer to realization in two ways: (1) each version has associated source code forming a partial implementation, (2) and each version also has an associated list of subgoals that are subordinate to it, that are required functions for completing the goal, and that form a problem reduction network among all goals in the catalog. The interleaving of goals and strategies leads naturally to a problem reduction search technique that allows the definition of high level strategies to be stated loosely in terms of subgoals, with design criteria, operational constraints and the various costs of alternative configuration playing a guiding role in the design process.

There is a one-to-many mapping relating each goal to a set of alternative strategies for implementing that goal. Achieving the goal requires the (eventual) adoption of one of these strategies. There is also a one-to-many mapping of each strategy contained within a goal onto names of subgoals, all of which must be achieved if that strategy is to be effective in implementing its containing goal. The structural effect of this relationship between goals and strategies is that the catalog contains a collection of three level Or-And tree fragments; the root node of each fragment is a named goal with Or structure; the middle level of And nodes corresponds to the various strategies for implementing the rooted goal, and the tip level of nodes corresponds to the names of subgoals for the strategies at the middle level.

There are two major attributes associated with each strategy that determine whether or not it will be selected in a given design session: applicability and cost. Thus, each strategy description includes specification statements that describe its applicability to the designer's requirements, in the form of an applicability predicate. When the SDL is used by the designer to select strategies for implementing a required goal, it evaluates the applicability predicate for each strategy. Those strategies deemed applicable are deleted from the candidate list. If a designer's specifications cannot be met by an available strategy, the SDL notifies the designer that his specifications are not implementable within the catalog. When a goal can be realized by a choice of many applicable strategies, the cost of each strategy becomes an influence in the selection. The SDL allows the cataloger to declare a cost function in terms of weighted resource consumption over any number of resources. The cataloger declares resources and expressed the resource consumption profile for each strategy in terms of design criteria. During the search for an effective application program, the total cost of each candidate program is used to direct search, so that the most resourceful system meeting designer requirements is the one configured.

The operation of the SDL can be summarized as follows. The cataloger creates the goals and strategies in an SDL catalog, and provides characterizations (including applicability and cost expressions) of each. The designer states a high level goal for the SDL to configure. Given the designer's high level goal, the SDL attempts to construct an implementation tree by combining the Or-And fragments until all tip nodes contain only source code (or other text), with no subgoal references. When the designer has completely configured an application system, the strategy inter-connection tree required to implement the system is stored within the SDL database, and is then used to configure application programs by manipulating the "pictures" (program source text and/or design-level descriptions thereof) that correspond to the selected configuration of strategies. The source program created in this way can be compiled and executed. Within the limits of its ability to distinguish between cataloged components and their relationship to the application area, the SDL uses the catalogued descriptions to act as an information designer so that the human designer can concentrate on exploring system requirements. Productivity improvement is dependent on the savings enjoyed by designers who use a catalog to create systems versus the

cost expended by the cataloger to create the descriptions which the designers use.

4. AN EXAMPLE

In setting out to demonstrate the viability of the SDL in the context of a realistic applications, military command and control information system (CCIS) was selected as an appropriate application to catalog. CCIS is an application which typically involves large systems that must operate in high-performance, high transaction rate environments on distributed system architectures. This is precisely the situation which leads to a series of customized systems in a common application area. Thus, the payoff of applying software reuse technology to this area is potentially great. The CCIS on which the initial catalog is based is a large, multisite display system that includes general applications software whose functions are (1) to display text and graphical data; (2) to generate, communicate, and process messages; and (3) to maintain a data base and notify selected users of changes to the database. This software supports a wide variety of special applications software, performing functions in the area of availability, logistics, weather, threat response, rescue operations, and many other areas. Processing at each site is allocated among host computers, workstations processors, and database processors to optimize system performance. If this function allocation is varied, a different strategy for implementation results.

A subsystem of CCIS, involving the display of "totes" to an operator, was the first major program to be cataloged. Totes are text data for display. They are generally tabular, with (perhaps) repeating fields of slowly time-varying data. Their structure is defined by setup preprocessing. Totes can be invoked for generation and display by selection from a menu, selection of a function key, (special function keys), special application programs, execution of a sequence file, monotoring of other displays, and display transfer between workstations. Tote formatting builds the tote display file and handles processing related to the tote. Text transfer converts information in the tote display file into a format suitable for display. Function key formatting formats function key sets (which can be associated with totes) for display.

A major portion of CCIS software for totes has been cataloged, resulting in a total of 90 modules in the catalog. The associated catalog graph contains a total of 116 goal and strategy nodes, and has a depth of 7 levels of design goals. The catalog was developed to allow post processing two forms - the actual software design (represented by structured English, or "pseudocode"), and design documentation in the form of pure text. (There was no actual running source code available for cataloging at the time of cataloging.)

There are two costs measured in the CCIS totes catalog. The simplest to measure is lines of pseudocode, with each strategy node incorporating its portion of that cost. The other cost, system response time, is a much more significant cost for CCIS. It is also more difficult to measure and is not conveniently modeled as a sum of constituent costs. Hence an estimated average system response time was assigned to each strategy at the highest level in the graph at which the strategy distinction exists. All subordinate strategies are assigned a system response time cost of O.

The CCIS totes catalog is subdivided into the three functional areas mentioned earlier - tote formatting (including tote, initiation and tote display file access primitives), text transfer, and function key formatting. Each of these areas is cataloged in a hierarchical fashion, reflecting the underlying structured design. The catalog representation decomposes the formatting of a tote for display into the formatting of the constituent parts of a tote (header, nonscrollable data, vertical lines, page number, page data), and central processing necessary to display the tote. It is important to note that this representation of text transfer is in terms of visible components of a tote which can be related directly to tote format requirements. The intent is that this will provide "hooks" inside the catalog, allowing reuse of selected subsystems as appropriate to the requirements of a new system.

In this catalog, several goal nodes have two strategies - one if processing is performed in the host computer and one if processing is performed in the workstation processor. The catalog reveals the pervasive effect such a choice of processor has on the software, even at the language independent detailed design phase. This illustrates the importance of developing design approaches which hide the architecture from as much software as possible.Note also that the catalog is quite finely structured, with more catalog nodes than modules in any one implementation. The reasons for this are as follows:

(1) The existence, and pervasive extent, of two strategies results in multiple versions for many of the modules. Thus, counting module versions, the number of nodes is smaller.

(2) An effort was made, for the sake of the experiment, to include catalog structure wherever it could reasonably be justified. In practice, fewer nodes would probably be used.

(3) To guarantee that any implementation selected from a catalog contains a logically complete and consistent set of source code statements, it is occasionally necessary to define nodes which do not perform "visible" processing.

(4) Most important, a guideline was adhered to on CCIS to create a goal for every "atomic visible" function. A visible function is one which can be observed and related to a requirement. For example, "display tote" is an operation performed by the system which results in text data appearing on a display screen; this is a visible function. However, it is not atomic; it is composed of smaller "building blocks", such as "display header", "display page number", and "display data". The significance of this guideline is two fold. First, the visibility of the functions associated with catalog goals allows designers of the functions associated with catalog goals allows designers of future associated with catalog goals allows designers of future systems to assess the relevance of software in an existing catalog to their problem. Second, using building blocks to represent functions in a catalog decomposes a function in a manner consistent with stepwise refinement and structured design, resulting in small, relatively encapsulated and well-understood building blocks. The semantics of these small pieces should be easier to characterize, leading more easily to a characterization of the application domain of the problem area in general.

(5) Finally, it should be noted that the catalog provided empirical evidence in support of information hiding as a design principle to enhance the reusability of the resulting software. Primitive routines were written to allow access to the Tote Display File - a major data file needed by several parts of the system - thus providing functional access and hiding the internal file structure from the rest of the system. If this approach had not been taken, it would have been very difficult, if not impossible, to encapsulate tote display information in the catalog. This approach was taken at a minor overhead cost (for calling primitive subroutines)in order to provide uniformity and control of tote display information among all using functions. Another part of the system did not use this technique. The resulting data coupling will make this area more difficult to reuse in another setting.

5. CONCLUSIONS

Based on cataloging experience to date, the following observations can be made:

(1) Software can be cataloged, using the Software Design Library, for potential reuse in future systems. The result is a "structured catalog" which contains a representation of the design structure in a form that reinforces stepwise decomposition. A cataloged system is thus a synthesis of subsystems which can be used as building blocks to configure new systems. The quality of a catalog is tied to the quality and style used to

design the system. Observations below specify design and cataloging guidelines which promote the reuse of software.

(2) Information hiding is a key design guideline which significantly enhances software reuse. This was expected to be true and was confirmed by the experience gained to date in cataloging CCIS software. Specifically, the tote display modules defined which hide the details of the file structure and provide only the information the caller wants. Certain other data structures in CCIS software do not have similar access routines, and in those cases the cataloged software is closely tied to the data structures. The degree to which software can be reused in future projects with slight differences in requirements will depend largely on the degree such data coupling can be avoided.

(3) Processor independence is another key design methodology which enhances software reuse. When performance considerations caused the reallocation of processing from the host computer to the workstation processors, the software was substantially changed at the detailed design level. This problem is not as well understood as information hiding and has some code-level implications (such as programming language used) which cannot always be avoided. One possible approach to reduce the impact on detailed design of reallocation of processing would be to implement a mapping (perhaps an extension of the compiler) of processes to processors, with the detail allocation of processes to processors in a few modules, with a common communication mechanism (such as mailboxes) among distinct processes or tasks.

(4) Visible functionality of design goals in the catalog enhances the understanding of the representation of software in the catalog. As mentioned earlier in reference library, can be used to convey understanding to support reuse of the subject software in the catalog. A format has been developed to use the documentation facility to provide definition of terms as used on the project, an abstract describing the functions performed by a subsystem in the catalog, assumptions and limitations of the cataloged software, and a discussion of salient design issues (such as the use of information hiding). Information about the catalog itself can also be provided, such as an indication of relatively self-contained subsystems which are suitable for reuse in the modified context of another systems. The designer of a new system can then browse through the documentation supported by the Software Design Library to aid in the extraction of software subsystems for reuse.

6. FUTURE DIRECTION

Thus far, we have formulated concepts to support software reuse, built the prototype Software Design Library System to implement these concepts, and used it to catalog software on two projects. Based on this experience, the following activities are planned for the future:

(1) Rehost the Software Design Library (SDL) on a larger system. This step is necessary for using the tool in a production mode.

(2) Reuse software from the existing CCIS catalog on future command and control systems. Several candidate systems are in the preproposal stage and are being analyzed to assess the feasibility of using software from the CCIS catalog. Another project family (trainers) is also a candidate for using the SDL.

(3) Develop methodology guidebooks for designing and cataloging future software, particularly software to be designed and/or coded using DoD's Ada language. A working group is currently addressing design methodologies in the context of Ada which, includes several features intended to facilitate software reuse. One such feature is the Ada package, which consists of two parts: (1) the package

specification defines an interface to services (such as searching and sorting) that are used by other parts of a program; (2) the implementation of the services promised in the specification is contained in the package body. Naturally, the specification might be implemented in several different ways, and different implementations may be appropriate to different costs and applicability requirements. In Software Design Library terms, the alternative bodies for a given package specification may constitute differing strategies to achieve a programming goal.

(4) The other feature of Ada that is at least in part directed to the matter of reuse is that of generic program units, especially generic packages. Generics are a means of parameterizing program units by type and subprogram--procedure or function--as well as value. For example, one may thus define in Ada a generalized sorting procedure which sorts with respect to several ordering relations. The particular ordering relation would be provided when the generic procedure is "instantiated", by giving the subprogram a particular function which defines the ordering relation.

(5) Develop a better understanding of the semantics of a programming goal in a software catalog. Ideally, such goals could be viewed as primitives at a sufficiently high level of abstraction. They could then be viewed as extensions to blocks used to compose new products in the same product line. The semantics of a goal would need to account for the possibilities inherent in different implementation strategies. More importantly, the problems raised by the fact that such goals are not self-contained or "closed", but have interface dependencies with other goals, need to be studied.

(6) Use practical experience to develop ways to capture performance and cost measures in a catalog. Currently, all costs are modeled incrementally. This is fine for costs such as time and storage. But some performance measures, such as system response time, are not adequately measured as the sum of contributions from constituent parts of a system.

(7) Define and implement concepts and tools to hide architectural details from the implementation of projects. Some details (such as scaling precision and bandwidth) cannot be hidden for performance reasons. But it may be possible to develop a general-purpose communication protocol, for example, which would localize the project-specific protocol to a small subset of a system.

(8) The language of cost and applicability is presently restricted to sentential logic. Whether the introduction of quantifiers would add usefully to the expressibility of the system is certainly of interest.

(9) Develop a configuration control scheme and support tools to modify the catalog whenever the underlying design is modified.

REFERENCES

(1) E. Horowitz, and J. Munson, "An Expansive View of Reusable Software", University of Southern California.

(2) P. Freeman, "Reusable Software Engineering: A Statement of Long-Range Research Objectives", University of California, Irvine, Technical Report 159, 10 November 1980.

(3) Neighbors, J. "Software Construction Using Components," Technical Report 160, University of California, Irvine.

(4) J. Hearne, Software Design Library Cataloger Handbook, Hughes Internal Report, 1982.

(5) R. Cooper, MetaCAD: A Knowledge Based Software Support Environment, Hughes Internal Report, 1981.

(6) Hansford J. Myers, Reliable Software Through Composite Design, Van Nestrand Reinhold, 1975.

Notes on Cataloging Methodology
SUPPORTING DOCUMENT TO:
An Overview of the Software Design Library

Hughes Aircraft Company

Based on the experience cataloging CCIS software and consideration of issues to be faced in reusing this software on future CCIS systems, some observations on design and cataloging methodologies to promote software reuse have been made and are discussed below. In reality, methodologies for designing and cataloging software impact each other. In the discussions below, it is presumed that the software design team for a new project has access to catalogs of related systems in the SDL.

1. Include "directory nodes" in the catalog to aid the designer of a new system in determining the relevance of software existing in the catalog (see Figure 1). The highest-level directory node is the default starting node for the catalog, with one strategy which has no subgoals and no source or object code. It is purely a documentation node which tells the designer how to use the catalog. In addition to describing requirements and design information about the system, it also describes the constituent "visible" subsystems and gives their corresponding goal names. This tells the designer how to access the part of the catalog of interest to him. The directory node must also give the goal name corresponding to the highest-level goal in the catalog containing the actual software.

   Variations on the approach exist. The subgoals referred to by a directory node could in turn be directory nodes for subsystems, which names the start goal for that subsystem and the directory nodes for subordinate subsystems, giving a layered-menu definition of the catalog contents.

   Another approach is to embase this information in the catalog structure without using separate directory nodes. The advantage of this approach is that some redundant structurizing in the directory node approach is avoided, and the size of the catalog is reduced. The disadvantage is that, by starting at the default node, the designer would generate the document for the entire catalog and would have to wade through it to find what he wants. The documentation facility added to the SDL this year supports either approach.

This catalog contains the processing to support the display of totes in command and control information system. Totes are text data for display. The processing is divided into three areas:

GOAL NAME FUNCTION

TOTEFORM
: Tote Formatting defines the Tote Display File, which is the major data file defining totes for display.

FTDTRANS
: Formatted Tote Display Transfer extracts data from the Tote Display File and converts it into a form suitable for display as text data.

FKFORM
: Function Key Formatting formats a variable function key set associated with a tote for display. It also formats the fixed function key set and its subfunction key sets for display.

This is a documentation node which provides a high-level description of a subsystem in a catalog. It tells the designer the names of its constituent subgoals to allow the designer to choose

the subsystems he needs from the catalog and access them. This directory technique, supported by the SDL documentation facility, allows the designer to use subsystems as building blocks to form a new systems.

2. In addition to providing functional information about the composition of the catalog, the documentation facility should be used to describe its content. A format has been evolved to associate descriptive information with selected nodes. This format involves the definition of terms as used in the system, an abstract describing the functional processing performed by this node and its subordinates, the calling sequence to perform this function, any relevant assumptions and limitations, and related design information such as the principle objects and operations on them in this subsystem. It is important that the terms used be defined precisely as used, since terminology can vary even on similar projects. It is also crucial that limitations be explicitly noted; this is probably the most important source of information to a designer about the reusability of the software. This section should include global data structures assumed, performance constraints, architectural constraints and any other information describing the presumed context of the subsystem. See Figure 1 for an example of text from the CCIS catalog as supported by the documentation facility.

DEFINITION OF KEYWORDS

VFK a variable function key, part of a VFK set which is associated with a tote. When a tote is displayed, its associated VFK set is also displayed. A VFK set is associated with a screen; when the passive screen is made active, the display VFK set changes.

Display Message Unit

A format used to represent information for display. This format consists of 7-bit ASCII character codes, relative positions of the keys on the display, display control information, and processing control information which indicates whether the function key processing is performed in the host computer or workstation processor.

Abstract

Function Key Formatting formats function keys for display at an operator console. If a tote is to be displayed and the tote has an associated VFK set, Function Key Formatting formats the VFK set for display in the VFK area on the screen.

Assumptions and Limitations

Function Key Formatting requires the existence of the Function Key Set Definition File containing function keys in display message with format. Details of this file are referenced extensively throughout Function Key Formatting.

3. The structure of the catalog should reflect the structure of the underlying design. This is necessary to allow the SDL to extract the appropriate software to perform processing for a selected subsystem in the catalog. Indeed, the catalog for a new application area should be built by design team leaders as the system is being designed. This structural correspondence implies a hierarchical representation refers to the stepwise decomposition of "larger" goals in terms of "smaller" subgoals. This "hierarchy" is not strictly a tree structure, but can be represented as a tree (perhaps by replicating goals, as is done in structure charts), provided the goal-strategy network embodying the decomposition does not contain any loops (in which a design goal is refined partially in terms of another instance of that same goal). Such hierarchical representation allows a designer to select a "large" goal and automatically obtain the details inherent in the decomposition. This supplies on organizational structure on the content of the catalog, and thus helps to manage the information contained in the catalog and reduce its complexity.

4. Use visibility of processing as a guideline for the creation of goals in a catalog. If the processing corresponding to a node in catalog represents an observable phenomenon to a designer of a new system, he is in a position to assess its relevances to his needs. This provides conceptual "hooks" into the system which can be supported by a menu/directory approach as in (1). These "hooks" should, at least at high conceptual levels of the catalog, embody system requirements (since they represent natural "world views" of the system objects which are invariant across all implementation strategies for the system), and the catalog is then seen as a mapping of system requirements into design and implementation.

5. The catalog should reflect the objects and operations in the system conceptually and hide their structure as much as possible. This depends largely on the design approach taken, but has two applications for cataloging. First, after identifying major conceptual objects such as totes, the catalog can then be organized and represented in terms of all the things that can happen to these objects (e.g., format for display). These "things that can happen" are the operations and will be the goal nodes in the catalog. These will tend to relate to system requirements at high levels of the catalog structure and will provide "hooks" as in (4).

Second, the catalog can represent a form of object hierarchy. In the design of the software cataloged from CCIS, access primitives were defined to provide contents of the Tote Display File without the need to know the file structure. The Tote Display File (as represented in the catalog via the primitives which access it) is various parts of CCIS defining the hierarchy. In fact, this approach can be extended by grouping the primitives into subsets called by each subsystem, each pair of subsystems, etc. The using subsystems access the subobjects they need. A "phantom" goal node serves to group the subobjects together to form the complete Tote Display File; this phantom goal is not referenced by any other subsystems and merely unifies the hierarchy. In fact, the primitives make use of still other primitives which performs system support functions for files (open, close, read, write). This provides a layering of information hiding in the CCIS software and in the catalog.

RESUME

J.W. (Jim) Winchester
Manager
Software Engineering & Technology Department
Software Engineering Division

EDUCATION

PhD, Computer Science, UCLA
MS, Research Methods & Evaluation, UCLA
BS, Applied Physics, Cornell University

EXPERIENCE

9 years at Hughes
13 years total

HUGHES POSITIONS

Dr. Winchester directs an organization of 80 engineers and scientists tasked with conducting research, development projects in software and system specification, design, and development as ell as providing direct analysis support for engineering contracts. His department is responsible for the Hughes Software Development System; an integrated collection of computer aided tools to support the development of software for large C?3OI systems. His department also develops specialized real time operating system and display software. His particular research emphasis is in requirements specification languages and their relationship to the computer system development life cycle. Previously he was Head of the Research and Analysis Section, leading software research and development projects in software specification, design metrics, testing, and computer system simulation.

PREVIOUS ASSOCIATIONS

As 1Lt. U.S. Army, was Executive Officer of a Maintenance Management Detachment and responsible as project director for the development of the Coscom Automated Maintenance Management system (CAMMS). For 18 months directed 24 officers and men in the conceptualization, design and development of this data based management system.

PUBLICATIONS

Papers for five ACM & NCC Conferences, 2nd International Conference on Mathematical Modeling, the First Annual Symposium on Systems Development Technology and other professional symposia.

PROFESSIONAL ASSOCIATIONS & HONORS

Member of ACM, IEEE and Upsilon Pi Epsilon. Recipient of Howard Hughes Doctoral Fellowship, Army Commendation Medal For Meritorious Service.

REUSABLE SOFTWARE IMPLEMENTATION TECHNOLOGY REVIEWS

PRESENTED BY

J. WINCHESTER

HUGHES
AIRCRAFT COMPANY

SOFTWARE PROBLEMS AND OBJECTIVE OF THE REVIEWS

HUGHES

- PROBLEMS

  COST, RELIABILITY AND TIMELINESS OF SOFTWARE FOR NAVY EMBEDDED COMPUTER SYSTEMS

- OBJECTIVE

  TO EVALUATE AVAILABLE SOFTWARE DEVELOPMENT METHODOLOGIES WITH RESPECT TO REUSABLE SOFTWARE

REPORT ORGANIZATION

HUGHES

SECTION 1 - INTRODUCTION AND SUMMARY

SECTION 2 - CONCEPT OF REUSABLE SOFTWARE

SECTION 3 - SOFTWARE DEVELOPMENT METHODOLOGIES

SECTION 4 - SOFTWARE DEVELOPMENT METHODOLOGY REVIEWS

APPENDIX A - ADVANCED COMBAT DIRECTION SYSTEM

ANALYSIS APPLIED TO RSIP TECHNOLOGY REVIEWS

HUGHES

SELECTION OF 19 DEVELOPMENT METHODOLOGIES THROUGH RESEARCH, INTERVIEWS, ETC.

- METHODOLIGIES
  - PROGRAMMER'S APPRENTICE
  - SARA
  - SCR
  - PAISLEY
  - RNTDS
  - GYPSY
  - ETC.

ESTABLISH TECHNICAL AREAS OUTLINED IN STATEMENT OF WORK

- TECHNICAL AREAS
  - LANGUAGE TECHNOLOGY
  - STRUCTURING METHODOLOGY
  - DESIGN ENVIRONMENT
  - PERFORMANCE EVALUATION
  - MAINTENANCE
- NON-TECHNICAL AREA
  - MARKET FACTORS

ESTABLISH KEY ATTRIBUTES OF TECHNICAL AREAS

- STRUCTURING METHODOLOGY
  - PARADIGM
  - BREADTH OF PROBLEM DOMAIN
  - LIFE-CYCLE PHASE PRODUCT
  - SIZE OF PRODUCT
  - ETC.
- DESIGN ENVIRONMENT
  - MANAGEMENT SUPPORT TOOLS
  - TOOL INTEGRATION
  - ETC.

ESTABLISH POSSIBLE RANGE OF VALUES FOR ATTRIBUTES (WHERE APPROPRIATE)

- SIZE OF PRODUCT
  - SINGLE STATEMENT
  - PROCEDURE
  - SUBSYTEM
  - SYSTEM
- MANAGEMENT SUPPORT TOOLS
  - NONE CONSIDERED
  - MINIMAL REPORTS
  - MODERATE REPORTS
  - HIGH, FORMAL REPORTS
  - COMPLETE, AUTOMATED REPORTS

APPLY ESTABLISHED CRITERIA TO 19 METHODOLOGIES SELECTED

SUBJECT OF SOFTWARE DEVELOPMENT METHODOLOGY REVIEWS

HUGHES

- RESEARCH
  - PROGRAMMER'S APPRENTICE
  - HARVARD PDS
  - SARA
  - DRACO
  - PAISLEY
- COMMERCIAL
  - SCR
  - RNTDS
  - FDM
  - GYPSY
  - ISDOS
  - SREM
  - HOS
  - SADT
  - ADA-ORIENTED METHODOLOGIES
  - SMALL TALK
  - RAYTHEON
- INDUSTRIAL
  - HUGHES
  - GRUMMAN
  - BOEING

TECHNOLOGY AREAS ADDRESSED

HUGHES

- LANGUAGE TECHNOLOGY
- STRUCTURING METHODOLOGY
- DESIGN ENVIRONMENT
- PERFORMANCE EVALUATION
- MAINTENANCE
- MARKET FACTORS

CONCLUSIONS FROM REVIEWS

HUGHES

- NO METHODOLOGY PROVIDES CODE-LEVEL REUSE BETWEEN DISSIMILAR APPLICATION AREAS

- NO METHODOLOGY FOR LARGE-SCALE DEVELOPMENT PROVIDES A RELIABLE STORAGE AND RETRIEVAL MECHANISM FOR A CODE-LEVEL LIBRARY

- REUSE OF PERSONNEL IS PRIME MEANS OF REUSING SOFTWARE IN THE INDUSTRIAL SECTOR

CONCLUSIONS FROM REVIEWS

HUGHES

- LANGUAGE TECHNOLOGY, BY ITSELF, IS ONLY A COMPONENT OF THE SOFTWARE REUSE SOLUTION

- THE LARGER THE BREADTH OF COMPONENT REUSE, THE GREATER THE NEED FOR FORMAL SPECIFICATIONS

- THE LEVEL OF SPECIFICATION REQUIRED FOR REUSABLE COMPONENTS IS MORE FORMAL THAN IS CURRENTLY USED IN LARGE-SCALE DEVELOPMENTS

CONCLUSIONS FROM REVIEWS

HUGHES

- INFORMATION EXPLOSION COUNTER-BALANCES THE DESIRE FOR FORMAL SPECIFICATIONS

- SPECIFICATIONS LANGUAGE IS MORE IMPORTANT THAN THE IMPLEMENTATION LANGUAGE

- ONLY METHODOLOGIES SURVEYED USING FORMAL, SEMANTIC DESCRIPTIONS WERE GYPSY AND FDM

CONCLUSIONS FROM REVIEWS

HUGHES

- RESEARCH METHODOLOGIES CONCENTRATE ON SMALL WELL-DEFINED SYSTEMS
- INDUSTRIAL METHODOLOGIES ADDRESS LARGE-SCALE SYSTEMS THAT ARE INCOMPLETELY DEFINED
- A METHODOLOGY SHOULD BE DEFINED BEFORE IT IS IMPLEMENTED

A PROCEDURE OF CHOOSING A METHODOLOGY

HUGHES

1. DETERMINE SIZE OF SYSTEM YOU INTEND TO BUILD
2. DETERMINE BREADTH OF PROBLEM DOMAIN
3. DETERMINE ORGANIZATION DISTANCE FOR REUSE
4. CHOOSE LIFE-CYCLE PHASE FOR SOFTWARE REUSE
5. DETERMINE LEVEL OF SEMANTIC DESCRIPTION REQUIRED
6. CHOOSE METHODOLOGY COMPATIBLE WITH THE ABOVE

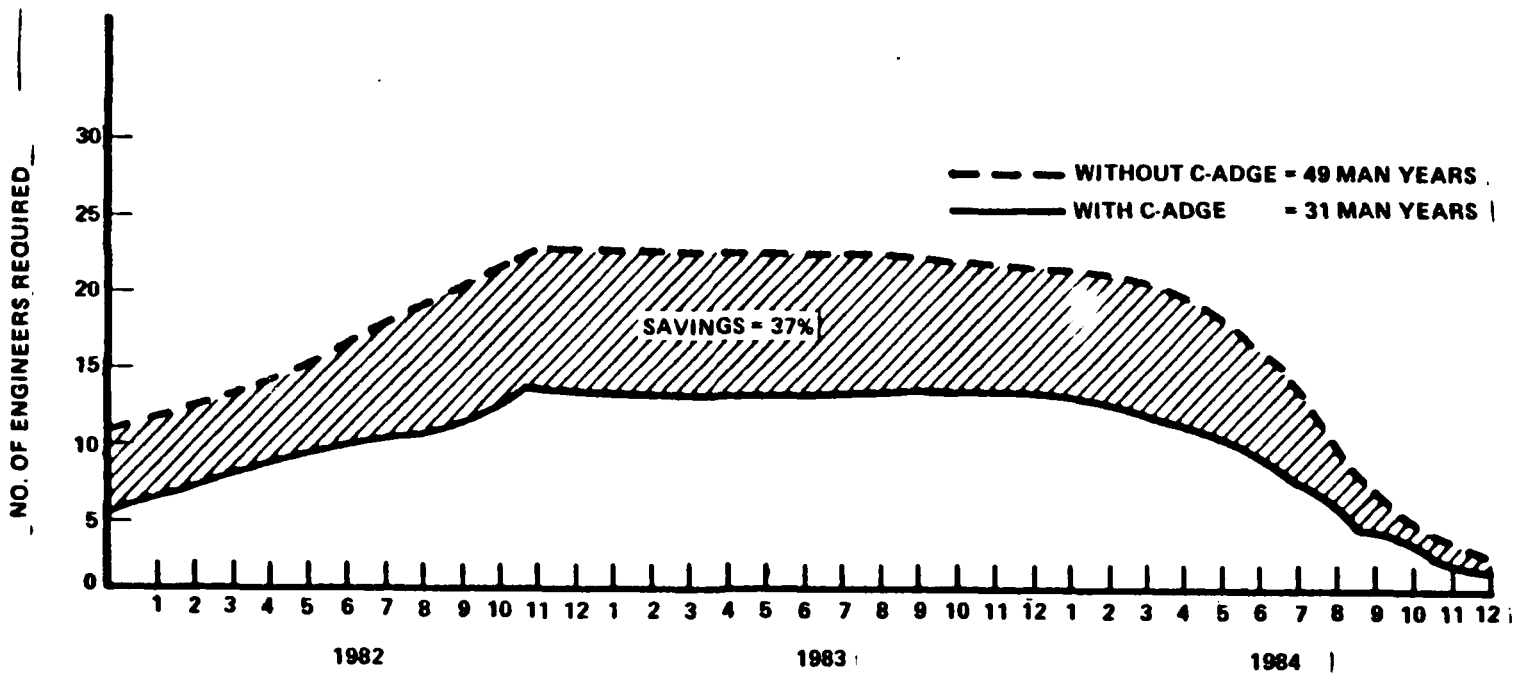
RECOMMENDATIONS

HUGHES

- PROVIDE STANDARD SOFTWARE DEVELOPMENT ENVIRONMENT
- REUSE SOFTWARE DEVELOPMENT ENVIRONMENT
- REUSE FIELD-SUPPORT SOFTWARE
- PROVIDE AUTOMATED SUPPORT FOR DOCUMENTATION
- REUSE REQUIREMENTS, SPECIFICATION, AND DESIGNS
- DESIGN FOR REUSABILITY
- RESTRICT APPLICATION AREA

PRODUCTIVITY IMPROVEMENT USING CADGE SOFTWARE

HUGHES



PRODUCTIVITY IMPROVEMENT IS MEASURED BY COMPARING THE NUMBER OF ENGINEERS REQUIRED IF EACH PROJECT USING C-ADGE PACKAGES WERE TO BUILD THAT SOFTWARE THEMSELVES VERSUS THE NUMBER OF C-ADGE ENGINEERS PROVIDING THE PACKAGES.

WORKSHOP PANELS

I. PART TAXONOMY/REQUIREMENTS

II. INCENTIVES

III. LIBRARY

IV. SYSTEM DESIGN/INTEGRATION WITH RESUABLE PARTS

V. METRICS

GROUP 1

PARTS TAXONOMY AND REQUIREMENTS

aka "PARTS IS PARTS!"

Bets Wald, NRL
Bob Kolacki, NAVELEX
Bob Fritz, CSC
Kaye Grau, Harris
George Mebus, RCA
Lorraine Griffin, Ford Aerospace
Glenn Murray, CDC
Alan Blair, General Dynamics
Steve Huseth, Honeywell
Miguel Carrio, Teledyne Brown
Barrie Baston, MITRE
Mike Glasgow, IBM
Geoff Mendal, Lockheed

ISSUES

1. Define forms and provide rationale for selection:

    a. collection of source code, design, report, to perform a function

        resolution: definition and rationale are feasible
        and acceptable for code level, less so for design,
        and difficult and uncomfortable for specification level

    b. individual items such as source code or document not addressed, but feasible

2. What is to be included in each definition and what information is needed for each of these items to reuse software at each level of the definition.

Levels		
Environment		
Conceptual	*Specification*	PPS
	Design	
Logical	Algorithms	FDS
Physical	Implementations	PDS

3. SDS Issues

- DIDS applicable to reuse at high flow levels
- DIDS modifications or alternatives
- information not in DIDS

    not fully examined; needs further review

REUSABLE SOFTWARE

Software reusability is the reuse of any information necessary to the development of software systems.

Reusable software includes any software information that may usefully be collected and used later to develop other software.

There are a number of levels of reusability.

At each level, the lowest reusable component is the PART.

With each part are necessary attributes that provide the characteristics and other information needed for reuse.

Attributes differ for the levels of reuse.

Three reuse compositions are recognized:

intact - the part is used without change
parametric - the part is altered through parameters
tailored - the part is modified by means other than parameters

REUSABLE PART ATTRIBUTES

General: Name of Part/Version
- Source/Date
- Description: Intended Use/Not intended use
- Warranty
- Liability
- Royalty
- Compositional Properties
  - dependencies
  - usage options, intact/parametric/tailored
  - domain identification
- Run time
  - target computer/os(exec)
  - implementation media

Requirements: Traceability information
Narrative Description: Including design rationale
Formal Description: PDL, formal semantics
Interface Specification: Type, mode, range, precision
Timing requirements
What and how
Quality, QA, Verification, Testing
Test reports, data, plans, tools, environment
Performance characteristics: actual target performance
Repair record: Version record, history of part
Notes: Comments, suggestions, hints
Existing implementations
THE ACTUAL PART

ADA REUSABILITY ISSUES

Does Ada focus harm acquisition of other language parts?

- It is mandatory that other language parts be accommodated.
- Ada does not preclude mixed language part composition.
- There may be run-time incompatibilities or problems mapping Ada to a ndw/sw configuration.
- Restricting languages restricts domains.
- Ada tasks/task model not universally reusable.
- Ada Reusability Style Guide needed.
  - prevent misuse/misunderstanding of Ada features
- Ada generics are a weak reuse aid.
- Fixed point types have tricky reuse problems.
- Chapter 13 features, programs, attributes may not be reusable.

= Useful enhancements for Ada

- package types
- procedure parameters ... priority tasking

Ada was designed for portability, reusability is serendipitous.

What means are there to communicate Ada problems and eliminate rediscovery?

Ada board, Ada-info, Ada implementors, SISAda/Ada letters

OTHER ISSUES

Configuration Management
    How many revisions?

Validation of parts
    How?
    How often?
    Who?

Development of knowledge-based systems
- Development of application rules
- Development of composition rules
- Development of generation rules

Definition and understanding of domain

Additional life cycle steps needed by reuse:
- in reports/design
- after completion - identification and insertion
    of result as reusable part
- other life cycle impacts
- alternate life cycles

Hierarchy of composition
  part - component - package?

Other views of level

Reusability support tools and aids
- conventional technology
- AI - knowledge based

PARTS IS PARTS!

ALL THE PIECES PARTS
IS ASSEMBLED INTO ONE BIG PART

FUSED...

THEN THE ONE GREAT BIG PART
IS CUT UP INTO LITTLE PIECES PARTS,

AND

PARTS IS PARTS!

STARS Workshop on Reusable Components of Application Software
PARTS TAXONOMY and REQUIREMENTS WORKING GROUP
SUMMARY OF SESSIONS, 9-11 APRIL 1985

Participants

Bets Wald, NRL, Government Chair
Robert Kolacki, Government Chair
Bob Fritz, Computer Sciences Corporation, Industry Chair
Kaye Grau, Harris Corporation, Session Recorder
George Mebus, RCA Advanced Labs
Lorraine Griffin, Ford Aerospace
Glenn Murray, Control Data Corporation
George Mendal, Lockheed Missiles and Space
Alan Blair, General Dynamics
Steve Huseth, Honeywell
Miguel Carrio, Teledyne Brown Engineering
Barrie Baston, MITRE
Mike Glasgow, IBM FSD

Charter

The Parts Taxonomy and Requirements (PTR) group was chartered to derive common terminology, to derive specifications for access and composition, and to address a number of related specific issues. These issues included:

a. Define terms and provide rationale for selection at the collection and individual item level.

b. What is to be included in each definition and what information is needed to reuse software at each level of definition?

c. How well can the SDS DIDs be used to guide reusability? How may they be modified or appended?

d. What effect does Ada have on reuse? Does Ada focus harm reuse of parts in other languages?

e. What are the most effective ways to communicate problems and fixes uncovered in Ada?

Issue - Definitions of reusable entities

Discussion - The PTR working group's discussion of definitions of reuse took most of the allotted working time. Even with this effort, a definition was only partly completed.

The PTR group defines software reusability as the reuse of any information necessary to the development of software systems. Reusable software includes any software information that may usefully be collected and used later to develop other software.

It was recognized that a number of different levels of reuse exist, and that there are a number of different compositions of reusable components.

number of attributes that provide the characteristics and other information for reuse. Attributes differ for the levels of reuse.

The levels of reuse are abstractions of a solution to a problem. The least abstract level is the physical level which is an implementation. The Program Design Specification is currently used to describe implementations. The next level of abstraction appears to be a range of abstraction and was designated the logical level. The logical level ranges from "designs" to "algorithms". The Functional Design Specification describes this level of abstraction. At a higher level is the conceptual abstraction, a range of abstraction that includes "specification" and "design". The Program Performance Specification is a document developed at this level. Though it is recognized that there is a higher level of abstraction that the working group called the "environment", what it is or how it may be used is uncertain and requires further investigation. The levels of reuse are summarized below. The boundary between conceptual and logical is fuzzy.

LEVEL	ABSTRACTION	DOCUMENT
Environment	?	?
Conceptual	Specification Design	Pgm Performance Spec
Logical	Algorithms	Funct Design Spec
Physical	Implementations	Pgm Design Spec

In discussion, it was determined that understanding of reusability at the physical level is best and development of methods to support reuse at this level will provide immediate benefit. As the level of abstraction increased the ability of the group to see how reuse at these levels could be accomplished decreased. Higher levels of abstraction are the areas of software and system engineering that are currently guided by experience and intuition rather than well defined sets of rules, making tool support other than as documentation aids difficult.

At each level, the reusable parts may be incorporated into a new system in a number of ways, called compositions by the working group. The part may be used intact, without change. Functions in a math library are examples of parts used intact. Parts may be modified parametricly, as in Ada generics. Tailored parts are modified by means other than parameters. Tailoring generally requires elaborate tools and metacommands that alter the part.

Resolution - Definition of reusable entities is easible but complex due to the need to describe reuse at a number of levels of abstraction. Reuse at low levels of abstraction is understood and provides immediate benefit, but methods to reuse parts at higher levels of abstraction must be developed.

Issue - Information needed in definition of reusable parts

Discussion - Attributes of reusable parts include information not now normally recorded. The PTR working group was able to draft the attributes for a part at the lowest level of abstraction. Some of this information should be included for parts at all levels of abstraction, but at higher levels other attributes are probably needed. The medium of description for the various attributes is intuitive (though not necessarily optimal) for the

abstraction, but at higher levels other attributes are probably needed. The medium of description for the various attributes is intuitive (though not necessarily optimal) for the attributes of physical level parts. The description medium for the currently undefined attributes at higher levels of abstraction needs to be defined. The reusable part attributes for the physical level component are described below.

REUSABLE PART

General Information: Name of Part
Version
Date
Source of Part
Description: Intended use/not intended uses
Warranty
Liability
Royalty

Composition Properties
Dependencies
Usage Options: Intact, Parametric, Tailored
Domain Identification

Run Time
Target hardware and operating system (or exec)
Configuration
Implementation media: Sw, Hdw, Firmware or combo

Requirements:
Original performance and function requirements
Traceability information

Host requirements: Development dependencies
Tools required

Formal Description: PDL, formal semantic description

Narrative Description: English prose, including design rationale

Interface Specification: Parameter type, mode, range, precision
Timing requirements
What and how

Quality: QA, Verification, Testing, Test Requirements,
Test Data, Test Plans, Test Tools,
Test Environment

Performance characteristics: Actual target performance, benchmarks

Repair Record: Version record- history of past
Problem tendencies

Notes: Comments, suggestions, hints

Existing implementations: This target, other targets

THE
ACTUAL
PART

This is a draft and additional data may be required by retrieval systems or code modification tools. The attributes are in many cases partially redundant. The redundancy may be reduced after more careful analysis.

Resolution - Describing the attributes of reusable parts is essential to the user oif reusable parts, to the parts developer, and to the parts management system. ASttributes of physical level parts can be enumerated, and may serve as the basis for attributes of higher levels of abstraction. Defining the reuse attributes for higher levels of abstraction is essential and requires additional research.

Issue - Ada and reusability

Discussion - The Ada and reusability issue has a number of facets, including reuse of components created in other languages, communicating problems and fixes, and features of Ada that promote or make difficult reuse of parts.

Ada does not preclude reuse of software parts implemented in other languages at the physical level. Ada was designed to interface with other languages, though by some views this is a limited interface. A severe problem is that the pragma INTERFACE is not a mandatory part of the language that is tested by the Ada Compiler Validation Capability and thus is not implemented in many compilers. It is mandatory that this pragma be implemented to provide this level of reusability. Additional pragmas may be necessary to make Ada compatible with various run-time configurations and constraints.

It is mandatory that parts in other languages be accommodated. Restricting reuse to Ada language parts restricts the domain of applications and the potential for reuse.

Some Ada features may limit reusability. Ada generics are a weak reuse aid, and do not supply the generality the name implies. The Ada task model is not a universal one for concurrent systems. Use of representation specifications, pragmas, attributes, or other features defined in Chapter 13 may limit reusability. Ada fixed point types have very tricky reuse problems based on the difference between the defined delta and the actual delta generated by the compiler for a particular implementation. The generation of a "Guide for Reusability with Ada" is needed to prevent the misunderstanding or misuse of Ada features that can be used to promote reusability.

Ada was designed for portability. Portability is a facet of reuse (an intact or parametric reuse of a physical component). Other levels of reusability based on Ada are serendipitous. Enhancements that would increase the domain of Ada solutions to problems and hence reusability include package types and passing of procedures as parameters. Package types would allow better description of abstract data types and allow Ada packages to be used at a higher level of abstraction. Procedures as parameters would allow tasks with dynamic priorities.

Resolution - Ada focus cannot preclude reuse of software components implemented in other languages. While some Ada features support reuse, a "Guide for Ada Reusability" should be developed. Future versions of Ada should be changed to support reusability by adding package types and procedures as parameters.

Issue - Do the SDS DIDs support description of reusable parts? What could be changed or added for better description?

Discussion - The SDS DIDs do not prevent use or description of reusable parts. They were not particularly designed for describing reusable components, and there are probably supplementary information that would aid reuse description. This issue was not thoroughly discussed by the PTR working group.

Resolution - The SDS DIDs need to be examined in more detail in the next working group session. Alternatives should be examined.

Other Issues - The PTR working group identified a number of related issues during its discussions. The issues were only described and require careful examination by PTR or other more appropriate working group.

Configuration Management: How many versions should be kept in a library? Suppose a number of users have successfully reused a part that is later shown to have a bug, is the previous version invalid? Should the early version be kept? What if the new "corrected" version does not work in the generated software?

Validation & Evaluation: Should parts be validated or evaluated or is it buyer beware? How should validation or evaluation be done? How often should a part be validated and evaluated; periodically or only when a new version is developed? How should version be defined? Who is responsible for validation or evaluation, the developer or an independent agency?

Knowledge Based Systems Support. Tools using this technology to support reuse may be a decade or more away. Three distinct sets of rules need to be developed; application rules, composition rules, and generation rules. Application rules guide the production of software for a particular application within a domain. Composition rules define the ways that reusable parts may be combined and changed. Generation rules guide the automated generation of applications software from higher level reusable components.

Domain: A more complete definition and understanding of domain is needed. Domain affects the selection and composition of reusable parts but the degree and bounds of the domains effects are not shown.

Life Cycle: Additional life cycle phases or activities are to be required to make reusability part of the software engineering process. At the beginning of the lifecycle, in the requirements, specifications, and design phases, there is a need to search for, acquire and integrate reusable components. The process for doing this is dependent on a number of outside factors such as government acquisition policy and library structure.

An additional reuse phase is needed after integration or delivery of a software product to identify it or a subset of it as a reusable component. One software product may produce one or many reusable parts at many levels. This identification of reusable components will have an effect on many of the phases of the current life cycle.

There may be a number of as yet undetermined effects on various phases of the life cycle. The current life cycle should be examined phase by phase to consider these effects.

Alternate life cycle models may be possible or necessary due to reuse. This should be considered as well.

Hierarchy of Composition. The part was arbitrarily identified by the PTR as the lowest reusable part, the software atom. It is apparent that there is a composition of parts to produce other software entities, and some of these entities stand alone for reuse. A hierarchy of composition should be established to describe these combinations.

Reusability support tools and aids. Reusability will be futile if there are no automated support tools and aids. Development of these tools and aids should be done on two parallel paths. Conventional technology, such as database management systems, will provide the most near and mid term support for selection, acquisition and insertion of reusable parts in a database. Simple tools, such as command procedures, can be used to incorporate reusable components into systems. Editors provide immediate capability for modification of source code parts. Initial reuse should focus on learning how to reuse parts at the level of understanding we now There are some immediate applications that show promise, but this technology will not provide the needed support for at least a decade, probably longer. The turn of the century will probably see the first AI tools in widespread use for software engineering. (AI technologies will probably be applied earlier to a great many simpler, better understood problem domains before they are useful in software engineering.) These tools should be viewed as long term developments. These technologies will probably be essential to the reuse of parts at higher levels of abstraction. At the current level of technology, AI/knowledge based tools and aids are high risk and offer little immediate payoff.

GROUP II
INCENTIVES

Marlowe Henne, Harris
Steve Strong, Naval Avionics Center
Thomas Arkwright, Lockheed
Joyce Mortison, Sperry
Norm Nise, Rockwell
Fredric Heilbronner, Advanced Technology
Dan Haggerty, Boeing
Rodney Bond, General Dynamics

INCENTIVES

ISSUE:

Proposal Evaluation Process

DISCUSSION:

Process currently disincentivizes reuse.

Lack of cost realism may be assumed.

Up-front costs may be disallowed.

RECOMMENDATIONS:

Include reusability
as part of evaluation criteria.

Properly evaluate as part of cost realism.

RFP to study the above.

INCENTIVES

ISSUE:

Should all RSW be incentivized the same?

DISCUSSION:

Reusability may not be apropos at times.
Determine up front in contract.
Three services do not agree
on criteria priority.

RECOMMENDATIONS:

Government/Industry up-front study.
RFP to study criteria
and establish priorities.

INCENTIVES

ISSUE:

Effects of reuse on competition.

DISCUSSION:

Mandatory reuse may encourage obsolescence.
Contractor business posture may suffer.
RSW library may increase competition.

RECOMMENDATION:

RFP for detailed analysis of trade-offs.

INCENTIVES

ISSUE:
The Government's Posture.

DISCUSSION:
Positive posture needed.

RECOMMENDATIONS:
Strong DoD pronouncements.
Orientation of DoD program personnel.

INCENTIVES

ISSUE:

Economic Incentives.

DISCUSSION:

Two Areas:

(1) Producing;

(2) Consuming.

Two methods to incentivize deposits:

(1) Dollars for producing.

(2) Dollars for consuming.

Will encourage high degree of reusability.

RECOMMENDATIONS:

Payment made when RSW part reused.

Incentives for tool use may differ.

Fund a study on levels required.

Study transfer of maintenance responsibility to the library.

INCENTIVES

ISSUE:

Contract types and clauses.

DISCUSSION:

FFP contracts encourage reuse.

CPFF (more common) tend to discourage reuse.

Savings-sharing encourages reuse if scope renegotiations aren't invoked.

Contract clauses needed for incentives (and metrics).

RECOMMENDATIONS:

Develop appropriate clauses.

Modify DIDS (minimize documentation)

Encourage electronic media transfer (etc.).

INCENTIVES

ISSUE:

Warranties and Liabilities.

DISCUSSION:

Liability opposes depositing and consuming.
Some limit to liability appears wise.

RECOMMENDATIONS:

No unwarranted liability for depositors.
Problems from reuse are users' responsibility.
Develop contract clauses to protect
paid developers voluntary depositors.

REUSE INCENTIVES REPORT

(MINORITY VIEW)

TOM ARKWRIGHT
LMSC
SUNNYVALE, CALIFORNIA

COMPETITION-BASED MODEL OF REUSE INCENTIVES

- ASSUMPTIONS
- CHANGES NEEDED
  - GOVERNMENT CHANGES
  - CONTRACTOR CHANGES
- RECOMMENDATIONS

ASSUMPTIONS

- A PRIVATE OWNERSHIP MODEL CAN CUT THE GOVERNMENT'S COST OF ENTRY TO NEAR ZERO, AND INCREASE ITS RATE OF SAVINGS
- COMPETITION SIMPLIFIES THE ADMINISTRATIVE MACHINERY THAT THE GOVERNMENT WOULD NEED TO OFFER TO INDUSTRY
- LIABILITY, WARRANTY, MAINTENANCE, PRICING, SECURITY, ACCESSIBILITY AND PORTABILITY ISSUES ARE MORE TRACTABLE WHEN OWNERSHIP IS NOT WITH THE GOVERNMENT
- THE GOVERNMENT OWNERSHIP MODEL IS A COUNTERINCENTIVE IN AND OF ITSELF
- THE INCENTIVE TO REUSE WILL BE THE DESIRE TO SUBMIT A WINNING BID
- NO INCENTIVE IS NEEDED TO STIMULATE DEVELOPMENT OF REUSABLE PARTS; THE NEED TO COMPETE IS ADEQUATE INCENTIVE
- REUSABLE SOFTWARE SHOULD NORMALLY BE A BY-PRODUCT OF A CONTRACT, RATHER THAN PRODUCED ON SPECULATION ABOUT A PUTATIVE NEED

ASSUMPTIONS (CONTD)

- THE EVOLUTION OF REUSE APPROACHES AND THEIR DIVERSITY WILL BE STIFLED BY GOVERNMENT OWNERSHIP (NOTE THE HISTORICAL FAILURE OF GOVERNMENT TO EXPLOIT ITS OWNERSHIP OF TECHNOLOGY, PATENTS, ETC.)
- THE COST OF FINDING, TESTING AND INTEGRATING REUSABLE WILL BE DRIVEN DOWN MORE QUICKLY IF OWNERSHIP AND COMPETITIVE EDGES ARE RETAINED BY CONTRACTORS
- THE COMPETITIVE MODEL REDUCES TEMPTATION TO GIVE THE GOVERNMENT USELESS REUSABLES, TO BEAT THE SYSTEM, ETC., BY ELIMINATING THE NEED FOR FORCED DEPOSITS AND WITHDRAWALS
- UNDER THE GOVERNMENT OWNERSHIP MODEL, THE NEED TO FILTER INPUTS CAN NOT BE EFFECTIVE ENOUGH TO PREVENT UNUSABLE INSERTIONS
- THE CHOICE OF AN INCENTIVE MODEL (COMPETITIVE OR GOVERNMENT) WILL IMPINGE ON AMERICA'S VIABILITY AS A PRODUCER OF DEFENSE SOFTWARE

ASSUMPTIONS (CONTD)

- NOT EVERY COMPANY WILL HAVE THE FORESIGHT TO STAY COMPETITIVE BY EVOLVING ITS REUSE POSTURE
- COMPENSATION TO CONTRACTORS (ZERO OR MORE DOLLARS) FOR SAVINGS GENERATED FROM REUSED SOFTWARE WILL BE A NEGOTIABLE MARKET-DRIVEN ITEM ON EVERY CONTRACT
- THE GOVERNMENT WILL BENEFIT MOST WHEN IT GIVES INCENTIVES FOR ACHIEVING REUSE LEVELS BEYOND THOSE BID (CONTRACTORS WILL SEEK OPPORTUNITIES AND BUY FROM PREVIOUSLY UNKNOWN SOURCES)
- BY NEGOTIATING REUSE COMPENSATION, SOFTWARE DEVELOPERS WILL BECOME INVOLVED (ALONG WITH HARDWARE DEVELOPERS) EARLIER IN SYSTEM DEVELOPMENT (INSTEAD OF AFTER SRR)
- AN ADEQUATELY DESIGNED PART IS REUSABLE; A SUPERBLY DESIGNED PART IS A DESIRABLE PRODUCT; A NONREUSABLE PART IS POORLY DESIGNED OR IMPLEMENTED; THUS, REUSE DOES NOT COST ALWAYS AND NECESSARILY EXTRA. REUSABLE PARTS MAY COST LESS TO TEST

CHANGES NEEDED-GOVERNMENT

- PRE-AWARD ASSESSMENTS OF CONTRACTORS' BIDS AGAINST EXISTING COST MODELS MUST BE ADJUSTED TO ACCOUNT FOR RESUSE LEVELS AND THE COSTS OF REUSE
- THE GOVERNMENT MUST SIGN A RELEASE ASSIGNING TO THE CONTRACTOR ALL REUSE RIGHTS. THE GOVERNMENT MUST RETAIN RIGHTS FOR PURPOSES OF MAINTENANCE
- THE GOVERNMENT MUST NOT CALL FOR SCOPE REDUCTIONS WHEN UNPLANNED REUSE OCCURS
- THE GOVERNMENT SHOULD SOLICIT INDUSTRY TO DEVELOP (AND ADMINISTER?) SEVERAL COMPETING SYSTEMS FOR CATALOGING REUSABLES TO BE AVAILABLE FOR SALE ON DEMAND (SECURE AS WELL AS OPEN), SO THAT FORMATS WILL NOT STAGNATE, SO THAT NON-CODE OBJECTS CAN BE SOLD, ETC.
- THE GOVERNMENT SOLICITS INDUSTRY TO DEVELOP COMPETING SYSTEMS FOR PHYSICALLY DISTRIBUTING THE SOFTWARE DESCRIBED IN CATALOGS
- GOVERNMENT PROPOSALS SHOULD DISCUSS THE NEED FOR REUSE VERSUS NEW TECHNOLOGY, AND ANY IMPACT OF REUSE ON PROPOSAL POINTS, SCHEDULE, ETC.
- RFPs SHOULD REQUIRE REUSE PLANS, SUBJECT TO DEFINED EVALUATION OR RATING CRITERIA
- THE GOVERNMENT SHOULD ACCEPT THE NEED TO REVISE "DIDs" BURDEN TO ACCOUNT FOR THE HIGHER DOCUMENTATION STANDARDS ASSOCIATED WITH REUSABLE PARTS
- THE GOVERNMENT SHOULD FOCUS ITS REUSE ATTENTION ON NEW ADA CONTRACTS

CHANGES NEEDED-CONTRACTOR

- THE CONTRACTOR'S BID SHOULD EXPLICITLY BREAK OUT THE IMPACT OF REUSE ESTIMATES, INCLUDING COST OF REUSE
- THE CONTRACTOR'S ADA DEVELOPMENT METHODOLOGY SHOULD PROMOTE ADEQUATE DESIGN (e.g., CARRY UNIT TESTS WITH THE UNIT) AND EARLY, CONTINUOUS FORMAL CONSIDERATION OF REUSE OPPORTUNITIES
- CONTRACTORS PAY FOR LISTING REUSABLE SOFTWARE IN THE VARIOUS CATALOGING SYSTEMS, AND HELP GET THEM STARTED WITH AN INITIAL ENTRY FEE

RECOMMENDATIONS

- RFP TO PROJECT THE EFFECT ON SOFTWARE COST OF BOTH MODELS (PRIVATE AND GOVERNMENT OWNERSHIP) AND TO EVALUATE EVENTUAL EXPOSURE TO SUPERIOR FOREIGN COMPETITION BY UNCONTROLLED SOURCING
- RFP TO PROJECT IMPLEMENTATION ADVANTAGES AND DISADVANTAGES OF BOTH MODELS (PRIVATE AND GOVERNMENT OWNERSHIP). EXPRESS THE DIFFERENCES IN TERMS OF COST TO THE GOVERNMENT PER REUSE TRANSACTION. PROJECT THE RELATIVE NUMBER AND VALUE OF TRANSACTIONS UNDER EACH MODEL. MODEL GOVERNMENT COSTS AND SAVINGS IN THE TWO CASES
- RFP TO REDUCE THE "DIDs" BURDEN TO REFLECT THE SUPERIOR DOCUMENTATION ASSOCIATED WITH REUSE
- RFP TO REVIEW, FORMALIZE, AND RECOMMEND ANY BEHAVIOR CHANGES NEEDED IN GOVERNMENT AND INDUSTRY TO SUPPORT ONE OR BOTH MODELS (PRIVATE AND GOVERNMENT OWNERSHIP)

GROUP III

LIBRARY

LIBRARY PANEL

Co-Chairpersons

Industry: Sue Mickel, General Electric
Government: Carl Ogden, Army

LIBRARY PANEL MEMBERS

Name	Representing
Steve Hopkins	Analytic Disciplines, Inc.
Francoise Yousseh	Institute for Defense Analyses
Ray Fryer	GTE Government System
Jon Squire	Westinghouse
Roger Smeaton	NOSC
Frank Friedman	CSC
Rick Bieniak	Ford Aerospace
Tom Leonard	Harris, Software Operation
Michael Broido	Intermetrics, Huntington Beach
John Litke	Grumman Data Systems
Rohn R. Mellby	Texas Instruments
Rick Jacques	U.S. Army Electronic Proving Ground
Sivey S. Hudson	Acquidneck Data Corp.
Ted F. Hobson	Computer Science Corporation

INTRODUCTION

The Library Panel addressed repository issues associated with reuse: how reusable parts are inserted into the library, how they are retrieved, and how they are maintained while in the library. These are extremely complex issues and in some cases the panel has recommended further research before a particular solution can be selected. The panel has, however, determined what functions the library should perfrom and, equally important, functions that the library should NOT perform.

It is critical to decouple the library from other functions, such as software maintenance, in order to keep costs reasonable and to focus attention on the basic capability the library must perform to effect reuse. The library must make it significantly easier and cheaper to locate and retrieve a part for reuse than it is to re-build that part from scratch.

LIBRARY MODEL

The panel developed a library model organized according to "levels of trust". It was recognized that from a library users' point of view perhaps the overriding concern is how well tested and documented or how reliable a part *is*. For example, parts currently being maintained within a nuclear or space application would be placed at the most trusted level. Handy, but undocumented, UNIX-like utilities would be placed at the least trusted level. There would be many intermediate levels. Depending of the application, the library user could restrict his search for parts to only the most trusted levels. The panel wishes to encourage reuse at all levels, with the expectation that through extensive reuse, parts at less trusted levels may migrate towards more trusted levels. Both the number of levels and the acceptance criteria to be applied at each level on areas that require further investigation.

Whether the library *itself* is physically distributed or centralized should be transparent to the user. Domain specific libraries are not recommended since there is significant overlap among domains. This would lead to confusion over which library a user should search. A single, centralized facility is, therefore, recommended.

A great deal of classified software and associated information is generated each year for the DoD. Reusability in this area is especially critical, not only to reduce costs but also to increase reliability. However, security procedures seriously hamper reusability by restricting distribution of information. The feasibility for multiple, classified libraries for reuse within and, to the extent possible, across classified projects must be investigated.

WHAT THE LIBRARY SHOULD DO

The library should be responsible for

- acceptance of new items
- cataloging
- retrieval
- configuration management
- on-line error reporting
- bulletin boards
- archiving

The library should apply acceptance.

Criteria to insert each item or part at the appropriate trust level, but it should not be responsible for screening out trojan horses or other types of potentially harmful parts. Certaintly, if a harmful part is reported to the library, the library should take appropriate action to notify users.

The panel strongly recommends that reusability not be limited to code. Designs, specifications, algorithms, test plans and procedures, etc. must be considered reusable as well. These items actually have the potential to effect larger reductions in cost than simple reuse of particular software parts. Because of this, both the mechanisms for cataloging and retrieving library items must be much more sophisticated than if software parts alone were to be considered.

Actual cataloging and retrieval mechanisms to be used are areas where extensive research is required. The panel recommends that techniques currently in use in other fields (medical, chemical, etc. not limited to software libraries) be examined and automated abstracting methods be investigated. Primary library access should be online, but other media must be available due to expected volume of information.

Online error reporting should be supported by the library as well as a bulletin board facility. Users should be able to query each of these facilities on the basis of individual parts of interest.

WHAT THE LIBRARY SHOULD NOT DO

Functions the panel felt the library should not be responsible for include:

- o screening submissions for trojan horses, etc.
- o software maintenance
- o proprietary software or information
- o subdividing contributed components for cataloging as lower level reusable parts

The panel recommends that the library be kept distinct from other functions. Thus, the maintenance of software of the library. This is not to preclude the government maintaining this software elsewhere, for example, at the life cycle software support centers.

The library should not accept or distribute proprietary items. Legal issues aside, the library should not become a marketing function for software vendors. This is not to preclude the library containing pointers to proprietary packages to facilitate reuse. All information actually contained in the library, however, must be publically accessib!

AREAS OF RESESARCH

The library panel recommends research in the following areas:

- o State of practice in library cataloging techniques
- o State of practice in techniques in maintaining software libraries
- o Estimation of library sizing parameters
- o Parts composition
  New technologies make feasible?
- o Submission criteria foe each library trust level
- o Establish standard measures of component performance, quality, etc.
- o Description methods for component dependencies versions, etc.

CHARTER
Library Panel

893

Address Repository Issues of Reusable Items

LIBRARY ISSUE III-A

What are the Submission Procedures for a New Item to Enter the Repository?

What is the Physical Mechanism?

LIBRARY ISSUE III-A (con't)

Submission:

Trust Level with Submission Procedures

Proprietary Software not Submitted

No Deletion from Library, Archive Everything

Research Area:

Submission Criteria for Each Trust Level

LIBRARY ISSUE III-B, Part I

How can the Library Deal with Classified Material?

Rules for Handling Classified Material Seriously Hinders Reuse Across Projects

Recommendation:

Declassify as Much Software as Possible

Availability of Classified Library Must be Made Known at Proposal Time

Investigate Distribution Methods for Classified Material

LIBRARY ISSUE III-B, Part 2

How can Integrity of Submittals be Maximized without Inhibiting Contributors?

Recommendations:

Library Submittals Should be Write-Only until Reviewed

Use "Confidence Levels", Based on Extent of Verification, to Quantify Degree to which Data Corruption, Trojan Horses and Time-Bombs have been Prevented

Rapid Notification of all Affected Users when these Problems are Detected

LIBRARY ISSUE III-C

What are the Necessary/Desirable Types of Information to Describe a Component (to a User of the Library)?

Descriptions

- All Information may not be Available
- Complete Information may be too Expensive

Component Trust

- High Trust - Complete Information
- Low Trust - Minimal Information

Component Classes

Source	Object Library
System	Executable
Package	Design
Subprogram	Algorithm
Information	Specification

Research - How to Describe

- Performance
- Quality
- Dependencies
- Variants on Original Component

LIBRARY ISSUE III - C (con't)

Summary of Description Classes

Identification
- Name — Summary
- Catalog Entry

Characteristics
- Language — Abstract
- Machine Dependencies — Performance
- Author — Side Effects
- Ordering Information (if not on library)

Code
- Source — Object
- Binary

Documentation
- Requirement — User Manual
- Design — Security
- Maintenance — Test: Plans, Proc., etc.

Usage
- Compilation Info. — Command Files
- Dependencies

History
- Revisions — Problems
- Usage — Current Users
- Notefiles (bulletin boards)

LIBRARY ISSUE III-D, III-J, III-K

How Can the Library Handle Proprietary Software?

DoD Should be a Match Maker to Promote the Reuse of Proprietary Software

DoD Should not Market Proprietary Software

Proprietary Software Should Not be Physically Contained in the Library

DoD Should More Agressively Persue Licensing & Rights

LIBRARY ISSUE III - E

Should the Library be Distributed or Centralized?

Recommendations:

Uniform Access Methods Regardless of Approach

Distributed Apoproach May be Dictated by Volume of Information

Approach should be Transparent to User

LIBRARY ISSUE III - F/G

What Soft of Cataloging Method Should be Adopted?

Factors Likely to Affect Choice of Cataloging are Search Mechanisms:

Sizing and Traffic Levels

Multi-Dimensional Nature of Parts Categories
- Applications Domain
- Type of Object
- ...

Domain Overlap

Need for Keyword/General Term Retrieval as well as Attribute Searches

Availability of Automated Techniques for Abstracting Parts Summaries, General Descriptive Terms and Keywords

Revisions and Variations

RESEARCH AREAS:
- Existing Software Libraries (COSMIC)
- Automated Library and Abstraction Facilities (Chem.,Legal,Med.)
- Other Cataloging Schemes (ACM, CR)
- Investigate Schemes Beyond Keyword Retrieval

LIBRARY ISSUE III - H

How Should the Library Facilitate Access and Distribute Holdings?

Primary On-Line, Other Media for Large Items

Make Library Search Mechanisms Available

Distribute Instructions for Search and Retrieval

On-Line Bulletin Board; Query by Library Entry

LIBRARY ISSUE III - I

How Should the Library be Maintained?

The Library Should Perform:

Configuration Management
Cataloging
Archiving
On-Line Error Reporting and Bulletin Boards

The Library Should Not:

Maintain Products

LIBRARY ISSUE III - L

What are the Legal Ramifications of a DoD Library?

What if Someone Puts Propriatary Software (e.g. LOTUS 1-2-3) into the DoD Library. Someone Takes it Out and Uses It? Who Has What Legal Responsibility?

What are the Legal Limits of DoD Competing with Private Industry by Providing a Software Library?

Is DoD Obligated to Maintain Software that a Contractor Extracts from the Library for Use in a DoD System?

LIBRARY ISSUE III - M

How Should the Library Handle a Submission Containing Subordinate Reusable Components?

Government Should Encourage Contributions at Lower Level of Granularity

Library Should not Break up Contributed Components

RESEARCH ISSUE:

Management of Components within Components, Versions Dependencies, etc.

LIBRARY ISSUE III - N

Should DoD Mandate Use of Existing Components from the Library?

Should not Mandate but Encourage

Include Reuse as an Issue in Life-Cycle Review Process

Permit Creation of New Versions when Justified

Allow the Economic, Schedule & Risk Factors
to Drive the Decision
(incentives vs. mandates)

RESEARCH AREAS

State of Practice in Library Cataloging Techniques

State of Practice in Techniques in Maintaining Software Libraries

Estimation of Library Sizing Parameters

Parts Composition
    New Technologies Make Feasible?

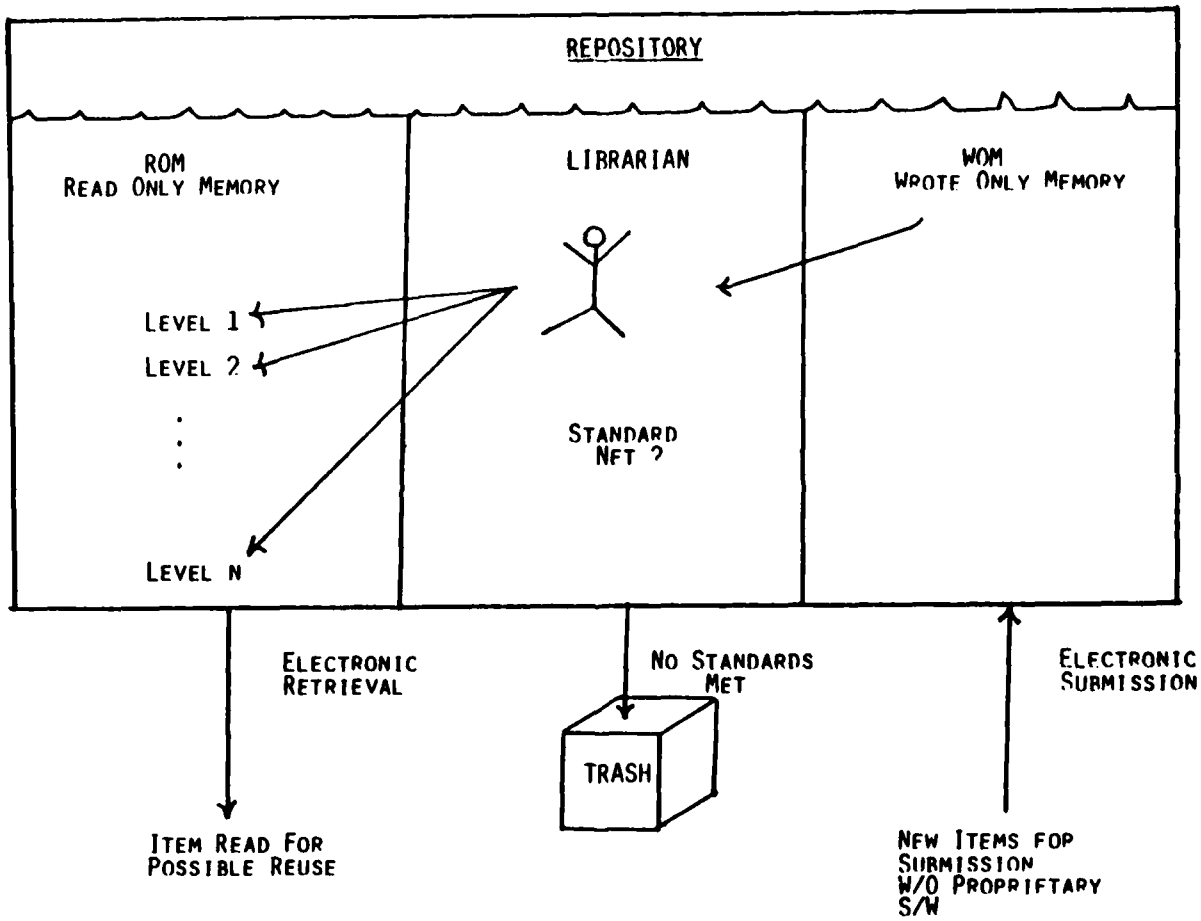
Submission Criteria for Each Library Trust Level

Establish Standard Measures of Component Performance, Quality, etc.

Description Methods for Component Dependencies, Versions, etc.

RELATED ISSUES

Funding, Fee Structures
Restriction of Access
Legal Issues


REPOSITORY
ROM
READ ONLY MEMORY
LIBRARIAN
WOM
WROTE ONLY MEMORY
LEVEL 1
LEVEL 2
LEVEL N
STANDARD
NET ?
ELECTRONIC
RETRIEVAL
NO STANDARDS
MET
ELECTRONIC
SUBMISSION
TRASH
ITEM READ FOR
POSSIBLE REUSE
NEW ITEMS FOR
SUBMISSION
W/O PROPRIETARY
S/W

GROUP IV

SYSTEM DESIGN/INTEGRATION

STARS APPLICATIONS WORKSHOP

DESIGN / INTEGRATION PANEL

9-12 April 1985

Co-Chairpersons

Industry: Elaine Frankowski, Honeywell
Government: Chris Anderson, Air Force Armament Lab

PANEL CHARTER

Develop Candidate STARS Projects Related to the Design/Development of Reusable Mission-Critical Parts and Their Integration into Mission-Critical Systems

DESIGN/INTEGRATION PANEL MEMBERS

Name	Representing
J.G. Snodgrass	E-Systems
William Wong	National Bureau of Standards
John Da Graca	General Dynamics
Jim Winchester	Hughes Aircraft
Daniel G. McNicholl	McDonnell Douglas
Ron McCain	IBM
Ray Dion	Raytheon
Nancy Ys. Kim	Rockwell International
C. K. Pian	Hughes Aircraft
Gregg Van Volkenburgh	Allied Canada, Inc.
Paul Hixson	General Dynamics
E. J. Startzman	Boeing
J. Roder	GTE
Christine Youngblut	Advanced Software Methods
M. Kemer Thomas	General Dynamics
Vicky Mosely	Westinghouse

DESIGN/INTEGRATION PANEL DISCUSSIONS

ASSUMPTIONS

ISSUES

PROJECTS

ASSUMPTIONS

Ada

Two Design Problems

Delivered Parts have Support

Sociological Problems of Reuse

Industrty-Wide Sharing is Far-Term

ISSUES / PROJECT MOTIVATION

Domain Analysis

Parts Development Methodology and
Parts Usage Methodology

Automated Parts Composition Systems

Library(ies) Location Affect(s) Parts Designs

ISSUES / PROJECT MOTIVATION

Lack of Confidence in Reusable Parts

Characteristics of Reusable Software Parts

Techniques to Produce Reusable Parts

Tool Support for the Reusability Techniques

STUDY TOPICS

DoD-STD-2167 (SDS) / DIDs

Structural Standards for Integrable Parts

Mechanisms for Interfacing Components

System Level Decisions / Software Reuse

Very High Level Languages

CANDIDATE PROJECT SUMMARY

1. Parts Methodology

2A. Horizontal Domain Identification
2B. Horizontal Domain Parts Development

3A. Vertical Domain Identification
3B. Vertical Domain Parts Development

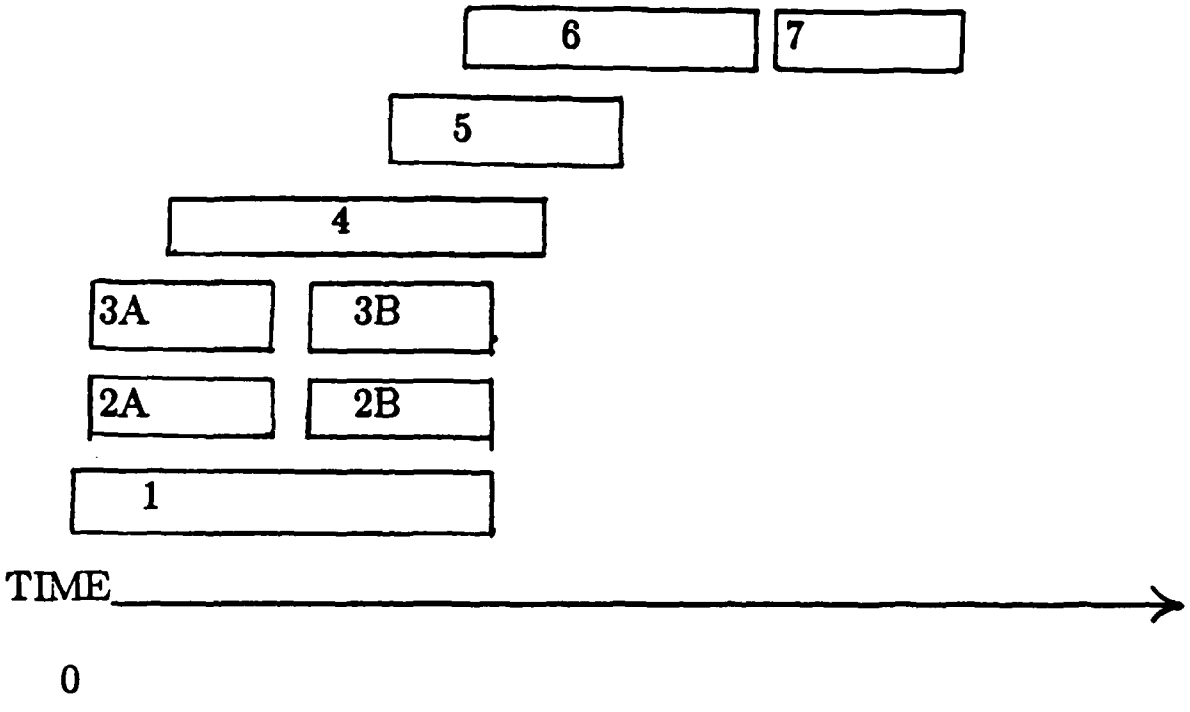
4. Software CAD/CAM

5. Software Reusability Demonstrations

6. Parts Certification

7. Parts Technology Showcase

PROJECT SCHEDULE



PARTS METHODOLOGY STUDY*
(PROJECT I)

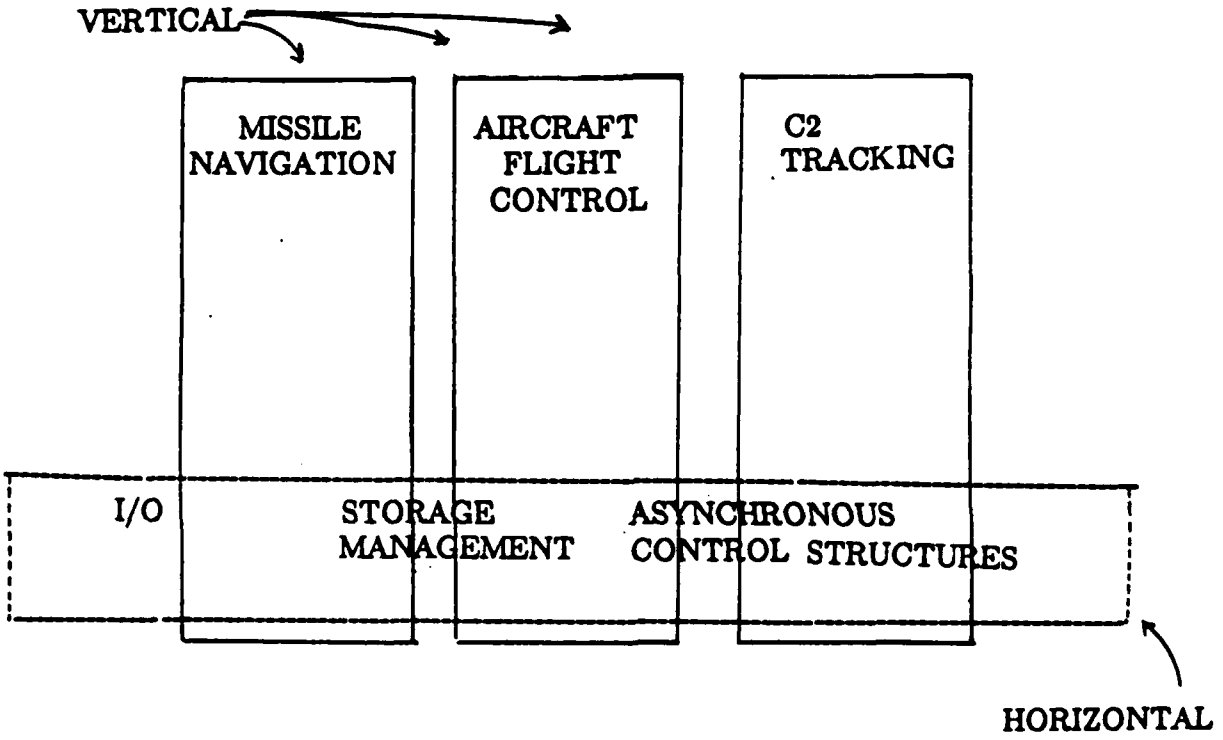
OBJECTIVE

To Identify a Compatible Set of Parts Reusability Methods Spanning the Domain Analysis, Parts Development and Usage

---

* To be Sponsored by STARS/Methodology

DOMAINS


VERTICAL
MISSILE NAVIGATION
AIRCRAFT FLIGHT CONTROL
C2 TRACKING
I/O
STORAGE MANAGEMENT
ASYNCHRONOUS CONTROL STRUCTURES
HORIZONTAL

HORIZONTAL DOMAIN IDENTIFICATION

OBJECTIVE

To Identify Domains within DoD Mission - Critical Applications which Span Multiple Application Areas (Horizontal Domains), in which Reusable Software can be Developed

HORIZONTAL DOMAIN PARTS DEVELOPMENT (PROJECT 2B)

OBJECTIVE

To Identify, Specify, Construct and Test Parts from the Horizontal Domains

Award Multiple Contracts to Demonstrate Various Parts Development Methodologies

VERTICAL DOMAIN IDENTIFICATION (PROJECT 3A)

OBJECTIVE

To Identify DoD Mission - Critical
Application Families (Vertical Domains)
in which there Exists
a High Degree of Reusability,
a Significant Life Cycle Cost Savings,
and an Opportunity
for Critical Expertise Propagation

VERTICAL DOMAIN PARTS DEVELOPMENT (PROJECT 3B)

OBJECTIVE

To Identify, Specify, Construct and Test Parts within Vertical Domains

Award Multiple Contracts to Demonstrate Various Parts Development Methodologies

SOFTWARE CAD / CAM (PROJECT 4)

OBJECTIVE

Investigate, Develop / Demonstrate Software CAD / CAM Technology as it Relates to Software Parts Development

Explore Role of Expert Systems ...

SOFTWARE REUSABILITY DEMONSTRATIONS (PROJECT 5)

OBJECTIVE

To Develop a Mission - Critical System (Sub-System) from an Identified Vertical Domain using Parts from the Horizontal and Vertical Domains.

Award Multiple Contracts to Demonstrate Parts Usage Methodologies Identified by the STARS/ Methodology Project.

PARTS CERTIFICATION (PROJECT 6)

OBJECTIVE

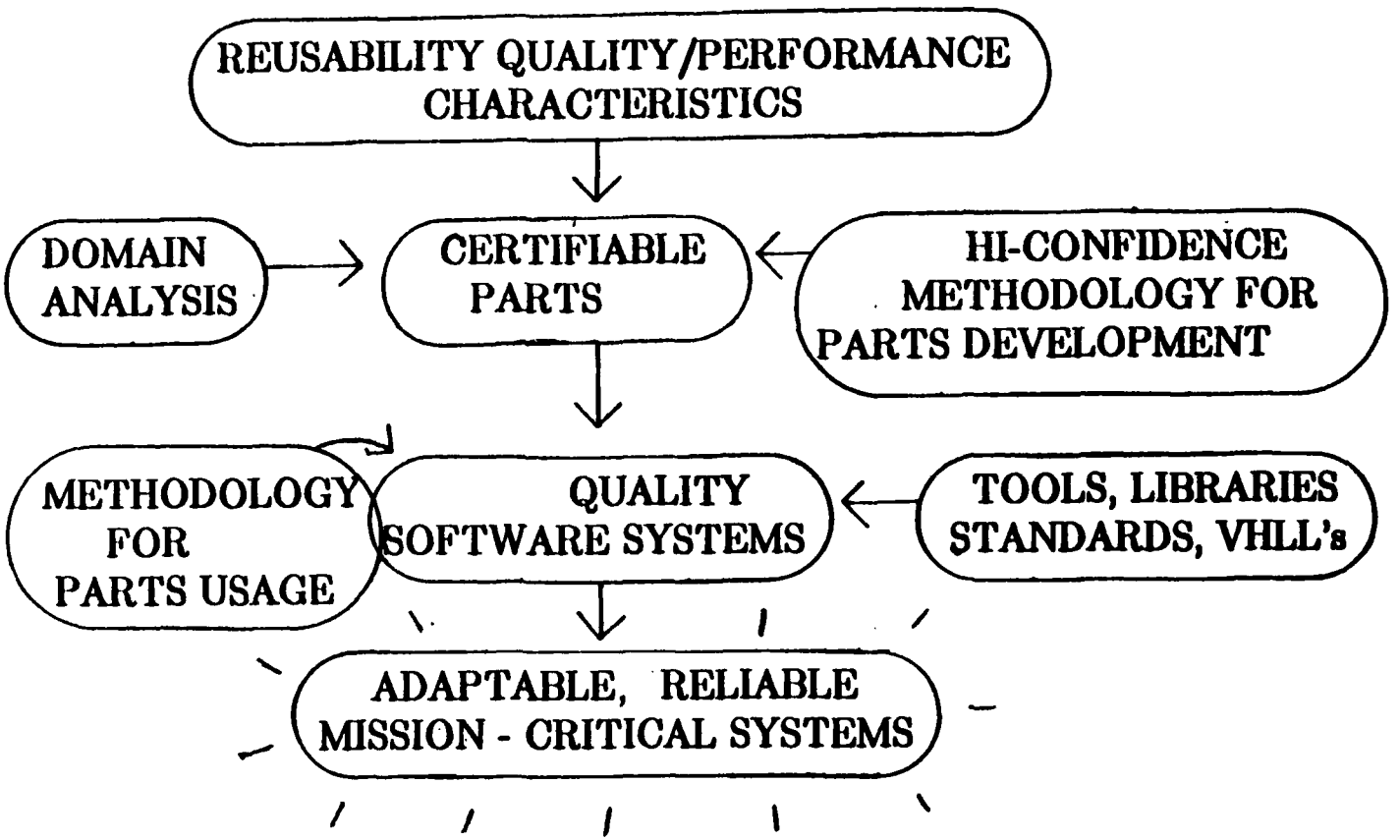
To Develop and Demonstrate Methods for Certifying Software Parts

PARTS TECHNOLOGY SHOWCASE (PROJECT 7)

OBJECTIVE

To Establish a Showcase Demonstrating
Parts Classification, Certification,
Library Cataloging, and
Evolution / Maintenance Procedures

SUMMARY


REUSABILITY QUALITY/PERFORMANCE CHARACTERISTICS
DOMAIN ANALYSIS
CERTIFIABLE PARTS
HI-CONFIDENCE METHODOLOGY FOR PARTS DEVELOPMENT
METHODOLOGY FOR PARTS USAGE
QUALITY SOFTWARE SYSTEMS
TOOLS, LIBRARIES STANDARDS, VHLL's
ADAPTABLE, RELIABLE MISSION - CRITICAL SYSTEMS

DESIGN/INTEGRATION PANEL REPORT

Elaine N. Frankowski
Christine M. Anderson

May 15, 1985

Executive Summary

During the past ten years the DoD (Department of Defense) has become increasingly sensitive both to the critical role that software plays in defense systems and to the software crisis facing DoD contractors. This crisis, caused by current software development methods, is manifested in rapidly escalating costs, schedule delays, and reliability problems. The defense needs of the 1990's and beyond will not be met if software continues to be developed line-by-line. Reusable software must be a part of any solution to this growing software crisis.

The primary emphasis of the April 1985 STARS Applications Area Workshop was reusable software. The objective of the Design/Integration Workshop Panel was to develop a set of candidate projects which would result in a robust reuse technology for mission critical systems. This panel recommends seven STARS projects, ordered in time to ensure robust, transferrable, reuse technology (see Figure 1). These projects are:

(1) **Parts Methodology Study**
To identify a compatible set of reusability methods for domain analysis, parts development and parts use.

(2) **Horizontal Domain Identification/Parts Development**
(a) To identify domains within DoD mission critical applications which span multiple application areas and for which reusable software can be developed; (b) to identify, specify, construct and test parts from these horizontal domains (See Figure 2)

(3) **Vertical Domain Identification/Parts Development**
(a) To identify relatively independent DoD mission critical application families in much there is a high potential for reusability, a significant opportunity for cost savings and/or an opportunity to capture critical expertise in solving complex problems; (b) to identify, specify, construct and test parts from these vertical domain. (See Figure 2)

(4) **Software CAD/CAM**
To investigate, develop and demonstrate graphic support for software development similar to the graphic support that so greatly enhances the hardware development process.

(5) **Software Reusability Demonstrations**
To develop a mission critical system or subsystem from a vertical domain using parts from the horizontal and vertical domains identified in Projects 2 and 3.

(6) **Parts Certification**
To define and demonstrate the parts certification process needed for process needed for parts that will become part of a high quality reusable library.

(7) **Parts Technology Showcase**

To establish a showcase demonstrating parts classification, certification, library cataloging and evolution/maintenance procedures.

1. Introduction

The primary emphasis of the April 1985 STARS Applications Area Workshop was software reusability. Reuse of software parts such as source code, designs, test data, documentation, and object code, can shorten the mission critical system development schedule, leading to the cost savings inherent in on-time system delivery. However, software parts reuse must be supported by reuse technology. Reuse technology includes methods and computer automated supoort tools both for developing reusable parts and for integrating them into mission critical systems. This technology must be developed, tested and transferred to the contractor community.

The objective of the Design/Integration Workshop Panel was to develop a set of candidate projects which would result in a robust reuse technology for mission critical systems. This panel recommended seven STARS projects, ordered in time to ensure robust, transferrable, resue technology (see Figure 1). The proposed projects address needs that were identified in previous reusability studies [IEEE84]. The most recent of those was the NSIA organized industry study on C3I software resuability [MORTISON84].

This paper is arranged as follows: Section 2 discusses the assumptions underlying the panel's deliberations. Section 3 reviews the design/integration issues that motivated the project definitions, and other issues important enough to merit study. Section 4 described the seven STARS projects that the panel recommends. Section 5 summarizes the effects of accomplishing the recommended work. Section 6 lists the panel members.

2. Assumptions

These are the assumptions that underlie the issues identified and the projects recommended.

(1) Every delivered part has support: for example, code is supported by documentation, a design, test data, and so forth. When we discuss a reusable part we are discussing the part and its supporting elements.

(2) Wherever feasible, Ada is the language of reusable parts. Ada is obviously the language of choice for reusable source code, since the DoD mandates it for mission critical software. In addition, Ada can be used to represent designs, and as a notation for documentating interfaces and other information about reusable code. This fact does not, however, exclude other languages from being notations for reusable parts where those languages are demonstratably more appropriate since it is the prescribed language for mission critical software.

(3) We recognize two design problems; the design of parts and the design of application systems that are composed or parts. In addition, we recognize that these problems have equal weight; both must be solved in order to develop mission critical software with reusable parts.

(4) We recognize that reusable parts must be engineered; not every software component that exists today or that will be developed in the future will be reusable. Retrofitting existing software for reuse may be prohibitively expensive and/or infeasible. Therefore, we are recommending projects to develop techniques for producing reusable parts; the techniques may produce reusable parts from scratch or by modifying existing parts. However, we are not specifically recommending the development of techniques for retrofitting existing software for reusability.

(5) We recognize that the sociological and technology transfer problems of software reuse are as important as, and perhaps more difficult than, the technical problems. Therefore, we are recommending projects whose success will provide technical grounds for increasing people's willingness to reuse parts and, confidence in the parts they reuse.

(6) We recognize current reluctance to reuse parts within the same company, even within the same division, the same department or the same section. Therefore, we think that a library of parts shared among entire industries is at least twenty years away and propose this scenario which leads eventually to that inter-company parts sharing.

o STARS supports the development of reusability technology – design methods, tools, and the like – and offers that technology to defense contractors;

o Contractors use this technology to develop reusable parts and reuse them internally;

o STARS or the DoD acquires some of this reusable software and make it available in some way that promotes sharing among contractors;

o Eventually contractors reuse software that has been made commonly available.

3. Issues

These issues motivate the projects defined in section 4. Section 3.1 presents the issues that led directly to the project defintions; section 3.2 discusses issues outside the proposed project framework.

The confidence problem cited in the next section is an umbrella issue that subsumes all the individual questions. Each of the projects proposed in section 4 provides another technical method or tool for the eventual high-confidence methodology.

3.1 High-Confidence Methodology

The sense of this workshop was that one of the major barriers, perhaps the single most serious barrier, to software parts reuse was the lack of confidence one industrial user would have in another's reusable software parts. To overcome that confidence gap, we recommend a study and implementation program to develop a methodology for producing "high-confidence parts."

We recognize that companies have or will develop individual software development methodologies, and would not accept one prescribed methodology. However, those methodologies can benefit by incorporating techniques (and their supporting tools) that specifically promote developing reusable software parts. The High-Confidence Methodology Project will develop a parts development methodology, some support tools for the techniques that comprise the methodology, and a collection of reusable parts. However, its major product will be the techniques that promote reusability and the individual tools that support those techniques. We assume that individual tools, supporting individual techniques, will be applicable in a variety of total methodologies.

The first phase of the program is a study that identifies the characteristics of reusable software parts. These characteristics need to be known so that the proposed methodology will lead to producing parts which have those characteristics and are, therefore, reusable. The study is undertaken assuming that we know what a "part" is and that we have collections of reusable and non-reusable parts to study. The work consists of studying reusable and non-reusable software components in light of the standard software quality characteristics called out in DoD-STD-2167 (SDS) and other sources and, determining which of these characteristics make a part reusable.

In the second phase, a software methodology(ies) will be selected (perhaps an RSIP (Reusable Software Implementation Plan) [GRABOW84] recommended methodology) and modified to include techniques that produce high-confidence reusable software arts, as defined by the characteristics identified in phase 1. We expect that the techniques will put special emphasis on validation, testing and certification, and that reusability metrics will be defined or adopted. The methodology will be developed by experiment; prototypes of needed support tools will have to be built to adequately test the methodology under development.

The final phase of the project will recommend the tools needed to support the methodology. We envision a companion project to develop these tools. In the companion project, the highest payoff tools will be developed first, and all tools will be insertable into the SEE.

3.2 Issues that Merit Study outside the Project Framework

Issue:

Are the concepts and terminology of DoD-STD-2167 (SDS) and its accompanying Data Item Descriptions (DIDs) applicable to reusable parts technology?

Recommendation:

(a) Study SDS/DIDs in light of the reusability characteristics identified in the study phase of

the High-Confidence Methodology Project. (b) Modify SDS/DIDs to reflect reusability concerns.

Issue:

If the long term goal of the reusability project is an inter-company library of reusable parts, structural standards for those parts will have to be very detailed so that parts will be integrable. Standards will have to meet both design/integration goals and library goals.

Recommendation:

Interface with the library panel.

Issue:

Only with something to "glue" them together will parts be reusable. For real time systems, glue candidates are the scheduling algorithms and executives that both provide a computational model and control the software in execution. For mission critical systems which do not have the stringent timing constraints characteristic of real-time systems, glue candidates are the Unix pipe/filter pattern, procedure files, or editor scripts. In every case, there is a need for some mechanism for interfacing components. One attribute specified for every component in a library should be its expected interface mechanism.

Recommendation:

A study to define some promising interface mechanisms for each type of mission critical software, so that a part's specification can include its assumed framework.

Issue:

How can system level decisions affect (encourage) software reuse? How does software reuse impact system level decisions?

Recommendation:

A study aimed at identifying heuristics to guide system engineers in allocating functionality among people, hardware, software and special electronics, with an eye to reusability. The study will assume:

(1) some inventory of reusable parts, such as an inventory comparable to the one identified by CAMP (Common Ada Misle Packages. Air Force contract F08635-84-C-0280);

(2) that the inventory arises in an ongoing development project.

Issue:

Very High Level Languages (VHHLs) address the needs of application specific domains and have proven themselves effective in increasing software generation productivity, in some cases (e.g., test generation with ATLAS) by allowing an application specialist to implement software without a software specialist, in other cases by improving the software specialist's productivity. This workshop has excluded VHLLs from consideration, whereas the STARS workshop held at Raleigh, N. Carolina in February 1983 saw it as the coequal of reusability.

Recommendation:

Do not limit the STARS Application Area to conventional concepts of reusable concepts of reusable components. Include VHLLs.

4. Projects

This section presents the seven projects proposed for STARS funding. The project definitions are responses to the key issues discussed by the Design/Integration Panel, and presented in Section 3. For this discussion, we define vertical domains as mission critical systems or subsystems such as missile navagation, aircraft flight control and command and control tracking; and, horizontal domains as functional areas such as input/output, asynchronous control structures, and storage management that cut across a large number of vertical domains. (See Figure 2)

Project 1

Parts Methodology Study. The objective of this project is to identify a compatible set of parts-reusability methods for domain analysis, parts development and parts use. The Panel recommends starting this study before the domain analysis/parts development efforts described in Projects 2 and 3 so that methodologies identified in this study can be used and evaluated in those projects. The Panel further recommends that the STARS Methodology Area Coordinating Team sponsor Project 1.

Project 2A

Horizontal Domain Identification. The objective of this project is to identify domains within DoD mission critical applications which span multiple application areas (horizontal domains) in which reusable software can be

developed.

Project 2B

Horizontal Domain Parts Development. This project is a follow-on to the previous effort. The objective is to identify, specify, construct and test parts from the horizontal domains. The panel envisions that several contracts will be awarded in order to apply and evaluate several parts development methodologies identified in Project 1. The panel recommends that this be a joint effort between the STARS Application and Methodology areas, with the majority of funds coming from the Application area.

Project 3A

Vertical Domain Identification. The objective of this project is to identify relatively independent DoD mission critical application families in which there is a high potential for reusability, a significant opportunity for cost savings and/or an opportunity to capture critical expertise in solving problems in a complex domain. The identification process includes some degree of analysis to justify the R&D value of the vertical domain selected.

Project 3B

Vertical Domain Parts Development. This project is a follow-on to the previous effort. The objective is to identify, specify, construct and test parts from the vertical domain. The panel suggests that multiple contracts be awarded so that each contract applies a different parts development methodology identified in Project 1. The panel sees this as a joint effort between the STARS Application and Methodology areas, with the majority of funds coming from the Applications area.

Project 4

Software CAD/CAM. Graphic/pictorial representations enhance the hardware development process greatly. The objective of this effort is to investigate, develop and demonstrate a similar graphic support capability for the software development process. Techniques, such as expert systems, that may enhance this capability will also be explored.

Project 5

Software Reusability Demonstration. The objective of this effort is to develop a mission critical system or subsystem from an identified vertical domain using parts from the horizontal and vertical domains identified in Projects 2 and 3. These mission critical systems may be selected from developing or inventoried defense systems. Typically, they will be objects of other research activities, but may also be subsystems of operational systems (e.g., F-15 flight control). Several contracts will be awarded in order to apply and evaluate different parts usage methodologies identified in Project 1.

Project 6

Parts Certification. It is assumed that during development a part is thoroughly tested. However, before a part is accepted into any library certification must be performed in order to ensure only quality parts are accepted. The objective of this effort is to define and demonstrate the parts certification process. Defining a parts certification process includes quantifying the characteristics of a "quality" part. The panel envisions that this effort will be jointly managed by the STARS Applications and Measurement areas, with the majority of funds coming from the Applications area.

Project 7

Parts Technology Showcase. The objective of this effort is to establish a showcase demonstrating parts classification, certification, library cataloging and evolution/maintenance procedures. It is envisioned that this showcase will be incorporated into the "Software Factory" at the Software Engineering Institute.

5. Summary

The path to adaptable/reliable mission critical systems made from reusable parts begins with "high-confidence" certifiable parts. The technology ele- ments leading to "high-confidence" parts are:

- o domain analysis of mission critical vertical domains and high-payoff horizontal domains.
- o definitions of the quality and performance characteristics required of reusable parts.
- o a tool supported, "high-confidence" parts development methodology

o parts certification methods,

The construction of high quality software systems from those parts requires:

o a tool supported methodology for parts use,

o system construction tools,

o libraries of certified parts,

o very high level languages to specify systems composed of parats

The projects recommended by the Design/Integration Panel are arranged chronologically to support the development of the needed technology, and attack the key issues identified as technical barriers to constructing adaptable/reliable systems from certified "high-confidence" reusable parts today.

6. Design/Integration Panel Members

These are the participants in the STARS Applications Area Reusability Workshop's Design/Integration Panel.

NAME	AFFILIATION
Elaine N. Frankowski	Honeywell, Inc. (Co-Chair)
Christine M. Anderson	Air Force Armament Lab (Co-Chair)
J. G. Snodgrass	E-Systems
William Wong	National Bureau of Standards
John Da Graca	General Dynamics
Jim Winchester	Hughes Aircraft
Daniel G. McNicholl	McDonnell Douglas
Ron McCain	IBM
Ray Dion	Raytheon
Nancy Ys. Kim	Rockwell International
C. K. Pian	Hughes Aircraft
Gregg Van Volkenburgh	Allied Canada Inc.
Paul Hixson	General Dynamics
E. J. Startzman	Boeing
J. Roder	GTE
Christine Youngblut	Advanced Software Methods
M. Kemer Thomas	General Dynamics
Vicky Mosley	Westinghouse

7 References

[GRABOW84] Paul C. Grabow, William B. Noble and Cheng-Chi Huang, Reusable Software Implementation Technology Reviews, Hughes Aircraft Company Technical Report N66001-83-D-0095 FR 84-17-660 RevA, December 1984.

[IEEE84] IEEE Transactions on Software Engineering, Vol SE-10 no. 5, September 1984.

[MORTISON84] J. E. Mortison, Systems Engineering Aspects of Software Reusability, NSIA Study Task ISTG 84-2, Slide Presentation to appear.

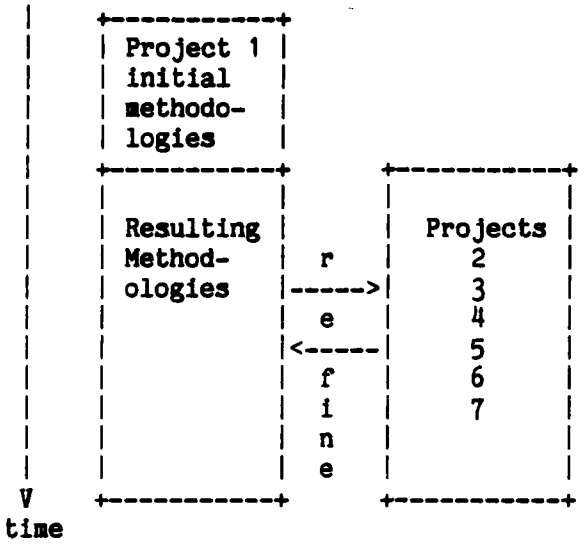


FIGURE 1: Timeline for the Design/Integration Projects

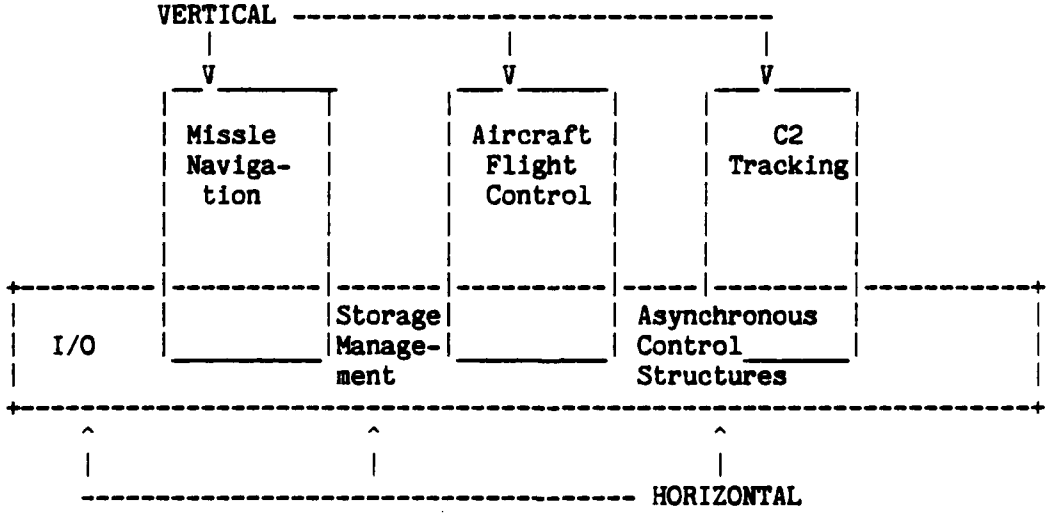


FIGURE 2: Vertical and Horizontal Domains

GROUP V METRICS

Tom Bowen, Boeing
Jerry Brown, US Army
Toni Shetler, Systems Engineering
Lyle Anderson
Cheng Huang, Hughes
Fred Rosene, Communications System
Ragha Singh, NAVMAT
Jim Kirkpatrick, Air Force
Arny Engelson, Grumman
Ted Taylor, Effective Software

METRICS

1. RECOMMENDATIONS

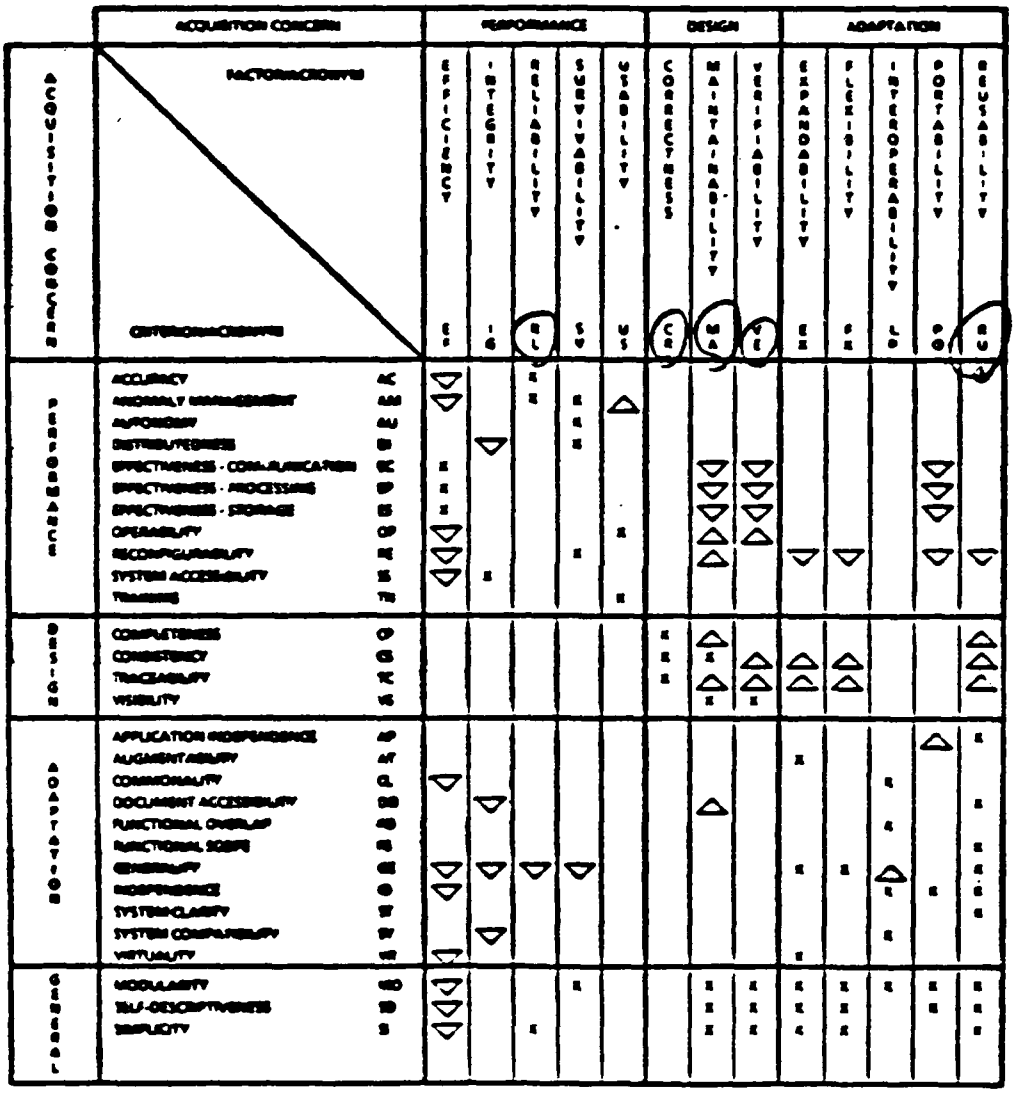
   a. DEVELOPMENT OF PARTS* TO BE REUSED
   b. USE OF REUSABLE PART
   c. RELATED ISSUES

2. QUESTIONS AND RESPONSES

3. WORKSHOP INPUTS

* = Scope of Term (reusable) part - includes requirements, design, code, architecture,...

Table 4.1.3-1 Effects of Criteria on Software Quality Factors



945

1.a. DEVELOPMENT OF PARTS TO BE REUSABLE

CHARACTERISTICS

- o Start with characteristics (criteria) identified in RADC quality framework (e.g., modularity, generality)
- o Consider factor interrelationships

METHODOLOGY

- o Use of specification & evaluation methodology for acquisition of reusable parts (see 2167/SDS)
- o Start with RADC methodology & procedures

MEASUREMENTS

- o Start with RADC metric worksheets & software evaluation reports of STARS measurement DIDs.
  Refine/Tailor for Ada/Validate
- o Add descriptive information (known to developer)
  (e.g., language, host processor, cost, validation, speed size).
  Select items from STARS measurement DIDs.

Note:

- o Other relevent sources - Ada E & V, SEI, KIT/KITIA,...

Considerations:

- o For all levels of abstraction
- o Development products should contain information for measurements

1.b. USE OF REUSABLE PARTS

Start with items from STARS measurement DIDS and develop part characterization, include:

- o Development information
  (language, host, cost, validation, test information, requirements information)

- o Quality levels
  (e.g., High reusability, medium efficiency)

- o Functionality (overlap with taxonomy)
  (What it does)

- o Operational characteristics
  (Documentation, execution speed, ...)

- o History (some items related to c.m. & incentives)
  (originator & reputation, cost (develop & reuse)
  (estimated & actuals), schedule, testimonial, number of requests & uses,
  degree of validation, change history, test drivers/data & results,...)

- o Interface
  (Relationship between environment invoking the part and functions performed by part; and characteristics that are asserted by the developer to remain constant - even in improved versions)

Note: Insure accessibility of parts - user will search by needs.

1.c. RELATED ISSUES

LIBRARY

- o Maintain all (?) old versions

- o Access 'versatility (adjustable "black box" view)
  should users be able to search/evaluate parts from different
  perspectives (shift level of detail)? - Y/N

INCENTIVES

- o Tie developer to maintenance costs. (How?)

- o Developer royalties fro actual use! (Details?)

- o Reward $ Savings due to reuse. (How?)

- o Contractor incentive for reuse is better competitive (bid) position.
  (Except Sperry).

WORKSHOP INPUTS

o It would be useful to create scenarios for various types of users in order to provide a relative evaluation of these recommendations. This analysis might also provide a priority scheme for the recommendations.

o Group V feels that is would be beneficial to allot more inter-group and intra-group communication time.

o assumptions
o redundant discussions
o related issues