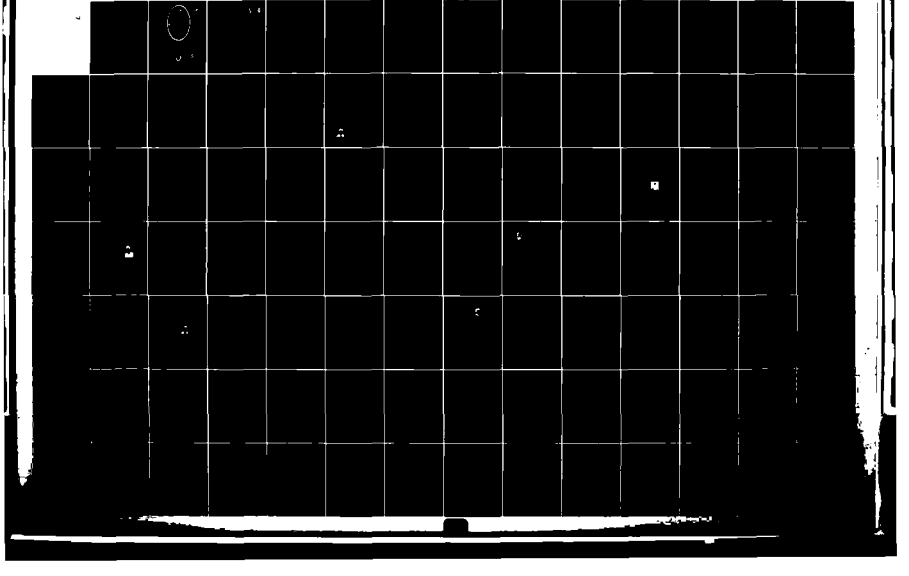
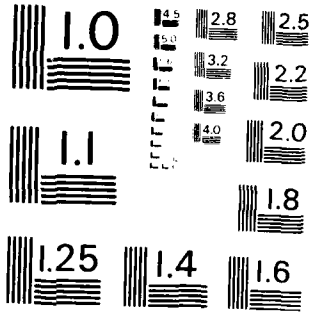
AD-A142 403 PROCEEDINGS OF THE ANNUAL CONFERENCE ON ADA (TRADEMARK) 1/2
TECHNOLOGY (2ND)..(U) ARMY COMMUNICATIONS-ELECTRONICS
COMMAND FORT MONMOUTH NJ CENT.. MAR 84

UNCLASSIFIED F/G 9/2 NL





Proceedings of the 2ND Annual Conference on ®Ada Technology

March 27, 28, 1984

SPONSORED BY U.S. ARMY CENTER FOR TACTICAL COMPUTER SYSTEMS
FORT MONMOUTH, NEW JERSEY
Host College - HAMPTON INSTITUTE, Hampton, Va.

®Ada is a trademark of the Department of Defense (Ada Joint Program Office)

COMPONENT PART NOTICE

THIS PAPER IS A COMPONENT PART OF THE FOLLOWING COMPILATION REPORT:

(TITLE): Proceedings of the Annual Conference on Ada (Trademark) Technology (2nd) Held at Hampton, Virginia om March 27, 28, 1984.

(SOURCE): Army Communications-Electronics Command, Fort Monmouth, N.J. Center for Tactical Computer Systems.

TO ORDER THE COMPLETE COMPILATION REPORT USE AD-A142 403.

THE COMPONENT PART IS PROVIDED HERE TO ALLOW USERS ACCESS TO INDIVIDUALLY AUTHORED SECTIONS OF PROCEEDINGS, ANNALS, SYMPOSIA, ETC. HOWEVER, THE COMPONENT SHOULD BE CONSIDERED WITHIN THE CONTEXT OF THE OVERALL COMPILATION REPORT AND NOT AS A STAND-ALONE TECHNICAL REPORT.

THE FOLLOWING COMPONENT PART NUMBERS COMPRISE THE COMPILATION REPORT:

AD#:	TITLE:
AD-P003 414	The Army Ada (Trademark) Education Program.
AD-P003 415	The U. S. Army Model Ada (Trademark) Training Curriculum.
AD-P003 416	Configuration Management with the Ada (Trademark) Language System.
AD-P003 417	Learning the Ada (Trademark) Integrated Environment.
AD-P003 418	Teaching Ada (Trademark) at the US Military Academy.
AD-P003 419	Experiences in Teaching Ada (Trademark).
AD-P003 420	Teaching Ada (Trademark) at Hampton Institute.
AD-P003 421	The CECOM Summer Faculty Research Program.
AD-P003 422	Teach Ada (Trademark) as the Student's First Programming Language.
AD-P003 423	An Ada (Trademark) Network: A Real - Time Distributed Computer System.
AD-P003 424	DCP (Distributed Software Engineering Control Process) - Experience in Bootst rapping an Ada (Trademark) Environment.
AD-P003 425	Ada (Trademark)for Business & other Non - DoD Applications.
AD-P003 426	Exerience with Ada (Trademark) for the Graphical Karnal System.
AD-P003 427	Military Computer Family Operating System: An Ada (Trademark) Application.
AD-P003 428	An Advanced Host - Target Environment for the Military Computer Family.
AD-P003 429	Ada (Trademark) Tasking in Numerical Analysis.
AD-P003 430	Ada (Trademark) as a Program Design Language - Have the Major Issues Been Add ressed and Answered.
AD-P003 431	Ada (Trademark) Design Language Concerns.
AD-P003 432	Seeding the Ada (Trademark) Software Components Industry.
AD-P003 433	Operating System Interface for Ada (Trademark) Instructors.

COMPONENT PART NOTICE (CON'T)

AD#: TITLE:

PROCEEDINGS OF SECOND ANNUAL CONFERENCE ON *ADA TECHNOLOGY

SPONSORED BY
U.S. ARMY CENTER FOR TACTICAL COMPUTER SYSTEMS
(CENTACS), FORT MONMOUTH, NEW JERSEY

HOST COLLEGE
HAMPTON INSTITUTE, HAMPTON VIRGINIA

SHERATON INN/HOLIDAY INN
MERCURY BLVD.
HAMPTON, VIRGINIA

*Ada is a Registered Trademark of the Department of Defense
(Ada Joint Program Office)

SECOND ANNUAL CONFERENCE ON ADA TECHNOLOGY

TECHNICAL SESSIONS as follows:

Tuesday Morning 27 March 1984

9:00 am	Session I	Government, Academia and Industry Speak
10:30 am	Session II	The Army Ada Training Initiative
10:30 am	Session III	Ada Programming Environment

Tuesday Afternoon 27 March 1984

2:00 pm	Session IV	Teaching Ada
2:00 pm	Session V	Ada Applications

Wednesday Morning 28 March 1984

9:00 am	Session VI	Application of Ada in the Mathematical Science
9:00 am	Session VII	IEEE PDL Working Group Report
11:00 am	Session VIII	Ada and the Future

PAPERS

Responsibility for the contents included in each paper rests upon the authors and the not the Conference Sponsor. After the Conference, all the publication rights of each paper are reserved by their authors, and requests for republication of a paper should be addressed to the appropriate author. Abstracting is permitted, and it would be appreciated if the Conference is credited when abstracts or papers are republished. Requests for individual copies of papers should be addressed to the authors.

CONTRIBUTORS

TRW
Redondo Beach, California

General Electric Company
Syracuse, New York

Softech Inc.
Waltham, Massachusetts

TABLE OF CONTENTS

Tidewater Room—Sheraton Inn

SESSION VII: IEEE PDL Working Group Report

Chairperson: Mark Gerhardt, Raytheon, Portsmouth, RI

Tidewater Room—Sheraton Inn

SESSION VIII: Ada and The Future

Chairperson: Joseph Kernan, US Army, CENTACS, Ft. Monmouth, NJ

THE ARMY ADA* EDUCATION PROGRAM

Dennis J. Turner

Center for Tactical Computer Systems (CENTACS)
U.S. Army Communications-Electronics Command (CECOM)
Fort Monmouth, New Jersey

In behalf of the U.S. Army, CENTACS is pursuing a comprehensive and aggressive Ada program. An important aspect of that program is the development and transfer of public domain Ada educational and training materials which are focused on the needs of the academic, industrial and government communities. This paper provides an overview of the Army's Ada education and training program and summarizes the products and materials which are being produced under contracts with Softech, Inc., New York University and Jersey City State College.

Summary of the CENTACS Ada Program

CENTACS has been actively supporting the DOD Ada initiative since 1975. Activities have been focused in five complementary areas: language definition, program support environments, methodology, education and training, and Policy.

Language Definition

CENTACS provided the Army representative to the DOD High Order Language Working Group (HOLWG) which developed the initial language requirements. Competing designs led to the selection of the so called GREEN language which was later to become Ada, as now defined in MIL STD 1815A. CENTACS contracts with Softech, Inc., Teledyne-Brown Engineering and New York University have been instrumental in the development of the test suite which is now used by the DOD Ada Joint Program Office (AJPO) to validate that compilers are implementing the language definition correctly.

Program Support Environments

CENTACS has participated in the development of requirements for Ada Program Support Environments (APSE's). These requirements are currently embodied in a document known as the STONEMAN.

In June 1980, CECOM awarded a contract to Softech, Inc. to develop an Ada Language System (ALS) which combines the Ada and STONEMAN initiatives. The ALS is a government owned, production quality Ada support environment which includes a rich set of powerful tools (in addition to a compiler) which will dramatically improve the productivity of programmers and technical managers. The ALS baseline system (VAX host) is scheduled for completion by the fall of 1984 and additional tools in support of targeting to the Army's Military Computer Family (MCF) by December 1985.

Representatives have also been participating in the service activities aimed at the development of a Common Ada Interface Set (CAIS), and the Joint Service Software Engineering Environment (JSSEE) Committee under the Software Technology for Adaptable Reliable Systems (STARS) initiative.

CENTACS is sponsoring a research effort which is developing a prototype system (called GANDALF) for experimentation with syntax directed editors, intelligent work stations and distributed processing, all in an Ada context.

The development and evaluation of powerful support environments are essential if we are to maximize the productivity of programmers.

Methodology

CENTACS has been pursuing a variety of methodology investigations since the early 70's. This important topic is the focus of a great deal of research and is a major thrust of the STARS initiative. The Army is strongly motivated to advance the state-of-the-art in this area in order to reduce costs thru effective methodology which can be applied uniformly across systems.

Education and Training

The Ada language and its supporting environments and methodologies are, obviously, of little value if the workforce is not able to put them to effective use. CENTACS has recognized the need for education and training and initiated a comprehensive program to provide academia and industry with necessary curriculums and materials to meet that need. Since this is the topic of this paper, it will be described in greater detail below.

Policy and Objectives

The DOD has established Ada as the standard programming language to be used across all defense

*Ada is a registered trademark of the Department of Defense, Ada Joint Program Office, OUSDRE (R&AT)

systems. CECOM and its parent Command, DARCOM, are also requiring the use of Ada as a Program Design Language (PDL) for Army systems. The Army seeks, further, to establish the ALS as a common support environment to be used across all Battlefield Automated Systems (BAS's). The overall objective is clear: common languge, common support environments, common methodology and common education/training. With software costs continuing to grow at an alarming rate, commonality is not just desireable - it is essential.

Ada Education and Training Initiatives

CENTACS activities in this area have been focused on the separate needs of the academic community (education) and those of Industry and Government (training). The common theme in both arenas is the timely development of Ada materials and products which can be used to cultivate widespread Ada literacy and skills.

Education Initiatives

The "Ada/Ed" Translator/Interpreter

When the Ada language first began to emerge, CENTACS recognized the need for an implementation that could be used for experimental but, primarily, educational (thus Ada/Ed) purposes. This project, which has been performed under contract with the Courant Institute of New York University, was initiatived at a point when the Ada language was not completely defined and has culminated in the first fully validated ANSI-Ada translator.

Ada/Ed was written in a very high level "set" language (called SETL). This approach enabled the translator to be implemented in roughly one-fifth the time that might otherwise have been expected. However, rapid implementation via this approach was achieved at the expense of execution speed (Ada/Ed is slow).

Ada/Ed runs on a VAX-11/780 and, despite its slowness, has been widely acclaimed as a valuable education tool, with a particularly friendly user interface.

Ada/Ed has been placed in the public domain and can be acquired from the National Technical Information Service (NTIS), U.S. Department of Commerce, 5185 Port Royal Road, Springfield, VA 22161, Phone: (703) 487-4650, for a nominal (reproduction) fee.

A continuation contract with NYU is currently focused on maintenance, efficiency improvements and re-validation through the AJPO.

Initiatives for Academia

If Ada is to be truly successful, we must address the need to educate member of our future workforce as they advance through our academic institutions. CENTACS recognized this need and has sponsored a contract with Jersey City State College which has thus far led to the development of two courses and the establishment of an Ada Technology Center.

The two courses - one undergraduate, one graduate - address Ada philosophy and concepts, syntax, semantics and provide complementary exercises and hands-on experience using Ada/Ed on a VAX-11/780. The undergraduate course focuses on Ada fundamentals while the graduate course provides advanced topics such as "tasking" and "generics".

The graduate course was taught at Fort Monmouth over an eight week period (2 hours, twice a week) with CENTACS personnel serving as the evaluators. The undergraduate course was given at Jersey City State College with students majoring in computer science serving as evaluators. Both courses have been favorably received and suggestions from the evaluations have already been incorporated.

Course materials include lesson plans, teachers guides, student guides, and viewgraphs. Again, all materials will be placed in the public domain and will be accessible through the NTIS.

An Ada Technology Center has also been established at Jersey City State College. It currently includes a VAX-11/780 and eight DEC GIGI terminals. The center was established to serve as a friendly site to government, industry and academia personnel pursuing Ada research, training or education.

Future activity with Jersey City State College will include the development of additional courses and an attempt to automate the courses (via the DEC "Course Authoring System") so that they can be both self-paced and portable to other sites.

Training Initiatives

Initiatives for Government and Industry

In considering the development of an Ada training program for Industry and Government, CENTACS, with strong contractual support from Softech, Inc., chose a very systematic approach. Three distinct data gathering activities were initiated as a means to formulate training requirements.

The first involved the use of Ada in the redesign of selected portions of the AN/TSQ-73 and the AN/TYC-39 by Control Data Corporation and General Dynamics, respectively. In the course of these "case studies", Softech monitors were able to identify the issues (language, environment and methodology) which are of greatest concern (i.e., candidates for training emphasis) in approaching the use of Ada on real Army systems.

The second activity involved a survey of the industry and government workforces in an attempt to identify and understand the functions which are performed by personnel working on the development of large-scale embedded computer systems. Each identified job category was characterized by the associated duties and the required technical and educational background.

The third activity involved a survey of industrial training methods and practices. Here, six large corporations, each with extensive involvement in large-scale systems, were queried to determine the most effective training approaches.

The results of the case studies, the workforce and training methods surveys were analyzed and 15 generic job categories (see figure 1) were identified with suggested course sequences within an overall model Ada training curriculum. Courses in the model Ada curriculum can be divided into three basic categories: Ada language, software engineering methodology and Ada program support environment. The courses (35 in all) within each category are identified in figure 2. CENTACS and Softech are currently developing 15 of these courses as summarized in figure 3.

In addition to these "generic" Ada training materials, CENTACS is alo developing material unique to the Ada Language System (ALS). Included here are a Users course, an ALS textbook, an ALS Administrator Manual and an ALS administrator course.

All of these materials, when complete, will also be placed in the public domain and be accessible through the NTIS.

Future activities here will include the development of additional courses within the model curriculum and the pursuit of the automation of selected Ada and ALS training material.

Summary

Through primary contracts with Softech, Inc., New York University and Jersey City State College, CENTACS has produced a great deal of educational and training material in support of the Ada language. While much has been accomplished, a great deal remains and the Army intends to remain active in this important arena to help insure the ultimate success of Ada.

Biographical Sketch

Mr. Dennis J. Turner holds BSEE and MSEE degrees from Monmouth College, West Long Branch, New Jersey. He has been a member of the U.S. Army Communications-Electronics Command for twelve years and is currently the Chief of the Software Technology Development Division within the Center for Tactical Computer Systems (CENTACS).

Mr. Turner has held industrial positions with DIVA Incorporated, Electronics Associates Incorporated and Frequency Engineering Laboratories.

His mailing address is:

U.S. Army CECOM
ATTN: DRSEL-TCS-ADA
Fort Monmouth, NJ 07703

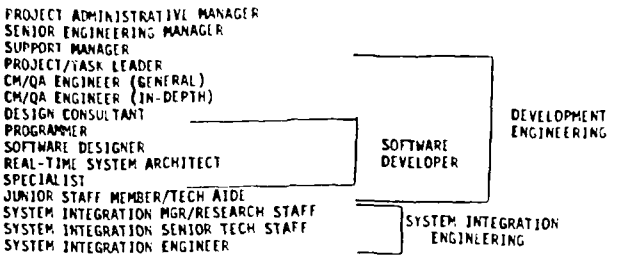


FIGURE 1
GENERIC JOB CATEGORIES

MANAGEMENT

NO.	TITLE
M101	SOFTWARE ENGINEERING FOR MANAGERS
M102	INTRODUCTION TO SOFTWARE ENGINEERING
M201	SOFTWARE ENGINEERING METHODOLOGIES
M202	OVERVIEW OF A SPECIFIC METHODOLOGY
M301	REQUIREMENTS METHOLDOGY
M302	DESIGN METHODOLOGY
M303	CODING METHODOLOGY
M304	SOFTWARE REVIEW METHODOLOGY
M401	INTRODUCING ADA TO YOUR ORGANIZATION
M402	PYSCHOLOGICAL ASPECTS OF RETRAINING

LANGUAGE

NO.	TITLE
L101	ADA ORIENTATION FOR MANAGERS
L102	ADA TECHNICAL OVERVIEW
L103	INTRODUCTION TO HIGH ORDER LANGUAGES
L104	BEGINNING PROGRAMMING
L201	ADA FOR TECHNICAL MANAGERS
L202	BASIC ADA PROGRAMMING
L301	USING THE ADA LANGUAGE REFERENCE MANUAL
L302	USE OF ADA FOR REQUIREMENTS
L303	REAL TIME CONCEPTS
L304	ADA READER'S COURSE
L305	ALGORITHMS AND DATA STRUCTURES IN ADA
L401	REAL TIME SYSTEMS IN ADA
L500	SPECIALITY COURSES

ENVIRONMENT

NO.	TITLE
E101	APSE CONCEPTS FOR TECHNICAL MANAGERS
E102	APSE OVERVIEW FOR PROGRAMMERS
E103	BASIC APSE OPERATION
E201	USER'S INTRODUCTION TO THE APSE
E301	COMMAND LANGUAGE
E302	PROGRAM DEVELOPMENT
E303	DATABASE
E304	DEBUGGING
E305	ASSEMBLING AND IMPORTING
E306	CONFIGURATION MANAGEMENT AND PROGRAM MANAGEMENT
E401	HOW TO ADD TOOLS
E402	SYSTEM ADMINISTOR'S COURSE

FIGURE 2
COURSE CATEGORIES

NO.	TITLE	DURATION
L101	Ada Orientation for Managers	1 day
L102	Ada Technical Overview	1 day
L103	Intro to Higher Order Languages	1 day
L201	Ada for Technical Managers	3 days
L202	Basic Ada Programming	1 week
L301	Using the Ada LRM	2 days
L302	Use of Ada for Requirements	2 days
L303	Real-Time Concepts	1 day
L304	Ada Reader's Course	1 day
L305	Advanced Ada Topics	1 week
L401	Real-time Systems in Ada	1 week
M101	Software Engineering for Managers	1 day
M102	Introduction to Software Engineering	2 days
M201	Software Engineering Methodologies	1 week
M303	Coding Methodology	2 days

FIGURE 3
COURSES UNDER DEVELOPMENT

THE U. S. ARMY MODEL ADA* TRAINING CURRICULUM

Putnam Texel
SofTech, Inc.

ABSTRACT

This paper describes the U. S. Army Model Ada Training Curriculum, developed by SofTech, Inc. for the U. S. Army, Ft. Monmouth, N.J. The curriculum consists of individual modules which can be grouped together to form the courses and training plans that best satisfy the needs of specific organizations. The paper describes the modules in terms of content, prerequisites, and status, as of the date of this conference. Finally the paper addresses how a manager might go about using this curriculum to satisfy the training needs of his organization.

Section 1

BACKGROUND

The U. S. Army Model Ada Training Curriculum defines a comprehensive set of training modules, or building blocks, which can be connected in a variety of ways to form the courses and training programs that best satisfy a given set of Ada training needs (see Figure 1).

It is well recognized that software development does not depend solely on a language, even Ada. Ada must be used in conjunction with a good methodology and systems must be developed within the framework of a rich and integrated programming environment. Consequently the modules of the U. S. Army Model Ada Training Curriculum cover three areas: the Ada language, methodology (both design and coding), and the environment. The modules within each of these specific areas are listed in Tables 1-3, along with brief descriptions.[1]

The modules of the curriculum differ in one or more of the following dimensions: area, depth, and viewpoint. Each of these dimensions is discussed below.

1.1 Area

Each module identifier starts with a letter. This initial letter indicates the area with which the module is to be associated. Modules beginning with the letter L address the Ada language. Modules beginning with the letter M and E address methodology and programming environment respectively.

1.2 Depth

Because each individual does not require the same degree of knowledge of a specific area, modules in the curriculum differ in terms of depth of material presented. The depth is indicated by the first digit occurring in the module identifier. A level-1 module, indicated by the digit 1, is a module having no prerequisites within the curriculum. A level-1 module may have some prerequisites in terms of general computer science knowledge not related to Ada. For example L103, Introduction To High Order Languages, has no prerequisites within the curriculum. The objective of L103 is to introduce assembly language programmers to the concept of high order languages, not to introduce a novice to the world of computing. Consequently L103 does require experience in assembly language programming as a prerequisite.

Level-2 modules, indicated by the digit 2, have prerequisites that can be satisfied by level-1 modules, and so on.

1.3 Viewpoint

Whereas a programmer needs detailed instructions on using the tools within the programming environment, a system administrator needs to know what demands those tools make on system resources. The fact that different individuals have different views of a specific subject matter, is addressed by the curriculum. As the module level increases, the viewpoint changes. Level-1 modules basically address managerial needs. Level-2 modules address programmer needs, and so on.

* Ada is a Registered Trademark of the Department of Defense (Ada Joint Program Office)

[1] C. L. Braun "Ada Training Considerations" 2nd AFSC Standardization Conference. Dayton, Ohio. Dec. 1982.

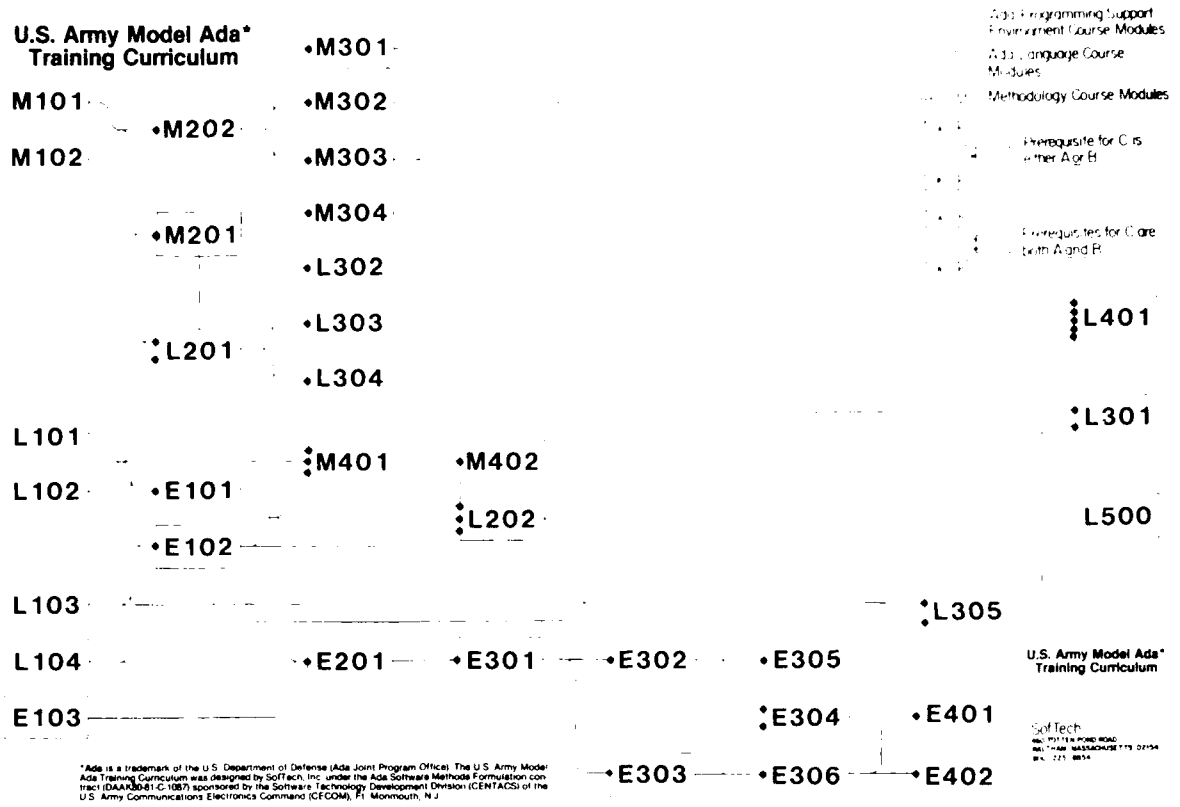


Figure 1. U. S. Army Model Ada Training Curriculum

Section 2

CURRENT STATUS

Currently SofTech, Inc. is under contract to develop a selected subset of the modules. Module development follows the software engineering process. A Preliminary Design Review (PDR) is conducted with the government. Upon government approval of the design, development begins. Upon completion of development a Critical Design Review (CDR) takes place at Fort Monmouth. Basically the CDR is the first teaching of a module and functions as an acceptance test. At the conclusion of the CDR, government feedback, along with student and instructors' suggestions, is considered for inclusion in the final product.

The specific modules under development, their length and expected completion date are shown in Table 4.

Section 3

PACKAGING

It is important to realize that one module can not satisfy the needs of an organization. Modules are packaged together to create a course for a specific organization. When selecting a course, the following must be kept in mind.

3.1 Define the Viewpoint

Who needs to be trained in your organization? Managers and practitioners? Once the viewpoint is selected, look for the modules that can be packaged together to train that viewpoint. Managers courses tend to be shorter and more concept oriented than programmers courses. But do not assume that the managers courses are therefore superficial; in many cases the emphasis on concepts (as opposed to details) makes the manager's courses deeper than any practioner's course.

3.2 Define the Level

The lower level courses contain the concept material required for managers. As previously stated they are prerequisites for the higher level modules. However for software managers who influence software without actually writing any code, or for QA personnel, consultants, analysts, etc., the higher levels are also appropriate.

3.3 Identify the Main Course

After having satisfied the prerequisites, an entry level programmer probably only needs L202 to

Table 1. Environment Modules

NO.	TITLE	DESCRIPTION	DURATION
E101	APSE Concepts for Technical Managers	broad overview of APSE emphasizing how it supports s/w life cycle	1 day
E102	APSE Overview for Programmers	broad overview of APSE for software developers	1/2 day
E103	Basic APSE Operation	introduction to APSE concepts, basic editing, etc., for people who will not be real users	1/2 day
E201	User's Introduction to the APSE	basic use of the APSE database, file system, command language; tool overview	3 days
E301	Command Language	command language, substitutors, I/O redirections	1 day
E302	Program Development	Compiler, linker, exporter, loader	2 days
E303	Database	files, directories, attributes, associations, access control, node sharing, program libraries, etc.	2 days
E304	Debugging	debugger, timing analyzer, frequency analyzer	1 1/2 days
E305	Assembling and Importing	assembly language, importer	1/2 day
E306	Configuration Management and Program Management	tools to support CM and PM, example tools one might build	3 days
E401	How to Add Tools	programming with the command language, KAPSE tool interfaces, examples of useful tools	2 days
E402	System Administrator's Course	user authorization and protection, installation, backup, system support	3 days

Table 2. Ada Language Modules

NO.	TITLE	DESCRIPTION	DURATION
L101	Ada Orientation for Managers	overview of development and features of Ada	1/2 day
L102	Ada Technical Overview	overview of language-introduction to language features in more depth than above	1 day
L103	Introduction to High Order Languages	key HOL concepts for assembly language programmers	1 day
L104	Beginning Programming	introduction to computer programming in an Ada context	4 weeks
L201	Ada for Software Managers	use of Ada for good systems design; packages, types, generics, portability features, etc.	3 days
L202	Basic Ada Programming	essentially the Pascal subset	1 week
L301	Using the Ada Language Reference Manual	how to use the alarm effectively as a reference	2 days

Table 2. Ada Language Modules (continued)

NO.	TITLE	DESCRIPTION	DURATION
L302	Use of Ada for Requirements	Ada as a requirements definition language	2 days
L303	Real Time Concepts	real time design concepts for technical managers	1 day
L304	Ada Reader's Course	reading an Ada design or program for its key points and overall structure	1 day
L305	Advanced Ada Topics	packages, access types, private types, discriminated records, generics, basic tasking, basic algorithms	1 week
L401	Real Time Systems in Ada	everything about tasking, external interfaces, low-level features	1 week
L500	Specialty Courses	numerical analysis, hardware diagnostics, man/machine interface database management, etc.	varying

Table 3. Methodology Modules

NO.	TITLE	DESCRIPTION	DURATION
M101	Software Engineering for Managers	software life-cycle, top-down concepts, documentation, testing	1 day
M102	Introduction to Software Engineering	life-cycle, top-down concepts, overview of various methodologies	2 days
M201	Software Engineering Methodologies	thorough coverage of major methodologies	1 week
M202	Overview of a Specific Methodology	overview of an organization's selected life-cycle methodology	1/2 day
M301	Requirements Methodology	requirements definition techniques and methodology	1 week
M302	Design Methodology	how to do design, with required	4 days methodology
M303	Coding Methodology	structured programming, coding standards, programming style, etc.	2 days
M304	Software Review Methodology	Walkthroughs, code reading	1 day
M401	Introducing Ada to Your Organization	how to use the recommended curriculum to meet specific needs	1 day
M402	Psychological Aspects of Retraining	techniques for overcoming resistance to change	1 day

Table 4. Module Completion Dates

MODULE/COURSE	LENGTH (IN DAYS)	AVAILABLE TO TEACH
L101 Ada Orientation for Managers	1	Now
L102 Ada Technical Overview	1	Now
L103 Introduction to High Order Languages	1	Now
L201 Ada For Software Managers	3	Now
L202 Basic Ada Programming	10	Now
L301 Using The Ada Language Reference Manual	2	Now
L302 Use of Ada for Requirements	2	April 84
L303 Realtime Concepts	1	April 84
L304 Ada Reader's Course	1	March 84
L305 Advanced Ada Types	10	Now
L401 Realtime Systems In Ada	10	May 84
M101 Software Engineering For Managers	1	Now
M102 Introduction To Software Engineering	2	Now
M201 Software Engineering Methodologies	5	Now
M303 Coding Methodology	2	April 84
E300* ALS Users Course	10	April 84
E402 ALS Administrator's Course	3	April 84
*E300 encompasses the majority of the E modules.		

become productive. A designer, on the other hand needs to progress on to L305, and the real time system programmer/analyst/designer should take L401.

For top level managers, L101 is appropriate. For senior QA personnel, program monitors, software managers, etc. L201 is appropriate.

3.4 Search for Related Courses

A language course without parallel courses in methodology and environment is like a car without an engine. The only reason that language courses appear in isolation is because different organizations use different methodologies and different environments. It is also possible that an organization may be proficient in software engineering and only need training in a specific language.

The point is that once a main course is identified, look for related courses. For all intensive purposes, the following courses are indivisible.

- L101 and M101
- L102 and M102
- L202 and M303
- L305 and M302

Additionally each pair listed may be taught sequentially or in parallel.

Ideally each of the pairs should be complemented by a third course from the environment. Currently environment courses are fewer in number. As a consequence L202 is usually supplemented with a brief introduction to the basic tools needed to develop homework assignments.

3.5 Do Not Forget the Prerequisites

One common mistake is to focus on the modules that include exercises to be coded and executed, such as L202 and L305. These two modules have prerequisites.

Another common mistake is to select the "meaty" modules, such as L201 and M201. Unless it is guaranteed that the students have the prerequisites, the students should take a few low level modules first.

3.6 Consider an Acceleration

In many cases an organization is faced with stringent time constraints. In these cases several of the modules in the low level may be compressed, with additional explanations and/or exercises

supplied, where appropriate, in the higher level modules. As a general rule however, acceleration is not a recommended practice.

3.7 A Couple of Exceptions

The curriculum has a few exceptions. For example L301, Reading the Reference Manual, can be viewed as a stand alone course. A solid understanding of Ada is really the only prerequisite for this course. How that understanding has been acquired is not relevant.

The prerequisite for M201 is really a good solid understanding of software engineering. Again, how this has been acquired is not relevant.

M303 can be taught after L202, during L202, or in parallel with L202.

Section 4

ISSUES NOT ADDRESSED BY THE CURRICULUM

The curriculum does define a set of precedences among the modules, as shown in Figure 1. The intended interpretation of Figure 1 is as follows: inputs to a given module define the prerequisites for that module. The Figure does not recommend paths through the curriculum. The line from L202 to L305 means that if there is interest in L305, L305 should be taken after L202. The line does not mean that after taking L202 an individual must procede to L305.

The curriculum does not state how these modules are to be packaged into a course. There are distinct courses that belong together. However each organization has its own needs and therefore will create its own "packaging" of modules into a course.

The curriculum does not state how much training is required by each individual within an organization or the total set of skills to be taught within an organization.

The curriculum does not address specific contents of modules that are particularly sensitive to a specific organization, e.g., M301, M302, M303, and M304. These modules are defined in general outline only and can be adapted to any methodology. The same holds true for some environment modules.

Section 5

DEVELOPING AN ADA TRAINING PROGRAM

The curriculum is not the starting point for a complete training program. The real starting point for an organization in developing a training program is to analyze the training requirements in terms of

- number of levels to be trained
- number of individuals at each level
- level of expertise required for each individual
- current skill level
- customization of course materials
- cost and time constraints
- supplemental training materials desired

Once these issues have been analyzed, courses must be scheduled. Generally it is effective to train managers before staff, designers before implementors.[2] Management commitment to the concepts of Ada is essential to its acceptance by employees. Designers can be designing while training the implementors is taking place. The implementors have the support of the designers and have the motivational level required.

Section 6

SUMMARY

Transition to Ada is a non-trivial process requiring a great deal of thought. Many individuals may find they are transitioning to Ada without really realizing it. The potential for the U. S. Army Model Ada Training Curriculum to aid in this transition is unlimited.

ACKNOWLEDGEMENTS

A great deal of this paper has been excerpted from material contained with the Preliminary Design Review documentation. That material was conceived and written by Nico Lomuto.

[2] Ibid.

Putnam P. Texel received a B.A and M.S. degree in Mathematics from Fairleigh Dickinson University.

She has been heavily invloved in the development of and instruction in U.S. Army Model Ada Training Curriculum. She is currently responsible for coordinating all instructional activities in Ada for the Federal Systems Division of SofTech, Inc.

Ms. Texel is Chairman of the Greater NY Area Local AdaTEC, a local Special Interest Group on Ada affiliated with the ACM Princeton, NJ chapter.

CONFIGURATION MANAGEMENT WITH THE ADA* LANGUAGE SYSTEM

Richard M. Thall

SofTech, Inc.

ABSTRACT

Three characteristics of large software projects and five basic configuration management capabilities are identified. The design of the Ada Language System (ALS) is then described in terms of these basic capabilities. The ALS is a computer programming support environment for Ada.

Section 1

INTRODUCTION

The emergence of software engineering as a distinct discipline has fostered examination of the methods used to program computers. This, in turn, has led to the development of a number of unified environments to aid programmers and improve their productivity. Many of these environments have viewed the programmer as an autonomous individual producing self-contained software. However, in most industrial, military, and commercial applications, it is much more reasonable to view the programmer as a member of a team producing software that must be precisely matched to the software produced by other members of the team. This fact has been acknowledged in the Ada programming language, where emphasis has been placed on the production of an entire coordinated software system rather than a collection of loosely coordinated modules. The Ada Language System (ALS) is a programming environment that supports the development of large systems in Ada. The ALS provides the underlying facilities necessary to coordinate programmers working in teams. The ALS was developed by SofTech, Inc. for the U.S. Army using the Stoneman Requirements [BUXT] as a guideline.

This paper first identifies three major aspects of large team-oriented projects which differentiates them from small one-man efforts. The ALS features which support such projects are described.

Section 2

CHARACTERISTICS OF LARGE SOFTWARE PROJECTS

Large team-oriented software efforts have three characteristics which differentiate them from small individual-oriented projects.

- Large projects are usually developing a family of similar programs rather than a single program.
- Configuration management is of critical importance.
- Close coordination of many programmers is necessary.

Although the discussion of these issues is separated, they are all heavily interrelated.

2.1 Families of Programs

Software is aptly named. It is the soft part of any computer system; it is the most malleable, easily changed part of the system; it is the part that is expected to adapt to changing requirements and changing hardware. In fact, the software is often specifically designed to be adapted to differing situations. It is the part that can be altered most rapidly at the least expense, provided changes are made in an orderly fashion. Even a perfect piece of software with no errors will still tend to accumulate changes for the following reasons:

The work described in this paper is being performed under US Army CECOM Contract No. DAAK80-80-C-0507.

This paper is a revision of a paper entitled "Large-Scale Software Development with the Ada Language System" which appeared in the Proceedings of the ACM Computer Science Conference, February 1983.

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

- the requirements of the original application have changed,
- the hardware configuration has changed, or
- the software is to be incorporated into a new application.

A change in the original requirements can result from a change in the external world or the identification of a shortcoming in the requirements as originally conceived. Hardware changes occur for many reasons: the correction of hardware problems, improved capacity and performance, reduced cost, production and supply problems, etc. Software is often designed at the outset to run on a variety of hardware to accommodate various sized applications. In general, each hardware configuration requires a different copy of the software even though the difference in the software might be as minor as the adjustment of a compile-time constant. Finally, bits and pieces of software tend to migrate from one application to another, from one computer to another, changing in some way each time a migration occurs.

Every change to a software component that can affect its operation must be regarded as creating a new component with different properties. It is a serious error to assume that the significance of a change is related to the amount of source text altered. A single character alteration can be just as devastating to the final operation of a system as a 10,000 character alteration. However, programs differing textually by only a small amount are related and should be treated as such. It is important to maintain the identity of such families of programs because an error in one member of the family is likely to exist in many members of the family. Members of a family may also be textually unrelated; e.g., they may be coded in different programming languages. However, if they are functionally similar, they may share errors. A program family may be loosely defined as those modules which have evolved from a common source text. The source text is often, but not always, the compilable text of a program; it may be the definition of an algorithm or pseudo-code. In general, the members of a program family will all perform similar functions [CARG] [TICH]. The notion of a program family is supported in the ALS by a database feature called a "variation."

Program families inevitably arise wherever there must be ongoing software support for multiple field installations. Unless the field installations are all identical and never change, there will be differing software for the various hardware configurations. Given the rate of change in the computer industry, it is inconceivable that any product would not undergo design changes for cost reduction alone. Many classes of products can be expected to undergo continuous field upgrades which require software alteration. The ability to control families of software may reduce the need to apply field upgrades to bring hardware into conformance with a standard. With strong support for program families, the software could be custom-generated to adapt to each hardware configuration.

2.2 Configuration Management

Configuration Management (CM) is the "consistent labeling, tracking, and change control of the ... elements of a system" [BERS]. There are a number of economic and technical forces which mandate increasing emphasis on CM for industrial grade software. Among these are:

- reliability requirements,
- complexity, and
- ongoing support requirements.

A growing number of computer applications have exceedingly high reliability requirements. In such applications as aircraft and spacecraft control, automotive control, weapons control, and medical systems, software failure can result in personal injury or loss of life. In such cases, strong CM is necessary to ensure that all operational software has been fully tested and that unproven alterations do not find their way into delivered systems. In very complex systems, the change control aspect of CM is used during development simply to ensure that the elements of a system are kept stable enough over time to be successfully integrated. In applications where software corrections and improvements are to be provided on an ongoing basis to remote field locations, CM is necessary to assure that delivered software is appropriate to the hardware configuration at that site.

CM is usually achieved by the creation of one or more "baseline" copies of the software. Each baseline has some official status. There can be working baselines updated frequently by the programming team, frozen baselines preserving exact copies of software delivered to field locations, etc. CM is obtained by management review of proposed changes to baselines and monitoring and recording of actual changes. In order to perform CM, one must be able to:

- absolutely identify the elements of a baseline at any point in time,
- absolutely identify the elements of a system placed in revenue service,
- account for and control all changes to a baseline,
- recreate exactly a system that existed in the past or exists at a field site, and
- control the correspondence between tested and delivered systems.

Even though companies and projects have diverse methods for performing CM, there are five capabilities basic to all CM. These are:

- absolute identification,
- change identification,
- change tracking,
- inventory control, and
- access control.

Absolute identification is the ability to reliably associate a name with a component of a system, usually stored in a file. When a component changes, no matter how small the change, a new component with a different name is created. Change identification is the ability to readily recognize when a change has occurred. Change tracking is the ability to record and review the sequence of changes to a component. Inventory control is the ability to record exactly what components constitute a system at any point in time. Finally, access control is the ability to guarantee that all changes to a baseline are authorized and documented.

A capability fundamental to all CM is absolute identification of software components. Most conventional file systems fail to provide the basic underlying support for absolute identification. Typically, the source code for a component, say a sine routine, is stored in a file that might be named SINE.SRC. Another file, SINE.OBJ, usually holds the object code. SINE.OBJ might be bound into any number of executable images with unrelated names. CM problems occur when a change is made. Typically, the change is introduced by in-place editing of the source file. The name of the source file remains unchanged after the alteration. A revi- sion history may be part of the source file, but the person inserting the change may not possess enough self-discipline to note the change, par- ticularly when the change is viewed as minor. The altered source is subsequently recompiled with the new object replacing SINE.OBJ. Since the file name is not altered, the change is invisible to programmers incorporating SINE.OBJ in their systems.

If the change engenders an unexpected problem, identification of the problem may be very time-consuming. In general, recompilation from source followed by file comparison is necessary to determine if a change in SINE.SRC was ever applied to SINE.OBJ. Determining if a given system has the new or old version of SINE.OBJ involves keeping explicit records of when the change to SINE was applied and when the system in question was last rebuilt. Such records are seldom kept. Partial rebuilds of systems complicate the situation even further. Very often it is easier to correct the present system than reconstruct a clear historical picture of what caused the problem. If an erroneous change finds its way into many systems, there can be many parallel efforts to identify and correct the same error.

A major part of this problem can be alleviated by incorporating a revision number in file names. Every time a file is changed, the revision number is incremented. Identification of changes is then readily accomplished by recording the names of files built into a system. The differences between two builds of a system can then be quickly identified by comparing the revision numbers of the components. Such visibility of changes is fundamental to the notion of absolute identification. Every change or closely related group of changes must be viewed as creating a new object with a distinct name. In short, the name must identify the object abso- lutely. (A single object may have several names, but one name must refer to a unique object throughout the lifetime of the name.)

A revision numbering capability supplies, at one stroke, both change identification and tracking mechanisms. As long as a new revision is created every time a change is made in a baseline, changes are easily identified by the high visibility of new file revisions. The numbered sequence of revisions for each file provides a change tracking mechanism upon which tracking mechanisms for entire baselines can be readily constructed.

Even with revision numbers, problems arise in absolutely identifying components when names have been changed. Component names should be highly mnemonic. But it is desirable to allow mnemonic names to be changed so they stay mnemonic as software development progresses. This broaches the possibility of renaming or deleting a file and then creating another file with the old name. This can result in two distinct components having the same name and revision number. To avoid any possibility of confusion, it is necessary to have a secondary naming mechanism where renaming and reuse of names is not possible. In the ALS, these secondary names are called unique identifiers. The mnemonic quality of unique identifiers is sacrificed for the uniqueness property. To absolutely identify a component, it is desirable to record both the mnemonic name and the unique identifier. The mnemonic name, while not absolutely necessary, helps human users. The unique identifier is used mostly by configuration control tools, to avoid the ambiguity which could otherwise arise if mnemonic names were used to compare configurations.

Inventory control is the ability to create and store a complete list of all the components of a baseline at some point in time. Saved inventory lists can be subsequently compared to

determine the changes in a baseline over some interval, or find the differences between an installed system and the current baseline.

Access control is the ability to guarantee that no undocumented or unauthorized changes find their way into a baseline. The term can be more broadly interpreted to encompass examination as well as modification of baselines. Of course, no guarantee can be absolute. Most computer systems can be penetrated by sufficiently clever and malicious users. In addition, hardware or software failure can always compromise access control. It is assumed, here, that ALS users are friendly and the hardware and software are sufficiently reliable.

The problem of supporting many installations with slightly differing configurations requires CM for families of programs. The inability to do this effectively usually results in software which must dynamically reconfigure itself or which must be completely rebuilt at each field site. CM mechanisms must be able to deal with conditional compilation or macro expansion techniques used to generate family members. In the ALS, revisions and derivations combine with variations to support CM for families of programs.

2.3 Coordination of Programmers

In multi-person projects, the effective coordination of programmers is vital. Lack of coordination results in costly redesign and retrofits during system integration. In complex projects, the lack of adequate coordination can jeopardize the successful conclusion of the project. An official working copy or baseline of the software is usually used to coordinate the efforts of the programmers. There is a spectrum of scenarios for using such a baseline. At the ends of the spectrum are:

- the total sharing scenario, and
- the private copy scenario.

Under total sharing, all incremental changes are immediately applied to the working baseline. The system is periodically rebuilt from the working baseline. Such a rebuild or relink usually occurs frequently, on the order of once or twice a day. Under the private copy scenario, each programmer has his own copy of the baseline which he can modify or rebuild at will.

The problem with sharing is that programmers interfere with each other, each making changes that affect the other. Much time is lost keeping up with alterations made by other programmers. Changes tend to proliferate, one change engendering others which engender still more changes. The rate of change and lack of testing of changes seriously reduces the chances of obtaining a system that works correctly. Sharing allows for little programmer freedom or the opportunity to apply changes experimentally. Even in a two-programmer team, total sharing may be unworkable.

The private copy scenario solves the problems of sharing, but does not provide any coordination. With private copies of the baseline, each programmer works independently. The longer a programmer works in his private area, the greater is the chance that his software diverges from software developed by others. On the other hand, the programmer has total freedom to make changes, even to components which are the purview of others. A programmer working with an isolated copy of the system does not benefit from improvements introduced by other team members.

In practice, some combination of the two scenarios is used to prevent the divergence of the software. Typically, this involves the use of private work areas for incremental development. When an element is completed and tested, it is then integrated with the official baseline. The integration is very often performed in a private area and the new system tested before it is placed in the project baseline. In addition, there are often administrative procedures for controlling baseline changes and for preventing changes to components one is not authorized to change. The ALS provides a general sharing mechanism as well as conventional copying to facilitate almost any scenario for programmer coordination. In addition, the Program Library services of the ALS gives programmers a totally isolated work area for building and modifying systems starting from a baseline copy.

Although the examples in this paper are confined to the source and object forms of computer programs, the discussion is equally valid with a more comprehensive interpretation of the word "software." The same problems occur with all types of documentation, e.g. requirements specifications, design specifications, user reference manuals, tutorial materials, etc. All of these should be included under the umbrella of the term "software." Similarly, although the discussion is illustrated by examples of software in the development phase, all arguments apply equally well to the maintenance phase of the software life-cycle. Indeed, there is no qualitative difference between the development and maintenance phases in relation to the issues treated here.

Section 3

ALS CAPABILITIES

This section describes the features of the ALS specifically designed to support large-scale programming projects. The user's view of the ALS database is presented in some detail. The use of these capabilities as they relate to

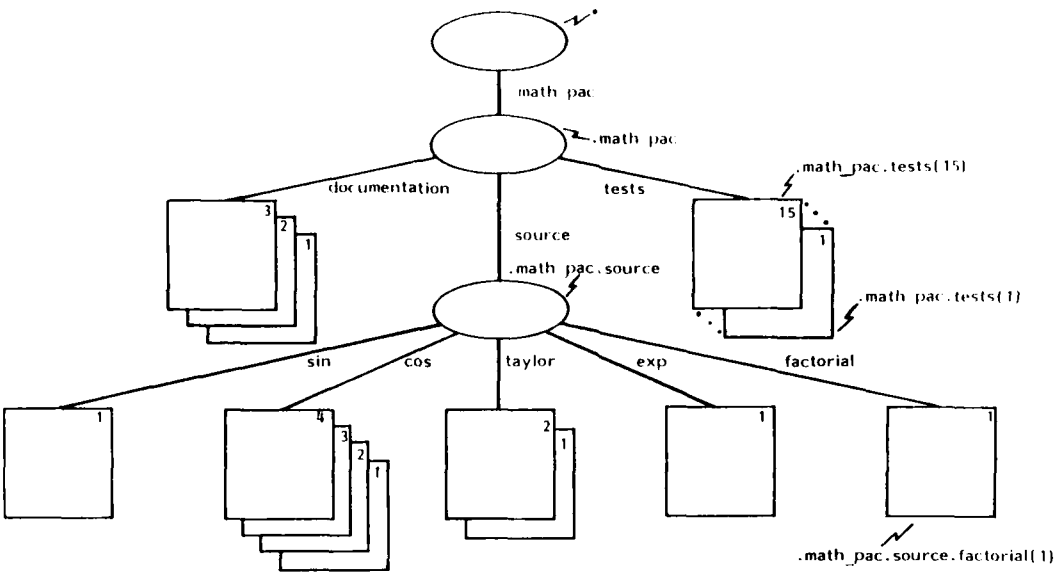


Figure A. Directories and Revision Sets

program families, CM, and programmer coordination is treated in the next section. Information on ALS features not related to CM can be found in [WOLF] and [THAL].

3.1 Nodes

The ALS provides users with a database capability that can be viewed as either a sophisticated file system or a rudimentary database management system. The database is a collection of objects called nodes. There are three varieties of nodes:

- files,
- directories, and
- variation headers.

Files correspond to the usual notion of a named data collection. Directories and variation header nodes are used to create groupings of nodes. All nodes in the database possess descriptors called attributes and associations. An attribute describes the node which possesses it. Associations establish relationships between nodes in addition to the relationships established by virtue of the groupings under directories and variation headers.

3.2 Node Naming and Structuring

Hierarchical data structures are built by using directories to group nodes. Directories are used to group any combination of files, variation headers, and other directories. Figure A gives an example. Every ALS has exactly one connected file structure with one root directory. (Strictly speaking, the root node is anonymous; however, it can be referenced with the name ".".) The root node of our example possess a single node named "math_pac". The reader is free to think of node names as being properties of either the node or the link to the node. However, because of node sharing, a single node may acquire aliases. Thus, it is more accurate to think of the names as properties of the links. Putting it another way, the name resides in the parent directory, not in the node itself. To avoid any ambiguity, the diagrams show node names on the links. In the example, "math_pac" is the child (or offspring) of "." which is the parent of "math_pac". A parent node is said to contain its offspring.

The "math_pac" directory has three offspring, the directory "source" and the files "documentation" and "tests". "Source" in turn, has five offspring: the files "sin", "cos", "taylor", "exp", and "factorial". Directories are shown as ellipses; and files are shown as squares. Just as in Ada, the identity of an object depends upon its position in the whole structure. The full name of an object is known as the pathname and is constructed by tracing the path to the object from the root and naming the links traversed along that path. The pathname of math_pac is ".math_pac". The name of the factorial subprogram is ".math_pac .source.factorial". Several pathnames are shown in Figure A. Users are encouraged to view the data structure as a tree; however, due to sharing of nodes, the structure is not strictly

a tree, it is a directed acyclic graph. The ALS excludes cycles from the structure. In other words a directory may not contain a subtree that contains that same directory.

3.3 Revision Sets

Every file in the ALS database is, in actuality, a member of a revision set. The revision set tracks the changes made to a file over time. Each member of a revision set is a snapshot of the file as it existed at some point in time. The members, called revisions, are ordered in chronological sequence and are automatically numbered in order starting from one. The most recent revision supersedes all previous revisions. Although a revision is most often a modified form of the previous revision, this relationship is not imposed. In some cases, a revision may come from a revision that predates the immediate predecessor in the same revision set or from some other source entirely. Most operations such as opening, reading, writing, and deleting, apply to individual revisions of a revision set. Sharing, however can only be accomplished for the revision set as a whole. If the last revision of a set is deleted, the number is not reused when the next revision is created.

To provide absolute identification, in-place editing of revisions is restricted. Only the latest member of a revision set may be modified in-place, and then only under certain conditions. The most recent revision supersedes all previous revisions. The latest revision can also be explicitly frozen, after which it may not be modified. A revision can become unmodifiable for three reasons:

- it was explicitly frozen by the user or a program,
- it is not the latest revision, or
- it has been used to generate another object which is under configuration control.

Only in the last case, when the derived object is removed from the database, can an unmodifiable revision again become modifiable. In the other cases, the action of freezing is irrevocable. In the first two cases, the revision is said to be frozen. Unmodifiability applies only to the text of a revision; it does not limit changes to attributes or associations. Each revision possesses a distinct set of attributes and associations. Revisions from which files under configuration management have been derived may not be removed from the database until the derived file has been removed.

Any revision can be named by attaching a parenthesized revision number subscript to the pathname of the file. If no subscript is given, the latest revision is assumed. If the subscript "+" is specified, the latest frozen revision is referenced. The latest frozen revision is either the last revision or the next-to-last revision. The use of subscript notation promotes the view that the revision set is an array possessing elements that are the individual revisions. Figure A shows the pathnames of three different revisions.

3.4 Unique Identifiers

The ALS automatically assigns each node an identifier which is temporally and spatially unique. In other words, once assigned, no other node in any other ALS database will ever have the same identifier, unless it is a copy of the original. Moreover, once assigned, the identifier cannot be changed. These identifiers are called unique identifiers or UIDs. UIDs have three fields:

- object serial number (10 bytes),
- ALS database identifier (7 bytes), and
- organization identifier (10 bytes),

An object serial number is assigned automatically by the ALS each time a node is created. To that is appended the database identifier which is unique for each database within an organization. Finally, the organization identifier, naming the organization owning the database, is appended. Database identifiers are administratively assigned by a specifically appointed person within each organization to which the ALS has been delivered. Organization identifiers are assigned by the government agency responsible for configuration management and distribution of the ALS. The name space is large enough to allow the creation of 10,000 nodes per second for the next 2.6 million years in each of 8 trillion databases in each of 1400 trillion organizations. With simple compression techniques, only 10 bytes out of the full 27 bytes would have to be stored for each node.

The ALS supplies tools for copying nodes from one ALS database to any other ALS database. When files are copied in this way, the original and the copy are automatically frozen. In the receiving database, copies are created with the same UIDs as the original. It is therefore possible to compare baselines on two hosts by comparing only the UIDs of the files in the baselines. Because both the original and copy are frozen, there is a reasonable level of confidence that the files are the same. Without this capability, it would be necessary to transmit the entire contents of all the files in the baseline to one of the hosts where an exhaustive file comparison would have to be run. Recording the UIDs of files from which an installed system is built provides a similar level of control for delivered software which may not reside on a host.

3.5 Variation Sets

To represent families of programs, the ALS provides a construct called the variation set. Members of variation sets are functionally similar software components that differ in their implementation details. Since variation set members do not supersede one another, they are named, not numbered. Nodes called variation set headers are used to represent variations sets in the ALS database. A variation set header can occur anywhere a directory node can occur except at the root of the database. The members of a variation set appear as offspring of the variation set header node. The members can be revision sets, other variations sets, ordinary subtrees (i.e., directories), or any combination of these. A default variation can be designated.

Figure 8 shows an example of the use of variation sets. Variation set headers appear as hexagons. In this example, the source for math_pac exists in two variations, one for integer hardware and another for computers with floating point hardware. The floating point variation is further divided into variations for long words and short words. Variation set headers are similar to directories except that in pathnames, references to their offspring appear in parentheses rather than being separated by dots from the preceding path element. In this respect, the members of a variation set are viewed as array elements, where the elements are named rather than numbered. Figure 8 shows how pathnames with variation references and nested variation references are formed. If empty parentheses are specified and a default variation has been designated, the default variation will be selected.

3.6 Node Sharing

To ensure that the ALS can readily support many scenarios for programmer coordination, there is a sharing mechanism in addition to the usual copying capability. Any node may be shared provided the sharing does not introduce a cycle into the database structure. In essence, sharing a node creates an alias for that node. A node may have two kinds of parents, true parents and foster parents. A true parent is the directory (or variation header) in which the node was originally created. Every node has exactly one true parent. A foster parent is a directory (or variation header) that subse-

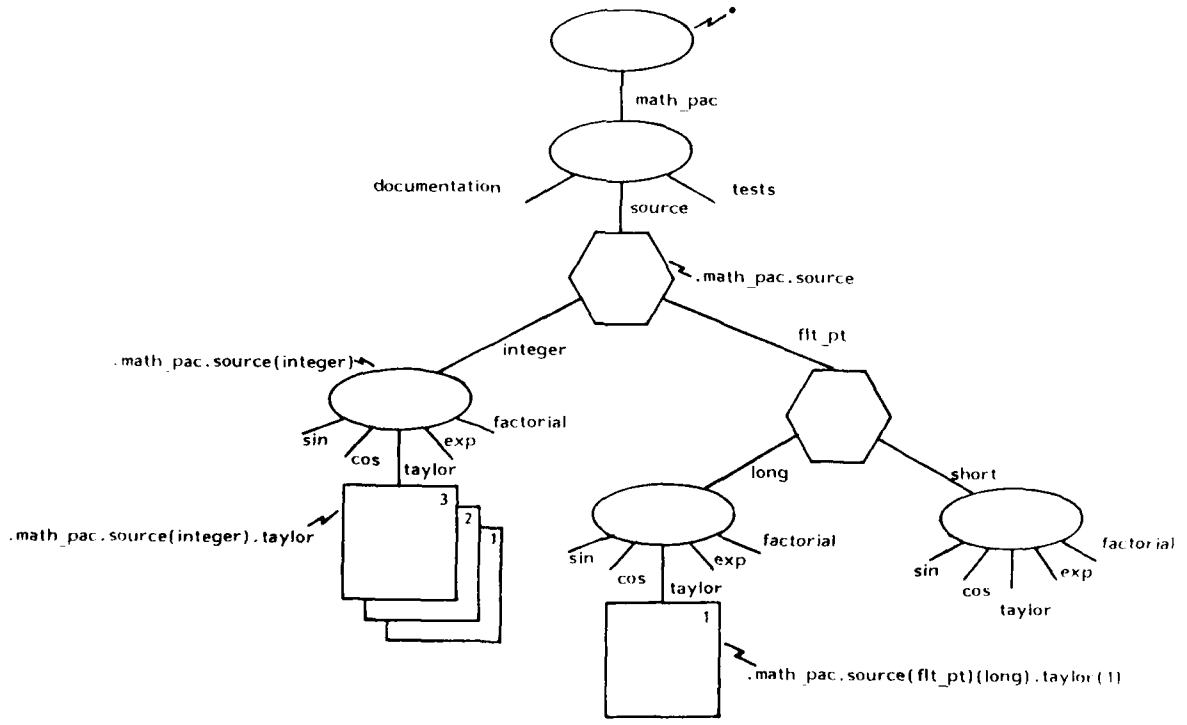


Figure 8. Variation Sets

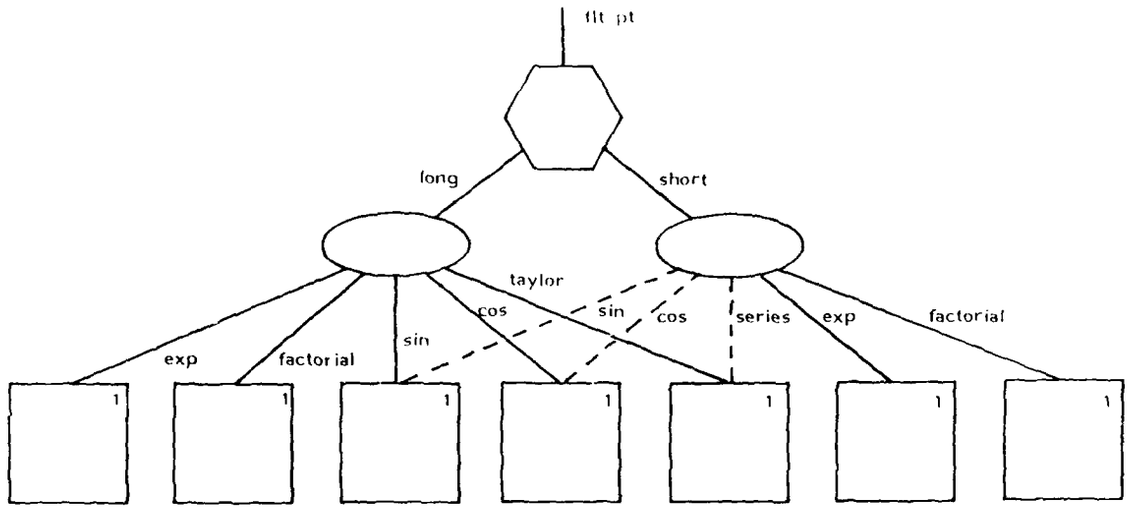


Figure C. Node Sharing

quently shares an existing node. A node may have an arbitrary number of foster parents. Figure C shows a node sharing situation. In this case, the short word length variation of math_pac shares the "sin", "cos", and "series" files with the long word length variation. Notice that the taylor series procedure has two names, ".math_pac.source(flt_pt)(long).taylor" and ".math_pac.source (flt_pt)(short).series". It is the same revision set, but shared with a different name on the link. Individual elements of a revision set cannot be shared, only the whole set. Directories and variation headers may also be shared.

3.7 Attributes

An attribute is a named character string used to describe the node which possesses the attribute. A node may have an arbitrary number of attributes. The ALS uses certain attributes to control the database and restricts the use of these attributes. Programs can create, delete, and modify any other attributes, subject to the normal access controls. There is no global list of attributes or registration procedure for attributes. Other than the attributes used for database control, there are no attributes that every node must possess. The values of attributes are strings which can be up to 64K characters long.

Attributes can be used to select variations. This is accomplished by giving a sequence of (name=>value) pairs in place of the variation name. The pairs are separated by commas. Figure D show an alternate organization for the example of Figure B. In this case, instead of nesting variations, there is only one variation header with variations named "va", "vb", and "vc". Variation "va" of "source" has an attribute named "mode" with a value "integer". Variation "vb" has an attribute named "mode" with value "flt_pt" and another attribute named "size" with value "long". Finally, variation "vc" has an attribute named "mode" with value "flt_pt" and an attribute named "size" with value "short". Figure D shows two examples of variation selection with attribute values. Additional variations and selection attributes can be added dynamically as the software configuration evolves. If attribute selection is used, the specification must select a single variation unambiguously.

3.8 Access Control

ALS access controls are based upon a conventional lock and key mechanism. Users and programs have keys and database objects have locks. The user and program keys must match the appropriate lock in order to obtain access. Attributes are used to store the locks and keys.

Each user has two keys: a user name and a team name. The user name is determined when the user enters the ALS from the host operating system. The team name may be chosen by the user from a roster of team names and team members controlled administratively. The key of an executing program is obtained from an attribute

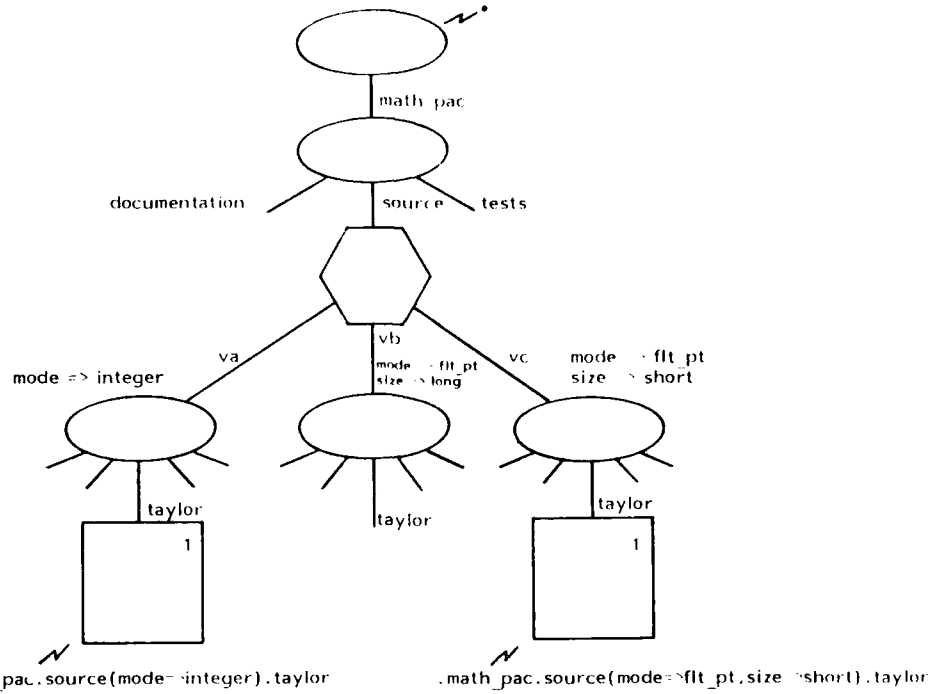


Figure D. Attribute Variation Selection

named "access_name" attached to the ALS file where the executable image of the program resides.

Locks are attached to each database object with attributes named read, append, write, attr_change, execute, and via. The values of these attributes are lists of the keys which will satisfy the lock. The lock can be satisfied by either the team name or the user name. An asterisk can be used to match a substring of all keys. For example, the lock "*Smith" will match all users with a key ending in "Smith". The key "*" matches all keys. If the read lock is satisfied, then the user may examine the file or may learn the offspring of a directory or variation header. If the append lock is satisfied, then the user may add to the end of a file, or add entries to a directory or variation header. If the write lock is satisfied, then the user may change a file or add and delete entries of a directory or variation header. If the attr_change lock is satisfied, then the user may alter the values of attributes and associations and add and delete attributes and associations. If the execute lock is satisfied, the user may place the executable image or command script into execution.

The via lock allows the creation of database objects that can be accessed only by programs intended for that purpose. If the via lock is not empty, then the key of the program used to access the object must satisfy the via lock. Even if the via lock is satisfied, either the user name or the team name must still satisfy the lock appropriate to the type of access desired. If the via lock is null, any program may be used, provided access is otherwise granted. This feature is utilized, for example, to prevent the user from altering object code produced by the Ada compiler.

3.9 Associations

Associations are similar to attributes, but used to document the relationships between nodes. The value of an association is a list of pathnames. The ALS ensures that the elements of the list are syntactically valid pathnames, but otherwise performs no validation or maintenance on the list. An example of the use of associations is the Ada compiler which records the names of previously compiled modules referenced during a compilation in an association named "depends_on". This association is subsequently

used by the linker to enforce the Ada compilation ordering rules by checking that no module named in a "depends_on" association has been compiled later than the module which possesses the association.

3.10 Derivations

The Stoneman calls for the generation of detailed histories of objects under configuration management. The ALS does this by means of derivations. Any ALS file can, potentially, possess a derivation. A derivation is a combination of attributes and associations that document the circumstances under which a file was created or modified. Comparison of derivations shows why files differ rather than the exact text of the differences. Although derivations are not intended or used for database backup, they contain enough information so that the contents of a file can be exactly recreated from the derivation if the files named in the derivation exist.

Files in the ALS database can only be created or modified during the execution of some program called the creating tool. The derivation is an accounting of the conditions under which the creating tool executed. The name of the program, the parameters passed to the program, and files opened and read by the program are automatically recorded in the derivation. The creating tool can modify the derivation based on specific knowledge that a particular input is insignificant or that some other unrecorded information is significant. The ALS internally maintains the information required for derivations. Whenever an output file is closed, the information is posted, if derivations have been enabled by the creating tool.

A derivation consists of the attributes derivation_text and the associations logged_inputs, derived_from and other_inputs. These attributes and associations collectively constitute the derivation. Derivations are controlled by the KAPSE and cannot be modified except by the creating tool. The functions of the derivation components are:

derivation_text

> This attribute conveys the name of the tools that created or modified the file, the parameters passed to those tools, and annotations posted by those tools.

logged_inputs

> This association lists the pathnames of files that were opened and read by the creating tool. References in this association engender the incrementation of the derivation_count of the named file.

derived_from

> This is a special association that contains, not pathnames, but the unique identifiers of the files named in the logged_inputs association. By using derived_from, the files named in the derivation can be found, even if they have been renamed. This is used by the ALS when decrementing derivation_counts.

other_inputs

> This association lists the pathnames of files that were open and read by the creating tool, but were not entered in the logged_inputs association because the citation was explicitly suppressed. References in other_inputs do not engender incrementation of the derivation_count of the named file.

Files which have been named in the logged_inputs association of the derivation of one or more other files possess a cited_by association and a derivation_count attribute. Cited_by contains the UID's of the files that name this file in their derivations. These are the back-links of the derived_from associations. In other words, cited_by refers to those files that have been created from the file possessing the cited_by association. Derivation_count is, simply, the number of entries in the cited_by association. Cited_by and derivation_count are managed automatically by the ALS and are not subject to direct alterations by tools or users. Entries in cited_by are removed when the named file is deleted. A revision cannot be deleted if it possesses a positive derivation_count. Since this makes deletion very complicated, the use of derivations is recommended only for baseline objects under configuration management.

Section 4

USE OF ALS FEATURES

This section describes how the features of the ALS can be used to overcome some of the problems faced in large-scale software efforts. For discussion purposes, the use of variations will be illustrated in the context of providing support for program families; the use of revisions, access control, and derivations will be outlined in relation to CM; and the use of sharing will be couched in the discussion of programmer coordination. However, in reality, the partitioning is not as clear. Variations are also necessary for CM; access control is necessary for programmer coordination; and all aspects of CM are intimately related to programmer coordination.

4.1 Program Families

All changes to software components fall into two classes:

- changes that make previous versions of the component obsolete, and
- changes that do not cause previous versions to become obsolete.

The first class of change is called revision; the second is termed variation.

Examples of changes of the first class are error corrections. Once an error is discovered and corrected, there is no reason, other than historical investigation of failures, to use old, erroneous, versions of a component in any new systems. The latest version supersedes all older versions. Revision sets are used to represent this type of change in the ALS database. Revision does not give rise to families of programs. If all components are changed by superseding the previous revision, then at any given time, there is only one current copy of the software incorporating all of the latest revisions of all components.

Examples of changes of the second class are changes in the function or implementation of a component. One common source of variation is testing. A program may have some components used only during testing and other, similar but not identical, components used in production variations of the system. In this case, the existence of a test variation does not make a production variation obsolete; the variations legitimately exist simultaneously. An error in one variation may or may not appear in another variation. Variations may also exist because of differences in implementation of identical functions. Our SINE routine, for example, might be coded in any number of languages for different computers. There may be a separate variation for computers that lack floating point hardware, or a separate double precision variation, etc. ALS variation sets are used to represent changes of this type. It is variation that gives rise to families of programs because multiple systems can be constructed by incorporating the latest revision of one or another variation of a component in each of the systems.

Variation set headers mark the places in the software where evolution of the families diverges. Components above variation headers are shared by all members of the family of programs. Components below variation set headers are specific to some subset of family members. In general, it is best to have the variation set headers as low in the structure as possible so that shared components do not appear below variation headers. In this sense, the example in Figure 6 is less than ideal.

A single functional variation often results in many changes distributed throughout the structure of the baseline. If a single variation header were used, it would have to be placed so high in the structure that many common components would appear in the subtree of the variation header. In such cases, it is better to use multiple variation headers for a single functional change. However, each of the resulting variations should either be given the same name, or should all have common identifying attributes. For example, if a variation is introduced in a system to support double precision arithmetic, then all components that are specific to single precision should be named "single" and components specific to double precision should be named "double." Using the variation notation, this would yield names like sine(single), cosine(single), etc., in one case, and sine(double), cosine(double), etc., in the other case. Alternatively, attribute variation selection could be used, in which case the corresponding component names would be sine (precision=> single), cosine(precision=> single), sine(precision=> double), and cosine (precision=> double), respectively. Several attributes can be used for selecting a single variation, e.g., sine(precision=> double, target=> 8086). In this way, variations with different names can be selected with a common set of attributes.

The ALS supplies these capabilities so that tools for constructing individual members of a program family can be readily developed. Such tools would be given the attribute values or variation names to use in selecting components from a baseline containing many variations. The tools would then collect the necessary components and bind them together to form an executable program. Combinations of many attributes and variation names could be used to generate a very large number of family members closely matched to the requirements of individual applications. Such a "custom tailoring" approach to software is often avoided simply because conventional methods for dealing with program families are cumbersome and expensive.

The proper use of variations can lead to substantial cost savings during maintenance. Conventionally, members of a program family are maintained in entirely separate baselines, often by entirely separate staff. This tends to encourage the continued divergence of the family members, even when it is unnecessary. By using variations, a family of programs can be stored in a single baseline. This approach keeps the evolution of the software from diverging to the point where a separate maintenance staff is required. Since all variations are readily visible, grouped under a single header, it is much easier to assess the effect of a software change on all members of a family. It is also much easier to prevent unnecessary divergence and easier to apply error corrections to all appropriate variations.

It is true that the notion of variations could have been supported by using directories. However, it is the author's view that the concept

will only work successfully if programmers are continually reminded of the difference between revisions and variations. Every time a change is introduced, the programmer must decide whether the change is a revision or variation and must use the appropriate structure to apply the change to the baseline.

4.2 Configuration Management

A design goal of the ALS was to provide the underlying database mechanisms to perform configuration management. It was recognized that there are many differing scenarios for CM and many tools that can be implemented to support these scenarios. Rather than impose one method, the ALS supplies the fundamental capabilities which make all CM tools easy to implement. Rudimentary CM tools can be implemented directly in the ALS command language without writing a computer program in the conventional sense. An implementor of CM tools is likely to rely upon the following ALS mechanisms:

- revisions,
- unique identifiers,
- variations,
- attributes,
- derivations, and
- access control.

Revisions and unique identifiers give the ALS user the means to absolutely identify software components. The use of variations has been treated in the previous section. Attributes supply a method of attaching descriptive information to an object. Derivations provide a detailed accounting of how an object was created and why it differs from a similar object. Finally, the ALS access control services give the CM tools flexibility in restricting access to baselines.

Revision sets and UIDs are the keys to ausolute identification; and absolute identification is the key to configuration management. Changes to baselines are made by appending revisions to revision sets and then freezing the latest revision. From then on, the name of the revision, say sine(6), stands for that object and that object only. Any change to sine would result in a new, highly visible, revision named sine(7), which has a new UID.

Revision sets facilitate the comparison of baselines with other baselines and installed systems, two fundamental CM operations. Suppose that there exists a baseline from which is generated a number of variants of a system. The systems are constructed by a tool such as described in the previous section. As a system is constructed, the tool prooduces a component list of the revisions incorporated. For each component, the list contains the full file name, including the revision number and any variation name, and the UID. Every system constructed for testing and every system generated for revenue service has its component list attached. The elements of any system can be readily identified by examining its component list. If a programmer needs to examine the source text of the system, he merely displays the contents of the revisions specified in the component list. Since the revisions cited in the component list are frozen, there is no question that the source text is exactly that used to generate the system. Any change between the given system and the current baseline can be rapidly identified by comparing the revision numbers and UIDs in the component list with the latest revision numbers and UIDs in the baseline. The system can be exactly recreated by extracting from the baseline the revisions cited in the component list. Finally, the correspondence between a test system and a production system can be easily verified by comparing the component lists of the systems.

In addition to system building tools, CM typically entails the creation of many tools for such tasks as installation and accounting of baseline changes, tracking of error reports, tracking of project status, baseline inventory and audit, error diagnosis, etc. Tools of this nature often require auxiliary information about the objects in the baseline, e.g. installation date, author, pending changes, systems in which the object was used, etc. ALS attributes are used to conveniently store such auxiliary information.

Attributes are a method of attaching descriptive information to an object without modifying the contents of an object. Without attributes, there are three choices: modify the object, build auxiliary files to contain the descriptive information, or use naming conventions. Modification of the object is very inflexible since it affects the programs that manipulate the object. This approach leads to such aberrations as highly coded control information embedded in comments in source code. Naming conventions are inadequate for the amount of information necessary for configuration management. If auxiliary files are used, each program that uses them must build and maintain the data structure of the auxiliary file. By providing attributes, much of the data manipulation burden is removed from the configuration management programs. Attribute values can be quite large, up to 64K characters. Attributes are used where the information is to be kept with the object being described. Auxiliary files will still be used where information about many objects is to be collected in one place.

Derivations are required by the Stoneman. In essence, they are a semi-automatic method for incrementally tracking the history of software components. In fact, the Stoneman uses the term

"history attribute." The ALS implementation of derivations is similar to the implementation proposed in the Ada Support System Study completed in the UK [STEN]. A common CM operation is the comparison of components to identify the differences between the previous software that functioned correctly and the current software that malfunctions. Unfortunately, direct textual comparison is often useless. For example, the textual comparison of object modules will usually establish that a difference exists, but rarely yields a clue about the significance of the difference. Textual comparison of source may not be much more enlightening about the relevance of any differences discovered. However, comparison of the derivations of two components can reveal that different revisions or variations of source were used to obtain object modules, or that different compiler options, e.g. optimization, were used in each case, etc. CM tools can post any relevant information in the derivation_text attribute. This might include a component list, or a short description of a change entered by the programmer during an edit operation. This type of information is significantly more useful than textual comparison by itself.

Access to baselines must be controlled to ensure that no unauthorized changes are applied. The ALS uses a relatively conventional paradigm for access control. For CM, the via lock is especially useful. With the via lock, it is possible to create subtrees in the ALS database that can only be accessed through the services of a tool or group of tools. In this way, access to baselines can be controlled by CM tools created for the purpose. Such tools are used to ensure that changes are applied in an orderly fashion, that all recording of changes is duly performed, that changes have been authorized, and so forth. This feature is used, for example, by the ALS Ada compilers to deny users direct access to program libraries where object modules are stored. In this way, the user is prevented from circumventing the recompilation ordering rules of the Ada Language.

4.3 Coordination of Programmers

This discussion will be limited to programmer coordination during the manipulation of source and object code. There are many other aspects of programmer coordination not treated here because the ALS currently provides no specific tools for interface control, design coordination, requirements analysis, etc. Some of these problems are addressed by the Ada language; others will be addressed by tools written for the ALS. It is expected that the features of the ALS already outlined will simplify the implementation of such coordination tools. Many of these tools will follow the CM paradigms established for baseline control.

For source code, most coordination will be done by the use of baselines. Source used by more than one programmer will be stored in a controlled baseline. Any modifications to the source will be accomplished by first locking the code to be modified, performing the modifications and testing them in a private area, then installing the modifications in the baseline, after suitable notice has been given to all interested parties. Locking prevents more than one person from modifying a component simultaneously. It also serves to alert other users that a modification may soon be applied.

The baseline can be used in three ways:

- source files can be copied from the baseline,
- any subtree can be shared, or
- the baseline can simply be referenced.

If source is copied, then the programmer is insulated from any changes that occur. He is also cut off from any error corrections or improvements. If the source is shared, new revisions of the source files will automatically appear in the sharer's area, potentially without notice. If the source is referenced, then there are a number of choices, references to explicit revisions and variations, references to the latest revision or latest frozen revision, and/or references to the default variation. Explicit references provide isolation, general references do not.

Sharing prevents unnecessary divergence of software. In more conventional systems, sharing is accomplished by copying. But once copied, the evolution of software components is likely to diverge because the copy will be overlooked during maintenance. With sharing, there is only one copy to maintain. If changes for one sharer are inappropriate for all, then a variation should be introduced to document the divergence. Keeping all the source logically in the baseline and only referencing it during compilation is a good compromise. Isolation can be achieved by using explicit revisions and variations, but the divergence of evolution is less likely. However, with referencing, deletion of old revisions must be controlled to avoid deletion of source text that is still in use. In some sense, this is an abrogation of the obsolescence property of revisions, and therefore should not be used in place of variations. In other words, explicit revision references should only appear when there is an intent to track the evolution of the source component; otherwise, a variation should be created.

The ALS supplies much stronger support for programmer coordination at the object code level. All Ada object code must be placed in a structure called a Program Library. In general, there is

one Program Library (PL) for each variation of an executable program. A PL is a collection of directories and revisions in one subtree. Via locks are used to restrict access to PLs. Programmers are encouraged to think of PLs as buckets into which they place components of a system. When all components are in the PL, they can be linked together to form an executable program. Revisions are used inside PLs so that one PL can be repeatedly used for recompilation and relinking during the system development. Ada recompilation ordering rules are enforced by all tools that operate on PLs.

Components can be placed into a PL by compilation or by acquisition from another PL. Suppose, for example, that an Ada package exists for trigonometry. The package can be initially compiled into a publicly available PL. Programmers who use the package can then acquire the object code directly without recompilation. This is done by using a tool named LIB, short for library. Acquisition is accomplished by reference, so that duplicate storage of the object code is avoided while maintaining isolation of PLs. Changes in the acquired-from PL do not automatically appear in the acquired-to PL. The addition of a subscription capability is anticipated. With this mechanism, the owner of an acquired-to PL would be notified if any changes were made in the acquired-from PL. He could then reacquire at his option. PLs provide the isolation of copying without the duplication of storage. Acquisition can be done from a baseline to a private PL to establish a private work area. The acquisition mechanism provides a method for easily sharing while still preserving some isolation. The guiding philosophy behind PLs is that neither a baseline nor a private PL can be altered without explicit action by the owner.

Section 5

CONCLUDING REMARKS

A major technical contribution of the ALS is the support for large-scale software projects. The ALS is one of the first production-quality programming environment to offer native, rather than tacked-on, support for configuration management of program families. Specifically, it is the first environment to offer:

- differentiation of revisions and variations,
- explicit named variations,
- freezing of revisions, and
- derivations.

The notion that there is a qualitative difference between revision and variation has been independently proposed by two other investigators, Cargil and Tichy. The ALS will test the value of this model by exposing the idea to a large number of software engineers in production situations. In the author's opinion, the distinction between revision and variation will prove to be a fundamental notion.

REFERENCES

[BERS] E. H. Bersoff, V. D. Henderson, and S. G. Siegel, "Software Configuration Management: A Tutorial," Computer Magazine, January 1979, IEEE, pp 6-14.

[BUXT] J. Buxton, Department of Defense Requirements for Ada Programming Support Environments "STONEMAN;" U.S. Department of Defense, February 1980.

[CARG] T. A. Cargil, A View of Source Text for Diversely Configurable Software; University of Waterloo, Dept. of Computer Science, 1980, 100p.

[STEN] V. Stenning, et al., Ada Support System Study; System Designers Limited and Software Sciences Limited, 1979 and 1980.

[TICH] W. F. Tichy, Software Development Control Based on System Structure Description; Carnegie-Mellon University, Computer Science Department, Jan 1980, 180p.

[THAL] R. M. Thall, "The KAPSE for the Ada Language System;" Proceedings of the AdaTEC Conference on Ada; October, 1982, ACM, pp 31-47.

[WOLF] M. Wolfe, W. Babich, R. Simpson, R. Thall, and L. Weissman, "The Ada Language System;" IEEE Computer Magazine, June 1981, pp 37-45.

LEARNING THE ADA INTEGRATED ENVIRONMENT

George Snyder

Intermetrics, Inc.
Cambridge, Massachusetts

The Ada Integrated Environment (AIE) is designed to be easy to learn and easy to use. It will be powerful, efficient, and friendly. This paper describes how these goals are addressed in the design of the Ada compiler, the MAPSE Command Language, and the Program Integration Facility. Plans for future tools are also described.

1. INTRODUCTION

The Ada* Integrated Environment (AIE) provides support for the development of Ada programs. A good environment should provide the power and flexibility to make program development as easy as possible. It should also be friendly and easy to use.

An environment must meet all these criteria if it is to be easy to learn. Program development tools which are slow or produce poor output discourage the user from learning by experimentation. Lack of flexibility in tools frustrates a novice user, and often force experienced users into arcane methods which are unreliable and difficult to maintain. Friendliness and ease of use help users get started in the environment. However, a user advancing into unfamiliar areas should not be hampered by unneeded

This paper describes the major user interfaces currently being developed for the Ada Integrated Environment: the Ada compiler, the MAPSE Command Language (MCL), and the Program Integration Facility (PIF). The MCL is part of the Minimal Ada Programming Support Environment (MAPSE)

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

2. COMMAND LANGUAGE

The MAPSE Command Language (MCL) [Shenker] is the primary interface between a user and the AIE. The fundamental role of the MAPSE Command Processor is to invoke programs, in response to a user's MCL commands. The overall philosophy of the MCL is similar to that of UNIX (R) [Bourne], a widely used and familiar system. Because MCL syntax is based on Ada end UNIX, users will find it easy to learn.

The MCP uses a "toolkit" approach, whereby a number of generalized tools can be easily interconnected for a particular purpose. All tools are available at the command level, or in scripts containing MCL commands. Because tools are programs, rather than being embedded in the command processor or operating system, the tool set can be expanded or modified. Following this philosophy, the MCP itself is a tool.

Like Ada, MCL may be typed in free format. Because MCL is primarily an interactive language, Ada syntax rules have been relaxed to reduce the typing of punctuation such as parentheses, commas, and semicolons.

2.1 Program Invocation

A program is invoked by typing its name. Suppose there is an Ada procedure:

```
procedure Compile ( Source: string;
                    Library: string);
```

This program could be invoked with a command like the following. Any combination of positional and named parameters may be used:

```
compile  Mysource  Library => Mylib
```

2.2 Pipes

The Ada predefined text input/output files STANDARD_INPUT and STANDARD_OUTPUT can be redirected either to disk files or to other programs via "pipes." Programs connected by pipes execute concurrently. In the following example, Sort reads its input from File1 and passes its output to Unique, which reads the result as its input and places its output in File2:

```
Sort -< File1 -| Unique -> File2
```

2.3 Language Elements

MCL provides a number of Ada-like constructs, as well as all Ada operators. Literals and implicitly declared variables may be of type integer, float, boolean, or string. Quotations around string literals are optional. A variable may also be given an aggregate value; its components may then be specified either by number (as an array) or by name (as a record). Certain attributes are defined for MCP variables, such as 'TYPE and 'LENGTH. The following examples illustrate some of these features:

```
%var1 := 4
%var2 := (A => 9, B => "Series 9")
%result := 5.0 + 3 * (4 / %result)
put %var2'type
put %var2.B'length
```

2.4 Control Structures

Control constructs include if-then-else, case, loop, and begin-end. These constructs may be nested, and may be invoked either from the keyboard or from a script. When a compound command is being entered from the keyboard, MCP prompts with line numbers until the command is completed. A compound command's output may be redirected. The following example sorts a list of colors and places the result in Sorted_Colors (a colon is the normal MCP prompt):

```
: for %color in
  2/ (green, blue, red, yellow) loop
  3/     put %color
  4/ end loop -| sort -> Sorted_Colors
:
```

2.5 Scripts

A frequently used sequence of MCL commands may be saved as a script, resulting in a new tool. A script may specify parameters, like an Ada procedure or function, and the parameters may have default values. A powerful aspect of this approach is that an Ada subprogram and an MCL script are invoked in exactly the same way. Thus frequently used scripts can be converted to Ada without having to change scripts which use them. Here is a script which performs a bubble sort:

```
%Data := ( 5, 2, 4, 0, 1, 3 )

put "Unsorted Data: ", %Data

for %i in reverse 1..(%Data'length-1)
  loop
    for %j in 1..%i loop
        if %Data(%j) > %Data(%j+1) then
            %Temp := %Data(%j+1)
            %Data(%j+1) := %Data(%j)
            %Data(%j) := %Temp
        end if
    end loop
end loop

put "Sorted Data: ", %Data
```

2.6 Help Facility

A help facility is provided, which allows the user to get information about any program or MCP script. Help is also available for a program's parameters, simply by typing a question mark where the parameter would normally be specified.

2.7 Other Commands

Any command may be executed in the background by terminating the command with "-&". The user will be informed when the background command completes. The WAIT command causes MCP to wait until a specified background command completes. A background command can be terminated with the ABORT command.

```
: comp:
  2/ compile Myfile Library => Mylib -&
    COMP EXECUTING
: abort comp
    COMP ABORTED
```

A user may end his MCP session with either LOGOUT or SUSPEND. In the latter case, the session may be later resumed at the state in which it was suspended.

3. PROGRAM INTEGRATION

The purpose of the Program Integration Facility (PIF) is to create & manage program libraries with minimal direction from users. In addition to the usual library support functions, PIF provides configuration management, including version control and automatic reconstruction of library objects.

3.1 Library Support

Multiple libraries can be maintained in the AIE, and one or more libraries may be available to each user. One of the parameters to the Ada compiler is the name of the library into which the compilation is to be placed. If the library does not exist, it is automatically created. More than one user can access the same library, so that members of a team can share program units.

In order to save space, a set of related library units which are frequently used can be stored in a catalog. In order to save space, catalogs can be shared between libraries in a manner analogous to a traditional library of object modules. The interface catalogs (specs) are maintained separately from implementation catalogs (bodies), and multiple implementations of an interface can coexist. Users can specify which interfaces and implementations are to be used for a particular library.

3.2 Configuration Management

The PIF supports numbered revisions and named versions of catalogs. Users can check catalogs out for modification, and check them back in afterwards. Interdependencies of objects in a library are tracked, and objects are reconstructed as needed so that any referenced object is up-to-date.

3.2.1 Version Control Since Ada units are heavily interdependent, maintaining revisions on a unit basis is impractical. A change in one unit's source may cause a change in the DIANA of every unit that depends on it. For this reason, versions and revisions are treated on a catalog basis. A user can link to the latest revision of a catalog, or to a particular revision number.

A version of an object involves significant changes, usually including unit specifications. A new version of a catalog is created by copying it to a new name, and changing a library's links to it. A revision is typically a change in implementation. A user creates a revision of a catalog by deriving from it a catalog with same name but a new revision number. Such catalogs can later be promoted to resource catalogs, and thus made visible to other catalogs.

3.3 Object Reconstruction

The PIF makes sure that every object in a catalog is up-to-date when it is referenced either directly or indirectly. This applies not only to Ada units, but to other kinds of objects. A user can change the rules and tools by which an object is updated.

3.3.1 Initial Form A library object may be present in more than one processing stage, or form, such as "source," "abstract syntax tree (AST)," "DIANA," "object module," "executable," "documentation," etc. Each object in the library has an initial form, which is not derived from other objects in the library. The initial form of an Ada compilation unit might be Ada source, or Abstract Syntax Tree (AST), for example, depending on how it was initially submitted to the library.

Every other object in the library is a generated form, and is derived from one or more initial forms. A generated form becomes out-of-date when one or more of its initial forms is replaced. An advantage of this scheme is that intermediate forms can be deleted from a library to conserve space, without causing generated forms to become obsolete.

3.3.2 Rules Rules are used to describe how to generate one form of a library object from another. The general form of a rule is:

```
precursor -> target: operation
```

Users can modify or add rules to cover other forms, such as foreign language conversions, documentation, and problem reports.

3.3.3 Approved Operations Operations are maintained in a list, which identifies for each operation a tool name and revision. Thus tools can be updated or replaced, without changing the list of rules. Because a target_form implicitly depends on the version of the tool that creates it, updating a tool may cause some objects to become out-of-date.

4. ADA COMPILER

A compiler may be a programmer's most important tool. The AIE compiler is designed to produce high quality code in a friendly and efficient manner. A number of optimizations are used to make the generated code efficient. Friendliness is achieved primarily through informative error messages and a powerful syntactic error recovery (parse fixup) scheme.

4.1 Compiler Optimizations

Compared to other languages, Ada presents four major areas of difficulty in producing optimized code. Constraint checks, several of which may occur in one statement, may create significantly more code than the programmer expected. Inline subprograms expanded in a simple way may make modularity expensive, thus defeating one of Ada's primary purposes. Tasking, if naively implemented, may be too inefficient for some synchronization needs. Expansion of generics must be optimized to avoid time wasted in recompiling generic bodies and space wasted by redundant code.

4.1.1 Constraint Check Elimination: The AIE Ada compiler eliminates many constraint checks at compile time, by keeping track of information known about each object. For example, a simple assignment of one variable to another of the same subtype does not require a constraint check, because the source variable must already contain a valid value. A use of a variable which was declared with an initial value does not require a constraint check, since the initial value has already been checked. The sum of two integers of discrete range need not be checked for integer overflow, unless the discrete ranges are large.

Implicit constraint checks which cannot be removed are flagged at compile time. With careful design, a programmer should be able to remove nearly all such checks. In fact, an implicit constraint check may often be taken as an indication of a flaw in coding.

4.1.2 Inline Subprograms: The AIE compiler fully supports inline subprograms, and optimizations are applied after such subprograms are expanded. Thus optimizations span the subprogram interface.

4.1.3 Tasking Optimizations: The AIE tasking implementation is optimized in several ways. Nearly all scheduling overhead is eliminated for the second task of a pair entering a rendezvous, since the other task is already waiting and one of the two must be the highest priority runnable task. The rendezvous is executed on the caller's runtime stack, so that parameters are passed with the same efficiency as a procedure call. The static link, which would normally point to the innermost invocation of the enclosing subprogram in the caller's stack frame, is adjusted to point to the called task's stack, so that up-level references refer properly to the scope containing the accept body.

In addition, the user may declare a task which does nothing important outside of accept bodies to be a "monitor" task. For such tasks, even more scheduling overhead and nearly all stack space is eliminated.

4.1.4 Optimizing Generics: Generics are stored as DIANA (an intermediate tree representation), and are therefore not recompiled for each instantiation. Where possible, code is shared among instantiations, to minimize redundancy.

4.2 Error Reporting

Accurate diagnosis of syntactic errors is doubly important. First, the location and nature of the error must be reported accurately. Second, the parser must recover from the error in such a way that artificial errors are not introduced. In addition, reporting a good parse fixup is often more helpful to a programmer than a good error message. The AIE compiler uses the syntactic error diagnosis and recovery method described by Burke and Fisher [BF]. Used in the NYU Ada-Ed compiler, this method has correctly diagnosed over fifty errors in an Ada

compilation of 111 lines. Some typical messages are shown in figure 1.

Semantic errors from the AIE compiler are designed to be as helpful as possible. Each such message will highlight the erroneous portion of the source line, describe the nature of the error, and give a reference to the relevant section and paragraph of the Ada Language Reference Manual [LRM].

The semantic error handling mechanism is modular and flexible. The compiler defines semantic errors at the levels of declarations, statements, and expressions. There is a unique exception for each semantic error, so that an error can be handled at any of several levels. Since an error handler can reraise the same exception or raise a different exception, there may be responses on more than one level. An error in an expression, for instance, might cause the statement in which it occurs to be skipped. Responses to an error can be easily changed. For example, messages describing possible causes of a semantic error and suggestions for fixes might be added.

5. SUMMARY

The AIE is a powerful, easy-to-use environment for the development of Ada programs. It provides a powerful command language based on Ada and Unix. Its program integration facility supports shared code, and helps to automate revision control and object reconstruction. The AIE compiler is production-quality tool which produces optimized code and and helpful diagnostics.

6. REFERENCES

[Bourne] S. R. Bourne, "The Unix Shell," The Bell System Technical Journal, Vol 57 No 6 Part 2 (July-August 1978), 1971-1980.

[BF] Michael Burke, Gerald A. Fisher, "A Practical Method for Syntactic Error diagnosis and Recovery," Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol 17 No 6, June 1982, pp 67 - 78.

[Fisher] Gerald A. Fisher, Jr., private communication to Len Tower.

[LRM] Reference Manual for the Ada Programming Language, MIL-STD 1815, March 1983.

[Shenker] Abraham Shenker, "A MAPSE Command Language," Journal of Pascal and Ada, January-February 1983, pp 35 - 39.

7. ABOUT THE AUTHOR

George J. Snyder has been a member of the Ada Systems Division of Intermetrics Inc., 733 Concord Avenue, Cambridge, Massachusetts 02138, since June 1982. Previously, he was Software Project Leader in the Electron Beam Lithography Division of Varian Associates Inc., in Gloucester, Massachusetts. He received BS degrees in physics and architecture from Massachusetts Institute of Technology in 1972, and an MS degree in computer science from Boston University in 1980. He is a member of the ACM, and the IEEE Computer Society.

Figure 1. Example Syntax Error Messages

```
   25       subtype c is range 1..30;
*** Syntax Error: "TYPE" expected instead of "SUBTYPE"

   39         x : = x + 2;
*** Syntax Error: ":=" expected instead of ":" "="

   46       begin
   47           declare
   48             x: integer;
   49             for i in 1 .. 2 loop
*** Syntax Error: "BEGIN" expected before this token
   50               b(i) := 0.0;
*** Syntax Error: "END LOOP;" inserted to match "LOOP" on line 49 at column 27
*** Syntax Error: "END;" inserted to match "BEGIN" on line 48 at column 21
   51       end;
   52
   53       function DAYS_IN_MONTH(M: MONTH IS_LEAP: BOOLEAN) return DAY is
*** Syntax Error: ";" expected after this token

   91           K: SHORT_INT = 1;
*** Syntax Error: ":=" expected instead of "="

  106        elseif z > w then
*** Syntax Error: Reserved word "ELSIF" misspelled
```

TEACHING Ada AT THE US MILITARY ACADEMY

Major Kevin J. Cogan

Department of Geography and Computer Science
US Military Academy
West Point, NY 10996

ABSTRACT--A five year history of teaching Ada* with the NYU Ada/Ed translator has evolved into an effective methodology for teaching top-down engineering design simultaneously with a bottom-up presentation of the Ada grammar. With emphasis on embedded hardware systems, students are confronted with successively more difficult design problems which must be written and executed on a VAX-11/780. Exposed to the Ada features of packages, concurrency, generics, and exception handling, students design, write and execute an extensive term project simulating a real-time embedded system using Ada. Projects approach the 1000 lines of source code limitation of the translator. Reusability of code is stressed by importing a previous year's package when feasible.

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

The United States Military Academy is located fifty miles north of New York City on the Hudson River at West Point. Every year nearly one thousand young men and women receive a Bachelor of Science degree and are commissioned second lieutenants in the Army. Like many other educational institutions throughout the country, an increasing number of West Point cadets enroll in the academy's rapidly expanding computer science curriculum each year. Over the last five years Ada has been part of that curriculum. The Department of Defense chose the Ada programming language as its weapon to combat the software crisis for embedded computer systems for the 1980's and beyond. It is a natural result that West Point, the Army's "college," has added Ada to its arsenal of computer science studies.

AN APPROACH TO ADA

Ada education at West Point began in the summer of 1979 when the academy was selected as one of the first locations to conduct an Ada workshop. Recognizing the impact that Ada programming was to have on the software industry as well as the direct applicability of Ada for the Department of Defense, a course in Ada programming was offered for cadets in August 1980. As many readers of this article are aware, any syllabus for a course in Ada has been largely experimental to date. A good case for a bottom-up approach to Ada education can be made by those who profess that the language is large and complex and must be digested at the syntactical level before the concepts unique to Ada can be introduced. Proponents of a top-down Ada course argue that programming in Ada requires the student to be reoriented in order to grasp the fundamental aspects of data abstraction and packaging in Ada first, and then master the syntax which should be a simple process. The correct approach may ultimately be decided by the textbook most widely used. In the last year many new Ada texts have been published since the Ada language definition was approved as an ANSI standard in Februaury 1983.

The primary objective of Ada education at West Point is to determine the framework for Ada as an undergraduate elective course concurrent with providing a rich and rewarding programming language experience for cadets. This effort has been in cooperation with the US Army's Center for Tactical Computer Systems (CENTACS) at Fort Monmouth, New Jersey. As the prime contracting agency for the Army's first production compiler, CENTACS has provided West Point with successive versions of Ada/ED, an Ada translator which can be implemented on a VAX minicomputer.

THE ADA/ED TRANSLATOR

Ada/ED is a product of New York University's Courant Institute of Mathematics under contract for the Army. Written in SETL which itself was developed at NYU, Ada/ED serves as an interim learning environment for Ada until a compiler is completed and released. To date, CENTACS has provided the US Military Academy with five versions of Ada/ED - 11.4, 13.5, 16.3, 17.2 and ANSI Ada/ED 1.1. All versions have been implemented on the Department of Geography and Computer Science's research computer, a VAX-11/780 minicomputer. Each successive version has resulted in faster translation and richer semantic error detection messages. Access to Ada/Ed has put an important and exciting tool in the hands of cadets. ANSI Ada/ED 1.1 was the first compiler or translator to receive a validation certificate from the Ada Joint Program Office. Validation certifies that a product fully complies with the ANSI Ada language definition. ANSI Ada/ED can translate 100 lines of Ada source code in approximately 200 seconds, complete with syntax error highlighting and semantic error messages, when appropriate. Although a production Ada compiler will be several orders of magnitude faster than this, ANSI Ada/ED has provided the necessary feedback for cadets and instructors to assess the learning skills acquired in the classroom. West Point's findings pertaining to undergraduate Ada education have been valuable to CENTACS and, hopefully, to Ada education in general.

COURSE FRAMEWORK

The first course in Ada at West Point was simply titled Ada Programming. An assessment of student background was made before deriving the first syllabus for a course never previously taught. Cadets choosing Ada Programming as an elective had, as a minimum, a course in FORTRAN programming during freshman year (required for all freshman cadets regardless of their academic major) and Structured Programming in Pascal during sophomore year. Accordingly, cadets were well prepared for Ada as another language. But it was realized also that Ada is more than just another programming language and not just an extension of Pascal. The birth of Ada's generic, package, exception, and tasking constructs requires a new orientation to programming if the full power of Ada is going to be realized. Beyond syntax, there are the concepts of readability, reliability, maintainability, and portability to be learned. These concepts were the genesis of the Defense Department's pursuit of a standard language for embedded computer systems. Therefore, it was reasoned that a course in Ada must somehow embody these concepts and make them an integral part of the course structure.

In January 1982 the course name changed to Ada Concepts and Programming. This placed emphasis on the fact that Ada's concepts were on a par with programming as required learning objectives. Cadets must demonstrate that they understand the concepts of Ada as a key to solving the embedded computer system software crisis. They must understand that the projected defense software budget of $36 billion for 1990 can be significantly reduced by fully utilizing the concepts of Ada.

INTEGRATION OF CONCEPTS WITH PROGRAMMING

Presently the course strives to integrate the key concepts of Ada with hands-on programming. It neither purports to be a top-down nor a bottom-up approach to Ada, but rather a weaving of these two approaches throughout the forty lesson attendances. To accomplish this, syntax learning objectives are coupled with what might be an actual embedded computer system application.

For instance, during an hour lesson devoted to arrays and the block-if, the well-known problem of counting change is used. The basic purposes of loops and if statements are already known by cadets having had FORTRAN and Pascal previously. The notion of strong and enumerated typing in Ada is encountered before this particular lesson. Therefore, to count change in Ada (see Figure 1.) requires only mastering the new syntax. An opportunity exists at this point to expand the problem for any monetary system based on 100 as in 100 cents/dollar or 100 pfennigs/W. German mark. An immediate problem is the fact that not all countries based on 100 have six distinct coins. Further it might be desirable for the coin machine to prompt the user with a voice synthesis module for the specific unit of currency. At this point Figure 1 is obsolete.

Figure 2 leads (almost) to a generic solution, although the students' acquaintance with Ada generic units has not yet been firmly established. Of particular note in this solution is the FOR LOOP in the procedure body. Type COINS is the range governing the number of iterations of the loop. It no longer is dependent on the integer range 1..6 given in Figure 1, but only on the number of values enumerated for type COINS according to the nations monetary system. Concurrently, because the loop index implicitly takes on the type and

current value of the range, execution of the statement PUT(N) outputs the current coin denomination desired followed by a user input to that coin prompt. Thus to convert this program for West German use, the programmer changes only three statements in the procedure specification making substitutions as follow:

```
type COINS is (ONE_PFENNIGS,TWO_PFENNIGS,
               FIVE_PFENNIGS,TEN_PFENNIGS,
               FIFTY_PFENNIGS,ONE_MARK,
               TWO_MARKS, FIVE_MARKS);
type MONEY is (MARKS,PFENNIGS);
VALUE : constant array(COINS) of INTEGER
 := (ONE_PFENNIGS => 1, TWO_PFENNIGS => 2,
   FIVE_PFENNIGS => 5, TEN_PFENNIGS => 10,
   FIFTY_PFENNIGS => 50, ONE_MARK => 100,
   TWO_MARKS => 200, FIVE_MARKS => 500);
```

This time there are eight enumerated values for type COINS resulting in eight iterations of the loop, but this should be abstract in the mind of the programmer. The executable part of the procedure requires no modification. Eight coin prompts will be in German.

The pedagogical advantages of this program should be clear. First there is the benefit of the problem solution itself. Secondly, aside from the issues of reliability, the classroom example embodies the conceptual goals of Ada. It is readable with virtually no documentation by the novice Ada programmer by selecting meaningful names for types and objects. It is maintainable due to the mere three statements that need to be altered for a different country. It is transportable, not only from the Ada language standardization point of view, but also physically transportable from a geographical and linguistic sense with a minimum of recoding. During a recent visit by an Australian official, the three statements mentioned above were quickly altered to reflect the five subunits of the Australian dollar (1,5,20,50,100) and thus a little bit of Ada is now at work "down under". A basic tenet of Ada is that there should be a minimum of recoding effort when changes are required. Readers familiar with Ada's generic construct can note how this same example problem can be modified and written later in a generic package.

HANDS-ON TRAINING

Few educators in computer programming languages could refute the benefits of actually running programs in the language being taught. Teaching Ada should not be an exception to this philosophy. Accordingly it has been found to be extremely advantageous to get students utilizing the translator as quickly as possible. One lesson in the syllabus is devoted exclusively to using the Ada/ED translator on the VAX. Of necessity, the treatment of input/output in Ada, chiefly in terms of package TEXT_IO, is moved to the beginning of the course syllabus at about the sixth lesson so that students may interact with the execution of their program. There is a twofold advantage in this approach. Not only does this allow students to run programs early in the course requiring syntactically pure solutions, but it also allows for the early introduction of the concept of packages in Ada. Package TEXT_IO in Chapter 14 of the language reference manual is rich in Ada style and diversity. By requiring its use early in the course of instruction, users get a rudimentary application of such constructs as generic package instantiation, subprogram default parameters - NEWLINE vs. NEWLINE(3), overloading procedure and function names, and implementation defined values for example. The WITH and USE statements must be introduced as well, thus providing a natural environment for the utility and application of the Ada package. Here is the essence of weaving the top-down and bottom-up approach to learning Ada. More details of subprograms, generics, and packages come later in the course, but by lesson 9, having learned the minimal set of control structures to solve any problem, students are ready for their first hands-on application of Ada. A representative problem statement is stated as follows:

A computer terminal manufacturer recognizes that terminals will continue to progress from computer terminals to communications terminals. Products such as direct connect modems should allow the user to directly dial a telephone number from the keyboard. A telephone handset would be connected to the terminal for voice communications when desired. A typical complaint from users, however, is that keyboards are not labeled with the alphabetic letters also found on the telephone dial or touch-tone pad. If a mnemonic such as ARMY is dialed from the West Point prefix 938, a tape recording for Army sports is connected. The user is hard pressed to remember the numbers associated with A, R, M, and Y as 2, 7, 6, and 9 respectively. Further, terminals should be intelligent enough for this to be transparent to the user. A typical telephone dial or pad is arrayed as follows:

	ABC	DEF	GHI	JKL
1	2	3	4	5
MNO	PRS	TUV	WXY	
6	7	8	9	0

The manufacturer desires to market terminals with embedded software written in Ada which allows the user to dial through a direct-coupled modem by either numeric or alphabetic values as found on the array above. Write a program which has two separate procedures to accomplish the following tasks: (1) Enter a four digit number from the terminal and output to the terminal on separate lines an array of dots corresponding to the number of each digit dialed; (2) Enter on separate lines four alphabetic characters (Q and Z not allowed) from the terminal and output to the terminal on separate lines an array of dots corresponding to the number associated with that letter.

An student's solution from this year's course is at Figure 3. It represents what can be accomplished by the tenth hour of instruction in a hands-on course in Ada. Readers apprehensive about the size and complexity of the Ada language may take some relief at this point. The student is in his junior year, with prior experience in FORTRAN and Pascal. By lesson 20 subprograms, packages, library units, separate compilation, and exception handling have been described. A problem statement is formulated by the instructor which incorporates and necessitates these constructs in the problem solution. Similarly, after Ada tasks and generics are taught, another hands-on exercise is required. With approximately 10 hours of instruction remaining from the 40 hours allocated, students formulate their own term project in consultation with their instructor. Term projects may produce a useful package such as for trigonometric functions which may subsequently be used by future student projects (this has already been done). Term projects also may emulate a present or future embedded computer system such as an auto-rotation procedure to safely land an incapacitated helicopter or a drone reconaissance aircraft's sensor systems. Both of these projects lend themselves nicely to Ada's task mechanism for concurrent processes. Term projects are not technically complete, but they do emulate such systems from a design viewpoint and typically range from 500 to 800 lines of code which must be executed on Ada/ED.

CONCLUSION

West Point is committed in its pursuit to offer quality Ada education. It requires striking a balance between student capabilities and the timely introduction of concepts unique to Ada. A mix of the top-down and bottom-up approach to learning Ada has evolved using the mechanism of example problems for embedded systems which can stress the advantages of an Ada problem solution. The luxury of having the Ada/ED translator available to fully exploit the education process has been an invaluable tool for instructors and students alike. Programs emulating a robotics application for optically recognizing resistor color codes and a self-service package mailing station for the post office have been written and successfully run by cadets. Stimulating problems such as these spark the imagination and reveal the power of Ada. Solving the software crisis of tomorrow requires sowing the seeds of Ada education today.

Major Kevin J. Cogan, Department of Geography and Computer Science, US Military Academy, West Point, NY 10996. Major Cogan was commissioned in the US Army Signal Corps. He received a BS degree in 1971 from the US Military Academy and an MS degree in Electrical Engineering in 1981 from Columbia University. He has served in various command and staff positions in the US and in Europe. Presently he is an assistant professor of computer science and the Ada course director at West Point.

EXPERIENCES IN TEACHING Ada

Philip Caverly, Charles Drocea, Philip Goldstein, Donald Yee

Ada Technology Center and Computer Science Department
Jersey City State College
Jersey City, NJ 07305

ABSTRACT

The first Ada course at Jersey City State College was taught three years ago. Since that time, the use of Ada on our campus has expanded considerably. We now offer a variety of Ada courses and applications of Ada have been incorporated into many other computer courses such as Software Engineering and Systems Programming. Under a contract with Fort Monmouth we (1) have established an Ada technology center, (2) have produced two training courses for Army use, (3) are currently developing case study extensions of these courses and (4) are exploring the feasibility of using CAI for training.

We have established a liaison with local industry to explore mutually beneficial ventures and in the near future, expect to offer special seminars for management, scientists, engineers and college faculty. We are also considering ways we can use Ada as a first or second course in the curriculum along with some software engineering concepts. Our goal is to put students in touch with cutting edge technology and realistic problems as soon as possible.

INTRODUCTION

The first Ada course at Jersey City State College was taught three years ago by Dr. Philip Caverly, currently Chairman of the Computer Science Department and Director of the Ada Technology Center at the college. Since that time, Ada courses have been given regularly to a wide variety of students - including undergraduate, graduate and visiting faculty. Ada has been used as a Program Design Language in our Software Engineering course and as a Systems Design Language in our Systems Programming course. In this paper, we review our experiences with the use of Ada in our courses and explore future directions.

Ada COURSES

We have structured all of our programming courses so that students can start writing complete programs almost immediately. As new topics are introduced, they are incorporated into programs. The package concept is introduced at the beginning of the programming courses and used as the unifying thread. We discuss the use of packages as types and as abstract objects along with the concept of information hiding. We have endeavored to use Ada as Ada and not just another language like FORTRAN or Pascal with somewhat differing syntax.

1. TRAINING COURSES FOR FORT MONMOUTH

In 1982 we obtained a contract from the Software Technology Development Division of CENTACS at Fort Monmouth to develop two Ada courses, a Professional Level course and a Technical Level course. The former is intended for engineers and computer scientists in Government service, while the Technical Level course is for application programmers. Each course was designed to take eight weeks for a class meeting twice a week for two hours per session.

Professional Course Content: The course is divided into seven modules: (I) Packages and Input/Output, (II) Encapsulating Data Types in Packages, (III) Encapsulating Data Objects in Packages, (IV) Encapsulating Finite State Machines in Packages, (V) Tasking, (VI) Blocks and Exceptions and (VII) Generics. Each module takes about one week to complete. One week during midcourse is used for a review, summary and breather.

Technical Course Content: (I) Introduction to Input/Output, (II) Introduction to Ada Structures, (III) Introduction to Packages, (IV) Elementary Data Structures and (V) Advanced Data Structures.

Testing of Professional Course: During the test and evaluation period the course was given at Fort Monmouth to about twenty five Fort Monmouth employees. It was a heterogeneous group of students with a variety of computer backgrounds. The course was restricted to those with no prior knowledge of Ada. In general, the course was received favorably. The main problems encountered were: (1) inhomogeneous student backgrounds, (2) students missing lectures due to job related travel, (3) insufficient computer time.

Testing of Technical Course: The material in this course has been tested in a one year undergraduate Ada programming course described below.

2. ONE YEAR UNDERGRADUATE Ada LANGUAGE PROGRAMMING COURSE

Currently, many computer science majors at Jersey City State College take a one year course in Ada. This course is not required, but it has gained great popularity among our students because many of our graduates have obtained good jobs due to their knowledge of Ada, and also because the students have a sense of excitement and enthusiasm

while working with a new language that promises to have a major impact on the world of computing.

Most students taking this course have had a one year course in PL/C. Nevertheless, they have limited experience in developing systems, hence the approach used in this course has been similar to the one used in the Technical Level Course developed for Fort Monmouth. In fact, the first semester has been the testing ground for much of the Technical Course. Topics covered in the course include: I/O, Predefined types, Compilation Units, Procedures and Functions, Packages, Enumeration Types, Arrays, Records, Scope and Visibility and Access Types. In the second semester, currently in progress, the student's point of view is shifted from being a user of packages toward becoming a designer of packages. The student is introduced to some of the advanced concepts of Ada adopted from the Fort Monmouth Professional Course, such as Private Types, Exceptions, Discriminated Record Types, Generics and Tasking.

3. SOFTWARE ENGINEERING

Software Engineering I, II are senior electives in our computer science curriculum. Most of the students in these courses have had at least one course in the Ada language. Therefore, in Software Engineering I, we use concepts and techniques from Ada-compatible and/or Ada based methodologies such as SAD (Systems Analysis and Design)[1], the Jackson Method[2] and CORE (Controlled Requirement Methodology)[3] as well as the Introduction of Program Design Language concepts and techniques. Also covered in this course are concepts such as top-down design, modularity, data abstraction and information hiding. These are implemented using Ada constructs such as packages, private types, separate compilation and program libraries.

A class project is required in Software Engineering I. This past semester, the project was based on the real-time, embedded Cardiac Treatment system discussed in Downs and Goldsack[3]. The system was studied at the requirements level, using the CORE methodology to specify the requirements of the system.

In Software Engineering II, currently in progress, the thrust is to design the system modules using Ada. Small teams of students will design each module. Then the system modules will be integrated into an Ada program library.

4. SYSTEMS PROGRAMMING COURSE

This course is intended to introduce students to the characteristics of system software. The current text is by Welsh and McKeag[4]. Students are not required to implement the results of their work, but rather concentrate on design through case studies. By going through the development of a compiler for a subset of Pascal, and by constructing a small operating system, students gain an understanding of systems programming concerns and methodologies.

Ada is used as [illegible] Program Design Language for the specification of packages, tasks, procedures, functions and generics. Students are expected to provide the modular decomposition of systems into [illegible] components and to pay close attention to interfaces with other components.

5. VISITING FACULTY COURSE

In the summer of 1983 a preliminary version of the Professional course was given by Dr. Caverly to a group of faculty from several predominantly black colleges, and to [illegible] faculty. This summer we are offering an Institute for college faculty with a curriculum based on the Professional course developed for Fort Monmouth. Attendees can opt to receive graduate credit.

EQUIPMENT

So far, all Ada programs developed by our students have been run under the New York University Ada/Ed interpreter running on a VAX. Initially, we did not have our own VAX, but fortunately the Rutgers University Computer Center in Newark allowed us to use their VAX, thus our students have always been able to get hands-on experience. In 1983, as a result of our contract with Fort Monmouth, we were able to obtain our own VAX-11/780, and to set up an Ada Technology Center. We are currently using the validated version of the Ada/Ed interpreter. Ada/Ed has been invaluable in enabling us, along with many others, to gain invaluable experience in using Ada. Without it, the use of Ada would not be as advanced as it is today. Unfortunately, Ada/Ed has some major shortcomings - it consumes enormous resources and it is very slow, both in compilation and in execution. As of this writing, we were making plans to acquire the Telesoft compiler. Hopefully, it will enable students to run large programs in a more reasonable time frame.

Ada TECHNOLOGY CENTER

One of the purposes of the center is to act as a friendly site for industry and academe and to help potential Ada users get started with Ada. As already mentioned, for the summer of 1984, we are offering an Institute for college faculty. Furthermore, we have already developed a liason with a number of local companies, and we are exploring various mutually beneficial ventures. It is too early to report any results at this time.

FUTURE PLANS

The trend today is toward the development of complex computer systems and large programs. Yet, many introductory text books still deal with basically the same set of problems and procedures that they did ten years ago. There may be a few more commentaries on structured programming or top-down design, but basically, these texts still deal with what might be called "programming in the small." That is, they deal with problems that are readily solvable at the lowest level of

program development. How do we get undergraduate students to actually build and implement large systems in some reasonable time frame? One way is to provide them with suitable building blocks, and an environment conducive to assembly and testing. Ada has many features that make it suitable for use as the language of choice for "programming in the large." If Ada is required early in a student's computer education, the student can spend more time in designing and implementing realistic systems without having to learn a new language for each computer science course, just because the capabilities of previously studied languages are inadequate. The catch, of course, is to provide the building blocks. They need to be reasonably well documented and debugged and there needs to be a fairly large number of such blocks. While many texts preach top-down designs, they do this in the context of programming in the small. It is a challenge to computer science educators to produce educational materials that will reverse this trend and enable students to program in the large.

REFERENCES

1. deMarco, Tom, Structured Analysis and System Specification, Yourdon Inc. 1978.

2. Jackson, M.A., Principles of Program Design, Academic Press. 1975.

3. Downes, V.A. and S.J. Goldsack, Programming Embedded Systems with Ada, Prentice Hall. 1982.

4. Welsh, J. and M. McKeag, Structured System Programming, Prentice Hall. 1980.

BIOGRAPHIES

Philip W. Caverly is Professor and Chairman of the Computer Science Department at Jersey City State College, and Director of the Ada Technology Center at the college. He is responsible for Ada activities and contracts at the Center, and teaches courses in software engineering and Ada. Dr. Caverly has been a consultant for the Federal Government and private industry in Ada related fields.

Caverly received his BS in Applied Mathematics from Stevens Institute of Technology and his PhD in Scientific Computing from New York University. He is a member of ACM, IEEE and SIAM.

Address: Computer Science Department
Jersey City State College
Jersey City, NJ 07305

Charles Drocea is an Assistant Professor of Computer Science at Jersey City State College and an active participant in the Ada Technology Center, located at the college. He presently teaches courses in Ada, Computer Architecture, and Computer Organization. Prior to joining the college he was a senior software engineer for IDR (Reuters) and Gould, Inc.

Drocea received his BS and MS degrees in Physics from Fairleigh Dickinson University and is currently continuing his graduate studies at Queen's College (CUNY). He is a member of the IEEE and IEEE Computer Society.

Address: Computer Science Department
Jersey City State College
Jersey City, NJ 07305

Philip Goldstein is Professor of Computer Science at Jersey City State College, and a member of the Ada Technology Center at the college. He teaches courses in microcomputers, Computer Organization and Computer Graphics. He has extensive experience in the use and development of real-time systems for medical applications, and has a number of publications in this field. He has also developed programs for use in physics courses. He has a BS in Physics from City College of New York, and an MS and PhD in Physics from Carnegie-Mellon University. He is a member of IEEE, IEEE Computer Society and AAPT.

Address: Computer Science Department
Jersey City State College
Jersey City, NJ 07305

Donald P. Yee is a half-time member of the Computer Science Department at Jersey City State College and on the staff of the Ada Technology Center. In addition, he is an Associate Professor and Chairman of the Computer and Information Sciences Department of Essex County College in Newark, New Jersey. He has incorporated Ada concepts in several of the courses he teaches at Jersey City State including: Systems Programming, Data Structures, Algorithms and Programming Languages.

Yee received his BA in Mathematics from Rutgers University and his MS in Applied Mathematics from New York University. He has over 20 years of experience teaching mathematics and computer science and is a member of ACM and the IEEE Computer Society.

Address: Computer Science Department
Jersey City State College
Jersey City, NJ 07305

AD-P003 420

TEACHING ADA AT HAMPTON INSTITUTE

[illegible]

Hampton Institute
Hampton, VA [illegible]

[illegible]

[illegible]

[illegible] Ada I would not be appropriate as a [illegible]

[illegible] Ada is a large language with many [illegible] more complex than other [illegible] There is always a certain [illegible] with the learning of [illegible] programming language; but I [illegible]

[illegible] will be well prepared [illegible] who have had FORTRAN or BASIC will be [illegible] prepared.

[illegible] high-level language.

[illegible] Overview of Ada I

[illegible] and suggestions and comments for teaching Ada I. Some of the remarks have general validity, while others apply particularly to Ada.

The instructor should develop a "philosophical framework" for Ada. An excellent starting point is [illegible], particularly the statement "Ada was designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency."

Throughout the course, the instructor should stress the importance of program maintenance. Students should be made aware of the fact that a program may be modified by someone other than the original programmer; and even though their programs will be small, students should realize that in the "real" world an application program is likely to be very large and written by a team of people rather than by an individual.

Encourage the students to write programs whose logical structure is conducive to maintenance. [illegible] regarding indentation to [illegible], and the use of lower-case letters for reserved words and uppercase for user-defined names. [illegible] meaningful comments and [illegible] names for objects in the program.

Stress the importance of good program [illegible] and structure. Point out that [illegible] program runs and produces [illegible] does not mean that it is a good program, and let the grading procedure reflect this [illegible].

Students will be exposed to [illegible] bewildering variety of structures, [illegible] it is very important that they not lose sight of the main [illegible] and features which are common to the various Ada units. For example, each [illegible] unit in an Ada program has the general form:

```
reserved word with name or label   (label is
   .                                optional
   .  declarations and              for blocks)
   .     clauses

begin
   .
   .  statements
   .

end;
```

Also, there is evident similarity among these forms:

```
  loop      procedure      case      if
   .            .            .         .
   .            .            .         .
end loop; end procedure; end case; end if;
```

Make sure the students understand the distinction between the two major forms of Ada instructions: declarations and statements. And point out that, in spite of the apparent complexity of the language, an Ada program is merely a sequence of instructions, each of which is either a declaration, clause, or statement; and each of which ends in a semi-colon.

Ada is a large, general-purpose language which gives the programmer the capability to [illegible] that which [illegible] more [illegible] a [illegible] there is in that, in general, if you [illegible], there is usually a way to do it, [illegible] may have to convince the compiler that you are aware of what you are doing and that you really do want to do it.

[illegible] Syllabus

The major topics in this syllabus can be covered in order, but it may be preferable to [illegible] some of the [illegible] derived types in [illegible] to a more appropriate point in the semester.

[illegible] History and Background

The history behind the development of Ada is [illegible] and the students should be made aware of it [illegible]. Explain what an embedded computer is and discuss the problem of the proliferation of languages for embedded systems. Tell the students who Ada was and why the language was named after her.

Discuss the concept of software maintenance and the software life cycle, and point out the [illegible] with software maintenance. You can then describe in a brief, qualitative way some of the features of Ada (such as separate compilation) which support effective software maintenance.

[illegible] Types

The discussion of types should include a [illegible] of [illegible] types and should make the point that an Ada type always has a [illegible] of values and a set of allowed operations on those values.

The [illegible]-typed nature of Ada should be emphasized as well as the rich variety of data types. [illegible] the Ada compiler [illegible] [illegible] FORTRAN [illegible]. An example [illegible] [illegible] integer and a floating point number will [illegible] this point, after which the [illegible] of types [illegible] can be explained. This [illegible] of the principle [illegible] of this paper.

[illegible] defined [illegible] [illegible] experience, except for those students who have already had Pascal.

Discuss the difference between the declaration of a type and the declaration of a variable of that type, and the difference between an implicit and explicit type declaration. Explain how and why subtypes and derived types are used, and explain the difference between the two.

[illegible]

[illegible]

[illegible] Structures

[illegible] if, [illegible] case, and the [illegible] [illegible] [illegible]

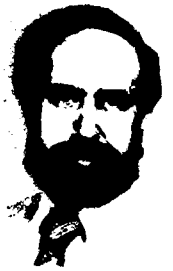
[illegible] between [illegible] [illegible].

[illegible]

[illegible] improve program readability. [illegible] variable [illegible] [illegible] with another variable name. [illegible] how the [illegible] of [illegible] and [illegible] can improve program readability.

[illegible] Subprograms [illegible]

This important unit is [illegible] the [illegible] in the syllabus for Ada [illegible]. This is where the student should see the [illegible] of an Ada program as a main procedure which [illegible] is [illegible] by other program units. The important concepts of scope, hiding, and [illegible] should be carefully developed here. The [illegible] different binding modes for parameters (in, out, in out) [illegible]



THE CECOM SUMMER FACULTY RESEARCH PROGRAM

Putnam Texel

SofTech, Inc.

ABSTRACT

[illegible] describes the U.S. Army [illegible] Faculty Research and Enhancement [illegible] The Center for Tactical Computer [illegible] located at Fort Monmouth, NJ, [illegible] the goal of fostering Ada [illegible] with the Historically Black College [illegible], primarily provides intensive Ada [illegible] professors of these colleges. This [illegible] trains the professors with the [illegible] expertise to include Ada within their [illegible] system curriculum as well as [illegible] the Department of Defense research [illegible]. This paper describes the program and [illegible] some educational issues and interesting [illegible] encountered during the course.

Section 1

INTRODUCTION

The Army is introducing an entirely new [illegible] cost effective way of doing software [illegible] development centered on the Ada language and [illegible] integrated environment (the Ada [illegible] system). Present activities include [illegible] a set of tools and [illegible] effective methodologies for use in conjunction [illegible] with Ada as well as developing curriculum [illegible] material for use in Ada training activities. [illegible], Sofitech is instrumental in [illegible] sharing Ada expertise to academia through [illegible] activities such as the CECOM Summer Program. The purpose of this program was 1) to transfer Ada expertise to professors from across the country therefore enabling them to introduce Ada into their curriculum and 2) to provide a source of research support for the Department of Defense.

1.1 Background

The original solicitation for [illegible] was a call for Ada programming language [illegible]. Classes were held [illegible] days a week, [illegible] for one [illegible], [illegible], through August 10, 1987 [illegible]. Materials were provided by lectures. Afternoons were devoted to laboratory sessions where the students coded and executed Ada programs using Verdix Verdix [illegible].

The universities represented in the program were:

1. Atlanta University
2. Howard University
3. Cheyney State University
4. Hampton Institute
5. Monmouth College
6. North Carolina A&T State University
7. Penn State
8. Prairie View A&M University
9. Spelman University
10. Tuskegee Institute
11. Youngstown University
12. University of Rochester

The course ended with a team exercise. The original plan was to use an Intersection control System (a basic Stoplight) as the exercise. Due to the delays caused by Ada/Ed the class was not able to code any tasking and requested that the group problem be reduced in complexity. Therefore, the project simulated a baseball game with certain refinements:

1. No [illegible] allowed.
2. Each player advances the same number of bases indicated by the hit (i.e., if a batter hits a double, each player advances 2 bases).
3. [illegible] are players on first and third and the batter walks, will the player on first advance?
4. [illegible] the [illegible] moment, we [illegible] but, a result of the batter's hit or not, the batter [illegible] walks, etc. [illegible] the end of each inning, the [illegible]

. The game will [illegible] projectile until one is thrown.

. The game cannot terminate prematurely (e.g. called on account of rain).

The class was divided into 3 teams [illegible] of 3 members, 4 members and 4 members respectively. The teams were given the above specification and given one week to produce the [illegible].

At the end of the week, the teams [illegible] code. A change in the specification of the problem was announced. The [illegible] was that when a hit occurs, each player [illegible] should now advance the number of bases [illegible] by a random number generator. Each team presented their view of how easy or difficult the code of another team was to change in light of the change in the specification.

It took each team only 1 day to modify the code. The reason for the short time required was because each team's code was well [illegible]. Each team had only one module, coincidentally called Advance_Runner, which required modification.

1.2 Facilities

Monmouth College, West Long Branch, New Jersey, under contract to CENTACS, provided the classroom facilities. Class was conducted each day in building 000, Room 305. Each student had his/her own work table with a VT-100 Terminal linked to a VAX* 11/780 with Ada/ed installed. The VAX was provided by the Avionics Research and Development Command (AVRADCOM).

Additional lecture halls were provided by the College, as required, for guest speakers.

1.3 Guest Speakers

Guest speakers were invited to participate to highlight certain areas of the language. The speakers and their respective topics were as follows:

Speaker	Organization	Topic
[illegible]	National Machines	1. Generics 2. Tasking
[illegible]	CENTACS	1. Proposal Writing
[illegible]	Temple University	1. Machine Representation Specifications
[illegible]	New York University	1. Generics 2. Compiler Implementation
[illegible]	CENTACS	1. STARS
[illegible]	CENTACS	1. Abstraction 2. Building Large Systems

*VAX is a trademark of the Digital Equipment Corporation.

Section 2

PEDAGOGICAL ISSUES

The following three sections describe various pedagogical issues raised during the summer.

2.1 Input/Output (I/O)

This issue centers around how (and when) to teach I/O in Ada. In Ada, I/O primitives are included as part of the language in package Text_Io. The basic file management capabilities (e.g. file creation, open, and close) are provided by subprograms in the package and are accessed by simple procedure and function calls. However, the actual procedures to obtain input and produce output are type specific. Some are available by a simple procedure or function call, while others are embedded within generic packages. For example, for type Character and String, I/O is provided by the overloaded procedures Get and Put. (Additionally Put_Line is provided for String). To utilize these facilities requires a simple procedure call and can be taught early in the course when "with"ing a package is introduced. To perform I/O on the other types - specifically the class of integer, real, and enumeration types - requires instantiation of generic packages located in Text Io.

If a top down approach to teaching Ada is utilized, the with clause is introduced early, and therefore "with Text_IO;" becomes an early clause for students to write and they can perform character and string I/O without much difficulty. But for the student to perform I/O on other types, even Integer, requires the introduction of generics and the concept of instantiation.

It is undesirable to introduce generics early in the course as the subject is complex and requires time to teach thoroughly and properly. On the other hand to give 2 lines of code to a student and tell him "Put this code here. It works. This topic will be covered later in the course.", is in general not the best pedagogical device. But this is exactly what you must do in Ada to allow the student to perform I/O on those types. The student must be told to write the following instantiation which will allow I/O on the predefined type Integer with little or no explanation as to what is transpiring.

```
package My_Integer_IO is new Integer_IO (Integer);
```

One temporary solution, to this dilemma is to provide a "buffer" between the students and Text_IO. This was done by creating a package, called Easy_Io, which included the necessary instantiations for the predefined types of the language, i.e. Integer, Float, and Boolean, and remaining declarations for Character and String I/O. By requiring the students to "with"

[illegible] Text_IO, the [illegible] the students, "We [illegible] It is that simple." This allows the student to concentrate on the [illegible] which typically involves the [illegible].

[illegible], however, is at best a [illegible]. It does not allow for I/O on [illegible] types, e.g. enumeration types or [illegible] types. As the use of [illegible] is vital to solving problems [illegible] and the use of user [illegible] types is vital to program [illegible] reliability, to postpone the [illegible] (because [illegible] I/O [illegible]) is a [illegible]. In order to do [illegible] Ada, it is [illegible] and user defined [illegible] instantiation [illegible].

[illegible] approximately [illegible] introduce [illegible] view, and [illegible], type X [illegible] was a simple [illegible] and [illegible] students [illegible] and [illegible].

[illegible] to code [illegible]. The [illegible] coded, [illegible] by the end [illegible], and therefore [illegible].

[illegible] Exceptions

[illegible] exceptions. The [illegible] the [illegible].

[illegible] write a [illegible] of a command [illegible]. For example, the program [illegible] the letter [illegible] and would [illegible] Edit. The full list [illegible] their expanded form is as follows:

First Letter	Expanded Command
[illegible]	Edit
[illegible]	Help
[illegible]	Print
[illegible]	List
[illegible]	Quit

[illegible] of enumeration types for this problem is obvious. Given the type declarations that follow:

```
type Input_Character_Type is (E, H, P, [illegible]);
type Expanded_Command_Type is (Edit, Help,
                               Print, List,
                               Quit);
```

and given the following object declarations:

```
Input_Character   :  Input_Character_Type;
Expanded_Command  :  Expanded_Command_Type;
```

and given the proper instantiations as follows:

```
package Input_Character_Type_IO
  is new Enumeration_IO (Input_Character_Type);
package Expanded_Command_Type
  is new Enumeration_IO (Expanded_Command_Type);
```

and given the proper use clauses as follows:

```
use Input_Character_Type_IO;
use Expanded_Command_Type_IO;
```

the executable code, assuming interactive I/O, includes the following statement;

```
Get (Input_Character);
```

The question "What happens if a value other than E, H, P, L, or Q is entered at the terminal?" is raised time and time again by the students. The correct response is that the predefined exception Data_Error is raised and the user has the option of writing an exception handler to take corrective action. Another correct response, although not as informative, is that the program will abort. The first response requires a discussion at that point to discuss exceptions. The second response answers the question but does not require a discussion of exceptions. One of the goals of an Ada language course is to teach proper use of exceptions. Use of a handler to take corrective action when Data_Error is raised is a good use of exception handlers and the example serves as a natural starting point for a discussion of exception handlers, and only exception handlers. The discussion does not need to elaborate on how to declare user defined exceptions or how to raise them. The student is told that this is done either by the language or by the package he is using. [illegible], he may be using a stack package which has Stack_Overflow and Stack_Underflow declared as exceptions, but the motivation exists here for the presentation of Ada's exception handling mechanism and should be capitalized on.

Furthermore, [illegible] presentation of exception handlers provides motivation for the [illegible] Ada, and [illegible], provides motivation for the [illegible] construct. For example, [illegible]

[illegible] ... [illegible] ... the text would provide ... [illegible] ... input of an invalid entry.

```
procedure Expand_Command is
     --
     --  local declarations
     --

begin --  Expand_Command

     Get (Input_Character);

     --
     --  remaining executable statements
     --  no exception handler present
     --

end Expand_Command;
```

Next, the student is told about the existence of the predefined exception Data_Error and how to code exception handlers. The following code fragment, the problem with an exception handler:

```
procedure Expand_Command is

     --
     --  local declarations
     --

begin --  Expand_Command

     Get (Input_Character);

     --
     --  remaining executable statements
     --

exception

     when Data_Error =>
          Put_Line ("Invalid Entry");

end Expand_Command;
```

The explanation is given that the text would print, if input to the program and then the procedure terminates. The statements after the Get statement and the exception handler are not executed. The code is then modified after the introduction of the block [illegible] as follows:

```
procedure Expand_Command is

     --
     --  local declarations
     --

begin --  Expand_Command

  begin

     Get (Input_Character);

  exception

     when Data_Error =>
          Put_Line ("Invalid Entry");

  end;

     --
     --  remaining executable statements
     --

end Expand_Command;
```

Use of the block to localize the exception handler to the Get statement, allows the sequence of statements after the block to be executed after the exception handler and prohibits the procedure from [illegible] termination. However, we still do not have the capability to return to the Get statement and try again. A simple loop is introduced and now the code is modified as follows to allow for successive attempts until a valid entry is received.

```
procedure Expand_Command is

     --
     --  local declarations
     --

begin --  Expand_Command

 loop

    begin

      Get (Input_Character);
      exit;

    exception

      when Data_Error =>
           Put_Line ("Invalid Entry");

    end;

 end loop;

     --
     --  remaining executable statements
     --

end Expand_Command;
```

The motivation is initially provided by a student's question and from that one question we will then develop it to include exception handlers, loops, and locks, all within a meaningful context.

2.3 Reagability

Old habits die hard. Which is easier to read, version 1 or version 2?

VERSION 1

```
WITH TEXT_IO;USE TEXT_IO;
PROCEDURE EXPAND_COMMAND IS
--
--LOCAL DECLARATIONS NECESSARY PLACED HERE
--
BEGIN--EXPAND_COMMAND
LOOP
BEGIN
GET(INPUT_CHARACTER);
EXIT;
EXCEPTION
WHEN DATA_ERROR =>
PUT_LINE ("INVALID ENTRY");
END;
END LOOP;
--
--REMAINING EXECUTABLE STATEMENTS
--
END EXPAND_COMMAND;
```

ACTUAL STUDENT CODE

VERSION 2

```
with Text_IO; use Text_IO;
procedure Expand_Command is

      --
      --  local declarations
      --

begin -- Expand_Command
      loop
         begin
            Get (Input_Character);
            exit;
         exception
            when Data_Error =>
                 Put_Line ("Invalid Entry");

         end;
      end loop;
         --
         --  remaining executable statements
         --

end Expand_Command;
```

ACTUAL STUDENT CODE

In any Ada course it is important to stress indentation and use of upper and lower case to enhance reagability of Ada code. For some students this is a very difficult transition to make.

2.4 Style

[illegible] to accept the first letter of a command and expand it to its full command as previously mentioned. Locks and exception handlers had not been covered at this point in time. The following 3 versions of code lend themselves very nicely to a discussion of:

1. Instantiation of Text_IO generic packages. When it is required and when it is not required.

2. Covering all choices in case statement alternatives.

3. Proper choices for identifiers.

4. Case statement versus if statement.

5. Modifiability. Which program is easier to modify? Version 3 requires no changes to its executable code.

The results of such a discussion were remarkable and a noticeable change in most students' code took place after this discussion.

VERSION 1

```
with Text_Io; use Text_IO;
procedure Exercise14 is

   Ch      : Character
   Command : String(1..13);

begin -- Exercise14

   Get (Ch);

   case Ch is

      when 'E'    => Command := "Edit         ";

      when 'L'    => Command := "List         ";

      when 'H'    => Command := "Help         ";

      when 'P'    => Command := "Print        ";

      when 'Q'    => Command := "Quit         ";

      when others => Command := "Try again    ";

   end case;

   Put_Line (Command);

end Exercise14;
```

ACTUAL STUDENT CODE

VERSION 2

```
with Text_IO; use Text_IO;
procedure Expand is
   Ch    :  Character;

begin -- Expand
   Get   (Ch);
   if Ch = 'E' then
      Put  ("Exit");
   elsif Ch = 'L' then
      Put ("List");
   elsif Ch = 'H' then
      Put ("Help");
   elsif Ch = 'P' then
      Put ("Print");
   elsif Ch = 'Q' then
      Put ("Quit");
   else
      Put ("Improper Input");
   end if;

end Expand;
```

ACTUAL STUDENT CODE

VERSION 3

```
with Text_IO; use Text_IO;
procedure Expand_Command is

   type Input_Character_Type is
     (E, L, H, P, Q);
   type Expanded_Command_Type is
     (Exit, List, Help, Print, Quit);

   Input_Character  : Input_Character_Type;
   Expanded_Command : Expanded_Command_Type;
   Position         : Integer;

   package Input_Character_IO is
     new Enumeration_IO (Input_Character_Type);
   package Expanded_Command_IO is
     new Enumeration_IO (Expanded_Command_Type);
   use Input_Character_IO;
   use Expanded_Command_IO;

begin -- Expand_Command

   Get (Input_Character);

   Position :=
     Input_Character_Type'Pos(Input_Character);

   Expanded_Command :=
     Expanded_Command_Type'Val(Position);

   Put (Expanded_Command);

end Expand_Command;
```

ACTUAL STUDENT CODE

Section 3

ERRORS

A few interesting errors encountered during laboratory sessions are shown in the following paragraphs.

3.1 Parameters of Mode In

The concept of a parameter of mode in as representing data flowing in to a function does not present any problems for the student. Additionally, when augmented by a discussion of how an in parameter acts as a local constant and therefore is not modifiable, students nod their heads in understanding. However, when coding, the concepts are lost and attempts are made to assign values to these parameters and to pass them as actuals to subprograms whose formal parameters are of mode in out or out. For example, one student had the following code to represent an open procedure:

```
with Text_IO; use Text_IO;
procedure My_Open ( Name : in File_Type) is
begin -- My_Open
   Create (Name, Out_File, "Data.dat");
end My_Open;
```

Name is a parameter of mode in and therefore may not be modified by the procedure. Yet the procedure passes this parameter as an actual parameter to procedure Create in Text_IO whose formal parameter is of mode in out. The error was detected at compile time, as shown in the following compilation listing.

```
   1 with Text_IO; use Text_IO;
   2 procedure My_Open ( Name : in File_Type) is
   3
   4 begin -- My_Open
   5
   6     Create (Name, Out_File, "Data.dat");
         <-------------------------------->
*** Semantic Error; inout actual parameter no. 1
   in call is not a variable (LRM 6.4.1)
   7
   8 end My_Open;
   9
1 translation error detected
Translation time:  23 seconds
```

ACTUAL STUDENT CODE

3.2 Discriminants with Default Values

Again conceptual understanding of a construct is quite distinct from correctly coding a construct. A common error encountered is exemplified by the following code:

```
with Text_IO; use Text_IO;
procedure Matrix_Multiplication is

   type Matrix_array is array
        ( Positive range <>,
          Positive range <>) of Float;

   type Matrix_type (Row_size, Col_size :
                     Positive := 10) is

      record

         Matrix : Matrix_array ( 1..Row_size,
                                 1..Col_size);

      end record;

   Matrix1, Matrix2, Result : Matrix_type;

   --
   -- Other declarations
   --

begin -- Matrix_Multiplication

      --
      -- Executable statements
      --

end Matrix_Multiplication;
```

ACTUAL STUDENT CODE

The code compiled with no translation [illegible]. However, at runtime the predefined exception Storage_Error was raised at the point [illegible] declarations for the unconstrained [illegible] Matrix1, Matrix2, and Result. The [illegible] a feeling of accomplishment as a [illegible] compilation only to be [illegible] by the raising of Storage_Error. The [illegible] is that the use of type [illegible] the type of the discriminants sets [illegible] values in the range 1 .. Integer'Last [illegible] objects. Elaboration simply [illegible] Storage.

[illegible] is to declare a subtype with [illegible] so that the objects will [illegible] storage. This solution is coded as [illegible]:

```
with Text_IO; use Text_IO;
procedure Matrix_Multiplication is

   subtype Positiv is Integer range 1..20;

   type Matrix_array is array
        ( Positive range <>,
          Positive range <>) of Float;

   type Matrix_type (Row_size, Col_size :
                     Positiv := 10) is

      record

         Matrix : Matrix_array ( 1..Row_size,
                                 1..Col_size);

      end record;

   Matrix1, Matrix2, Result : Matrix_type;

   --
   -- Other declarations
   --

begin -- Matrix_Multiplication

      --
      -- Executable statements
      --

end Matrix_Multiplication;
```

ACTUAL STUDENT CODE

The point is that a thorough understanding of elaboration for unconstrained and constrained record objects must be presented.

3.3 Scope and Visibility

Again the concept of scope and visibility is taught quite easily, but coding it is a different story. Consider the following code to develop Pascal's Triangle:

```
with Text_IO; use Text_IO;
package Pascally is

   type Row_Type is array (Natural range <>)
                          of Natural;

   --
   -- Other declarations
   --

   procedure PascalRow
       (N : Natural; R : out Row_Type);

end Pascally;
```

ACTUAL STUDENT CODE

```
with N;
with Text_IO ; use Text_IO;
with Pascally; use Pascally;
procedure Pascal_Row is

   type Row_Type is array (Natural range <>)
                          of Natural;

   M      : Natural;

   NthRow : Row_Type (0..N);

   --
   -- other declarations
   --

begin -- Pascal_Row

   --
   -- some executable statements
   --

   M := N;
   PascalRow (M, NthRow);

   --
   -- more executable statements
   --

end Pascal_Row;
```

ACTUAL STUDENT CODE

...; it turns out that N is a separate compilation unit representing a function which returns a value of type Natural.

The procedure Pascal_Row did not compile successfully, and received the following error:

```
 1  with N;
 2  with Text_IO; use Text_IO;
 3  with Pascally; use Pascally;
 4  procedure Pascal_Row is

 5      type row_type is array
             (natural range <>) of natural;

 7      M      : Natural;
 8      NthRow : row_type (0..N);
        --
        -- other declarations
        --
10  begin -- Pascal_Row
        --
        -- some executable statements
        --
23      M := N;
24      PascalRow (M, NthRow);
        <-------------------->
    *** Semantic Error:  invalid argument
        list for PascalRow
        --
        -- more executable statements
        --
29  end Pascal_Row;

1 translation error detected
Translation time:  114 seconds
```

ACTUAL STUDENT CODE

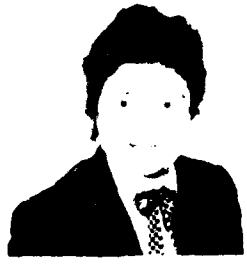
The actual parameter, NthRow, being passed to PascalRow is of type Pascal_Row.Row_Type while the procedure expects as its actual parameter an object of type Pascally.Row_Type. The inner declaration of Row_Type in procedure Pascal_Row hides the declaration of Row_Type in package Pascally that is imported via the with clause.

Obviously the solution to the problem is to remove the declaration of row_type in the procedure Pascal_Row since the declaration in the package is imported via the with clause. But the point is that again, understanding of a concept is not the same as coding it.

Section 4

SUMMARY AND CONCLUSION

Students of the program are now presenters at this conference.



Putnam P. Texel received a B.A. and M.S. degree in Mathematics from Fairleigh Dickinson University.

She has been heavily involved in the development of and instruction in the U.S. Army Model Ada Training Curriculum. She is currently responsible for coordinating all instructional activities in Ada for the Federal Systems Division of SofTech, Inc.

Ms. Texel is chairman of the Greater NY Area Local AdaIEC, a local Special Interest Group on Ada affiliated with the ACM Princeton, NJ chapter.

TEACH ADA® AS THE STUDENT'S FIRST PROGRAMMING LANGUAGE

M. Susan Richman

The Pennsylvania State University, The Capitol Campus
Middletown, PA 17057

In designing an Ada programming course within our colleges and universities, one of the first issues we must confront is the level of expertise we shall set as prerequisite to the course. Ada is a very rich and complex language. Must the student have experience with some other high order language in order to appreciate Ada? The speaker contends that programming in Ada can be taught in a meaningful way to the neophyte and, in fact, there are decided advantages inherent in learning Ada as a first language. Some suggestions are offered for coping with the size and complexity of Ada.

Introduction

The Ada® programming language is named for Augusta Ada (1815-1852), Countess of Lovelace, daughter of the English poet Lord Byron and generally considered to be the world's first computer programmer.

Ada was developed under the auspices of the United States Department of Defense (DoD) in response to the "software crisis." By the early to mid-1970s, programming languages used within the DoD proliferated; estimates range from 450 to 1500 languages and incompatible dialects. Studies projected that enormous savings would be realized if the DoD used one common high order language for all of its applications. Requirements were defined for a language to serve this purpose, as described in the Ada Language Reference Manual:

> Overall, these requirements call for a language with considerable expressive power covering a wide application domain. As a result the language includes facilities offered by classical languages such as Pascal as well as facilities often found only in specialized languages. Thus the language is a modern algorithmic language with the usual control structures, and with the ability to define types and subprograms. It also serves the need for modularity, whereby data, types, and subprograms can be packaged. It treats modularity in the physical sense as well, with a facility to support separate compilation.
>
> In addition to these aspects, the language covers real-time programming, with facilities to model parallel tasks and to handle exceptions. It also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, both application-level and machine level input-output are defined.

In order to meet all those requirements, Ada had to be a very large and complex language, indeed.

The features which make Ada such a rich and powerful language appear, on the surface, to argue against teaching Ada as the student's first programming language. In fact, most texts written on Ada are aimed at the "experienced programmer." Yet the possibility of teaching Ada to the uninitiated deserves careful consideration.

A number of Ada's features, while incorporated into the language because of their usefulness in complex systems rather than for pedagogical reasons, may actually significantly benefit the novice programmer. Those features will be considered without regard to whether or not, or in what ways, Ada may be a "better" language than other languages.

Identifiers

In the context of powerful tools, the matter of identifiers seems almost trivial, yet it can be of great importance. The Ada philosophy emphasizes that it is more important to be able to read and understand a program than to be able to write the program quickly. The program, if useful, will be read many times, but written only once. While the purpose of this objective is to improve the maintainability of software, a long-term goal, readability is also vital to the student. It cannot be assumed that a beginning student will naturally be able to read and understand his own program.

When a student begins to write code, he should have a clear mental image of the logical structure of his program. Attached to this

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

structure are various entities (data types, variables, subprograms, etc.). If these entities are given names that are strongly suggestive of the roles they play, then the structure, with objects attached, becomes *a unified whole*. If the form of the identifiers is severely restricted, names become cryptic, are only vaguely suggestive, and confuse the student's mental picture rather than reinforce it.

In Ada, the length of an identifier is restricted only by the length of line. Consequently, the name of a data object or function or package can be chosen to clearly reflect the nature and role of that entity. One won't have to wonder whether CMPMAT refers to COMPONENT OF MATRIX or COMPLEX MATRIX or COMPOUNDED MATRICIDE or COMPLIMENTS TO THE MAITRE D; it will be clear from the identifier chosen.

In choosing identifiers, learning FORTRAN prior to learning Ada can be a disadvantage. A friend who became a skilled FORTRAN programmer before learning Ada, persisted in using FORTRAN-like identifiers in his Ada code -- terse and cryptic -- apparently suggestive only to himself. This habit is difficult to break and makes the intentions of the programmer unnecessarily obscure to the reader, even when the reader is the programmer. With careful choice of identifiers, Ada code that is ready to be compiled can be read almost like clear English prose.

Strong Typing

Ada is a strongly typed language. This means that every data object used must be declared to be of a specific type, either pre-defined (integer, float, or character, for example) or user-defined to fit the particular situation. Furthermore, each object must be used in a manner consistent with its type so that, for example, one cannot compare men to women if they have been declared to be of different types. Usually, comparisons of different kinds of data objects, such as distance and mass, are unintentional. Ada actively discourages this kind of error. If you really are misguided and want to compare men and women, you may do so by making your intentions explicit. You can declare both men and women to be derived types of another people type and then compare them after doing type conversions. But *if you haven't made your intentions clear*, comparing men to women will result in the compiler saying that you can't do that.

All of the type declarations and the involved rules associated with them can be *rather daunting* to the beginning programmer and tedious to anyone. However, the student benefits. Why? He is forced to give careful thought to the design and planning of the program before he begins coding. Studies have shown that the programmer who invests a lot of effort in the planning stages of problem solving writes programs with fewer statements, cleaner logic and greater efficiency than the programs of a student who starts writing immediately, modifying and adjusting as the solution progresses. Thoroughly planned programs also take less time (including planning time) to be developed and require less terminal access time (an important consideration in these days when few computation centers have enough terminals). So Ada enforces a strict discipline that results in improved design practices, with immediate benefit to the student.

Modularity

Even a beginning student is likely to write a program with enough complexity to make it worthwhile to subdivide it into smaller, more manageable units that can be separately designed and tested. An Ada program is written as follows: it is partitioned into logical pieces, each dealing with some part of the problem. This partitioning (which, of course, could be done in an informal way with any language), when coupled with the capability for separate compilation of each unit, is useful to the student, as well as to the experienced programmer. Each of the logical units can be designed, then checked separately by the compiler for logical consistency within itself and with others in the program. This permits the student to concentrate on smaller portions of the problem at one time and to handle complexity in a more reliable and efficient manner.

Abstraction

When a student is grappling with a problem he may feel that he must understand most details of his solution (structures of the data, exactly how variables should be incremented, and so forth) before he can compose the code. Since there is a limit to how many facets of the problem the mind can handle at one time, this can lead to "blank page paralysis." Ada encourages a different approach.

If the student feels the need for objects which have certain properties, Ada permits him to declare a data type with the necessary properties, without specifying the details of the internal structure at that time. This is an abstract data type. Ada also allows for the abstraction of operations, the separation of *what* the operations do and *how* they are implemented. At the top level of design the structural and implementation details are irrelevant and sometimes confuse the picture.

As the design is refined, the internal structures and related operations are defined. If some of these are quite complex they may be defined in terms of simpler, but still abstract, types until all are defined in terms of the basic types that are pre-defined by the language. The amount of this *stepwise refinement* is yet another way in which Ada assists the student in simplifying and understanding the problem.

Debugging

As much as Ada encourages good design practices, errors will occur. Inherited from Pascal is the philosophy that the data structures and the algorithms should be specified precisely and clearly and that the compiler should detect as many errors as possible (typographical, syntax, logical inconsistencies, etc.). This compile-time detection is greatly preferable to permitting errors to remain until run-time when they are much more difficult to locate and more time-consuming to trace. Ada's ability to detect many errors, early and automatically, relieves the student of some of the burden of program checking and helps compensate for the complexity of the language.

When errors are detected, Ada assists the student in making the necessary changes. Modularity, together with the rules for scope and visibility of variables and the precisely defined interfaces of the modules with each other, prevent a change in one part of the program from propagating unpredictable effects throughout the entire program. If a change is made to a variable within one subprogram, any effect outside that subprogram will be clearly delineated through the interface of that unit with other program units. There should be no unpleasant surprises ten lines into another subprogram. So, the correction of errors is not the monumental task it can be in a monolithic program, with the effects of modifications rippling throughout the entire structure.

Exception Handling

[This is a topic which some instructors may want to, and certainly can, postpone to a more advanced course in Ada. But, exception handlers can be used, on an elementary level, to advantage.]

Not all errors in a program can be detected before compilation. Some errors occur only during actual execution when specific data values, either input or calculated, result in an anomalous situation.

Most languages assume that, when a program has compiled correctly and is ready to run, things will go smoothly. The expression whose square root is to be found will not turn out to be negative. If the month is February, the date will not be given as 31. Two matrices to be multiplied will have compatible dimensions. When things are not as they should be, the typical response of the computer is to issue a warning message and then abort the program.

In contrast, Ada is designed to permit fault-tolerant programming. If the programmer can anticipate certain classes of abnormal situations, such as incorrect data being supplied or a calculated value being outside the appropriate range, the system can be programmed to acknowledge the abnormal situation and deal with it in whatever way the programmer has determined, allowing the program to continue running rather than to terminate unnecessarily. In a realistic situation, for example, exception handling permits the controlling program to continue running even after receiving erroneous data from an instrument. (This may keep an airplane flying rather than unnecessarily terminating, but that is not our concern at this point.)

The particular value in the classroom is in notifying the student, in a helpful way, that an error has been detected during execution. Rather than reacting in the usual unforgiving manner of a program crash, the program will continue to run, perhaps to flag another error, but possibly to complete the execution.

Why Ada First?

Once the instructor believes it is possible to teach Ada to the computer-innocent student, he or she might ask, "Why?" Undoubtedly the student would have an easier time learning FORTRAN or Pascal, so why not teach either of those before moving on to the more complex and ambitious Ada?

Ada was designed after the "software crisis" focused the attention of the software community on the problems of the 1970s. The discipline of software engineering analyzes these problems and provides a methodology which offers great promise in their solutions. Ada supports the basic software engineering principles in ways that earlier languages cannot. It is essential, or at least desirable, that these principles of good program design, as encouraged by Ada, be instilled in potential programmers before they acquire the customs that are more consistent with FORTRAN.

The reason for teaching Ada before Pascal is different. In a beginning Ada course, one normally teaches essentially those features of the language that closely resemble Pascal, so why not just start with the easier language? Ada is a rich and powerful language, much more powerful than either Pascal or FORTRAN. If the goal is ultimately to achieve the full power of Ada, why begin by teaching syntax and structures that must eventually be superseded? Clearly, it is preferable to teach the "Pascal subset" of Ada, which can later be expanded, rather than teach Pascal, which must then be modified in learning Ada.

Teaching Ada

When the instructor has decided to teach Ada as the first programming language, the next consideration is "how?"

A program is a sequence of instructions that directs the computer in the performance of some computation, and these instructions must be expressed in a form that can be understood by the machine. Until the program is actually submitted to the computer, any writing of code is just an academic exercise. Therefore, the student should

be writing and executing programs as soon as possible. In order to do this, the student learns quickly how to declare basic data types and to use these with correct syntax, proper structure, and good logic in simple procedures. Unfortunately, even a simple program requires more than just writing the Ada code. The student must also interact with the hardware and the operating system. He must learn how to log on, how to issue commands to the operating system, how to react to unexpected responses from the operating system, how to use the editor, and how to create and manipulate files. He will have to learn how to describe the interfaces between his files and his program, how to create and use a program library, and how to issue commands for compilation and execution. These skills are probably second nature to the instructor who may consequently lose sight of the sheer quantity of information which the student must absorb on the numerous fronts.

All of this can be rather overwhelming to the student who just wants to be able to test his program to compute the square root of 64. He may feel as if he has been stranded in Paris without command of the French language. In other parts of France, any attempt to speak French, however poorly, is likely to be greeted with friendliness and eagerness to communicate, on whatever level is possible. In Paris, however, as with the computer, if the syntax or vocabulary is not correct, attempts to communicate will probably be met with a blank stare or perhaps with a response that is, to the student at least, unintelligible. This is most discouraging.

The basic problem is that the student must assimilate an enormous quantity of information before he can get any useful feedback from the computer with regard to his Ada code. It is clear that the instructor needs to minimize the amount of information the student must master in order to get a meaningful response. One way to accomplish this is to provide the student with examples of complete, tested Ada programs, together with the step-by-step instructions for interacting with the hardware and operating system. These examples should make it clear just what is part of the Ada language, what is part of the interface with the operating system, and what is optional to the student. Initially, the student might be expected to demonstrate only his ability to follow instructions and to type correctly; admittedly, the sense of accomplishment will not be as great as if he were working independently, but at least failure will probably be avoided. (Remember: An assignment is so simple that it can't go wrong.) At first, of course, the student will not understand the specific instructions:

```
package Day_IO is new enumeration_IO(Day);
use Day_IO; .
```

but, the computer will understand them, so the program will run. In the process the student will have absorbed information about a proper, working program. When the student is no longer unfamiliar with the equipment, assignments might consist of the student supplying missing sections of otherwise complete code. The regions of missing code could then be expanded as the course progresses, while the amount of code provided is decreased, until eventually the student is writing the complete program. In this way, the student will advance, learn how to interact with the computer, and learn Ada, at each stage receiving positive feedback from the computer.

Providing numerous examples of complete programs will help in other ways, as well. Too often, in illustrating various features of the language, code segments are given as examples. The instructor should understand that it is a non-trivial exercise for the inexperienced programmer to assemble these segments: the complete examples will provide the often missing information of the way the pieces of the puzzle are put together *in a functioning* whole. The instructor may illustrate particular features by focusing on the relevant sections; the details may be safely ignored, yet will be there when needed. Complete, working examples will also provide the student with the opportunity to learn, by experimentation, the effects of localized modifications on a successful program. The examples will serve, also, as models in the development of good Ada programming style.

There is an ancient Chinese proverb:

> I hear and I forget.
> I see and I remember.
> I do and I understand.

The examples provide not only the "see," but are of enormous help in the "do" that is absolutely crucial in learning any programming language.

Topics in an Introductory Course

A reasonable syllabus for the first course in Ada would include program structure, discrete and structured data types, statements, declarations and blocks, subprograms, and packages. Instantiation of the generic Text_IO is unavoidable and, conveniently, not difficult. User-defined generics are more ambitious and may easily be postponed. A number of the features that make Ada such a rich and powerful language may also be quite difficult for the beginning student. Fortunately, some of these (access types, tasking, low-level IO, and user-defined generics, for example) can safely be ignored in an introductory course.

Possible Texts

Unfortunately, I have not yet seen an Ada text which is suitable, in itself, for a beginning course. Either the text is at an elementary level and does an inadequate job of covering the language, or it assumes experience with some other high order language and tends to cover the basics of Ada rather quickly. My preference would be to choose one of the latter type, supplement it with

materials on the basics, and move fairly slowly at first. Of these texts my favorite is Young which has beautifully clear and complete explanations, while an updated edition of Barnes (soon to be available, I am told) might run a close second. Booch and Gehani are excellent also, but for more advanced courses or for supplemental reading in this course.

Conclusion

Language helps shape thought processes. This is true of programming languages, as well as human languages. The Ada programming language was designed with the principles of readability, understandability, and modularity of paramount importance. Ada supports the development of good programming practices, logical structuring, and an awareness that the programmer has a responsibility to make his program understandable to future users. With the increasing complexity of large scale computer systems, the instructor should encourage the modular approach to a problem rather than the sequential approach used with most earlier programming languages. The student learns a powerful program design methodology with Ada, as well as the language itself.

Ada is a large and complex language, and it will not be easy to learn or to teach. But the potential benefits are tremendous.

References

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815-1983, page 1-1.

Valuable resource texts are found in the following:

2. J. G. P. Barnes, Programming in Ada, Addison-Wesley, 1981.

3. Grady Booch, Software Engineering with Ada, Addison-Wesley, 1983.

4. Narain Gehani, Ada: An Advanced Introduction, Prentice-Hall, 1983.

5. A. N. Habermann and D. E. Perry, Ada for Experienced Programmers, Addison-Wesley, 1983.

6. H. Ledgard, Ada, An Introduction, Springer-Verlag, 1982.

7. I. C. Pyle, Ada Programming Language, Prentice-Hall, 1982.

8. L. A. Saxon and R. E. Fritz, Beginning Programming with Ada, Prentice-Hall, 1983.

9. R. W. Wiener and R. Sincovec, Programming in Ada, John Wiley, 1983.

10. S. J. Young, An Introduction to Ada, John Wiley, 1983.

AN ADA* NETWORK [illegible]

[illegible], and Bryce M. Bardin

[illegible] Aircraft Company
[illegible], CA

[illegible]

1. Introduction

For many years the hardware architecture typical of military real-time systems has been distributed. That is, many computers are connected together by communications cables. However, the software designed for these systems does not provide for incremental growth, one of the advantages possible with distributed architectures. In general, the software exhibits a high degree of coupling to the hardware, the operating system, and the communication methods, due to the fact that the functions to be performed are written for specific computers in a machine dependent manner. Thus, the software has to be redesigned each time another computer is added.

Another characteristic of real-time military computer based systems is the importance of high reliability. Many times this is achieved by having a duplicate system, known as a hot standby or hot string, so that should the primary system fail, processing would then be transferred to the hot string. Techniques are currently being developed to detect failures of hardware devices and software modules in distributed systems. Processing could then continue using the remaining resources of the system. This represents a potentially less expensive alternative to the hot string approach for achieving reliability.

Although the hardware in the above systems is distributed, the software is not, making the potential advantages (incremental growth, enhanced reliability) of distributed systems impossible to realize. A methodology for distributing software is needed which will minimize its complexity and singularity, and at the same time provide capabilities for fault-tolerance and incremental growth with a minimum amount of redesign effort.

Ada, the new programming language designed for the DoD, contains features which could support the construction of distributed software [5]. However, there isn't any agreement as to how the various language constructs should be used, or even how constructs such as tasking should be implemented for the distributed environment. The implementation chosen will impact the way the features can be used to write programs. Software support for decentralized programs in the form of process and memory management will be needed once the other issues have been resolved. Finally, some method for assigning software units to processors is needed.

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

This paper reports on a project to construct a prototype distributed system implemented in Ada. An earlier version of the paper was published in [10]. The intent of the project is to: (1) develop a software architecture that supports transparent distribution of application software in a local network of communicating processors, (2) take advantage of the potential benefits offered by distributed architectures, (3) assess the use of Ada as a programming language for real-time embedded systems, (4) define fault detection and recovery techniques which allow the system to degrade gracefully, and, (5) evaluate the overheads of the Ada network (having the capabilities just described). Section 2 of the paper describes the hardware configuration and the facilities used to develop software. The third section discusses some of the motivations and methods for distributing software in a local network of loosely coupled processing elements. Section 4 describes the model chosen to distribute Ada programs across the network, and finally, the last section summarizes the contents of the paper.

2. Ada Network Hardware Configuration

The hardware is configured as a local area network (see Figure 1) consisting of two Intel 432/670 systems, and an Ethernet contention bus as the communication medium. Each 432/670 system will be referred to as a 432 processing element due to the fact that a 432/670 system contains multiple processors. One of the 432 processing elements is connected to a microprocessor development station (MDS). Some software development is done on the MDS; in addition it is used to load and debug the 432 systems. The MDS is connected to a VAX 11/780 where the Ada compiler is hosted [8].

The architecture of the iAPX/432 is somewhat different from conventional computer architectures [7,9]. An iAPX/432 system contains general data processors (GDPs), interface processors (IPs), and attached processors (APs). The attached processors (there can be many) manage any peripheral devices in the system; one AP and it's associated peripherals is sometimes referred to as a peripheral subsystem. The GDPs (again there can be many) are the actual 432 processors; they execute Ada programs. The interface processors, IPs, are used to transfer data between the GDPs and the APs. In our configuration, each iAPX/432 contains two GDPs, one AP, and one IP (see Figure 2).

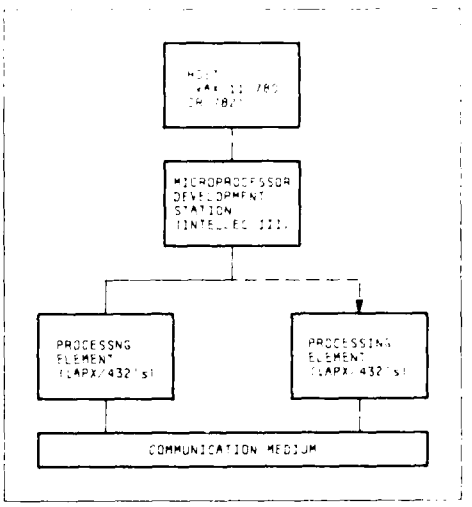


Figure 1. AdaNet and Host Computer

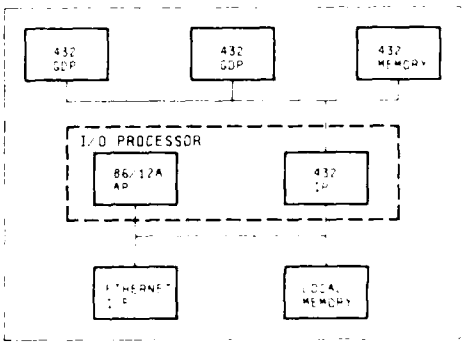


Figure 2. One iAPX/432 Processing Element

The AP is an 86/12A which uses the iRMX88 operating system. The AP in both iAPX/432 systems will be used as a communications processor; it will have software to send and receive messages over the Ethernet. Software for the 86/12A is developed in PL/M or 8086 assembler. There are two IPs, one for interfacing to the 86/12A, and the other for communicating with the MDS (it resides in the MDS). Software to control the IP exists on both the GDPs and the AP. Finally, there are two GDPs. It is possible to expand these systems with either more GDPs or APs.

The MDS executes the ISIS-II operating system which contains software development tools for the 86/12A: a PL/M compiler, a linker, and a loader. The loader is an iSBC 957B monitor; it is also used to debug the software on the 86/12A.

In addition, the MDS has a different link and software for loading and debugging the GDPs. The microprocessor development station is connected to a VAX 11/780, where all pretest software development for the GDPs is done. An Ada cross compiler and linker for the 432 GDPs resides on the VAX. Ada program modules are compiled and linked together on the VAX. One executable load module is downline loaded to the MDS.

The primary reasons for choosing this computer are (1) the Ada language can be exercised -- the iAPX/432 is one of the few systems which has an Ada compiler; and (2) the iAPX/432 has software transparent multiprocessing capabilities. Transparent multiprocessing refers to the fact that it is possible to add more GDPs to the 432 system by just plugging the boards into the backplane. The software will automatically incorporate the new processor and assign processing to it. Some other features of the 432 are: (1) hardware operating system primitives, and (2) fault tolerant capabilities. Some operating system primitives have been incorporated into the hardware in an attempt to: (1) improve their performance, (2) accelerate the software development process, and (3) standardize such operating system primitives. Fault tolerant capabilities exist in a couple of forms. In the simplest form, should one of the processors fail (within a processing element), the software will detect the condition and use only the remaining GDPs. It is also possible to have more sophisticated capabilities by using Intel chips which provide for quadruple modular redundancy (QMR) and shadowing. Shadowing is when two processors execute the same code and detect any discrepencies between them. QMR is when there exist two pairs of shadowing processors, one pair designated a master and the other a slave. If the master detects a discrepency, then the master disables itself and the slave becomes the new master. Thus, the 432 system has some new features that are not offered by more conventional computers.

3. Motivations for Ada Network Software

As mentioned before, distributed computer systems have the potential for increased reliability over single processor systems, and for incremental system growth with a minimum of hardware and software redesign. One of the objectives of the Ada network project is to develop a prototype system which exhibits these capabilities. In order to have these capabilities, both hardware and software must be decentralized in a manner which is consistent with Enslow's [2] definition of a distributed system. As shown above, the hardware resources are decentralized. Thus it remains to devise a method for constructing decentralized software.

One method advocated by researchers for decentralizing the control and data structures of software is to construct programs as collections of autonomous communicating processes [1,2,3,4,6]. The processes are designed to be autonomous entities since they may be on separate processors. Interactions between processes are well defined and independent of the actual distribution. Between interactions, processes on separate processors are capable of running in parallel. Thus, the process requires no centralized control and is the software unit of distribution and parallelism.

Coordination between autonomous processes to perform some function requires the ability to exchange information. Because the run time configuration is variable (e.g. in the case of failures), the actual configuration should be transparent to interprocess communication. This goal can be achieved by passing messages between processes.

Because processes are autonomous and have no centralized control, some means must be provided for process synchronization. For example, a process which serves as a device handler must have control over when it accepts requests, perhaps how many it accepts, and from whom. Also, a process using the service might need to wait for its completion before continuing.

The absolute timing of a particular process cannot be predicted due to the fact that processes exist on separate processing elements, each with their own environment. Since the timing of a single process cannot be predicted, neither can the timing of interactions between processes. For example, a process which is to be used by several other processes cannot always guarantee beforehand the ordering of requests. This inability to predict or guarantee a sequence of events means that decentralized and distributed software is inherently nondeterministic, and a distributed system language must be designed to tolerate some degree of nondeterminism in the interactions of processes.

Ada contains language constructs which support all of the features mentioned above. The packaging and tasking constructs can be used to create software processes which can be distributed. The Ada rendezvous is a mechanism which allows tasks to exchange information. In addition, the rendezvous is a form of so-

sation mechanism. Finally, the Ada select statement provides a means of handling the nondeterminism characteristic of process interactions in a distributed system.

4. Ada Implementation

The software for the Ada network is constructed following a typical layered approach. The top layer is the application program(s), written in Ada. Underneath the application Ada code is some support software to execute Ada programs. The support software takes the form of an operating system kernel which supplies a minimal set of primitives. The operating system that Intel supplies with the 432 is iMAX, which contains process and memory management functions for programs executing on a single processing element. The iMAX functions will be extended to support distributed Ada programs. Finally, the bottom layer is communication network software which supports the transmission of messages between processing elements.

The issues being addressed in implementing a first version of the Ada network (without fault tolerant processing) are: (1) the form of a distributed Ada program, (2) the implementation of the kernel to support the distributed Ada program, (3) system startup, and (4) the assignment of software units to processing elements. The rest of this section will discuss the first two issues.

One plan for implementing distributed Ada programs assumes that a single Ada program will be distributed. The implications are the following: (1) the semantics of the Ada rendezvous and Ada programs in general will be preserved and implemented in a distributed fashion, and (2) the compiler will be used to perform the usual compiler checks before the program is executed; thus, syntax checking of a distributed rendezvous and, in general, a distributed program is possible.

Assuming that tasks will be used to distribute processing among processing elements, the types of programs that will be allowed will be a subset of the programs that Ada allows. It is possible to share global data among tasks in an Ada program by incorporating tasks inside a package, and having data declarations to which all tasks inside the package have access. This type of structure will not be allowed for two reasons. First, it does not conform to the model of autonomous processors, and second, it is neither straightforward nor efficient to implement global data sharing between processing elements which do not share memory.

Another potential problem in distributing tasks is passing access types (pointers or addresses) as parameters. If an access type is passed to a task which resides on another machine, currently there is nothing to indicate to which machine the address refers. In order to be able to pass access types in a distributed system, some method for incorporating the processing element identification along with the address passed for the access type must be devised.

The problem in trying to implement this model is that it requires modifications to the compiler, linker, loader and runtime system. The reason is that during the implementation of the compiler and associated Ada support software, the assumption made was that a program would execute on a single processor or a multiprocessor with shared memory. Although it is assumed that some changes to the runtime system will be made, compiler modifications are outside the realm of this project.

An alternative model of a distributed system is one where the entities that are distributed are tasks, but where there is (at least) one Ada program per processing element. In order to exchange data or messages between tasks in different programs it is necessary to define a convention. This convention could not be checked by the Ada compiler for syntax or other errors. The advantage to assuming this type of program model is that it would be easier to implement. It would not be necessary to modify any Ada software support tools. Also, the issues described above (global data, access types) are not of any concern in this case. The convention or protocol defined could be implemented as "application" Ada tasks. The disadvantage of this method is that some location transparency is lost. It is possible to have parallel activity and incremental growth, but now many of the tasks are application specific and thus become the responsibility of the designer/programmer, whenever a change has to be made to the system.

The model of distributed program chosen for the Ada network is the second one outlined above. Although the first model is the preferred one, the level of effort required is not within the limits of the resources available on this project. Thus some of the desired capabilities of the Ada network will not be realized with this model. However, with this choice of implementation, it is possible to prototype an experimental capability in a reasonable amount of time and then assess the overhead involved with distribution and the use of Ada.

4.1 Pseudo Rendezvous

The following discussion gives the details of a method, called a pseudo rendezvous, for implementing a distributed rendezvous assuming that the software unit of distribution is the task in independent Ada programs. This method does not require changes to the compiler, linker or other support software. At the same time, the intent is to preserve as much of the Ada rendezvous semantics as possible. The primary support needed for such a model of distributed programs is some communication network software. Since the entities on separate processing elements are programs, the process and memory management algorithms that exist in iMAX can be used.

A normal rendezvous, shown in Figure 3, forces two tasks to synchronize at their respective CALL and ACCEPT statements. The caller, task A, specifies an entry point in another task, B. When task B reaches the specified entry point (an ACCEPT statement), the rendezvous takes place. Input parameters can be passed, and then some processing can take place (in Ada this is done by the server task). When the processing is finished, output parameters are passed and then the caller task resumes execution.

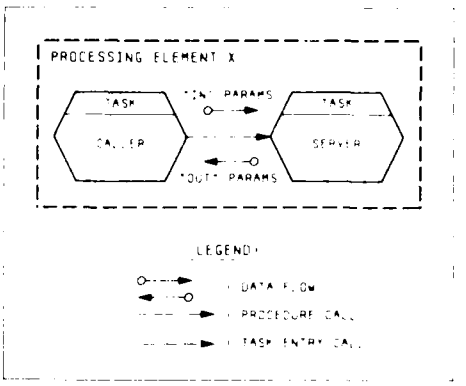


Figure 3. Normal ADA Rendezvous

Figure 4 shows the same rendezvous in distributed form. The caller now calls a surrogate procedure. The purpose of the surrogate procedure is to transmit the in and in out parameters, if any, to the communication subnet along with the destination name of the intended task, and to wait for the results of the rendezvous (status and out parameters, if any) which will then be returned to the caller. The destination name specified by the surrogate procedure is the name of a surrogate caller on another processing element. The surrogate caller is a task whose purpose is to wait for "calls" from the communication net; when one is received, the surrogate caller calls task B. After task B completes its ACCEPT statement, the surrogate caller sends the results of the ACCEPT back to the surrogate server procedure.

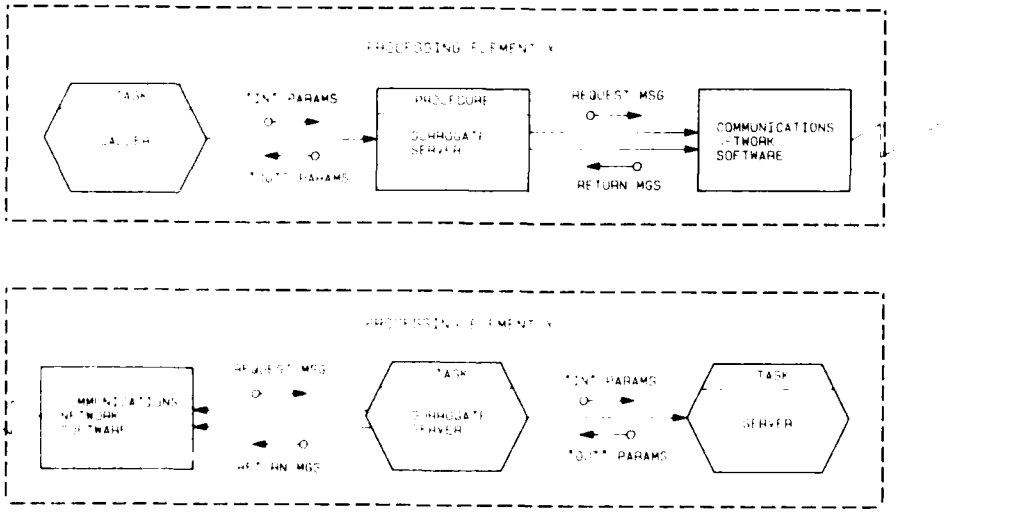


Figure 4. Distributed Pseudo Rendezvous

In this model there is a one-to-one correspondence between distributed entry callers and surrogate tasks. This allows queuing of remote callers just the same as local callers. Therefore each calling task can know the (unique) address of its surrogate procedure, and each surrogate can know the (unique) address of its caller. Additionally, each surrogate calling procedure knows the address of the surrogate server task; each surrogate server task knows the address of the real server task, B in this case.

In addition to the parameters, the messages sent and received during a distributed rendezvous must provide for communication about exceptions. If the server has an exception during a rendezvous, this will be propagated to the surrogate caller which can then request the surrogate server to raise it in the caller.

If the caller is killed during a rendezvous, the server will be unaffected as required, but the subnet may be burdened with a message that will never be read. A similar comment can be made about the case in which the caller dies before or after the rendezvous but while the request is still in transit. It will be necessary for the communications subnet to time out connections and clean up any remains. Any such time out schemes will be application dependent.

If the surrogate dies, the same sort of problem exists: communication is effectively cut off. However, the subnet time outs will handle this case also, and the calling task can be given a disconnected status.

It is possible in the case of a simple rendezvous to preserve normal Ada semantics, when the tasks involved in the rendezvous are on different processing elements by using the pseudo rendezvous approach. Even though each surrogate task or procedure must be coded for the signature of the entry being handled, the coding for all of these is fairly simple, and easy to replicate.

5. Summary

This paper has described an Ada network, a distributed system designed to prototype distributed software concepts and techniques for real-time embedded systems. The Ada language contains constructs which can be used to write distributed software. It also supports many software engineering techniques (e.g. data abstraction, parameterized types, strong typing) to improve the quality of software. However, the compilers and associated support software that are currently available have supposed that an Ada program will execute on a single machine, or a multiprocessing system with a shared memory. Thus a method, called a pseudo rendezvous, has been proposed which will allow Ada programs on different processing elements to exchange data in a manner that preserves the Ada semantics of a simple rendezvous. This allows the Ada network to be implemented within the project's time and resource constraints, and will support the evaluation of the performance overheads associated with using Ada and distributing software.

REFERENCES

[1] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, 21, 11, (Nov. 1978), pp. 934-941.

[2] Enslow, P., "What is a Distributed Data Processing System?", Computer, 11, 1, (Jan. 1978), p. 13.

[3] Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, 21,8, (Aug. 1978), pp. 666-677.

[4] Jones, A.K., and K. Schwans, "Task Forces: Distributed Software for Solving Problems of Substantial Size", Proceedings of the 4th International Conference on Software Engineering, Munich, Germany, (Sept. 1979), pp. 315-330.

[5] -- Reference Manual for the ADA programming language, U.S. Department of Defense, July 1982.

[6] Liskov, B., "Primitives for Distributed Computing", Proceedings of the 7th Symposium on Operating System Principles, Pacific Grove, CA, (Dec. 1979), pp. 33-42.

[7] -- iAPX 432 General Data Processor Architecture Reference Manual, Intel Corporation, (Dec. 1981).

[8] -- Introduction to the Intel 432 Cross Development System, Intel Corporation, (Dec. 1981).

[9] -- System 432/600 Reference Manual, Intel Corporation, (Dec.1981).

[12] Lane, D., G. Huling, and B.M. Bardin, "Implementation of a Real-Time Distributed Computer System in Ada", Proceedings of the AIAA Fourth Conference on Computers in Aerospace, Hartford, CT, Oct. 1983, pp. 325-331.

Debra S. Lane is a member of the technical staff in the Software Engineering Division of Hughes Aircraft at Fullerton. She is currently engaged in research and development of software for real-time distributed systems. Ms. Lane has a B.A. in Mathematics from SUNY at Potsdam, NY and an M.S. in Computer Science from the University of Connecticut.

George Huling received a B.S. in Mechanical Engineering from Duke University and an M.S. in Mathematics from the Stevens Institute of Technology, Hoboken, NJ. Mr. Huling is a member of the technical staff in the Ada Projects Section in the Software Engineering Division of Hughes in Fullerton. His current activities include leading an Ada design methodology working group.

Bryce Bardin is the Technical Director of the Ada Projects Section in the Software Engineering Division of Hughes Aircraft in Fullerton. He received a B.A. in Physics from the University of California at Berkeley and both a M.S. and PhD in Physics from the University of Colorado at Boulder. Dr. Bardin has participated in many of the public reviews associated with Ada and internal studies of the application of Ada to air defense systems. He has presented a number of lectures on Ada, both publically and internal to Hughes.

The authors may be contacted at Hughes Aircraft Co., P.O. Box 3310, M/S 618/P215, Fullerton, CA 92634.

DCP - EXPERIENCE IN BOOTSTRAPPING AN ADA ENVIRONMENT

STEVE PARISH AND ANDRES RUDMIK

GTE COMMUNICATION SYSTEMS R&D

ABSTRACT

This paper describes our experiences in developing a Distributed Software engineering Control Process, DCP[1]. The DCP is a portable distributed Ada programming support environment that provides centralized project management and control facilities integrated with an off-the-shelf Ada compiler and associated development tools. A goal of the DCP is to support the reuse of Ada programs and packages. This capability is supported in part by the DCP database which maintains descriptions of Ada packages and can be used to locate packages for reuse. An Ada PDL and methodology is being developed to support the development of reusable programs and packages as well as a methodology for building programs from existing packages. The goal of DCP portability is addressed by building virtual interfaces to the user, the database and the host environment. The development methodology supported by the DCP is being used to develop the DCP, thereby bootstrapping itself. The methodology is currently supported by manual controls and procedures, but as the DCP capabilities are realized, they will be replaced by automated controls and procedures.

INTRODUCTION

The DCP is a self contained Ada programming support environment. The fact that it is self contained means that one can use the DCP, once it has attained a certain critical mass, to support its own development, thereby, bootstrapping itself. The advantage of this approach is that the DCP developers are its first users and can learn from using it as it is being developed. Some of the lessons learned from building the DCP that will be discussed in this paper are: developing systems in Ada, developing virtual interfaces in Ada, designing reusable components in Ada, designing portable programs in Ada, an Ada PDL supporting reusability and portability, and methodologies for developing reusable and portable components. The experiences described in this paper are based on about twelve man years of work on the project during which time about sixty thousand lines of Ada PDL and code have been produced. The project is expected to continue for another year during which time further tools and capabilities will be developed.

The design of the DCP is heavily influenced by the need to provide support for the entire software development life-cycle with particular emphasis on managing and controlling the development process. Much of the current Ada environment activities are centered around the compiler with emphasis on supporting the code development phase. The DCP system goes beyond the Stoneman requirements in that it takes a broader and more general view of programming environments as embodying and supporting the complete integrated process of program design and evolution (Stoneman section 1.3). The DCP project assumes that adequate programming support tools will be available and that they can be integrated into the DCP with minimal effort.

The approach taken emphasizes the maintenance and management of information about the development process and the objects under development. Typical of these information requirements are the documents which describe the system, the accuracy of these documents, and the identification of reusable objects such as packages and their associated documentation. Since projects tends to reinvent objects, the DCP places a high emphasis on sharing components across many projects, possibly spread across different development hosts, and possibly utilizing differing hardware.

The DCP deliverable is a system that supports the development of tools and applications in Ada. Our approach to develop the DCP was to manually perform the operations which we will ultimately deliver. This has allowed us to exercise the DCP concepts early and has in many cases allowed for changes prior to development. Also, when the DCP capability which the manual process has been supporting is developed, the data necessary to populate the DCP database is already available. An example of this is our use of ADA PDL; each package we develop contains the PDL information that can be extracted and put into the database when these extraction tools become available.

---

[1] The DCP development is funded by the WIS JPM Technology Directorate under contract MDA 903-88-C-0212.

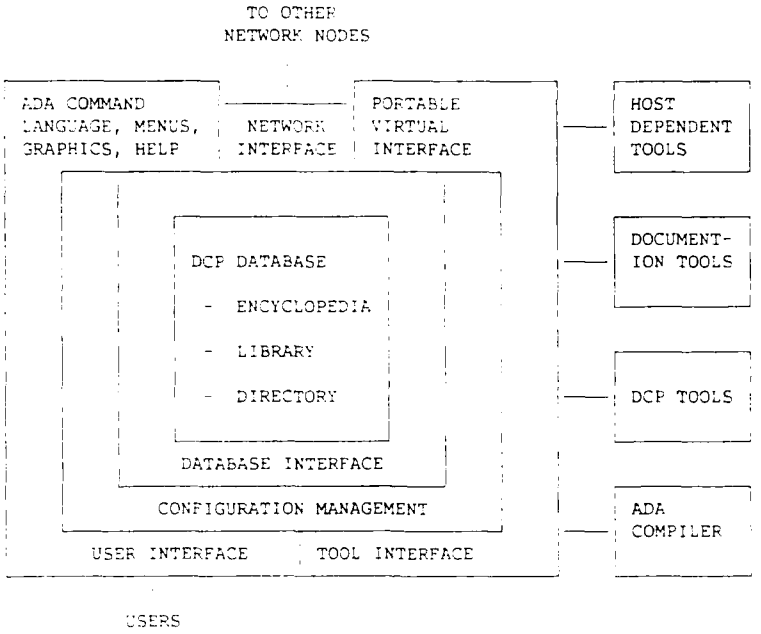


Figure 1.

DCP ARCHITECTURE

The DCP has a layered architecture as illustrated in Figure 1. The heart of the DCP is a database that maintains and manages information about the DCP users, the objects under development, and the development process. The DCP database can be viewed as a single centrally controlled database containing all the system wide application information and documentation. All the database accesses will be controlled to ensure consistency of data so that a DCP user can obtain complete, accurate, and current descriptions of applications, all their parts and the relationships between their parts. Conceptually, the DCP database can be viewed as consisting of three kinds of information:

1. An encyclopedia which maintains descriptions of the DCP objects and relationships between them.

2. A library which contains these objects.

3. A directory which maintains a mapping between logical object names and host dependent physical file names.

Although the encyclopedia, the library and the directory have been described as separate entities, they are all logically part of a single database. These terms are used to provide a handle by which different categories of database data can be described.

The encyclopedia information is used to manage and control the configuration of DCP objects, to generate documentation, to support a development methodology, and to enforce project standards. The encyclopedia information is structured to support the reuse of packages in multiple Ada programs, to provide information to support their reuse, and to support the analysis of proposed changes against packages.

The DCP library objects include Ada text, document text, and the data derived from these text forms such as object and load modules, and completed documents. The objects are typically stored in host files using host physical file names.

The directory allows DCP users and tools to refer to DCP objects in a host transparent manner by maintaining a mapping between logical and physical file names. The DCP user communicates with the DCP using logical file names and invokes DCP tools using these names. The tool interface is responsible for converting these logical names into physical file names and then directing the tools to operate on these files. The DCP has adopted this approach so that the system would support a distributed development environment where the user would not be concerned with where the files are stored and how they are accessed.

Surrounding the database, there is a portable database interface[2] that allows the DCP to be ported to other hosts where there may be different but compatible databases. The database interface will provide uniform access operations to all of the DCP facilities, and a standard query interface to all DCP users. These same interfaces can be used by application programs to provide a portable interface to application databases.

On top of the database, we have built a configuration management system to ensure that all applications and database designs developed using the DCP are maintained in a consistent form. The design of the configuration management system uses the database to manage all the information about the constituent components of a document or a program. Configuration management tools will allow users to define a configuration, to add components, fetch components, store components, compile programs etc. These tools will contain logic to ensure that the DCP user is performing a valid operation and is not violating the project development procedures and standards.

Each object in the DCP database is identified by its name, the revision of a system in which it resides, a version or modification number, and a position in a user definable hierarchy of development states. Examples of positions are production, test and development. The DCP promotion mechanism provides for the orderly movement of objects between these different positions.

The outer layer of the DCP provides a portable interface to the DCP users and the DCP toolset. The user interface supports the use of Ada as a command language, a full screen menu system, a help facility, and an on-line documentation capability. Some of the tools are host dependent and will therefore be different on each host, in which case, the tool interface would be modified to accommodate these tools. The DCP user interface to the tool would remain the same with the only differences occurring when the user is interacting directly with the tool, for example the host edi-

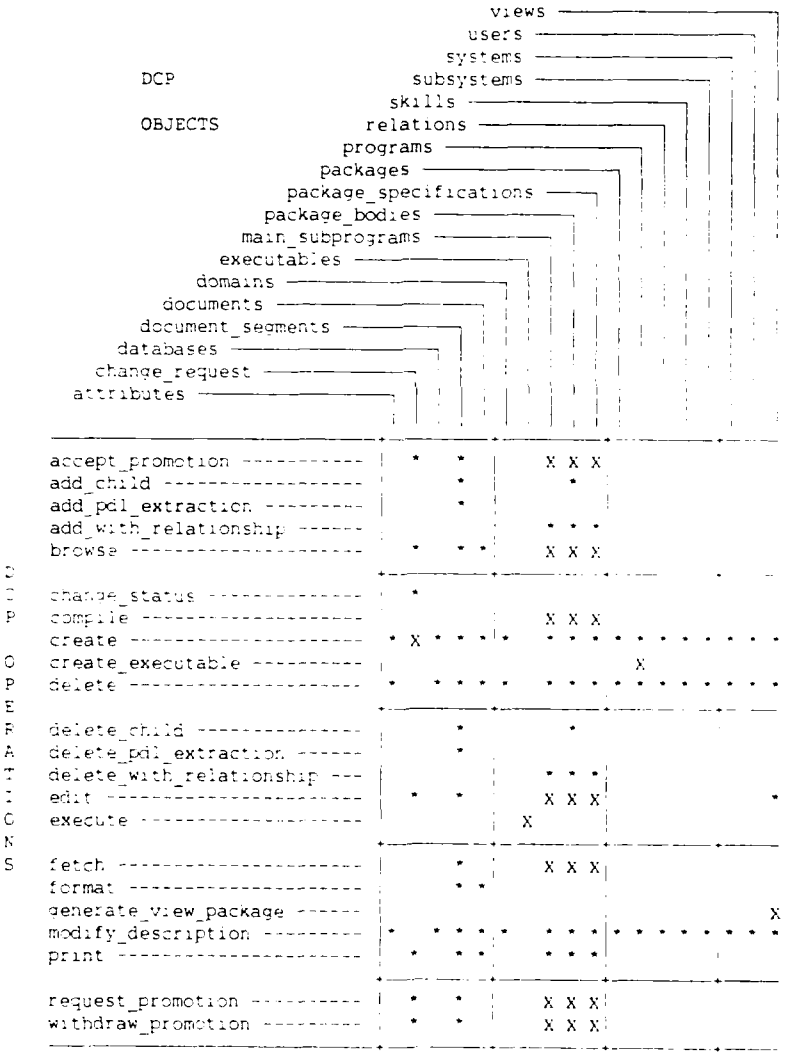


Figure 2. DCP objects and their operations.

tor. Even these difference can be eliminated in time as portable tools are developed in Ada.

The initial DCP system consists of tools to support configuration management, Ada program development, and an Ada command language interpreter. Some of these tools have been developed specifically for the DCP system as well as incorporating existing host supplied tools. Figure 2 is a matrix of DCP objects and the operations which the user can perform on these objects. An operation is identified by an 'X' if it is part of the initial automated DCP system and by an '*' if it is part of the a more complete system.

Given the initial DCP toolset, along with manually entered and controlled data, it is possible to execute a stable base of these development functions and to build incrementally on that base. This incremental approach demonstrates the viability of the DCP design, allows for early use and evaluation of the facilities, and supports the evolution of the initial system into a production quality product.

Even though the initial system is not complete, it provides sufficient project management control to support effective development of medium scale projects such as the DCP. Adequate information is maintained in the encyclopedia to track the constituents of a change, the constituents of a load module, and where-used for a package or smaller source component.

Our approach to building the DCP is to maximize the use of off-the-shelf tools and concentrate our efforts on building an integrated environment by developing virtual interfaces between the tools, the user, and the DCP database. A network interface will allow other DCP development hosts to be interconnected to provide a distributed development capability. The distributed properties of the DCP will be fully realized when the distributed database capability is available.

DESIGNING REUSABLE COMPONENTS IN ADA

The DCP addresses the goal of Ada package reusability by both encouraging the development of reusable packages and by providing a documentation database that supports the identification of packages for reuse. Our goal in building the DCP was to maximize the reuse of packages in the DCP interfaces and tools. Consequently, the design methodology and the Ada PDL used in the project emphasized the development of reusable packages. The following is a description of some of the lessons learned in this project.

First, Ada packages must be designed and documented with the objective of producing reusable packages. Having a design methodology that encourages the development of reusable packages is important in that, even though Ada has direct language support for reusability, it does not enforce the construction of reusable packages. Second, one can extract from the package text information to support its reuse by providing data on which queries can be based to search for packages. The DCP uses both of these approaches placing a heavy emphasis on how packages are designed and then extracting the necessary information from the packages to build the encyclopedia.

An object oriented design methodology as described by Booch[3] adopted and an Ada PDL supporting this methodology was developed. In developing this methodology special consideration was given to supporting the design and implementation of resuable packages. We feel that it is of ut ost importance to design packages with reuse in mind if the intended objective is to reuse them in other applications. This approach has some far reaching implications on the use of Ada PDL, the design methodology, and the documentation support tools. We feel that an approach that forces the designer to develop packages that implement simple abstractions with well understood properties is the key to the development of reusable packages. By forcing the designer to categorize the kind of abstraction represented by a package, we can define guidelines to complete the remaining package design descriptions to conform to the selected category. The primary advantage of this approach is that it produces packages that have well understood properties and that are singular in their function, thereby promoting their reuse in other applications.

Our methodology incorporates the use of structured comments to provide additional descriptive information in Ada PDL. Some of these comments are extracted and stored in the encyclopedia to support library searching and for documentation purposes. We are currently storing the following kinds of information in the encyclopedia:

1. Package category: All packages must be categorized into one of five categories: declaration group, operational abstraction, state machine, abstract type and abstract object.

2. Keywords: Each package will have a set of keywords associated with it to assist in locating packages.

3. Summary: Each package will contain a brief summary of the services provided by the package.

4. Description: An expanded description of the services provided by the package.

5. Data Flow: Packages that transform data (operational abstractions and state machines) will have data flow descriptions that help to identify and describe the transformations performed and the data types that characterize these data flows.

6. Package Specification: Each package specification is a document that provides a complete and detailed description of the package including the data types, the operations, and the user defined exceptions.

There are many ways the encyclopedia data can be partitioned to support the searching for reusable software components and documents. One such approach is through the use of keywords to categorize packages by function and application. Even though this seems like a reasonable approach there are some difficult problems in defining a set of keywords that can be used to describe packages. Attempts to generate a standard set of keywords have run into difficulties because of the compromise between the requirement that a keyword be both general enough to be widely applicable, and specific enough to effectively limit the search. Keywords must have readily available meanings and this implies maintaining a dictionary of valid keywords. Also to be effective the keywords must be validated two ways, first, against the keyword dictionary, and second, against the code they describe. This latter validation requirement implies that one can check consistency between the keywords and the code being described.

Another way to partition the encyclopedia descriptions of packages is by the descriptions of the data flows from a package to its environment. When one attempts to build a new program by reusing existing packages it is necessary that the data flows be consistent within this new program. One measure of data flow consistency is that the data types associated with the data flow out of one package and into another package be the same. The Ada PDL identifies the package data flows, provide a textual description of the data, and associates the data flows with Ada data types. Some of the more difficult issues that must be addressed are data flows that result from the use of relational and non-relational files. In this case, the designer must specify these data flows in the package specification.

DESIGNING PORTABLE COMPONENTS IN ADA

The DCP is intended to provide a portable development environment. Portability is taken to mean that the DCP can be easily installed on different hosts, and that the user interfaces to the DCP in a consistent manner on each host.

The DCP provides a uniform and portable Ada development environment by supporting the use of Ada as a design language, as an implementation language and as a command language. As part of the DCP project, we are developing a portable Ada command language interpreter, that allows the user to define Ada command programs in the same manner that he would construct his application program. An advantage of this approach is that packages can be shared between command programs and application programs providing a consistent development environment where complete type checking can be performed between command programs and the invoked application program. As a consequence of this approach, the DCP presents a consistent Ada based environment to the software developer.

In order to support type checking between the command and application programs which are developed as separate programs we must provide a mechanism to allow the parameters to be passed as Ada typed values. A problem that we encountered, was that different host systems provide different ways that parameter information can be passed to the invoked program. In most cases, this information is passed as strings, but some systems may impose restrictions on string length and in some cases content. We wanted to define an interface to the DCP tool set which would be host independent and in addition provide Ada type checking across the interface. We identified several problems that must be addressed.

The Ada LRM states the "each main program acts as if called by some environment task; the means by which execution is initiated are not prescribed by the language definition. An implementation may impose certain requirements on the parameters and on the result, if any, of a main program."[4] The description of how programs interface to the command language must be described in Appendix F of the Ada LRM, which defines the implementation dependent characteristics for each Ada implementation. The problem that this raises is that we cannot count on the compiler to support a particular program invocation protocol.

The DCP solution was to define some standard Ada packages that will be used by each DCP tool to access parameter values from the command invocation. These packages would hide the DCP host dependent representation of the parameters and provide Ada type compatible values to the program. If one uses Ada as a command language then the command parameters are typed in the same way that they would be within the called program. Type compatibility between the command parameters and the program command input is preserved by sharing the package that defines these data types in both the command and application programs.

In transporting the DCP tool set to other host systems, we would have to modify the bodies of the command interface packages, but not the programs that use them. This approach has made the DCP tool set independent of the particular host command parameter passing mechanism.

Not only did we want to be able to port the DCP to other hosts, but we also wanted to port existing tools not necessarily written in Ada into the DCP system. An obvious tool that needed to be included was an Ada compiler. In attempting to do this, we encountered further problems. Different Ada compilers made different assumptions on how the Ada libraries were implemented and used. In the DCP, we were required to support multiple versions of the library objects, with multiple developers creating objects at different development sites. In addition, we needed to support multiple levels of these libraries for development, testing, and production. Tools were required to retrieve objects from different libraries dependent on some search path specified by the developer. We found some of the existing Ada library designs to be inadequate to meet these requirements in a natural and efficient manner. To further encourage tool

portability and database transportability to different project management environments, it is necessary that the Ada library facilities be general and flexible.

Since the DCP makes extensive use of a database, a major portability issue was developing a database interface. This interface must support both users, application programs, and database administration functions. The DCP project is using Ingres, a relational database, for the encyclopedia since it provides a powerful organization of data which is capable of modelling both hierarchical and network databases. In order to use the database we had to construct a portable Ada interface.

The usual approach to interfacing a programming language to Ingres is to use a precompiler which processes specially marked statements in the source and generates modified source containing calls to the DBMS interface routines. This precompiler approach places a source level dependency on the underlying DBMS which compromises portability. For example, each DBMS will have a different form of database directive in the source for the precompiler using different query languages. A solution to this problem is to define a standard DBMS call level interface to be used by all tools and applications.

The call level interface was achieved by examining the source level expansions produced by the Ingres precompiler for other languages and analyzing the data dependencies in that generated code. For operations on views defined in an Ingres database, the only data sensitivity is in a retrieval operation which constructs a view one attribute at a time. In Ada we wish to express the view being processed as a record, therefore we have developed a fetch operation which builds this record from the individual attributes of the view. The construction of this record requires knowledge of the mapping between the view definition and the Ada representation of the record. All other operations, namely insert, update, delete, and select, require only the name of the view and are not sensitive to the data content of the view. One of the DCP tools is a view package generator which extracts a view description from the DCP database and generates an Ada package supporting that view, a retrieval operation on that view, and an Ada record representing the view. The generated view package addresses the data sensitive portion of the DBMS interface. This package together with a standard DBMS package containing the data insensitive operations forms our interface to the DBMS for a given view.

Each DBMS we have examined either provides a call level interface which can be used directly, or a precompiler similar to that of Ingres which can be analyzed in the same way. The formal parameters used in our call interface can be translated easily to other relational DBMSs eg. Oracle, IDM, SQL DS, . This means that if the DCP is installed on a machine which does not support Ingres, but has another relational database, then the only code requiring rework is the package body of the call level interface.

The descriptions of both the DCP database and user databases are held in the DCP encyclopedia. Tools are provided to interface with the underlying DBMS to support the database administration operations such as defining tables and views, and restructuring the database. These tools provide a uniform user interface which is independent of the language of the underlying DBMS.

The DCP project experience indicates that tool portability can be achieved by defining the appropriate interface packages. Our approach is similar in concept to that defined by the Common APSE Interface Set (CAIS).

Some of the portability and transportability problems that we encountered were due to the close coupling between the compiler and the Ada library. In conclusion, great care must be exercised to ensure that these interfaces are not tool technology dependent or that their use does not significantly affect performance.

EXPERIENCES USING ADA

Even though the DCP system is a medium scale project, we have learned a number of lessons that are applicable to both medium and large scale projects and we have identified some areas for further investigation.

Training is a more extensive problem with Ada than with some other languages. It requires a considerable amount of time and effort to train programmers to become proficient in Ada and in the use of object oriented design methodology. Most of the people who are working on the project had a strong Pascal background and in general had considerable programming experience. We found that it took about three months for these programmers to become fluent in Ada and able to think and design effectively using the powerful abstraction mechanism in Ada. Some programmers still have problems in effective use of abstraction concepts. In some cases programs were overdesigned, resulting in packages that were too complex and very inefficient. It seems that it will take considerable experience before one can make effective tradeoffs between efficiency and elegance in design.

In using Ada, there are a number of areas that need to be considered further. For example, what design methodology and design discipline must be introduced to control the use of "with" clauses in structuring programs. In the past, the design of the program architecture included the specification of the various dependencies that program modules had on each other. These dependencies were defined carefully and controlled during development. On the other hand, Ada allows one to include "with" clauses on both the package specifications and the bodies as part of the Ada text. These "with" clauses essentially define the package dependencies. A development environment and

methodology must control the use of the "with" clauses to allow the design to proceed in an organized and controlled manner.

The design and implementation of medium to large scale software systems requires more additional support than is normally provided by minimal Ada programming support environment. Simply the fact that all the Ada packages are under configuration management control requires that the Ada compiler must provide considerable flexibility on how Ada object libraries can be structured and maintained. In the DCP project we encountered a number of problems where the compilers view of the Ada program library was too restrictive for the DCP task. The DCP requires that many levels of Ada libraries be maintained and that the compiler can easily be directed to the appropriate libraries and packages within these libraries. Furthermore, these libraries may be distributed between different host computers requiring that the compiler library interface be handled by the DCP tool interface.

Another characteristic of large scale development is that the programming support environment must provide more support for documentation, design and implementation to ensure that these program descriptions be maintained in a consistent state. For this reason the DCP encyclopedia maintains key design information that supports consistency checking, impact analysis and traceability between various development phases. For example, the DCP database tracks the use of data types by identifying the package in which a type is defined, and the packages and programs that depend on that definition. This concept also handles the tracking of database "views".

Since Ada relational DBMSs do not exist at this time, only limited Ada type support can be applied to the database. For example, in Ingres, only character strings, integers, dates, and float are supported. Enumeration types do not map to database implementations since the action of the type in a program is changed when a new member is added to the list at compile time; whereas adding a new entry in a database would affect the action of the type as a run time operation.

Another database problem relates to Ada requiring the unique identification of a type. Several common Ada types may exist in different relations or views in a database and it is difficult to manage the attribute type definitions because different programs require different subsets of the types. Two extreme approaches are possible, first, place each discrete type in its own package, possibly without any operations, or second place all the types in the database in one package. The first solution involves proliferation of "with" clauses but provides good granularity against change. The second solution is easy to implement, but unless aggressively managed is expensive in recompilation. Possibly, the best approach is to be aware of the underlying types needed by a program and to generate a package for each program based on the simple types that it needs. Our solution in the DCP is to place all database types in two packages, split by functional requirements. However since the view package generator accesses these types when generating a view, and since package names need not be unique in a system, we have the information in the DCP database to enable us to develop the option which customizes a package of underlying types on a per-program basis.

DCP DEVELOPMENT ENVIRONMENT

The DCP is being developed using a VAX 11/78 located at Eglin AFB Florida running VMS. Two 9600 baud lines are used both for terminal connection and for remote printing via an RJE connection to an IBM mainframe. The Irvine Ada compiler is used for development. This compiler is not validated and does not currently support full Ada, but was chosen because it supports our interface to Ingres and has a more flexible library interface. Some C and VAX macro routines have been developed for system dependent operations.

SUMMARY AND CONCLUSIONS

The approach presented in this paper has helped us achieve our goal of building a portable Ada development environment. This approach included the development of virtual interfaces to the database and the host operating system, the use of a database to manage the development process, the development of a methodology including object oriented design and Ada PDL, and the development of an Ada command language interpreter with a portable typed interface to Ada programs.

Ada has supported this effort well. Where tools or capabilities are not yet available interfaces have been successfully implemented to non-Ada processes, including the database, the text editor, and the host operating system.

We have addressed the software reusability goal of Ada by supporting a methodology for both developing and using reusable components. This methodology includes object oriented design, and the use of Ada PDL. The DCP provides tools to extract information from design specifications, and to populate the DCP encyclopedia which provides on-line documentation and supports queries.

In developing the DCP we have identified some potential problem areas. Ada compilers interface to their Ada libraries in different ways, which can make it difficult to incorporate an arbitrary Ada compiler in a development environment. This problem would be reduced if a standard method of interfacing with the Ada compiler were defined.

Finally, Ada makes greater demands on developers than some other languages and it takes several months before developers become productive. As a consequence of our experience we feel that more investigation and development of design methodologies is needed to better understand and utilize the Ada language concepts.

The following topics represent the future areas of investigation for the DCP: distributing the DCP across different hosts; expanding DCP portability by investigating use of a standard non procedural DBMS query languages; adding Ada tools to the DCP as they become available. Furthermore we need to expand the DCP support for Ada oriented logical design, configuration management, and design of databases. Finally, we plan to provide both guidelines and mechanisms for incorporating Ada systems developed externally into the DCP.

BIBLIOGRAPHY

1. Department of Defense. "Requirements for Ada programming support environments". Feb 1980.

2. Vines, T.H., "An interface to an existing DBMS from Ada IDA ", ACM SIGMOD Conference Proceedings, 1984, submitted for publication.

3. Booch, Grady. Software engineering with Ada 1983

4. Ada Programming Language, ANSI MIL-STD-1815A, section 10.1.

5. Department of Defense. Draft specification of the Common APSE Interface Set (CAIS . Version 1.0. 26th August 1983.

The authors are members of technical staff at GTE Communication Systems Pub. 2500 W Utopia Road, Phoenix, AZ 85027.

Steve Baird is the lead engineer of the DCP database group and is responsible for DCP tools and the database design. His research interests are development methodologies and design tools. Steve received a BSc Hons 1 in mathematics from the University of Manchester, England in 1976.

Andres Rudnik is the lead engineer of the DCP Language Group and is responsible for the DCP user interface. He is active in the PIT PITIA team to define the CAIS. His research interests include Ada design methodologies, Ada reusability, programming support environments and compiler technology. Andres received a PhD in electrical engineering from the University of Toronto in 1978 and is a member of the ACM, Sigma Xi, and IEE.

AD-P003 425

ADA [illegible] AND THEIR NON-DOD APPLICATIONS

Ralph E. Crafts

INTELLIMAC, INC., Rockville, MD [illegible]

ABSTRACT

A primary motivation for the U.S. Department of Defense in sponsoring Ada, was to provide a common portable computer language for real-time military embedded computer systems. Ada incorporates many features and characteristics that are also appropriate for commercial, non-DoD applications. This paper will discuss some [illegible] Ada business applications, ongoing [illegible] Ada efforts, the benefits derived from [illegible] Ada [illegible] environment, and the potential [illegible] business-oriented Ada applications.

[illegible] Ada Business Applications

[illegible] non-trivial applications [illegible] oriented. [illegible] facility went [illegible] multi-state payroll [illegible] written [illegible] Ada. The user requirements [illegible] application were [illegible] support the pay- [illegible] initially, and be expandable [illegible] and hourly [illegible] tax [illegible] the federal tax [illegible] every user [illegible] experienced [illegible]

[illegible] Ada. The system [illegible] required [illegible]

[illegible] continues to be very dynamic, with [illegible] exceeding [illegible],000 separate components, and [illegible] multiple-user access to the database. This system has also functioned well for almost two years.

A third major business-oriented application is an integrated general ledger accounting system currently undergoing testing at a [illegible] facility in Maryland. This system [illegible] integrated accounting functions, [illegible] payroll, receivables, payables, general ledger, [illegible] accounting, and financial report [illegible]. The system has been written entirely in Ada, and is scheduled for full implementation [illegible] 1984.

Other Commercial Non-DoD Applications

In addition to the business applications discussed earlier, there are numerous other commercial Ada applications in the field:

- Industrial process control. A large international corporation [illegible] to the use of Ada for real time manufacturing process control software. This firm began using Ada in early [illegible] and has realized significant benefits from [illegible] than, the commitment to the language.
- Artificial Intelligence Applications. A [illegible] subsidiary has awarded a contract for a language translation system written in Ada. Using AI algorithms, the system enables the user to translate English technical text into a target foreign language. [illegible] prototype has been developed that [illegible] Ada [illegible] output, and a system will be delivered in 1984 that provides bilingual [illegible] text in French and English.
- Parallel Processing Models. Due to the tasking features of Ada, [illegible] with Ada packages, initial implementations utilizing parallel processing are now in the field. [illegible] powerful application of the Ada language [illegible] significant step toward the fifth generation of computer architectures,

the [illegible] development team could be confident that the [illegible] be [illegible] relatively error free and functional. When the final debugging of the software took place, the testing concentrated on the [illegible] of the software that was totally new. The result: [illegible] development time, functional software, [illegible] of errors, and most importantly [illegible] satisfied, operational end user.

- A prototype accounting system, consisting of [illegible] lines of code, was put together by one person in seven working [illegible], by making use of already written [illegible]. At first glance, this might not seem too impressive, since the code was already there and was simply reused. This particular effort, however, was to show a proof of concept to a particular [illegible] government, for an accounting application. The ability to demonstrate functioning software for specific applications in a short period of time is a powerful tool for in-house and outside [illegible].

The two efforts [illegible] cited are [illegible] examples of [illegible] productivity-enhancing capabilities when [illegible] existing software. What are typical [illegible] productivity when the Ada program under [illegible] of [illegible] Bell [illegible] over the last three years, average [illegible], allowing for design, testing, and debugging, a team of Ada programmers [illegible] per [illegible] during the development of a [illegible] Ada project. This level of productivity [illegible] comparable [illegible] lines of [illegible]. These figures [illegible] level of experience with Ada of at least six months, and [illegible] development.

An earlier reference was made to the ease of [illegible], and [illegible] Ada software [illegible]. Assuming the use of [illegible] of [illegible] system, [illegible] Ada software [illegible] environment. For example, [illegible] of the Ada [illegible] software, the manufacturing facility [illegible] environment [illegible] software [illegible]. After [illegible] Ada [illegible] system, [illegible] requirement [illegible] week [illegible] "system [illegible]." [illegible] features of the Ada [illegible] software life [illegible].

[illegible] expenditures [illegible] of software [illegible]. Typically, one would [illegible] and analysis [illegible] directly [illegible] software [illegible]. [illegible] so that, [illegible] in order to [illegible] software [illegible] commercial software [illegible] the user in the [illegible] will be much more [illegible] software that [illegible] properly [illegible].

In [illegible] software [illegible] significant [illegible] of the annual [illegible]. The cost of an audit is usually directly proportional to the amount of testing an auditor is required to [illegible] confidence in the accuracy of the financial reports. In a computerized system, these tests are often in the form of internal control checks of personnel and procedures, because the nature of the software makes it virtually unauditable (at least at a high level of confidence). These tests typically consist of manual verification of automated numerical calculations [illegible] extensive [illegible] sections of the accounting [illegible]. The Ada accounting software in the [illegible] under the scrutiny of two "[illegible]" accounting firms. After the initial [illegible] large scale accounting software that was [illegible] written in COBOL, the auditors [illegible] they [illegible] peruse and verify [illegible] the Ada code because of the [illegible]. Even auditors [illegible] they could [illegible] the [illegible] of the source code [illegible] the accuracy of the software. For specific tests, they were able to "pull" Ada packages and test them individually for features like proper truncation, rounding, precision, etc. The end result was the [illegible] for Ada, and the [illegible] that Ada code is very "auditable".

Future ADA Commercial Applications

The future for Ada is rich with promise, in both government and commercial applications. Empirically new, Ada has proven to be superior [illegible] for business-oriented applications. The power and clarity of the language greatly facilitates the development process, enhances productivity, and reduces software life cycle [illegible]. [illegible] time, Ada will enjoy a much greater [illegible] than [illegible] in the commercial marketplace. [illegible] the emergence of a new industry of Ada "[illegible]", [illegible] software components [illegible]. These vendors will provide transportable software packages, [illegible] variety of applications from accounting, [illegible], expert systems, AI applications, [illegible]. A primary attraction [illegible] target market [illegible] computer [illegible]

[illegible] Ada environment. The current [illegible] hardware [illegible] language will [illegible] the Ada community. A software development [illegible] with the confidence that the language will [illegible] variety of powerful hardware [illegible]. For the first time in the corporate [illegible] number of computer manufacturers, [illegible] will have to become "[illegible]" [illegible] Ada will mandate that compatibility.

Summary

[illegible] advantages of using Ada for business [illegible] applications are [illegible]. The benefits [illegible] with the use of Ada have been proven [illegible] variety of applications.

The reader [illegible] will be aware that Ada is not [illegible] that can [illegible]. Throughout [illegible], I have [illegible] to emphasize the [illegible] input, [illegible], I have heard [illegible] unwieldy [illegible] with the comment, "I wish we [illegible]!" Ada provides [illegible] and a challenge to properly [illegible] software develop- [illegible].

Mr. Crafts is the Vice President of Operations and Marketing at INTELLIMAC, Inc. of Rockville, Maryland. For the past 3 years INTELLIMAC has been actively involved in the design and development of commercial Ada applications software. Mr. Crafts has spoken at numerous AdaTEC functions and other industry related symposiums. His technical background includes eight years of service in the U.S. Marine Corps as a computer technician and jet pilot, as well as extensive experience in the design of information systems in both small business and large corporate environments. Mr. Crafts graduated Magna Cum Laude from the University of West Florida with a degree in Accounting & Financial Analysis.

EXPERIENCE WITH ADA FOR THE GRAPHICAL KERNEL SYSTEM

Kathleen Gilroy

Harris Corporation
Government Information Systems Division
Melbourne, Florida 32901

Abstract

This paper describes the effort to produce an Ada* language binding to the Graphical Kernel System (GKS) and to implement a subset of the GKS functionality in Ada. It presents an overview of the GKS/Ada project, discusses some of the issues raised during development of the GKS software, describes the results of a post-coding analysis comparing the binding and prototype code, and comments on the lessons drawn from this experience.

Introduction

The Graphical Kernel System (GKS) is a proposed national and international standard for an application level interface to a graphics system. GKS provides device-independent support for most graphics applications, with capability ranging from simple output primitives to complex interactive graphics. The set of GKS functions is intended to be implementable in many programming languages. A language binding defines the syntactical interface to GKS from graphics programs written in that language. Bindings to ANSI languages are included as part of the GKS standard. Standardization of the bindings promotes portability of both programs and programmers, and facilitates validation of an implementation of GKS.

The project described in this paper was part of the multiphased GKS/Ada effort to develop Ada graphics capabilities conforming to standards currently being developed by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). This first phase, sponsored by the World Wide Military Command and Control System (WWMCCS) Information System (WIS) Joint Program Management Office (JPMO), provided an Ada language binding to the Graphical Kernel System. Other work involved development of prototype software in Ada (both device-independent and device-dependent) to demonstrate the capabilities of the GKS/Ada system.

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

The binding was developed in coordination with the ANSI Language Bindings and Conformance Subcommittee of the Graphics Technical Committee (X3H34). It was designed to employ the full capabilities of the Ada language while conforming to the specification defined in the GKS standard. The attempt to synthesize GKS and the Ada mind-set resulted in a few difficulties, which were presented to ANSI as Ada binding issues. About twenty issues were identified, recorded, discussed, and hopefully resolved. Some of these issues may be categorized as generic binding issues, or issues applicable to GKS implementations in any language. Subsequent analysis shows that some of these issues are also applicable to Ada implementations in general. A few of these Ada language issues are discussed in this paper.

The prototype software was developed on a microcomputer interfaced to a graphics monitor, with a partial implementation of Ada supported on the development system. The prototype was coded entirely in Ada, and demonstrated the feasibility of programming graphics in Ada. Both machine-dependent and machine independent facilities were required in implementing the software, and lack of support for full Ada presented some problems in the development effort. The prototype code, while certainly valid Ada, was limited to the use of those language features supported by the compiler. The results of a post-coding analysis of the binding and prototype specifications highlight the differences in the use of the language under these two approaches.

Overview of the Graphical Kernel System

The Graphical Kernel System defines a set of language-independent functions providing a standard interface to a two-dimensional color graphics system. GKS supports machine and device-independence in the production and manipulation of pictures, yet allows device tuning to best employ the features of a specific device for a particular application. As a standard, GKS promotes portability of graphics programs, facilitates the development of applications, and provides guidelines to manufacturers for future device capabilities. GKS supports most graphics applications and devices. Types of applications for which GKS could be employed include CAD, CAI, management graphics, simulations, games and contouring. Devices which might be interfaced to GKS include plotters, storage tube displays, printers, digitising tablets, vector refresh CRT's, and raster CRT's. Six sets of logical input

[illegible] is described below.

[illegible] and Workstation Description and Control

[illegible] control of the [illegible] the [illegible] provided through an object's state [illegible]. [illegible] the concept of an abstract workstation, representing a configuration of a [illegible] display surface and zero or more input devices. [illegible] supports multiple workstations of [illegible]. Workstations are [illegible] according to their capabilities, and [illegible] capabilities. An escape mechanism is provided for [illegible] access to and control of non-standard device-specific features.

[illegible] Attributes and Output Attributes

[illegible] provides basic graphics output primitives such as drawing lines, displaying filled areas, and writing character strings. Associated with each primitive are attributes which control the appearance of the [illegible] element, such as line style, fill area color, and character size. Attributes may be set individually, or in bundles [illegible] attributes. Individual attributes are, in a workstation independent manner, while bundles may be used to control the appearance of the output for each workstation individually.

[illegible] Units and Transformations

[illegible] defines three Cartesian coordinate systems for the [illegible] representation of pictures. The World Coordinate System is used by the application program, the Normalized Device Coordinate System is used for workstation independent representation, and the Device Coordinate System is used to represent the display surface of each workstation. GKS performs mapping to and from each of these coordinate systems, using the concepts of windows and viewports to specify the [illegible].

Segment Manipulation and Segment Attributes

A segment is a collection of output primitives which are treated as a single unit. Segments provide the basis for [illegible] storage and reuse of pictures. Attributes of segments are visibility, highlighting, detectability, transformation, and [illegible] priority.

[illegible] Input and Device Interactions

[illegible] devices are classified as logical input devices, [illegible], locator, choice, pick, [illegible]. Physical input from the physical [illegible] is mapped into a value of the [illegible] the logical input device, [illegible]. Three modes of [illegible], request, and sample) [illegible] application program and the [illegible] input device. [illegible] to be detected, the operator [illegible] must be [illegible].

Inquiry Functions

Inquiry functions obtain information about the state of the graphics system, or implementation dependent characteristics of the system. They also provide information about the capabilities and characteristics of a workstation.

Metafile Functions

GKS provides for the generation and interpretation of a [illegible] (or audit trail) metafile. This metafile is used to record and recreate the sequence of GKS operations performed during a session at a workstation. Another type of metafile, the Virtual Device Metafile (VDM), is used for the long-term storage of pictures (picture capture metafiles). The pictures are stored in a workstation independent form for later recreation on the same or a different system.

Error Handling

GKS supports error checking for a finite number of exceptional conditions. The number of conditions is sufficient to provide precise identification of recognized errors. GKS also provides an error logging facility for storage of information about the source and type of an error.

Utilities

GKS provides utilities for performing matrix manipulation for segment transformations.

GKS Level Concept and Conformance

The functionality provided by GKS covers a wide range of graphics capability, however, not all of the functions defined by GKS are needed by every application. Graphics capability supporting passive output on a single workstation is defined as minimal. The introduction of new capability is controlled through the partitioning of GKS into twelve valid levels. The level model [illegible] with the addition of input capabilities treated independently of all other functional capabilities.

The matrix of GKS functionality by level is shown in Table 2.10-1. Capabilities are indicated for the level at which they first appear. Capabilities at subsequent levels consist of at least all previous capabilities along each of the axes, plus any new capabilities. For simplicity, capabilities which are only optionally supplied by a level are omitted from the table.

An implementation of GKS is said to conform to a level of GKS if it contains at least the functions and capabilities defined for that level. For example, an implementation of level 0b must contain those functions defined for level 0b, plus those from levels 0a, 0c, and 1a. An application program is said to conform to a level of GKS if it does not reference any functions or capabilities outside of that level.

OUTPUT LEVEL	INPUT LEVEL A	B	C
M	No input. Minimal control. Subset output. Individual attributes.	Request mode. Initialization. No pick input.	Sample and event mode.
B	Basic control. Full output. Predefined bundles. Multiple normalization transformations.	Viewport input priority.	
1	Full bundles. Basic segmentation. Metafile workstations.	Pick input.	
2	Workstation Independent Segment Storage.		

TABLE 2.10-1

MATRIX OF GKS FUNCTIONALITY

GKS/Ada

GKS/Ada is a multiphased effort to develop Ada graphics capabilities, including development of an ANSI standard binding of GKS to the Ada language, a production-quality implementation of the full GKS functionality, a suite of ANSI-approved metafiles for GKS validation, and a suite of device-dependent software drivers. The GKS/Ada system conforms to the set of graphics standards currently being developed by the ANSI Committee on Information Processing Systems (X3).

The GKS/Ada system model (Figure 3.0-1) shows the elements of an Ada graphics system and the interfaces between them.

An Ada application program must access GKS through the Ada Binding Interface (ABI). This interface will conform to one of twelve levels as defined in the GKS and ABI standards. The GKS system environment includes interfacing with the Virtual Device Interface (VDI) and Virtual Device Metafile (VDM), both of which are draft ANSI standards.

GKS employs the VDI as the interface to the device-dependent software which drives the graphical devices. The Virtual Device Interface provides a device independent interface at a lower level of functionality than GKS. The Virtual Device Metafile provides for device-independent storage of graphical picture information for later recreation of pictures on the same or a different system. Metafile generation and interpretation are accomplished through the VDM. A suite of metafiles could be used to validate a GKS Ada implementation. Although not shown in this figure, the VDI and VDM are both directly accessible to Ada programs for use in other graphics systems.

The accomplishments of Phase I of this effort included production of the Ada Binding Interface, currently a draft ANSI standard, and a prototype implementation of GKS to Level 0a, which demonstrated the feasibility of the GKS/Ada concepts that have been defined. The following two sections discuss the binding and prototype in more detail.

Ada Binding Interface

The major emphasis of the GKS/Ada project was the development of an Ada language binding to the Graphical Kernel System. Every effort was made to embody the philosophies expressed by the GKS standard and the ANSI Language Binding subcommittee in developing the binding. The binding consists of Ada package specifications containing all of the data types, subprograms, and exceptions used to interface with a GKS system implemented in Ada. The binding is currently an official work item on

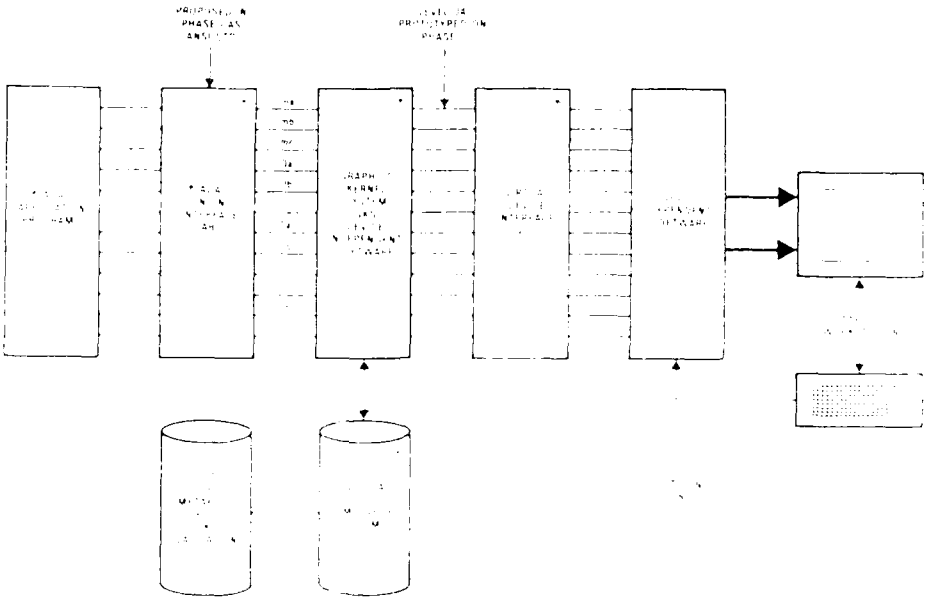


FIGURE 3.3-1.

GKS/Ada System Model

ANSI and ISO agendas, and is expected to undergo some evolution as binding and language issues are resolved by these standards organizations.

Binding Philosophy

The following guidelines were applied in defining the Ada language binding to GKS, and in attempting to choose among alternative interface specifications:

- The binding should be transportable. Changes required to rehost an implementation of GKS on a different system should be minimized.
- The binding should be extensible. Upgrades of GKS should result in minimal changes to graphics application programs and GKS implementations.
- The binding should support the GKS level concept, with the interfaces for each level upwards-compatible.
- The binding should support portability of applications programs using GKS. Programs might use any implementation of GKS with minimal changes to the source. Features of the Ada language should be used to support portability.
- The binding should follow the semantics of the standard wherever possible.
- The full capabilities provided by the Ada language should be used to best advantage, with compatibility with Ada philosophy used over some other method.
- The binding should be as similar as possible to other language bindings to promote programmer portability.

These goals were not always mutually obtainable, and involved trade-offs. Many of these or similar issues are discussed in [1].

Binding Specification

The mapping of GKS into Ada was fairly straightforward, although the GKS specification incorporates some features which are in conflict with the Ada language philosophy. For the most part, these areas of potential difference were recognized by the GKS developers, and allowance made for variation. The approach taken in development of the Ada Binding Interface (ABI) is presented for the general categories of data typing, functionality, error handling, packaging and naming conventions.

Mapping of GKS Data Types to Ada

The most difficult part of defining the Ada language binding rested in the area of data typing. The GKS standard defined several simple and compound data types used in describing the semantics of the GKS functions and data structures. The GKS data types were implemented as a variety of Ada

[illegible] and constant types. The [illegible] provided [illegible] data types to Ada data types is [illegible].

[illegible] data type	Ada data type
[illegible]	private or enumeration type
[illegible]	floating point type
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible] or enumeration type
[illegible]	enumeration type
[illegible]	array or record type
[illegible]	array type
[illegible]	access to [illegible] or LIST of
[illegible]	MATRIX [illegible] or
[illegible]	[illegible] MATRIX or
[illegible]	record type
[illegible]	private type or [illegible]

[illegible]-1

[illegible] to Ada

[illegible] of the data types are for the [illegible] of predefined [illegible], [illegible] and of Ada [illegible] for data declaration and [illegible] [illegible] [illegible]. Ada has [illegible] [illegible] of higher levels of [illegible] that is not in the GKS standard. It [illegible] [illegible] to the binding, [illegible] [illegible] to the stand[illegible].

[illegible] Functionality to Ada

[illegible] of the GKS functions were bound to Ada [illegible], with an almost one-to-one correspondence to the GKS functions [illegible]. The ordering of the [illegible] was not preserved, but some [illegible] were eliminated as unnecessary when the [illegible] could be provided by an Ada attribute [illegible] the length of an array). Ada [illegible] are more appropriate for mapping of the [illegible], but their use forces the declaration of different [illegible] data types for return values. [illegible] could be used for each [illegible] parameter, as suggested by other Ada [illegible] ahead. It was, however, there are some [illegible] parameters which [illegible] [illegible]. This problem is discussed further [illegible] [illegible] issues.

Error Handling

[illegible] handling features are specified in the binding as a replacement for the error handling [illegible] defined in the GKS standard. The [illegible] between the two methods of [illegible] are [illegible]. This is [illegible], because the GKS standard makes some assumptions about the execution environment of the system which [illegible] in Ada. For example:

> [illegible] that all raised errors are [illegible] by [illegible] subroutines, but is [illegible] the errors defined to be would be detected at [illegible] otherwise detected outside of the [illegible] of GKS.
>
> GKS provides no consideration for a multi-tasking environment, such as the [illegible] of putting GKS into an error state which disallows all but a limited set of functions to be called.
>
> The GKS error handling mechanism appears to be designed explicitly to accommodate languages which do not support recursion (which is implicit for Ada subprograms).

The binding describes a mechanism for handling errors which was intended to offset the disadvantage of not knowing the source of the error when an exception occurs. This approach would also be problematic in a multi-tasking environment, and it is probably best to rely exclusively on the exception facilities defined by the language.

The binding consolidates many of the GKS error conditions into exception classes to simplify the user's interface. This appears to be in conflict with the GKS philosophy to provide precise error description.

Packaging

All of the data types, subprograms, and exceptions are bound as a single package GKS. Three other packages are defined external to GKS. One is a generic package providing facilities for declaring and using a Cartesian coordinate system. A second is a generic utility package providing for the declaration and manipulation of lists of GKS elements. The third is also a utility package providing text manipulation capabilities. The GKS interface, vary in content depending on the level of GKS which has been implemented. That is, the implementation must at least contain all of the data types, subprograms, and exceptions defined for that level in the binding specification, plus declarations from all lower levels as described earlier. The three support packages must be provided at all levels of GKS.

The packages above defined in the Ada binding was intended as a preliminary definition. Another possibility is partitioning GKS in the same manner as GKS conceptual model, with the inquiries distributed among the other packages as appropriate. This approach may result in inconsistencies with the level concept. Use of packages defined by other standards should also be considered. For example, the Proposed Common Ada Interface Set (CAIS) defines a package for text manipulation which is more comprehensive than that provided in the bindings.

Naming Conventions

One of the most heated discussions regarding the definition of the Ada language binding to GKS fell in the category of naming conventions for Ada identifiers. The issue has implications for many other high level language bindings. The binding

[illegible] identifiers which closely [illegible] the descriptive names provided in the GKS standard. Since no uniform conventions for data type names are provided by the standard, the names used were selected for consistency with the names used for the functions. Parameter names were chosen to provide consistency throughout the bindings, to avoid conflict with data type names, and to be descriptive enough for use in the body of the function.

Other naming issues addressed the specification of version numbers in package names (they should not be), possible conflict of GKS names with those used by the application program (don't have to use the GKS package), and use of abbreviations. The ANSI subcommittee has defined a set of standard abbreviations to be used in all language bindings. For a language like Ada, abbreviations are not required, but if used, they must be uniformly consistent with the abbreviations defined.

Evolution of the Binding

During the development of the Ada language binding to GKS, a number of issues were identified. Many of the issues were discussed by the ANSI [illegible] subcommittee and other interested [illegible], and tentative resolutions were reached. Some of these issues can be classified as "generic binding issues," or issues which are applicable to [illegible] bindings. The resolutions which have been adopted for the Ada binding will be used as the basis for decisions in developing bindings to other similar languages (such as Pascal and PL/I).

Among the changes which have been suggested for the next iteration of the Ada binding are:

- Allow implementations to define the interfaces for escapes and Generalized Drawing Primitives, and contain them in a new package GKS_EXTENSION.
- Specification of a package GKS_STANDARD, which would contain implementation-dependent definitions and constants related to the [illegible] configuration.
- Make GKS a generic package. Types for world coordinate [illegible] segment names, and color table indices are among the possible candidates as generic parameters.
- [illegible] functions instead of procedures for the GKS inquiry functions. Make an attempt to improve the correspondence between the GKS [illegible] and Ada.
- Make better use of the data abstraction capabilities provided by Ada through greater use of private types and [illegible] operations.
- Avoid use of the predefined type STRING, and provide a new definition for a string data type.
- Employ a means of fine tuning device capabilities with regard to machine [illegible], etc.
- Provide more precise [illegible] [illegible] a greater number of [illegible].

Prototype Implementation

Prototype software was developed which supported a meaningful subset of the GKS functionality. Several application programs were also written which demonstrated the use Ada capabilities, included a graphical description of the features of the GKS system itself. All of the test cases and demonstration programs were written entirely in Ada.

Development environment

The configuration for the development system was comprised of an IBM PC, with 512k bytes of RAM, 1 M bytes of hard disk, a 5-1/4" floppy disk drive, and an 8087 numeric co-processor chip. An AED512 graphics board from Control Systems was interfaced to a Mitsubishi monitor with 1024 x 768 pixels. The graphics system was driven by a Hitachi 7220 Graphic-Display Controller chip.

The software development environment was TeleSoft's Programmer Support Environment (PSE) for the IBM PC, composed of the TeleSoft-Ada compiler (Version 1.0A), a linker, librarian, the A-Kernel operating system, a screen editor, Ada run-time support package, and Ada packages for device I/O, input and output, and machine level interfaces.

The TeleSoft Ada compiler implementation which we used did not support the full capabilities of the Ada language. This presented several problems during development of the prototype system, but the implementation was complete enough to allow us to develop a fairly good working subset of the graphics system. The deficiency having the greatest impact on the prototyping development was the inability to take advantage of the separate compilation feature of the Ada language. Any package submitted for compilation had to contain both the specification and body, so any time that modifications were required in the body of a unit, the entire unit had to be recompiled. Inter-dependencies among the packages further led to recompilation of many other packages, consuming development time which could have been devoted elsewhere.

Device Independent Software

The set of GKS functions implemented roughly corresponds to Level 1a, and included:

- The control functions to OPEN and CLOSE GKS, and control functions to OPEN, CLOSE, ACTIVATE, DEACTIVATE, and CLEAR the workstations.
- The output functions POLYLINE, POLYMARKER, FILL AREA (partial implementation), TEXT (partial implementation), and two generalized Drawing Primitives (GDP1 and GDP2).

- [illegible] of the output attribute [illegible]

- [illegible] of the inquire functions.

[illegible] this system included a single [illegible] workstation with sixteen predefined [illegible] line styles, five marker types, two [illegible] character fonts, one line width, [illegible] , and settable bundles. The [illegible] implemented did not provide any settable normalization transformations.

The internal architecture for the GKS system was [illegible]. Separate packages were designed [illegible] of the following:

- The [illegible] state, one per system, holds the current value of the operating [illegible].

- The [illegible] Description Table, one per system, [illegible] information about availability [illegible] workstations.

- The [illegible] State List, one per system, holds the current state of workstation independent information. In a full implementation [illegible], this list also includes the input [illegible].

- The Workstation Description Table, one per workstation type available in the implementation, describes the capabilities [illegible] workstation. This table cannot be changed by an application program.

- The Workstation State List, one per every open workstation, contains the current settings of workstation dependent attributes.

- The GKS Error State List, one per system, contains information including the current error state.

The GKS device independent software was developed concurrently with the development of the [illegible]nding Interface, but the ABI specification required modification to conform to limitations imposed by the compiler. For example, the binding declares the type of an object from the Normalized Device Coordinate System as follows:

```
type NDC_TYPE is digits PRECISION
   range 0.0..1.0;
```

The following declaration had to be substituted in the prototype code, since programmer-defined floating-point types were not supported:

```
subtype NDC_TYPE is FLOAT;
```

Other data typing facilities which were needed but not supported included integer type definitions, derived types, record typing involving discriminants, and array aggregates. We generally relied on the predefined data types and statically constrained arrays to work around the problems, although the semantics of their use was inconsistent with the intention of the binding specification. Limitations on symbol table size also presented problems in attempting to compile GKS as a single package. We ended up dividing GKS into five separate packages. One package contained all of the GKS interface data types, and each of the others contained a subset of the GKS functions.

Device Dependent Software

The device driver software interface was designed to prototype the capabilities of a draft standard for the Virtual Device Interface. The prototype VDI contains routines for initialization of the workstation, clearing the screen, drawing a line, drawing a hollow or filled rectangle, drawing a hollow or filled circle, displaying a marker, and writing a line of text.

The device driver software was translated into Ada from existing programs written in the language C. This code performed the machine and device dependent functions, such as initializing values of registers and memory addresses, defining bit patterns for character sets and line styles, and performing data format conversions.

The package SYSTEM was incomplete, and hardware addressing had to be accomplished through the use 8086 package supplied by TeleSoft. We also had to work around the lack of representation specifications.

Language Issues

This section discusses issues identified during development of the GKS/Ada software, and which may be applicable to various Ada applications and other Ada standardization efforts.

Extensions

It is the capability in Ada to declare "programmer-defined" operations that prompts this issue:

> Should the binding (or other interface) provide for the definition of basic operations appropriate to the data type, but which are properly extensions to the functionality required by GKS (or other interface requirements)?

Examples of such operations from the GKS binding are:

```
function "+" (LEFT, RIGHT : POINT)
  return POINT;
function IS_IN (ITEM : ITEM_TYPE;
                THE_LIST : LIST_OF)
  return BOOLEAN;
```

Unless the data type is private or limited private (in which case all needed operations should be provided), the user of the type could define his own routines for performing such operations. However, it is advantageous to standardize common operations for purposes of portability.

This advantage is offset by the additional burden which is placed on the developers and maintainers of such interface specifications.

Another issue regarding non-required/ implementation-defined extensions:

> should such interfaces allow the introduction of non-required and implementation-defined operations?

Examples from the Ada binding include:

> possible operations on objects of type GKSM_ITEM_TYPE
>
> new Escape functions or Generalized Drawing Primitives

If these declarations are allowed in the specification, then it is no longer "standard" for purposes of validation. An alternative which is attractive for the Escapes and GDP's is placing them in an external package. However, forcing operations on private types to be in an external package doesn't mesh with the concept of encapsulation of types and operations. It also makes implementation of the operations awkward. Another alternative is to have such extensions contained in a package EXTENSIONS within the package of reference. This way, the user of the extensions must prefix every use of the element with the word EXTENSIONS, or must explicitly indicate use of the extensions through a "use" statement (this is nicer for overloaded operators). The problem of visibility of the declaration of private types is also solved.

Exceptions

Current practice suggests that a minimal number of different exceptions be declared by a program. The philosophy of GKS suggests that the preciseness of description provided through a large number of distinct exception conditions outweighs the disadvantages of dealing with a large number of possible exceptions.

> Which is preferable, and under what circumstances?

An example from the binding in which consolidation of many error conditions under one exception is desirable is the state error. GKS defines eight possible cases in which a state error would be reported, each treated separately by GKS. During execution of an Ada graphics application program, if any one of these state errors were to occur, it indicates a serious logic error in the program. Exception handlers would have to check for all eight of the exceptions to detect if a state error had occurred. It is suggested that a single exception, STATE_ERROR, is more appropriate. It should be noted that because of the way in which GKS defines error handling, the error handling routine would have no way of knowing the context in which the call causing the error was made, and must treat errors uniformly. In Ada, the programmer determines the context in which the exception handler exists. There is some loss of information in using the prescribed approach, since the exact state error is not known. There seems to be a problem with exceptions in general; the exception which is detected by a program may not express the true nature of the cause of the error condition.

Data Typing

Which approach better promotes portability of Ada programs, use of programmer-defined data types, or predefined data types?

For example, a program could choose between the declarations:

```
type NUMERIC_TYPE_A is new FLOAT;
type NUMERIC_TYPE_B is digits PRECISION;
```

A program which uses "B" is not guaranteed that PRECISION digits of accuracy are supported by every implementation. "A" is guaranteed to have some implementation on every machine, but a given implementation may not be appropriate to the needs of the program. Another consideration is that "B" is implemented, but in an inefficient way. It is assumed that "A" would employ the most efficient implementation.

Another issue of interest is tightness of data typing.

> How strongly typed should GKS (or another interface) be?

It is possible to define the GKS function parameters in such a way that a maximum amount of checking be performed on parameters and other data objects. Emphasis is intended on detecting logic errors at compile time, but run time checking would also be performed. The strong data typing of Ada allowed us to off-load checking for many of the GKS errors on to the compiler. This seems to be a good thing, but in order to implement it to the maximum extent, the binding would be lost in a sea of data type declarations which would be confusing at best. There are two points of view to consider as well. The application program desires assurance that the function performs as promised, and the GKS implementation must always check the validity of the parameter values. Both desire to off-load as much checking as possible on the compiler.

Another consideration is "who" detects the error. For example, the value of an integer type parameter should fall in the range 1..5. The type of the parameter might be a subtype declaration which restricts to this range. In this case, a value outside that range would be raised as a CONSTRAINT_ERROR exception to the calling unit, and the called function never gets control. Alternatively, the type of the parameter could be left as an integer, and the value of the parameter checked on entering the function. In this case, the function could raise a more descriptive and distinctive exception, such as INDEX_INVALID.

Functional Mapping

The GKS specification defines several functions which employ a parameter whose contents are interpreted differently based on the value of a record parameter, or the value of one of the components of the parameter (GKS calls this a data record type). There are several options within Ada, including:

- Define a single function using a simple data record type (such as a string), which has a complex interpretation. This solution also includes the use of private types, with associated operations.

- Define a single function using a complex data record type (such as a record with discriminant components and variant parts), which has a simpler interpretation.

- Overload the function using the specific data record types needed.

- Provide multiple unique functions using the specific data record types needed.

Since it is likely that the structure of the data record will change over the life of the system, or that the structure will vary greatly over various implementations, the second option was discarded. The third and fourth options were discarded since it is necessary to be able to inquire the contents of the data record without knowing in advance what the structure looks like. This left us with the first option, and whether to use strings or private types. It made sense to be Ada-like and opt for the private types, and define functions for accessing the components of the various data records. Problems with this are that an ordering is implied in calling the functions to determine which of several possible components may be accessed (similar to variant parts of records). The semantics of this ordering are not implicit in the definition of the functions as they are with the visible record type declaration.

Tool Support

The DoD position that there shall be no Ada subsets avoids the problems associated in binding languages like FORTRAN and PL/I (in which authorized subsets exist), requiring alternative bindings for the same language. However, current Ada compilers support the Ada language to varying degrees of completeness, and it may be some time before validated compilers are available for all machines which could support GKS. It will be even longer before all these features are implemented in an efficient manner. One solution would be for tools such as GKS to employ only those Ada features which are implemented by all compilers[6]. This approach would preclude use of generics, aggregates, overloading, or tasking. Use of language features outside of that subset would prohibit the portability of both implementations of GKS and of application programs which use GKS. The binding defines the use of all of these features except tasking. The prototype implementation described in this paper is one of many alternate mappings of the GKS binding to fit the limitations of a compiler. The position expressed by ANSI X3H7 is that the fullest possible definition for the language should be used. It should be recognized that certain modifications are necessary in the interim and will hinder efforts to verify conformance of an implementation to the standard. However, the Ada Joint Program Office (AJPO) predicts at least six validated compilers by mid-1984[7]. There should be more than sufficient support for Ada as GKS implementations become available.

The use of a compiler which does not provide the full capabilities of the language would result in the inability to implement the system as intended, or to exploit the power of the language as it was designed to gain the benefits of reliability, extensibility, etc. A comparison of the data type distributions in the binding and prototype specifications highlights the differences in the utilization of the Ada language features in order to accommodate the subset compiler (Table 4.4-1).

Most of the differences are in the use of predefined versus programmer-defined data types. Use of generics in the binding allowed the implicit declaration of many additional data types through instantiation of packages for coordinate systems and list manipulation facilities. Because generics were not supported, corresponding data types had to be explicity and individually declared in the prototype. Many of the types provided through the generic instantiations are not necessarily used, however. The table is limited to data type declarations from levels ma and 0a, since that is the level implemented for the prototype. The full binding utilizes record types, private types, and enumeration types much more heavily, as shown in Table 4.4-2.

Conclusions and Recommendations

The following conclusions can be drawn from this experience with Ada for the Graphical Kernel System:

- Ada may successfully be used to program graphics applications at both the machine-dependent and machine-independent levels.

- Graphics programmers from various applications areas should consider use of the Ada Binding Interface, and to report any problems with the binding. The application programs which were written to demonstrate GKS/Ada were written to interface with the prototype specification, and the interface has not yet been fully tested.

- GKS should be studied for possible use as part of the CAIS, including compatibility with the CAIS definition.

	Binding		Binding (Including generically instantiated types)		Prototype	
	Number	Percent of Total	Number	Percent of Total	Number	Percent of Total
BOOLEAN	0	0.0	[illegible]	[illegible]	[illegible]	[illegible]
CHARACTER	0	0.0	[illegible]	[illegible]	[illegible]	[illegible]
Other enumeration type	28	49.1	28	17.[illegible]	26	32.9
INTEGER	1	1.8	1	0.6	1	1.3
POSITIVE	0	0.0	0	0.0	4	5.1
NATURAL	0	0.0	0	0.0	8	10.1
Other integer type	10	17.5	28	17.0	0	0.0
FLOAT	0	0.0	0	0.0	10	12.7
Other floating-point type	8	14.0	14	8.5	0	0.0
STRING	2	3.5	2	1.2	0	0.0
Other unconstrained array	0	0.0	36	21.8	1	1.3
Other constrained array	0	0.0	0	0.0	9	11.4
Record with discriminant						
Variant part	1	1.8	1	0.6	0	0.0
No variant part	1	1.8	37	22.4	0	0.0
Other record	5	8.8	17	10.3	20	25.3
Private type	1	1.8	1	0.6	0	0.0
TOTALS	57	100.1	165	100.0	79	100.0
Types	53	93.0	158	95.8	56	70.9
Subtypes	4	7.0	4	2.4	23	29.1
Derived types	0	0.0	3	1.8	0	0.0
TOTALS	57	100.0	165	100.0	79	100.0
[illegible]	3	5.3	3	1.8	23	29.1
[illegible]	54	94.7	162	98.2	56	70.9
TOTALS	57	100.0	165	100.0	79	100.0

(Table 3.4-1)

[illegible] Distribution for Binding and Prototype
(Levels 0a and 0a only)

- [illegible] should be used regarding [illegible] and [illegible] of [illegible]

- [illegible] should be required in [illegible] mapping from non-Ada system [illegible] into Ada, especially in the [illegible] error detection and handling.

- An [illegible] and system description [illegible] should be employed as early in the life [illegible] as possible if Ada is to be used [illegible] implementation language.

- It will be difficult to validate [illegible] non-Ada system [illegible] into Ada (that is, if [illegible] bridge to Ada are to be used).

- The full power of the Ada language should be employed in defining the system, and any modifications and/or optimizations performed later.

- The programming support environment is very important. A non-validated compiler can be a lot of trouble if features likely to be needed are not implemented. The same goes for support packages. Validation also does not guarantee that the run-time system provides the necessary support for a GRS implementation.

- A programming support environment should include a sufficient program support library. We sorely missed having a math package, or packages for low-level I/O. A validated compiler should be employed.

	Binding		Binding (Including generically instantiated types)	
	Number	Percent of Total	Number	Percent of Total
BOOLEAN	0	0.0	0	0.0
CHARACTER	0	0.0	0	0.0
Other enumeration type	41	45.1	42	17.5
INTEGER	1	12.1	37	15.4
POSITIVE	0	0.0	0	0.0
NATURAL	0	0.0	0	0.0
Other integer type	11	1.1	1	0.4
FLOAT	0	0.0	0	0.0
Other floating-point type	11	12.1	17	7.1
STRING	2	2.2	2	0.8
Other unconstrained array	0	0.0	52	21.7
Other constrained array	1	1.1	1	0.4
Record with discriminant				
Variant part	4	4.4	4	1.7
No variant part	6	6.6	58	24.2
Other record	7	7.7	19	7.9
Private type	7	7.7	7	2.9
TOTALS	91	100.1	240	100.0
Types	83	91.2	229	95.4
Subtypes	8	8.8	8	3.3
Derived types	0	0.0	3	1.3
TOTALS	91	100.0	240	100.0
Predefined types	3	3.3	3	1.3
User-defined types	88	96.7	237	98.7
TOTALS	91	100.0	240	100.0

TABLE 4.4-2

Data Type Distribution for Binding
(all levels)

- The package SYSTEM and Appendix F should be examined for support for machine-dependent requirements.

Acknowledgements

This paper is the result of work performed under contract number F49642-83-C0083, sponsored by the World Wide Military Command and Control Systems (WWMCCS) Information System (WIS) Joint Program Office (JPMO), and guided by American National Standards Institute (ANSI) X3H34, the Language Bindings and Conformance Subcommittee of the Graphics Technical Committee. The author also wishes to acknowledge the contributions of Geri Cuthbert, Sam Harbaugh, and Greg Saunders. The opinions expressed in this paper are not necessarily those of WIS-JPM, ANSI, ISO, or Harris Corporation.

References

1. Schmucker, Sparks, Post, Journey, Cutnert, Preheim, Carson, French, and Skall. "The Language Bindings of GKS," future issue IEEE Computer Graphics and Applications, proposed panel session of SIGGRAPH 1984.

2. Graphical Kernel System (GKS), Version 7.2, draft proposed American National Standard of ISO/DIS 7942, ANSI document X3H3/83-25R1, ANSI Project 362, 19 July 1983.

3. GKS Binding to Ada, draft American National Standard, ANSI document X3H3/83-95, 18 October 1983.

4. Ada Programming Language, ANSI/MIL-STD-1815A, 22 January 1983.

5. Hopgood, Duce, Gallop, and Sutcliffe, Introduction to the Graphical Kernel System, Academic Press, 1983.

6. Saib, Sabina, "Making Tools Transportable," Kernel Ada Programming Support Environment (KAPSE) Interface Team : Public Report, Volume I, 1 April 1987.

7. "Ada Joins the Army," Defense Electronics, December 1983, p. 70.

8. GKS Ada Specification and Prototype Software: Final Technical Report, Contract No. F49642-83-C0083, 18 October 1983.

9. Ada Interface of GKS 7.2 Issues List, ANSI document X3H3/83-XX, 7 December 1983.

10. Draft Specification of the Common APSE Interface Set (CAIS), Version 1.0, 26 August 1983.

Biographical Information

Kathleen A. Gilroy is a Senior Engineer at Harris Corporation in Melbourne, Florida, where she has been employed since 1982. Ms. Gilroy is a member of the Methodology Group, chartered to research and develop advanced software engineering techniques, methodologies, and tools. She also works closely with the Programming Support Environment Group, responsible for developing a comprehensive automated software engineering environment for real-time software systems. Ms. Gilroy's current work is as program manager responsible for the evaluation of Ada compilers and APSE's, development of a trial real-time multi-tasking problem, and performing an upgrade to Harris' Ada PDL Guide. Previously she worked on Phase I of the GKS/Ada project, where she was the principal developer of the Ada Binding Interface to GKS. Ms. Gilroy received her B.S. degree in Computer Science from the University of Central Florida in Orlando, Florida.

MILITARY COMPUTER FAMILY OPERATING SYSTEM: AN ADA APPLICATION

Frederick E. Wuebker
RCA Government Systems Division
Missile and Surface Radar
Moorestown, New Jersey

Summary

The Military Computer Family Operating System (MCFOS) Program is an interesting Ada* application effort. Not only will the operating system be one of the early Ada applications, but it is also an Ada operating system for a new family of machines and is designed to support fielded, real-time Ada applications programs. Finally, the operating system will be the first Ada program designed to be a formally verified multilevel secure system. This ambitious but completely doable program will certainly stretch the state of the art of ADA programming, if not actually advance it. This paper explores some of the Ada issues that have a major impact on the MCFOS program.

Introduction

In August 1982, the U. S. Army's Communications and Electronics Command (CECOM) at Ft. Monmouth awarded two competitive contracts for a Military Computer Family Operating system (MCFOS). The two contractors who began a competitive definition and design phase were RCA and TRW. The MCFOS contract required the definition and top-level design of extensions to the Ada Language System necessary to support the construction of Ada programs targeted to the MCF machines. The contract also required the definition and top-level design of a family of operating systems for the MCF machines. The MCFOS was to provide functional capability to support the variety of applications programs currently in the Army's inventory and those projected for the future. The MCFOS was to be written in Ada and, of course, to support applications written in Ada and targeted for the MCF machines. The MCFOS family is to support the complete range of security requirements, including a multilevel secure operating system that is capable of passing the Class A1 criteria of the DoD Computer Security Evaluation Center. Those criteria require machine proofs of the design verification as well as code compliance assurance. Although proof rules for Ada have not yet been published, the RCA team of RCA, Intermetrics, and Odyssey Research Associates has been examining the language relationship for security and formulating procedural usage rules to insure provability.

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

Brief Description of the MCF Machines

The Military Computer Family Program began in 1979, when the design of an Instruction Set Architecture (ISA) was undertaken by Carnegie-Mellon University under Army sponsorship. The ISA is a 32-bit general-register architecture with byte-addressable memory. This architecture, known as NEBULA and specified as MIL STD 1862B, has the following features:

- Two active context stacks
- Mapped memory with access protection
- Variable-length instruction with variable number of operands
- Use of privileged instructions for resource control
- Vectored interrupts and exceptions
- Virtual I/O with I/O processors
- Explicit addressing of all instruction operands
- Procedure-based control structure with a local register set for each procedure

The Military Computer Family consists of a minicomputer, a microcomputer, and a single-board microcomputer. The minicomputer is suited for large systems, such as command and control, large phased array radar systems, and intelligence data handling systems. The microcomputer is intended for smaller embedded applications such as fire control, vehicle control, or networked communications. The single board microcomputer is a compatible machine intended for use in embedded weapons control, backpack radio control, and intelligent data entry systems.

Table I shows the expected performance for each member of the family.

TABLE I. MCF COMPUTER PERFORMANCE

	Minicomputer	Microcomputer	Single-Board Microcomputer
SPEED	3 MIPS	500 KIPS	500 KIPS
MEMORY	4 Mbytes	1.25 Mbytes	256 Kbytes
POWER	100W	20W	5W
WEIGHT	40 lb	10 lb	0.75 lb

Synopsis of the MCF Operating System

The MCF operating system is to be a family of compatible operating systems targeted for the MCF machines. This family is to satisfy the Army's needs for a real-time operating system to support the myriad fielded applications that will exist on the MCF machines. The operating system is to be written in Ada and is to be compatible with and supportive of applications written in Ada and or embedded NEBULA. The operating system must be efficient and easily understood. It must support multiple Ada programs as well as the multitasking inherent in those Ada programs.

Finally, the family of operating systems must satisfy the security requirements of systems that will range from dedicated secure through multilevel secure. The multilevel secure operating system must be capable of verification as a Class A-I system as defined by the "Computer Security Evaluation Criteria," dated 15 September 1983 and published by the DoD Computer Security Evaluation Center.

An operating system built for multilevel secure applications is faced with stringent requirements. More important, the MCFOS will be the first Ada program to be formally verified.

Ada Issues for the MCFOS

Three major areas of concern must be investigated to insure that the operating system can support real-time Ada applications. These are:

- Control of the machine and its resources
- Real-time performance
- Security

Each of these areas interacts with the others. For instance, control of the machine is not only required to insure that multiple Ada programs can co-exist within the machine, but is also subject to some very stringent rules imposed for security; security, in turn, increases the penalty of overhead for performance. The following paragraphs discuss each of these topics.

Control of the Machine and its Resources

An operating system that supports Ada programs must be capable of terminating each individual program's Run Time Support Library (RSL) to schedule and request task initiation. Although the operating system has no knowledge of the scheduling logic within the program (that is, for example, the operating system does not know the task completion status), it must honor the task request and start the task on the machine.

In effect, the operating system takes the place of the machine control or "nucleus" portion of the RSL for the application program. The operating system must retain control of the machine since the operating system controls the scheduling of the various Ada programs that it supports. Therefore, when an Ada program is suspended, the task context must be preserved and provided to the RSL of the suspended program. As an added complexity, the operating system is also an Ada program that interfaces with the nucleus RSL.

In a multiprogrammed and or secure environment, the operating system must maintain control of the memory management. It must honor the program's request for data objects and insure that the security rules for those data objects are observed. The operating system must also honor the request, by each of the RSLs, for working memory space. The Ada RSL may ask for space in small increments until such time as the maximum memory space required by the program is achieved. The RSL retains that maximum amount until the program completes or is terminated. For the operating system, this creates the problem of providing small increments and having the memory returned in a large block. The memory management for MCFOS must run mapped because a secure operating system requires paged or segmented mapped memory to assure protection without an extraordinarily complex logic and a high overhead for memory management.

The machine-dependent portion of the RSL, referred to as the nucleus RSL, will actually control the target machine. Many nucleus RSLs will exist but ideally the machine-independent portion of the RSL should be identical for all Ada programs. Two sets of interfaces are involved.

The first set, to insure portability at the source level, consists of user interfaces. The RSL should conform to the common Ada Programming Support Environment (APSE) interface set as defined by the Kernel APSE interface team. As seen now, there will have to be "RSL-like" extensions to that interface definition for various environments. These extensions must be built as a set that does not impact the common set and can be added to the machine-independent portion of RSL as needed for the implementation. The MCFOS requires a set for such functions as interprocess communications, device control, and process control. These are Ada packages that will be added to the common APSE interface set.

The second set of interfaces is much more difficult. This set involves the interfaces between the machine-independent RSL and the nucleus RSL. A common interface should be designed to simplify portability to various target machines. The current set of compilers interfaces with the host machine's operating system, i.e., TeleSoft and SofTech interface with VAX VMS and ROLM Data General interfaces with AOS. Eventually a common interface should be defined between the machine-independent portion and the RSL nucleus. Figure 1 depicts the various interfaces necessary to go from Ada constructs to execution.

Real-Time Performance

The MCFOS is to be a real-time operating system, and efficiency is therefore of paramount importance. The RSL and its relationship with the Ada programs must be investigated and understood. The reader must remember that the Ada

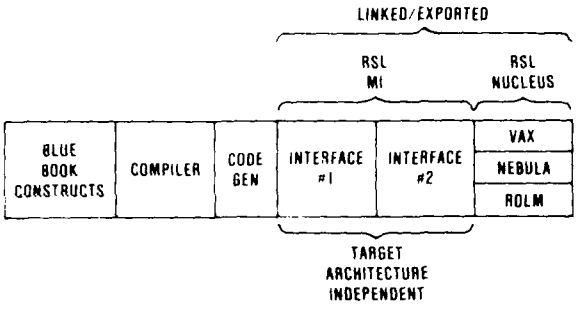


Figure 1. Ada Construct-to-execution Flow.

programs, operating under the MCFOS, communicate through the machine-independent portion of their RSLs to an operating system. That operating system is an Ada program that must verify and interpret the request and transmit the desired action to the nucleus RSL for execution. That Ada program, called MCFOS, must control the application Ada programs its supports.

The NEBULA architecture, is an excellent (though not perfect) instruction set architecture for the implementation of Ada. A measure of the efficiency may be taken from the number of instructions that must be executed to support an Ada request such as task execution, task termination, memory request, etc. To do this an Ada RSL has been constructed and instrumented to provide statistics related to the execution of Ada programs on an MCF computer. The statistics consist of instruction counts and response times for each of the requested Ada operations. Table II shows sample values collected for a multiple Ada program implementation.

TABLE II. ADA PERFORMANCE ESTIMATE

Process	Average Instruction Count
Program Initiation	99
Task Initiation	185
Entry Call	27
Accept	36
Rendezvous Initiation	28
Rendezvous Completion	34
Task Scheduling	108
Task Termination	22

Although it is expected that there will be some improvement in these figures, analysis shows that an order-of-magnitude improvement will not occur. These types of statistics will definitely influence various factors considered in the architecture of an Ada program.

Security

Since the MCFOS is to be Class A1 multilevel secure operating system, the design is to be formally verified and machine-proven. The [illegible] design must then be shown to satisfy the security policy. This "proof" is a manual process that relies on proof rules that reflect the semantics of the data structures and control constructs of the high-order language used. Proof rules for Ada have not yet been published, but the work done by Odyssey Research Associates shows that Ada is verifiable if its usage is restricted. Such restrictions do not violate the "thou shall not subset" commandment. They represent a self-imposed discipline with the same status as an in-house program development methodology.

The restrictions chosen have been assembled from a variety of sources, and although they are not optimum, they do provide a baseline. If during implementation it is found that the restrictions are too stringent, they will be reviewed.

On the basis of the study conducted by Odyssey, a set of nine restrictions will be used. It is felt that Ada programs written using the discipline listed here can be proved correct:

1. No aliasing; that is —
   - do not use a global variable as an actual parameter in a subprogram or entry call.
   - In calls, distinct actuals should be used for distinct formals of mode out or in out, and no such actuals should be used within an expression bound to a formal in parameter.
2. No variant records.
3. No generics with subprogram parameters.
4. Documentation of exception propagation and exception handling; no handling of the task failure execution.
5. No shared variables between tasks, i.e., all tasks communicate explicitly through entry calls and accept statements.
6. No access variables as formal parameters to entries.
7. No dynamic task creation using task types and task access variables.
8. No delay statements or conditional entry calls or timed entry calls.
9. No real types.

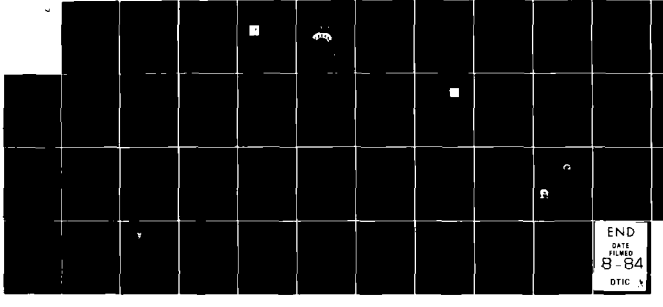
Current Status

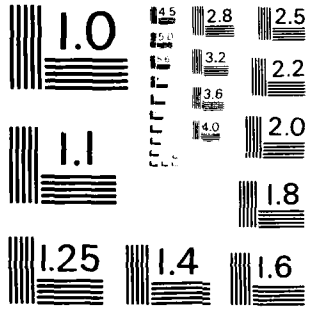
The MCFOS program has completed the Concept Definition Phase, with the top-level design. The second phase, about to start, will be the Interim Operating System phase. This phase will involve building an interim secure operating system that will be a subset of the MCFOS, targeted for the MCF Advanced Development Model machines. Also during this period the formal specification and secure verification of the multilevel secure operating system will be accomplished and the detailed design completed.

During this period the RSLs for the MCF implementation will be completed and the real performance of a full-scale Ada implementation can be assessed.

AD-A142 403 PROCEEDINGS OF THE ANNUAL CONFERENCE ON ADA (TRADEMARK) TECHNOLOGY (2ND)..(U) ARMY COMMUNICATIONS-ELECTRONICS COMMAND FORT MONMOUTH NJ CENT.. MAR 84 2/2

UNCLASSIFIED F/G 9/2 NL





AN ADVANCED HOST-TARGET ENVIRONMENT FOR THE MILITARY COMPUTER FAMILY

Hal Hart
Ruth Hart
Isabel Muennichow

TRW
Redondo Beach, CA 90278

Abstract

As part of the Military Computer Family Operating System (MCFOS) project, extensions to the Ada Language System (ALS) are being constructed which allow software for the MCF computers to be developed and tested in a host/target environment. These extensions are collectively known as the ALSE. ALS facilities are used for editing, compiling, linking, and exporting Ada programs, while ALSE facilities are used to download the software into a connected MCF computer and execute the software on the MCF, thus providing state-of-the-art high level debugging and performance monitoring facilities in an embedded target environment. This paper describes the components of the ALSE from a user viewpoint, concentrating on how an applications programmer would use MCFOS and the Extended Ada Language System to develop software.

Introduction

In August 1982, TRW was awarded a contract by the U.S. Army Communications-Electronics Command (CECOM) to develop requirements and top-level design of two operating systems for the Military Computer Family (MCF) computers. Both operating systems were to be written in Ada and were to be designed to support real-time Ada applications. The Dedicated Secure Operating System was to be run in either a dedicated or system high mode, and was to be optimized for efficiency, while the Multilevel Secure Operating System was to be designed to support multilevel secure applications.

As part of this Military Computer Family Operating System (MCFOS) contract, extensions to the Army Ada Language System (ALS) were to be designed which would allow the user to develop and test software for the MCF computers using the facilities of the ALS. Together, the Extended Ada Language System and MCFOS provide support for all phases of software development: design, coding, debugging, optimization, testing, and deployment.

Ada and MCFOS

The MCFOS project uses Ada in several different ways. First, MCFOS itself is written in Ada. Second, MCFOS supports Ada applications. An applications program can interface with MCFOS in three different ways: through Ada language semantics such as tasking or exception handling, through Ada standard packages such as those for input/output, and through MCFOS packages which provide capabilities beyond those which are part of Ada. Therefore, MCFOS serves as a replacement for and extension of the runtime support library provided with the Ada compiler. It should be noted that the real-time and security requirements imposed on MCFOS require that much of the ALS runtime support library be replaced.

The Ada Language System Extensions

The Extended Ada Language System consists of the Army's Ada Language System (ALS) augmented by the Ada Language System Extensions (ALSE). Together, the ALS and ALSE provide facilities for developing Ada programs targeted to an MCF computer. To do this, a host/target configuration is used, in which the host computer is a VAX 11/780 and the target computer is any one of the three computers in the Military Computer Family - an AN/UYK-41, AN/UYK-49, or the Single Module Computer. As many as four MCF computers can be connected to the VAX via a hardware link.

The ALSE has three components: target dependent tools, software development monitors and a VAX/MCF link. The target dependent tools consist of the MCF/Ada Symbolic Debugger, MCF/Ada Timing Analyzer, MCF/Ada Frequency Analyzer, and ALSE Profile Display Tool. There are two software development monitors: a Single User Monitor to support a single application executing on a bare MCF machine, and a Virtual Machine Monitor to provide a multiuser capability for Ada program development in the host/target environment. The VAX/MCF link consists of a hardware connection between the VAX and the MCF, software to support the link, and the VAX/MCF Loader, which downloads MCFOS/applications software from the VAX into the MCF.

Figure 1 shows the Extended Ada Language System, with shading indicating the ALSE components to be developed as part of the MCFOS project.

Software Development Monitors.

The ALSE has two types of Software Development Monitors, a Single User Monitor (SUM) and a Virtual Machine Monitor (VMM). Both monitors provide a bare machine interface to Ada programs executing on an MCF computer, support the other ALSE tools being used for Ada program development on the MCF, and provide for simulation of unavailable peripheral devices. The Virtual Machine Monitor also provides a multiuser capability for Ada program development in a host/target environment. The monitors execute primarily on the MCF, with the peripheral simulation function executing on the VAX.

All software developed using the Extended Ada Language System will run under control of one of the monitors. The VMM provides the capability of simultaneously running one or more Ada applications for debugging or testing purposes. The SUM provides a more efficient capability for debugging or testing a single Ada application.

Developing Software With MCFOS and ALSE.

In this section, we illustrate how applications software would be developed and tested using the facilities provided by MCFOS and the Extended Ada Language System. It is assumed that the applications software is designed to run under MCFOS.

First, already existing facilities of the Ada Language System are used to develop the software on the VAX 11/780, as shown in Figures 2 and 3. These facilities include the ALS Editor, the Ada compiler with MCF code generator, the MCF Linker, and the MCF Exporter. The MCFOS SYSGEN function then combines the exported load module with MCFOS to produce a tactical system suitable for loading into the MCF computer. Note that this MCFOS function executes on the VAX rather than on the MCF. Figure 4 depicts the process of building a tactical system.

The output from SYSGEN is now ready to be downloaded across the hardware link into the MCF computer. This is accomplished by the VAX/MCF Loader. Three types of loads across the link can occur. First, the Software Development Monitors can be loaded onto a bare machine. Later, applications software that is to run under control of a Software Development Monitor can be loaded onto the MCF. Finally, a system ready to be deployed can be loaded through the MCF directly onto an MCF peripheral, from which it can be loaded onto an MCF in the field. These operations are depicted in Figures 5, 6, and 7.

The VAX/MCF Loader also provides overall control (on the VAX side) between elements of the ALSE and responds to user commands. That is, other ALSE tools such as the MCF/Ada Symbolic Debugger, the MCF/Ada Timing Analyzer, and the MCF/Ada Frequency Analyzer are invoked via loader subcommands.

After the MCFOS/applications load module has been loaded it is ready to be executed. It can either be executed directly, with or without timing and frequency analysis, or it can be executed under control of the MCF/Ada Symbolic Debugger. This tool is functionally identical to the ALS VAX/VMS Symbolic Debugger. It allows controlled, incremental execution of the target program, symbolic display of the state of the program, symbolic display of program entities, and symbolic modification of program variables. Figure 8 shows the operation of the Debugger within the ALSE.

If the software is to be executed directly, a different loader subcommand is specified. Parameters to this subcommand indicate if timing analysis and/or frequency analysis are to be performed. The MCF/Ada Timing Analyzer is functionally identical to the ALS VAX/VMS Timing Analyzer. It monitors the execution time characteristics of Ada programs executing on an MCF computer, by counting how often each block in an Ada program is in control at the end of a specified time interval. Likewise, the MCF/Ada Frequency Analyzer is functionally identical to the corresponding ALS VAX/VMS tool. It monitors the execution frequency characteristics of Ada programs executing on an MCF computer, by counting how many times each block in an Ada program has been executed. In both cases, the collected timing and frequency data are stored in an ALS file (on the VAX system) whose name is specified on the execution command. This data can then be displayed by the ALSE Profile Display Tool, which executes entirely on the VAX and produces histograms which measure how often or how many times a particular subprogram or block was executed. Like the other ALSE target dependent tools, it is functionally identical to its corresponding VAX/VMS tool. Figures 9 and 10 show the operation of the Timing and Frequency Analyzers and the Profile Display Tool, respectively.

Summary

Together with MCFOS, the Extended Ada Language System provides support for all phases of software development, from design through deployment. It allows a programmer to develop, debug, and test Ada

programs designed to be run on an MCF computer using a single integrated system, and allows software to be debugged on a real MCF computer instead of being simulated.

Hal Hart is the Ada Chief Scientist in the Software and Information Systems Division (SISD) of TRW. His main responsibility is to forecast Ada technology needs, and to plan and direct technology development/transfer activities. He is principal investigator for TRW's Ada PDL research project, and he is a member of KAPSE Interface Team (KIT) for Ada environment standardization. Hal was a member of the Air Force Ada Selection Team and contributed to the Stoneman requirements for Ada support tools environments. He previously supported Air Force programs by developing software standards, compiler requirements, and HOL development tools. Hal holds a BA in Mathematics from Carleton College and the MS in Computer Sciences from Purdue University. Prior to joining TRW in 1974, he was an instructor and PhD candidate in CS at Purdue. He co-developed and co-taught Ada courses at UCLA and TRW, and he has collaborated on development of a new Ada certification test. Hal belongs to ACM, numerous SIGs, and the IEEE Computer Society. He is currently an ACM national Lecturer on Ada topics. Hal has been an executive committee member of both AdaTEC & the Ada-Jovial Users Group since their founding. is the founder of Los Angeles AdaTEC, and is past chair of LA SIGPLAN and LA SIGSOFT.

Ruth Hart has worked on the MCFOS project since its inception. She was the Technical Volume captain of the MCFOS proposal, was in charge of the Ada Language System Extensions, coordinated writing of formal specification, and wrote MCFOS Users Manuals. Ruth Hart worked at TRW since 1974 on the design and development of software tools, and the analysis of high order languages. Prior to joining TRW, she was an instructor in the Computer Sciences department at Purdue University for five years. She has an AB in Mathematics from Cornell University and an MS in Computer Sciences from Purdue. She is a member of ACM.

Isabel Muennichow has over 20 years experience in the software industry, both as a software manager and developer. Her areas of expertise include: real time operating systems, support software tools, tactical systems, and man-machine interface. She is currently the project manager of the Military Computer Family Operating System, a project wihich is designing a secure operating system for U.S. Army systems written in Ada. Her last assignment was as Manager of the Real Time Operating System subproject of the Marine Integrated Fire and Air Support System (MIFASS) on the AN/AYK-14 computer. She also managed the system generation work unit on the System Technology Program (STP) and an executive software work unit on a USAF telemetry processing language compiler (MITOL). Her tactical systems experience includes MIFASS at TRW and the Army's Tactical Automatic Data Processing System (TADPS) at Litton Industries. Ms. Muennichow has a BA degree from the City College of New York and has done graduate work at the University of Wisconsin.

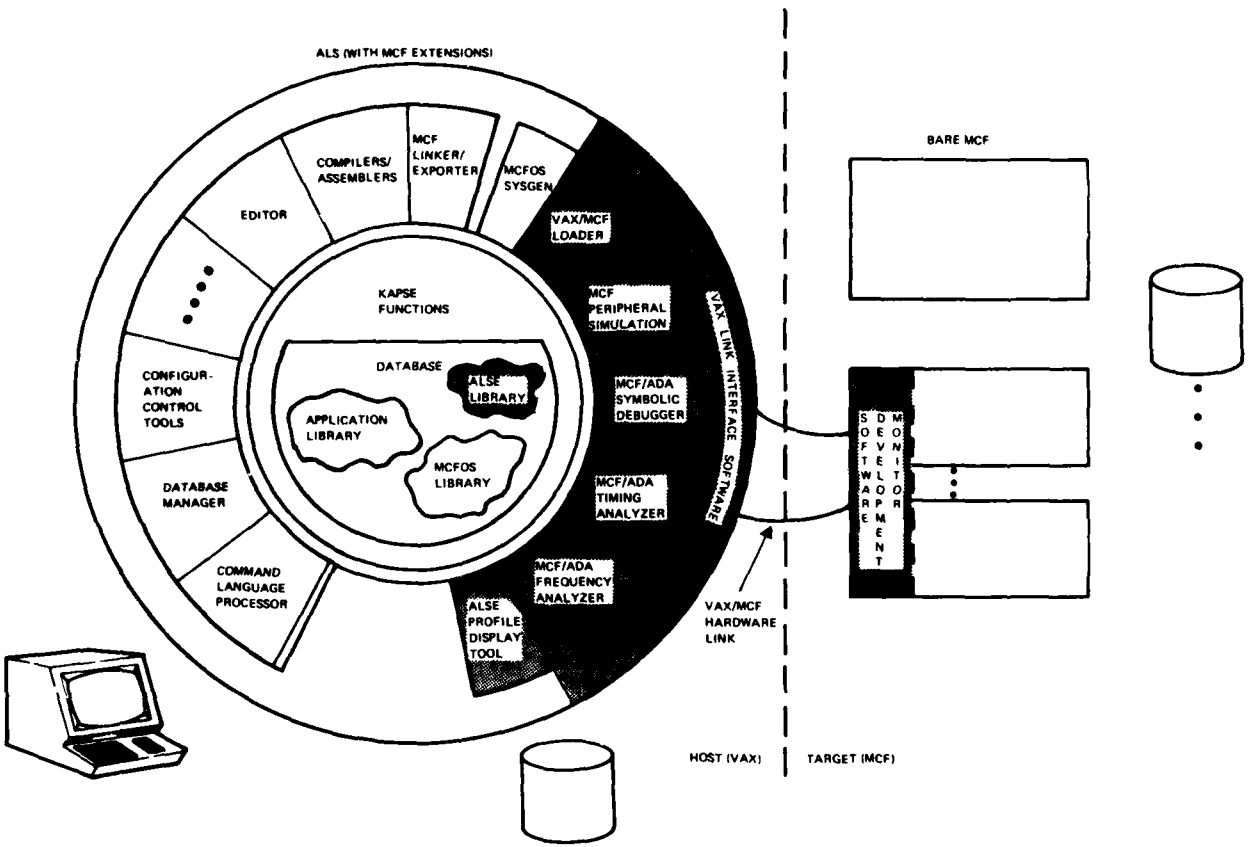


Figure 1: The Extended Ada Language System

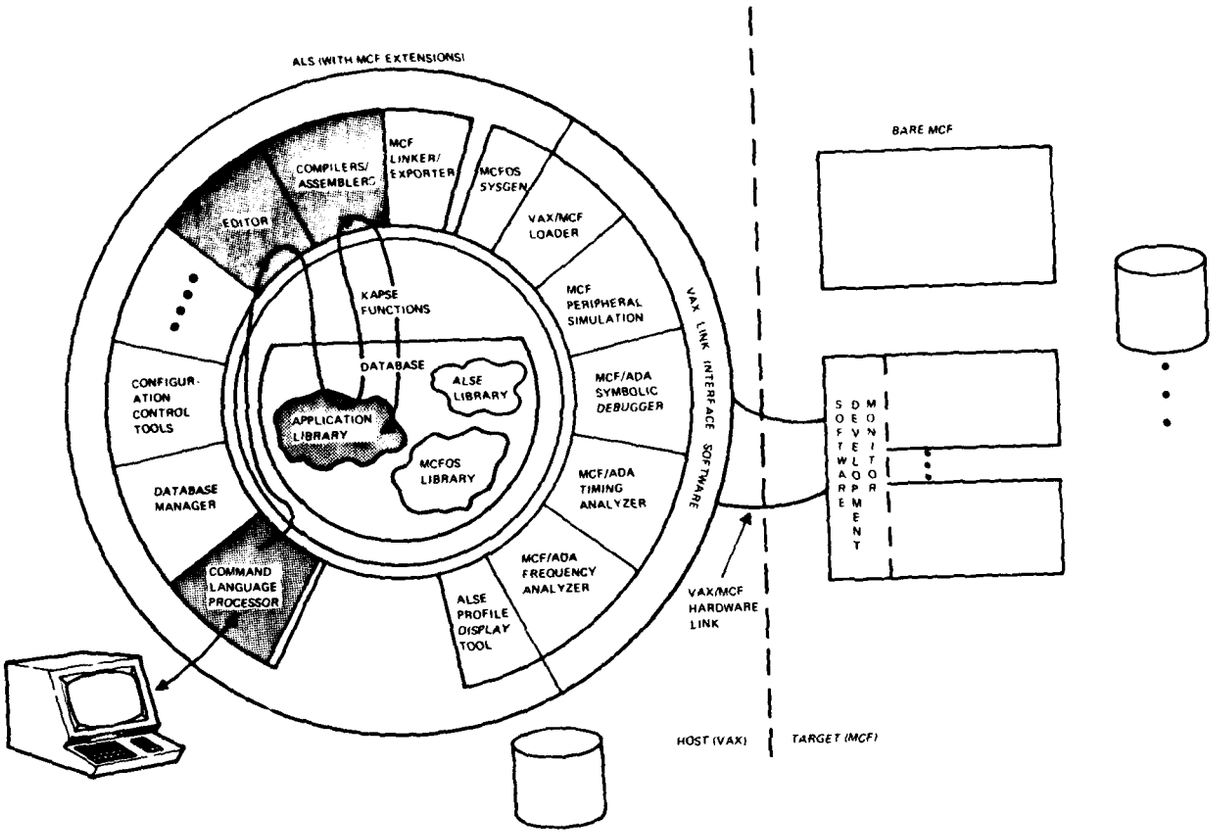


Figure 2: Coding and Compiling

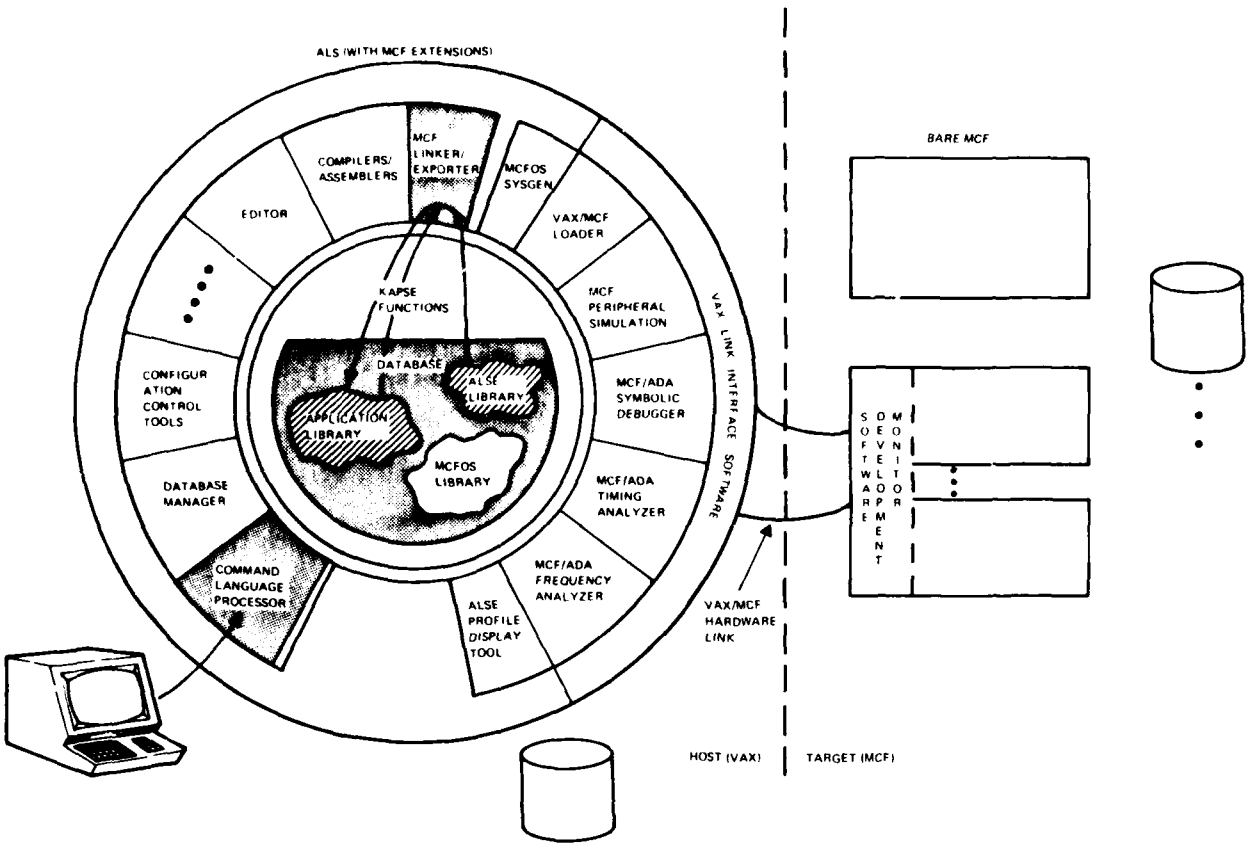


Figure 3: Linking and Exporting

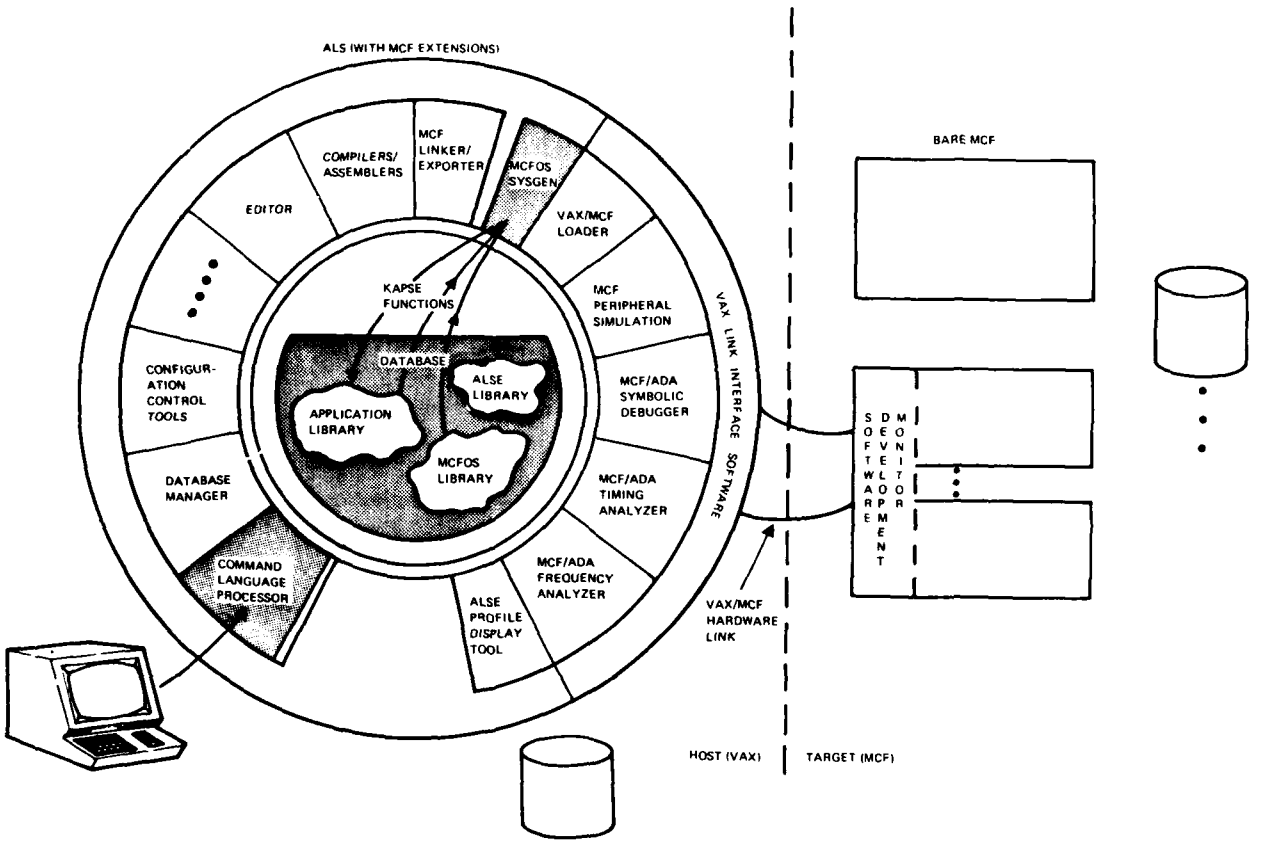


Figure 4: Building A Tactical System

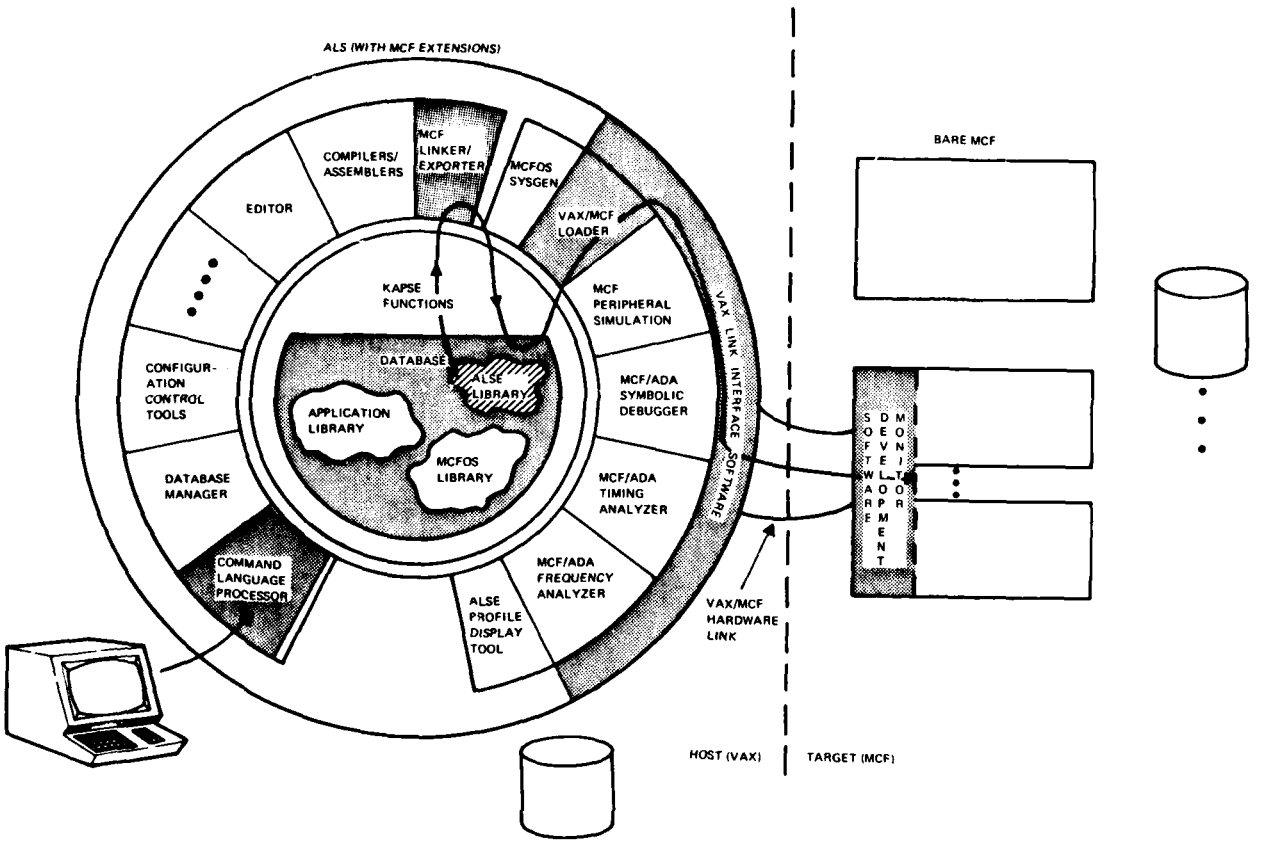


Figure 5: Loading a Software Development Monitor

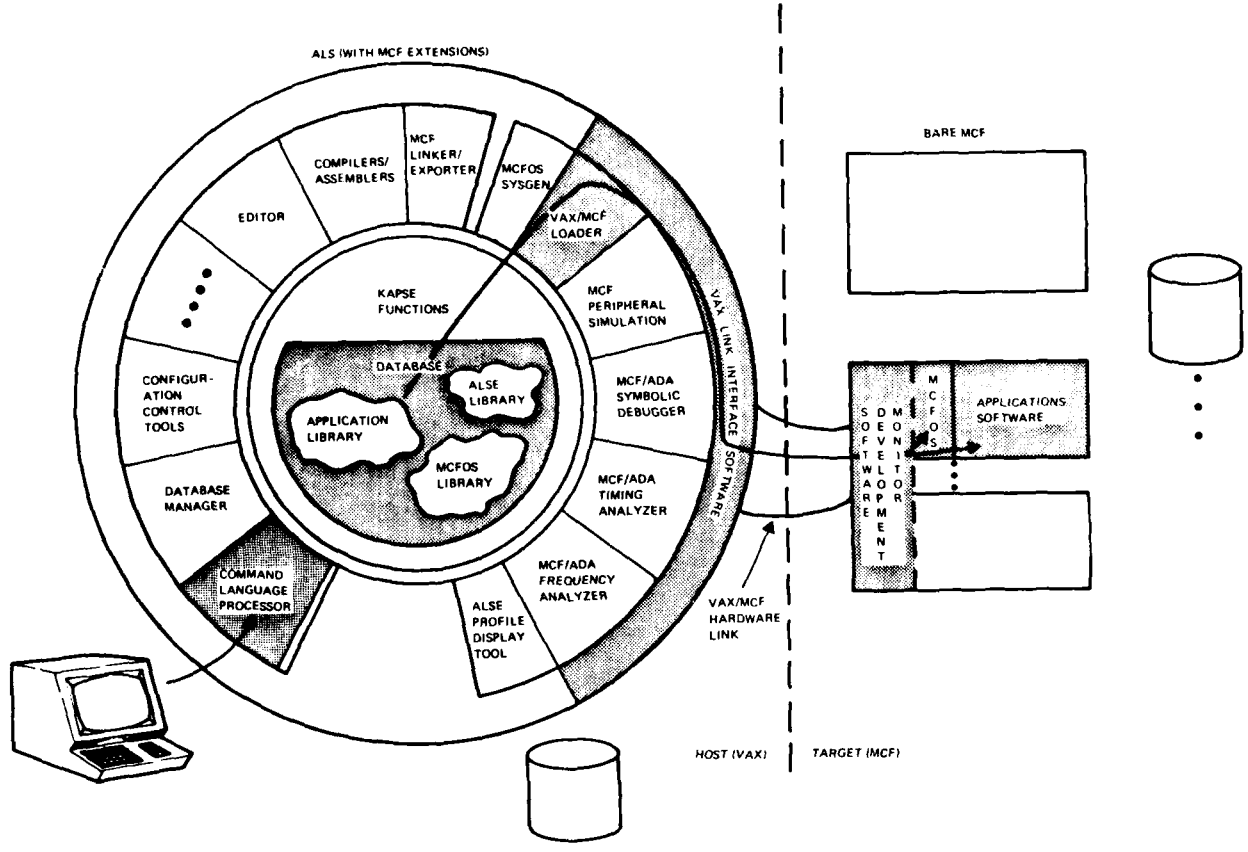


Figure 6: Loading a Tactical System

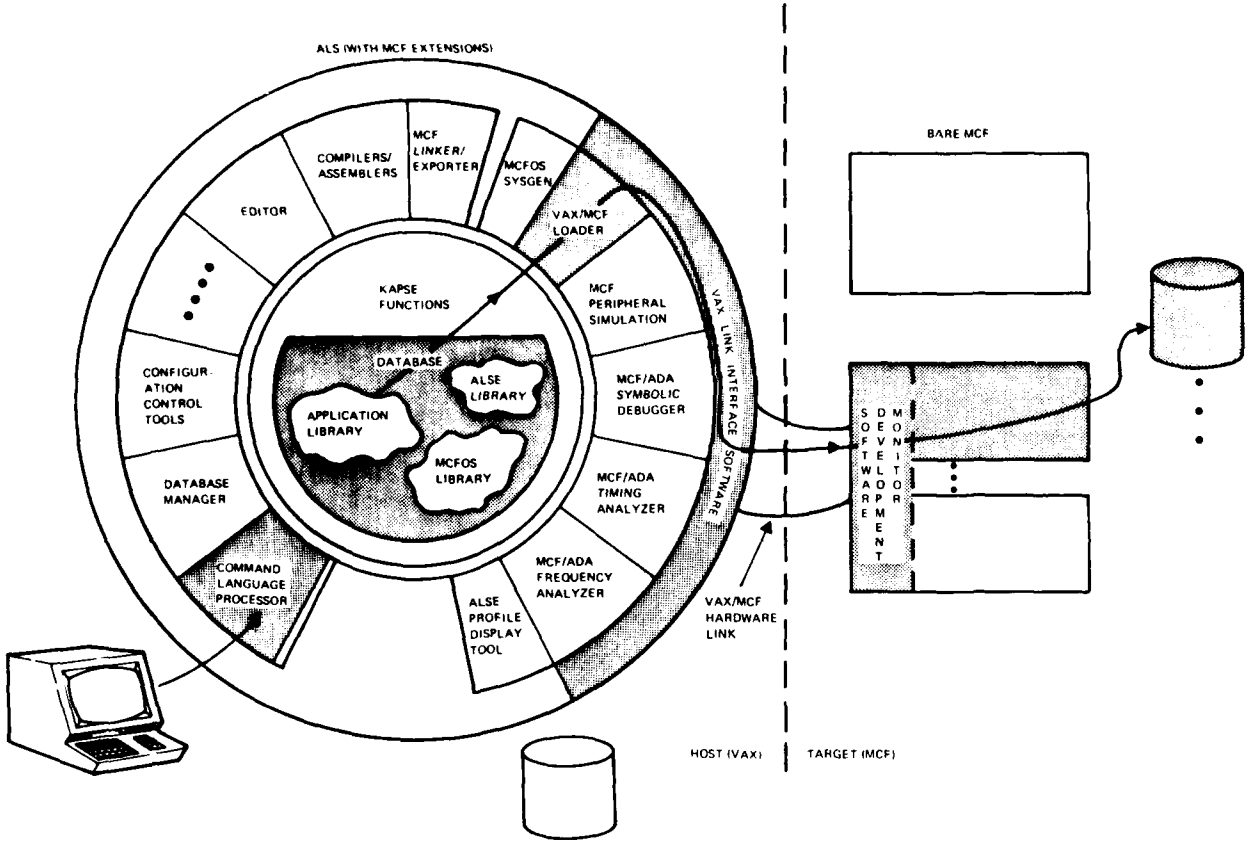


Figure 7: Loading an MCF Peripheral

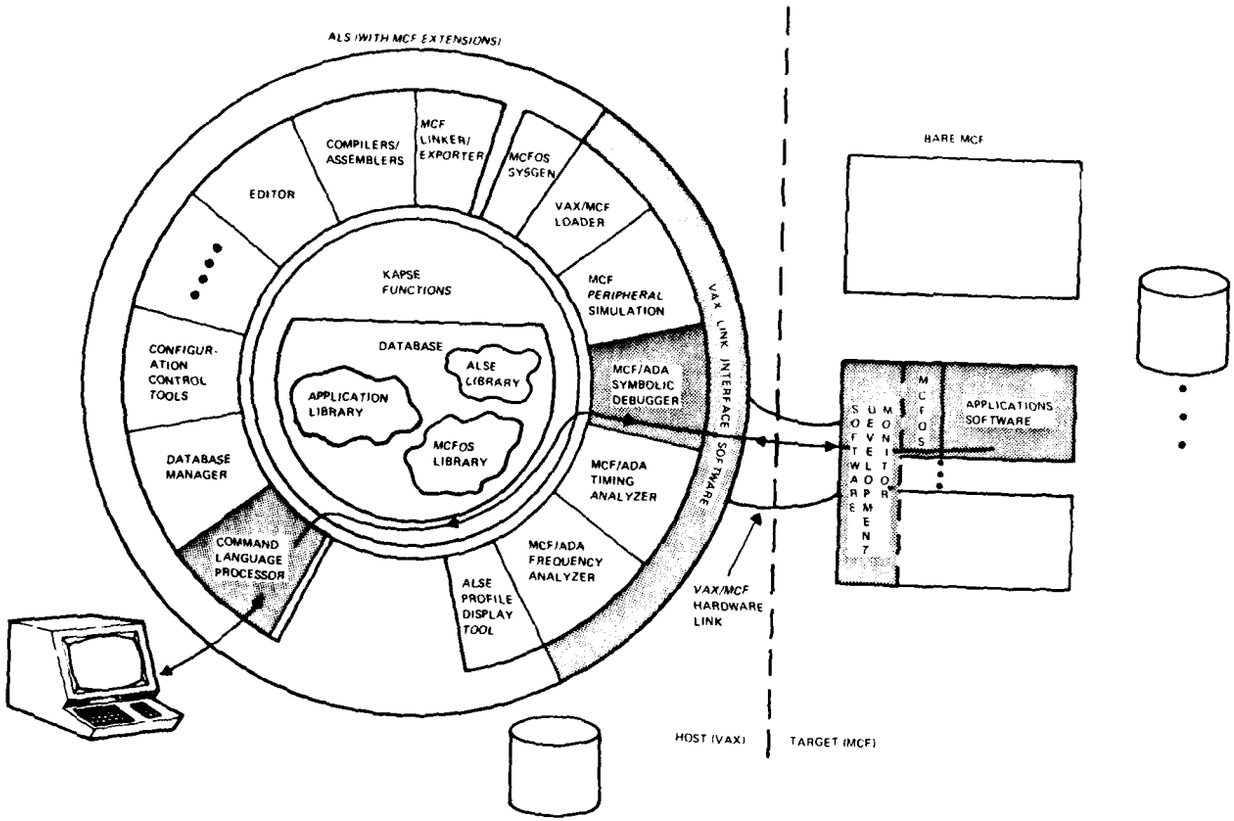


Figure 8: Using The MCF/Ada Symbolic [illegible]

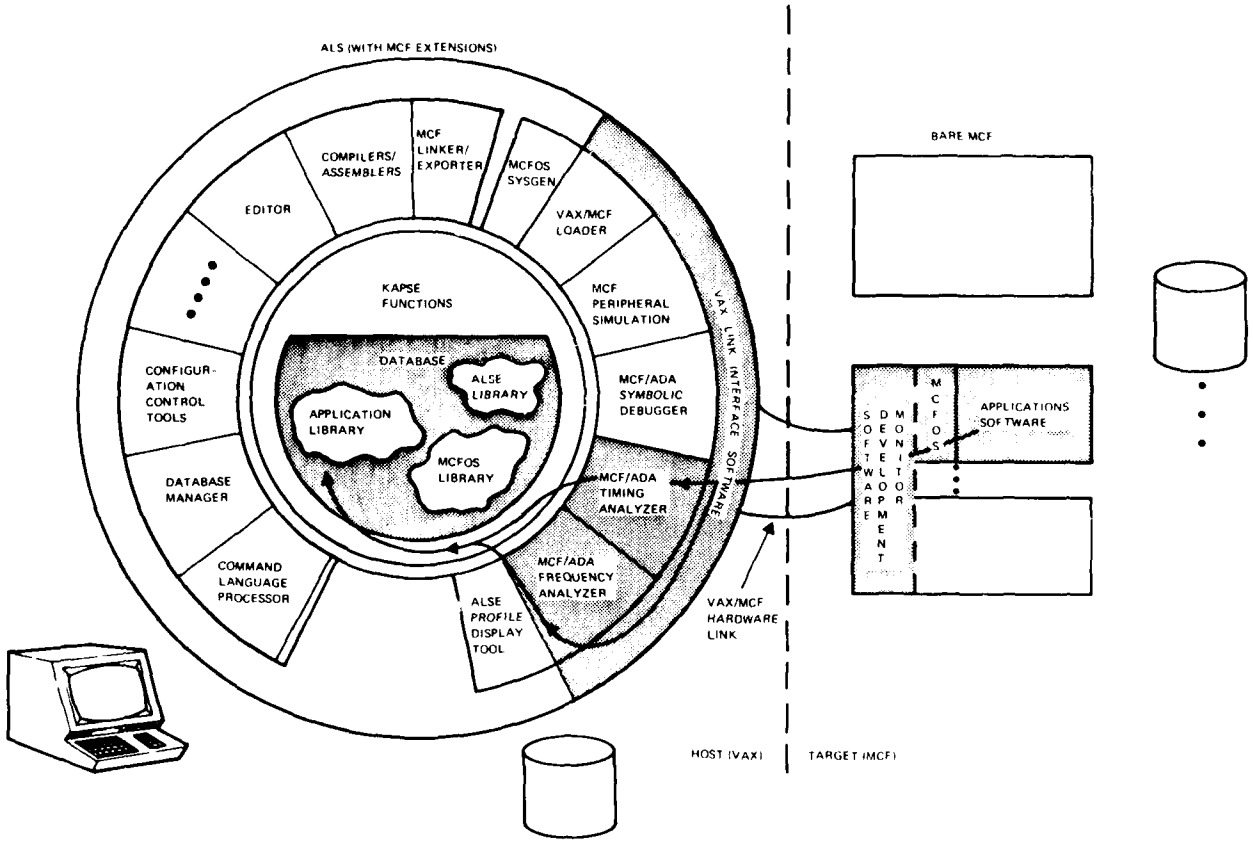


Figure 9: The MCF/Ada Performance Analyzers

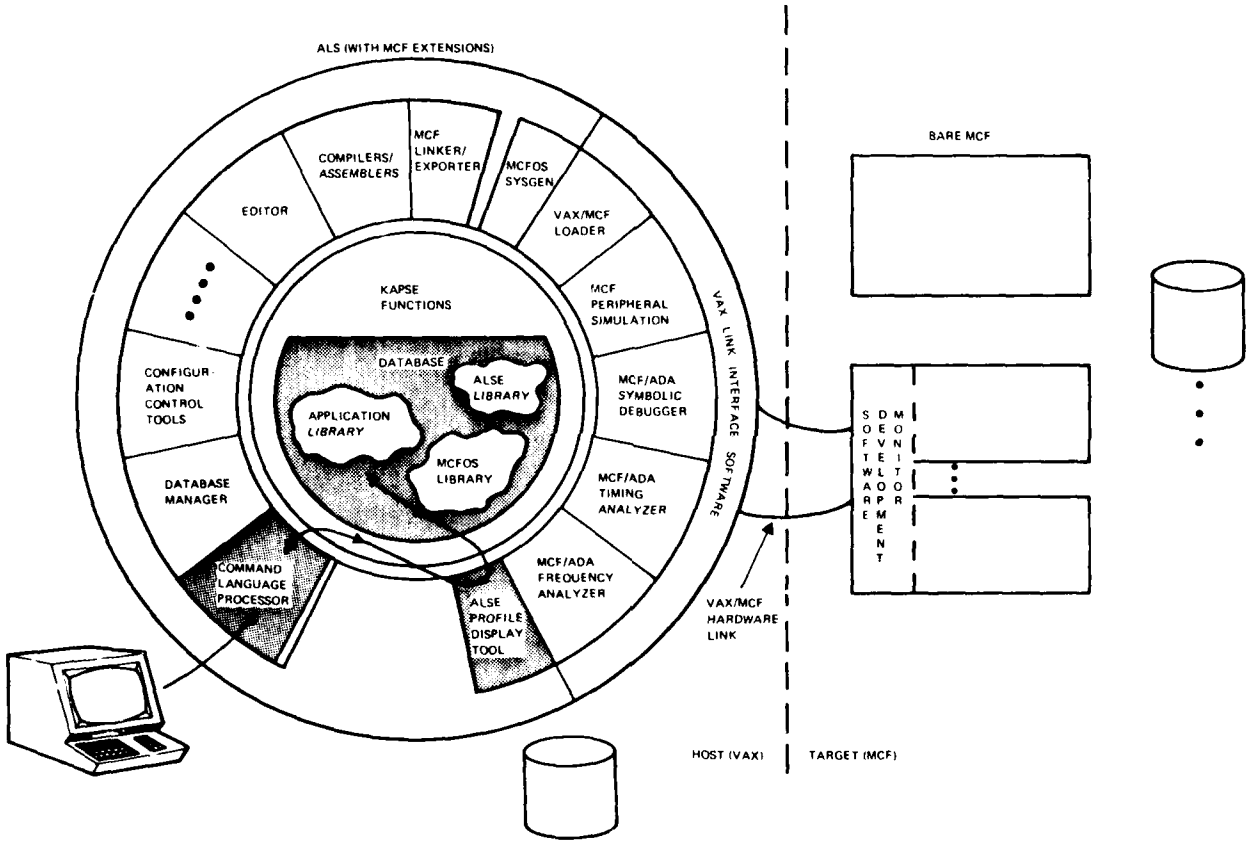


Figure 10: The ALSE Profile Display Tool

MATHEMATICAL SUBROUTINE PACKAGES FOR ADA

BENJAMIN J. MARTIN

Atlanta University
Atlanta, Ga. 30314

ROBERT F. BOZEMAN

Morehouse College
Atlanta, Ga. 30314

ABSTRACT

The authors demonstrate the feasibility of converting the Linpack routines for analyzing and solving systems of linear equations from FORTRAN to Ada. This is done with minimal alteration of the original program structure, thus requiring very little re-orientation by current users of LINPACK.

INTRODUCTION

A number of questions have been raised as to whether Ada is an appropriate language for general scientific computations. In order for Ada to be so used, it is required that a number of software packages now used in FORTRAN be adaptable to Ada. One software package that is widely used and highly developed in FORTRAN is LINPACK.

LINPACK is a collection of FORTRAN subroutines which analyzes and solves various systems of simultaneous linear algebraic equations. It is the aim of this paper to demonstrate the feasibility of recoding the LINPACK routines to the Ada language.

It has been pointed out (see Morris (2)) that a deficiency in the Ada language is its failure to include internal representation of arrays. The FORTRAN standard requires that arrays be stored in column major form, that is, columns are stored together one after the other. This standard along with the absence of strong typing allows the programmer to access the elements of a matrix as if it were a vector and always get the right element. It is this standard among others, that has permitted the development of the LINPACK routines. This paper demonstrates an approach at recoding the LINPACK routines that requires minimum re-orientation by current LINPACK users.

METHODS OF CONVERSION

There are three approaches to the recoding of the LINPACK routines which seem obvious and straightforward. The first approach is to merely insert the appropriate codes for the various BLAS routines in the various places where the subroutine calls are made. This would reduce the work required in performing the conversion. The effect of this insertion on execution time should be minimal since no extra code is being executed. In fact, the overhead of the subroutine call is saved. However, it would significantly increase the space requirements for the program code. It would also destroy the modularity and the readibility of the routines.

The second approach is to process the matrices and vectors prior to the BLAS subroutine call and postprocess them after the BLAS subroutine call. The preprocessor would remove the appropriate portion of the column of the matrix or the appropriate portion of the vector. This approach is placed back in its place in the matrix or vector. This approach maintains the readibility and modularity of the program. The increase in space requirements is minimal in that only four short routines are needed in addition to the usual ones. The increase in time requirements is also minimal since the only thing these routines do is transfer several data items back and forth.

The third approach is to define the matrix to be an array of vectors. After the matrix to be used is properly defined the appropriate vector is sent to the BLAS subroutine. In order to use this technique the vectors to be transferred must be the columns of the matrix. This may require a routine to compute the transpose of the matrix before proceeding. Therefore, the increase in space requirements and in execution time should be minimal. The Ada concept of slices may prove useful in this approach.

THE IMPLEMENTATION

The first alternative was dismissed as being the least attractive alternative. The second alternative appeared to be the easiest to implement quickly, and so was considered first. The third alternative is presently being pursued.

In order to implement the second alternatives, LINPACK and BLAS routines for a general system were coded in Ada. The changes made in the programs were of two types. The first type of changes simply involved the use of structured programming technique making use of control structures not available in FORTRAN. The second type of change involved writing four new routines call CONVRTM, RECONVRTM, CONVRTV, and RECONVRTV. The first two routines work on matrices. CONVRTM takes a given portion of a column from a matrix and stores it in a vector. RECONVRTM takes a portion of a column from a vector and replaces it in a matrix. The other two routines do similar things for a vector. The routines are used in conjunction with the BLAS routines as follows:

The FORTRAN statement CALL SAXPY(N-K,T,A(K+1,K), 1,B(K+1),1)
is replaced by the Ada sequence :

```
CONVRTM(N-K,K+1,K,A,X,1);
CONVRTV(N-K,K+1,B,Y);
RECONVRTM(N-K,K+1,K,A,X,1);
RECONVRTV(N-K,K+1,B,Y);
```

The structure of the Ada package that implemented the LINPACK routines is listed below. The package bodies were coded under ADAED, version 16.3. Because of the nature of ADAED, extensive testing is not possible. A simple system of five equations in five unknowns took 30 minutes of CPU time to compile and execute.

```
PACKAGE LINPACK IS
FUNCTION ISAMAX(N:INDEX;X:VECTOR) RETURN INDEX;
FUNCTION SASUM(N:INDEX;X:VECTOR) RETURN REAL;
FUNCTION SDOT(N:INDEX;X,Y:VECTOR) RETURN REAL;
PROCEDURE SAXPY(N:INDEX;S:REAL;X:IN OUT VECTOR);
PROCEDURE SSCAL(N:INDEX;S:REAL;X:IN OUT VECTOR);
PROCEDURE CONVRTM(N,K,L:INDEX;A:MATRIX;V:OUT VECTOR
  :INC:INDEX);
PROCEDURE RECONVRTM(N,K,L:INDEX;A:OUT MATRIX;V:
  VECTOR;INC:INDEX);
PROCEDURE CONVRTV(N,K:INDEX;B:VECTOR;V:OUT VECTOR);
PROCEDURE RECONVRTV(N,K:INDEX;B:OUT VECTOR;V:VECTOR);
PROCEDURE SGESL(A:IN OUT MATRIX;LDA,N:INDEX;IPVT:
  IN OUT INTVEC;B:IN OUT VECTOR;JOB:INDEX);
PROCEDURESGEFA(A: IN OUT MATRIX;LDA,N:INDEX;IPVT:
  IN OUT INTVEC;INFO:OUT INDEX);
END LINPACK;
```

CONCLUSIONS

While it may not be possible to retain all of the characteristics of the LINPACK routines in a conversion to Ada, it has been demonstrated that such a conversion is possible. Because of the limitations of ADAED, especially its execution speed, any real testing must await a compiler. Nevertheless, it is clear that the routines can be fairly easily recoded in Ada. The costs incurred in this recoding cannot be determined at this time. Other approaches may also be available besides the ones indicated above. The third alternative may be the best of the three. There are indeed unanswered questions, but we must conclude that it is feasible to salvage at least a portion of a "quarter century accumulation of logic and code."

BIBLIOGRAPHY

1. Dongarra, J.J., LINPACK User's Guide, Siam Press Philadelphia, PA., 1979.

2. Morris, A.H., Jr., "Can Ada Replace FORTRAN for Numerical Computation?" ACM, Sigplan Notices, Vol. 16, number 12, (Dec. 1981).

ADA TASKING IN NUMERICAL ANALYSIS

John J. Buoni*

Mathematical and Computer Sciences
Youngstown State University
Youngstown, Ohio 44555

Abstract

Recently the interests in the use of iterative methods for the solution of Partial Differential Equations has been revived. Also, the advent of multiprocessor computer systems will lead many to reformulate much of the existing theory of numerical analysis. It is felt that Ada's portability and rich resources will play an important role in this rekindled interest. The purpose of this paper is to discuss three different implementations of a classical iterative methods for the solution of a numerical problem using several Ada tasks.

I. Introduction.

In this paper we shall comment on some methods for solving linear systems of the form

$$Au=b \tag{1}$$

where A is a given real NxN matrix and b is a given real column vector of order N.

We shall describe three different implementations of an elementary iterative method for solving (1) which uses Ada tasking. Such methods appear to be ideally suited for problems involving large sparse matrices.

In order to illustrate our discussion we shall consider the following model problem: let G(x,y) and g(x,y) be continuous functions defined in the interior, I, and on the boundary, B of the unit square $0\le x\le 1, 0\le y\le 1$. We seek a function u(x,y) continuous in I+B, which is twice continuously differentiable in I and which satisfies Poisson's equation,

$$\frac{\partial^2 u}{\partial x^2} + a\,\frac{\partial^2 u}{\partial x \partial y} + \frac{\partial^2 u}{\partial y^2} = G(x,y) \tag{2}$$

where a is a constant.

On the boundary, u(x,y) satisfies the condition

$$u(x,y)=g(x,y). \tag{3}$$

If G(x,y)=0 and a=0, then (2) reduces to Laplace's equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0. \tag{4}$$

We shall be concerned with the linear system arising from the numerical solution of the model problem using a five-point difference equation. We superimpose a mesh of horizontal and vertical lines over the region with a uniform spacing, $h = M^{-1}$, for some integer M. We seek to determine approximate values of u(x,y) at the mesh points and use the approximations

$$\frac{\partial^2 u}{\partial x^2} \sim [u(x+h,y)+u(x-h,y)-2u(x,y)]\,h^{-2},$$
$$\frac{\partial^2 u}{\partial y^2} \sim [u(x,y+h)+u(x,y-h)-2u(x,y)]\,h^{-2}. \tag{5}$$

Replacing the partial derivatives in (2) and multiplying by $-h$ we obtain the difference equation

$$\begin{aligned} &4u(x,y)-u(x+h,y)-u(x-h,y)-u(x,y+h)-u(x,y-h)\\ &=-h\,G(x,y)\\ &4u(x,y)-u(x,y+h)-u(x,y-h)-u(x+h,y)-u(x-h,y)\\ &=-h\,G(x,y) \end{aligned} \tag{6}$$

To be specific, if G(x,y)=0 and if h=1/5, we have the situation shown in Figure 1. We seek approximate values of $u_i=u(x_i,y_i)$ for i = 1 ...4. The values of u at the points labeled 5,6,... are determined by (3). Thus we have $u_5=g_5$, etc. From (6) we obtain

$$\begin{aligned} 4u_1-u_2-u_3-u_6-u_{16}&=0\\ 4u_2-u_{13}-u_4-u_1-u_{15}&=0\\ 4u_3-u_4-u_9-u_7-u_1&=0\\ 4u_4-u_{12}-u_{10}-u_3-u_2&=0. \end{aligned} \tag{7}$$

II. Vector Product.

Let us consider the following example for a moment and recall the multiplication of a matrix A(i,j) by a vector u(j) where i=1...n, j=1...m and perform the matrix operation Axu. This matrix operation is performed by multiplying each row of the matrix A by the vector u. It is part of the folklore in mathematics that these operations can be performed in parallel.

A partial Ada solution is the following code presented here only for a proper historical approach.

Example 1

```
type Matrix_Row is array(integer range <>)
of float;
type Pointer is access Matrix_Row
task type Vector_Product is
  entry Receive_Value(P  :out float);
  entry Send_Value (Vector1,Vector2:
```

* Dedicated to Bernard J. Yozwiak on the occasion of his sixtyfifth birthday, July 5, 1984.

```
    in Matrix_Row);
  end Vector_Product;
  Task body Vector_Product is
    Product :  float;
    Vector_Pointer1 :  Pointer;
    Vector_Pointer2 :  Pointer;
  begin
    accept Send_Value(Vector1,Vector2:
    in Matrix_Row) do
      Vector_Pointer1 := new Matrix_Row'
      (Vector1);
      Vector_Pointer2 :=new Matrix_Row'
      (Vector2);
    end Send_Values;
    Product:=0.0;
    for i in Vector1 all'range
      loop
        Product:=Product+(Vector1(i)*Vector2(i));
      end loop;
    accept Receive_Value(P : out float) do
      P:=Product;
    end Receive_Value;
  end Vector_product;
```

Although such an algorithm is interesting, for large matrix problems many of the entries are zero (sparse) and the non-zero entries are banded along the diagonal. This motivated Karush et all[8] to derive a procedure for matrix multiplication by diagonals suitable to certain parallel processors. See also Jordan[7] and for other related work Kowalik et all[9].

III. Synchronous Approach.

The prospect of using a multiprocessing computer system to solve linear problems is quite appealing and appears to date back to Rosenfeld et all[10]. If one considers Figure 1, one obtains a grid point stencil of the following form:

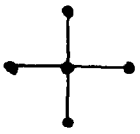


Figure (a).

If we generate as many tasks as there are interior grid points, each interior point and its associated stencil of components could be assigned one Ada task i.e. i=1,...,4, as in Figure 1. The boundary nodes are not assigned any tasks, but instead their values are stored in the tasks that need them, see Figure 1. The data rendezvous between tasks is performed using other coordinator tasks. Before considering a draft of the algorithm necessary to solve this problem, let us consider the following definition:

Definition. Assume T is a task assigned to an interior node. Any other task is said to be a logical neighbor of T if and only if it shares data with task T.

Notice that in the above example, refer to Figure 1, the task at point 1 and at point 2 are logical neighbors but the task at points 1 and 4 are not logical neighbors. An algorithm may now be given

Algorithm. For k=1...iteration-limit

1.) Solve for u(T,k+i).
2.) Send u(T,k+1) to its logical neighbors.
3.) If there is no significant difference between the present and past iterates raise this node's convergence flag i.e. $|u(T,k+1)-u(T,k)|<e$ for some e>0.
4.) If all the tasks have raised their convergence flags, then it is finished else continue.
5.) Accept u(N,k+1) from task T's logical neighbors N.

The equations for the updates at the interior nodes were given in (7). Notice how the tasks for neighboring nodes located at 1 and 4 must send their updated values to that task associated with the node 3 etc...

The Ada code for such a system is for the most part straightforward with the more challenging portion being the design of the communication paths between the various tasks and the protection of the convergence flag. Figure 2 illustrate the communication channels used. Both of these ideas can trace their origins to Hibbard et all[6].

It is worth noting that it is necessary for the coordinator to accumulate all the necessary updated values from all neighboring tasks before they are sent to the proper interior node task which the coordinator is coordinating. Furthermore, the convergence flag is maintained in a protected task and it's code may appear as follows:

Example 2.

```
task_type Protected Convergence-Counter is
  entry Initialize_Counter(Z : in integer);
  entry Increment_Counter(Z:  in integer:=1);
  entry Decrement_Counter(Z:  in integer:=1);
end Protected_Convergence_Counter;
task body Protected_Convergence_Counter is
  Convergence_Counter:  integer;
begin
  accept Initialize_Counter(Z:  in integer)
  do
    Convergence_Counter  :=Z;
  end;
  loop
    select
      accept Increment_Counter(Z:  in
      integer:=1) do
        Convergence_Counter:=Convergence-
        Counter + Z;
      end;
    or
      accept Decrement_Counter(Z:  in
      integer:=1) do
        Convergence_Counter:=Convergence_
        Counter - Z;
      end;
    or
      accept Read_Counter(Z;  out integer:
      =1) do
        Z:=Convergence_Counter;
      end;
    or
      terminate;
    end select;
```

end loop;
end Protected convergence counter;

where the initialize counter is equal to the number of regions being used. In our model, there are four regions.

Obviously, the delay in Figure 2 caused by the rendezvous with neighboring tasks becomes more noticable as the stencil becomes more complex. For example, the elliptic partial differential equation of the form (2) with a being non-zero yields a stencil of the form:

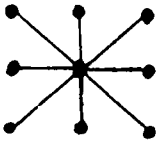


Figure (b).

where each interior node has eight neighbors. Hence, the rendezvous process becomes more involved and perhaps more time consuming depending on the cost of a rendezvous in Ada, i.e., there are eight logical neighbors with which a task coordinator must have rendezvoused prior to redezvousing with the task for which it is coordinating.

For more spacious problems, inner square regions are used rather than nodes. The geometric picture which would appear as in Figure 3 where the interior dots illustrate interior nodes of the interior regions. In our previous example each interior node is a region.

IV. Asynchronous Approach.

With the understanding that the above solution was complicated by the extensive updating of the boundaries of the region, a second approach was taken by Baudet[2].

Motivated by previous work on chaotic relaxation[3,5], he developed an asynchronous approach to the solution of (2). Briefly, the idea is to perform various computations in parallel and to utilize shared memory in order to avoid the many task rendezvous. The benefit of such an approach is the avoidance of the overhead used by tasks when a rendezvous is performed. The cost is that one task may run significantly faster than the other resulting in an unusual number of extra computations.

With this in mind let us consider Baudet's solution of the LaPlace equation as modified by Hibbard et all[6]. In this approach, access is available to all the variables of the matrix (shared) and updates are done at will. An algorithm for this approach is the following:

Algorithm. For k=1...iteration-limit do

1.) Solve for u(T,k+1).
2.) If there is no significant difference between the present and past iterate raise the convergence flag for that region.
3.) If all the region's convergence flags are raised then it is finished else continue.

Since the array u(i,j) are shared variables there is no need to communicate the updates between tasks. However, the tasks must communicate with a central task where the protected-convergence-counter resides. This fosters the need for a coordinator task for each task. Figure 4 explains this communication.

The communication with each of the coordinators as depicted in Figure 4 can get more involved as the stencil is more involved as in Figure (b). Note that the coordinator task is not required to rendezvous with the other region tasks as in Figure 2. Since all the region tasks access the shared discretized values u(i,j) without any additional protocol, this may lead to some unforeseen difficulties. The effects of simultaneous access to a shared variable are discussed in section 9.11 of the Ada language reference manual[4] where strong words of warning are given to those who violate the assumptions given therein. Specifically, some shared variables while being read by one task may be read incompletely due to the fact that it is also being written by another task. Hence, the read may receive a value that is neither the previous value of the variable nor the new value. Obviously, for the solution to be reasonable requires that the operation of reading and writing the shared variable be indivisible with respect to each other. There are other difficulties with this shared variable approach. See Hibbard et all[6].

V. Multi-Coloring.

Consider the model problem (4), partitioned as in Figure 1, the grid points by the Red/Black schemes as shown in Figure 5. The Red points are numbered from left to right, bottom to top followed by the Black grid points in the same fashion. As in Young[12] [p.271], the difference equations may be written in the partitioned matrix form

$$\begin{bmatrix} D & C \\ C^T & D \end{bmatrix} \begin{bmatrix} u_r \\ u_b \end{bmatrix} = \begin{bmatrix} b_r \\ b_b \end{bmatrix} \qquad (8)$$

where D is the diagonal matrix and u_r and u_b denote the vectors of unknowns associated with the red and black grid point respectively. The iteration scheme can be easily written as

$$\begin{aligned} Du_b &= -C^T u_r + b_b \\ Du_r &= -Cu_b + b_r \end{aligned} \qquad (9)$$

and each part of (9) can be effectively implemented by Ada task. Again, the coordinator approach is adopted with a communication pattern as in Figure 6 where the usual centrally located protective counter is utilized.

An algorithm where C is a color and N is an adjoining color is

Algorithm 3. For k=1,2...iteration-limit do

1.) Solve for u(k+1,C).
2.) Send u(k+1,C) to a logical neighbor's coordinator.
3.) Receive u(k+1,N) from the task's own coordinator.
4.) If there is no significant difference between u(k+1,C) and u(k,C) raise the convergence flag.

5.) If all tasks have raised their convergence flags, then it is finished else continue.

The stencil of figure (b) may also be colored with a red/black coloring while more complicated stencils may require more complicated coloring patterns.

Of course, we have just touched the surface of the coloring approach for more complicated stencils one derives more complicated coloring. The muti-coloring approach has been puursued by L.Adams[1], in what seems to be the start of a huge project. In that thesis, various iteration methods, e.g. SOR and SSOR among others are considered. The Numerical experimentation was done on the Finite-Element Machine.

VI. Conclusion.

The above methods are intended for use with an implementation of Ada on a multiprocessor system, that will have some number P of processors. Our N Ada tasks (2r+1 where r is the number of regions/colors) will be scheduled onto these processors by the underlying Ada system. Suppose that P is less than N, or even that P is equal to 1. The above methods can be executed correctly regardless of the number of processors that are devoted to a program and even executed on a single processor. Furthermore, if this be the case then the results can be interpreted in such a way as to guarantee the results when the code is moved to another system.

VII. References.

(1) Adams, L.M. (1982). *Iterative Algorithms For Large Sparse Linear Systems on Parallel Computers.* N.A.S.A. Langley Research Center, Hampton, Virginia.

(2) Baudet, G. (1978). *Asynchroneous Iterative Methods for Multiprocessors.* Journal Association of Computing Machinery, 25, pp. 226-234.

(3) Chazan, D., and Miranker, W. (1969). *Chaotic Relaxation.* Linear Algebra and Appl. 2, pp. 117-128.

(4) Department of Defense (1983). "Ada Programming Language", Ada Joint Program Office, Washington D.C.

(5) Donnelly, J.D. (1971). *Periodic Chaotic Relaxation.* Linear Algebra and Appl. 4, pp.117-128.

(6) Hibbard, P., Hisgen, A., Rosenberg, J., Shaw, M., Sherman, M. (1983). "Studies in Ada Style", Springer-Verlag.

(7) Jordan, T.L. (1982). *A Guide to Parallel Computation and some Cray-1 Experiences.* Parallel Computation, G. Rodrigue, Editor, Academic Press.

(8) Karush, J.I., Madsen, N.K. and Ridrigue, G.H. (1975) *Matrix Multiplication by Diagonals on Vector/Parallel Processors.* Rep. UCID 16899, Lawrence Livermortre National Laboratory, Livermore, California.

(9) Kowalik, J.S., Lord, R.E., and Kumar, S.P. (1983). *Design and Performance of Algorithms for MIMD Parallel Computers.* (preprint).

(10) Rosenfeld, J.L. and Driscoll, G.C. (1969). *Solution of the Dirichlet Problem on a Simulated Parallel Processing System.* Information Processing 68, North Holland Publishing Co.

(11) Varga, R. (1962). "Matrix Iterative Analysis." Prentice-Hall. Englewood Cliffs, N.J.

(12) Young, D. (1971). "Iterative Solution of Large Linear Systems." Academic Press, New York.

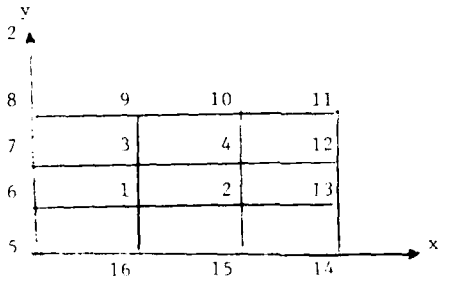


Figure 1

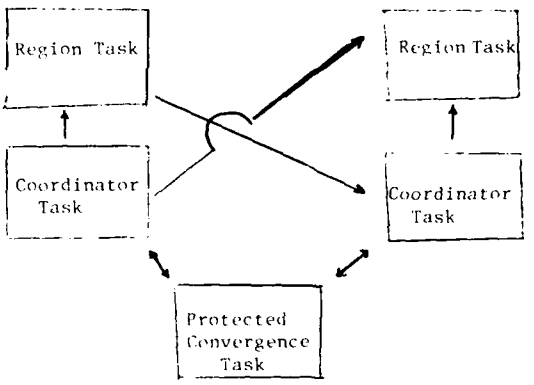
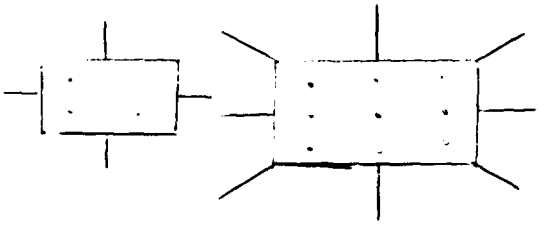


Figure 2



where the ▭ indicates an enclosed region

Figure 3

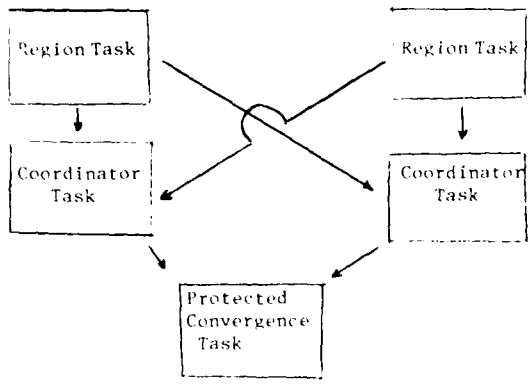


Figure 4

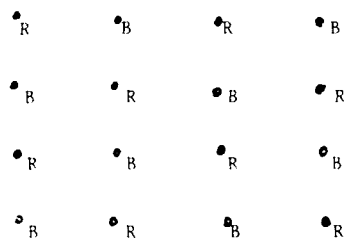


Figure 5

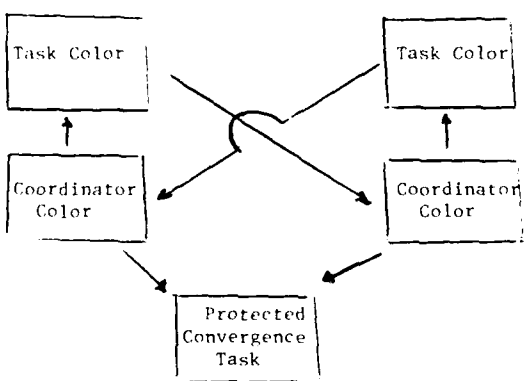


Figure 6

John J. Buoni

Department of Mathematical and Computer Sciences
Youngstown State University
Youngstown, Ohio 44555

John J. Buoni is a Professor Of Mathematical and Computer Sciences at Youngstown State University where he is primarily engaged in the instruction of Computer Science. After completing his doctorate at the University of Pittsburgh in 1970, he joined Youngstown State University. He spent the '78-79 academic year visiting Kent State University. Much of his research has been in various aspects of Pure and Applied Mathematics. He participated in the Army Research Faculty Program at Fort Monmouth, New Jersey where he became involved with the Ada language.

Ada and Statistics

Arthur M. Jones

Morehouse College

This demonstrates a method by which a small college computer science department can introduce Ada into the curriculum without the burden of costly additions to its faculty. It is suggested that such a department should enlist the support of non-computer science departments as conveyors of Ada in the Problem domain.

The example cited here illustrates Ada as a vehicle to describe a statistical problem in data analysis.

This is intended to address the challenge to the computer science program of the small college presented by the technological changes brought forth by the Ada programming languages. Undoubtedly the introduction of Ada will spawn sweeping changes in software technology, for otherwise the Ada initiative will have been a massive failure. Small colleges will be hard pressed to instantly accommodate such program perturbations without an increase in faculty size.

A close examination of the situation, however, suggests that the outlook may not be as dismal as it appears on the surface; to the contrary, Ada may indeed force us to develop a curriculum which will better serve the interest of the student and the nation at large. Traditionally the computer science department at a small liberal arts college has grappled with the dual mission: 1) to educate the student with sufficient breadth and depth in the theoretical framework of computer science so that he/she can cope with graduate school; 2) to provide practical experience in applications sufficient to prepare the student for immediate employment. Yet, in many cases, each graduate tend to be strongly skewed toward one of those two poles, despite the diligent efforts of his department to promote some balance between them. By the time he graduates, the student who prefers applications to theory is usually a programmer delux, who hacks away at his stylized code in a dialect that only he can fully comprehend. Now, if Ada can successfully modify such behavior among the practical software developers, then surely it must be of some benefit to the student before his poor habits are formed.

It is suggested here that, first and foremost, the computer science major should be trained to become a good problem-solver. He should be well-informed that though his solution may be cast in a computer science context, his problem usually has roots in mathematics, science, engineering, or business. With this approach it not only will encourage the computer science major to develop a stronger appreciation for non-computer science subject matter, but it may also pursuade the computer science department to forge stronger ties with other departments. The introduction of Ada and software engineering techniques into the undergraduate curriculum may provide the impetus and the opportunity to implement this strategy.

It is suggested here that the computer science department in the small college targets its initial Ada training program to the mathematics, business, and science faculty members. Emphasis should be put on the design goals of Ada, with some hands-on experience with Ada as a programming language. The comnuter science department should provide numerous examples from each area of problems to be specified in Ada. By so doing, the department, with the support of other faculty, can immediately place Ada in the problem domain for the student. The student would then bring to the computer science course some knowledge of the structure of Ada as well as a readiness to apply Ada in the solution domain.

Statistics courses; which computer science majors are generally required to take, are cited as an example of fertile ground for this kind of application. Ada may be used to describe the problem, the data base and the associated analytical models. The instructor may do this without impairing his freedom to specify a method of solution. He may specify that the problem be solved

manually, by a computer program written in OFR-TRAN or some other suitable language, or by a computerized statistical package. The point here is that the exposure to Ada *in a multi-discipline* problem-solving environment may produce a more disciplined and resourceful software developer/designer than the traditional one-dimensional prospective provided by the computer science *department alone.*

The following is an example of problem specification through Ada.

Program Unit Specification

```
With DATA_ENTRY; with DATA_RANKS; with SUM_OF
SQEARES; PACKAGE ANALYSIS, _OF_VARIANCE use
DATA_ENTRY;
    Problem:  statistical data analysis
    data source:  tree population on a south cen-
                  tral Georgia plantation the ex-
                  periment span a 21-year period
    objective:    Compare seedlots relative to
                  survival and growth character-
                  istics
    model:        Randomized Block Design
```

```
Type PLANTATION is
   record
   Plot_NUMBER:  INTEGER range 1..9999 ;
   TREE        :  INTEGER range 1..99;_ _ tree no.
                  within plot
   SEED_LOT    :  INTEGER range 1..9999;_ _ seed
                  source
   REPLICATE   :  INTEGER;
   NPLOT       :  INTEGER;_ _ no. of seedlings
                  plated in plot
   HEIGHT_3    :  REAL;_ _ height in ft. at age 3
                  years
   HEIGHT_5    :  REAL;_ _ height in ft. at age 5
                  years
   HEIGHT_10   :  REAL;_ _ height in ft. at age 10
                  years
   HEIGHT_21   :  REAL;_ _ height in meters at age
                  21 years
   DIAM_10     :  REAL;_ _ breast-high diameter in
                  in. at age 10 years
   DIAM_21     :  REAL;_ _ breast-high diameter in
                  cm at age 21 years
   end record;
   PLOT_NUMBER is experimental unit
   suggested method:  Analysis of variance
                      (assume normality)
   suggested alterative:  Friedman ANOVA by ranks
                          (Conservative)
type fixed is DELTA 0.01 ..999.99

   ANOVA_TEST_STATISTIC:  FIXED
   FRIEDM_TEST_STATISTIC: FIXED

   end ANALYSIS_OF_VARIANCE;
```

ADA AS A PROGRAM DESIGN LANGUAGE — HAVE THE MAJOR ISSUES BEEN ADDRESSED AND ANSWERED?

Robert M. Blasewitz
RCA Government Systems Division
Missile and Surface Radar
Moorestown, New Jersey

Summary

Department of Defense requirements to use the higher order language Ada* will create challenges to developers of military software that encompass these major concerns: (1) developing a core of Ada software personnel, (2) achieving productivity and software gains that have been targeted as Ada life-cycle objectives, and (3) transitioning to a language that embodies a capability to express software solutions eloquently, clearly, reliably and efficiently. Ada is more than a programming language, it is the basis for a modern perspective of software design and engineering. The IEEE working group on Ada as a PDL has been addressing the issues involved with the use of Ada as a design mechanism for nearly two years. This working group has recently generated a draft guideline that addresses the key issues.

The extent of industry's involvement with Ada PDLs and the status and final form of the IEEE product will substantially impact both the acceptance of the Ada language and the efficiency and correctness of its use.

Introduction — Program Design Languages and Ada

During the past several years, industry has seen an explosion in the cost of generating and maintaining software, coupled with a decline in the quality and reliability of the software product. A need for a radically different approach to the development of software is readily apparent.

One of the first tools for documenting software—the flowchart— was developed from the belief that a program should be documented after it is written. Today, the view is that program design and documentation, at the very least, must precede coding.

A current tool for software design and documentation is the program design language (PDL). PDLs are based on a common theme of software engineering that complex,technical developments require an iterative approach. Other noted reasons for the use of PDLs include:

1. productivity is increased, since the PDL can be used as a design documentation that can be used by ancillary tools to check for completeness and consistency
2. communication complexity is reduced since the PDL facilitates communication at the proper level and in the required detail throughout its use
3. a single notation can be utilized that expresses design throughout all phases of the software life cycle
4. software quality gains are facilitated by the early detection and correction of errors in the design process.

Although most PDLs consist of a mixture of language-oriented control key words and English-like statements to concretely describe an abstract design and concurrently support the above goals, other support objectives can also be noted:

1. focus attention on appropriate levels of design detail without becoming overwhelmed with minor issues
2. provide a process that is amenable to the creation of well-structured programs
3. replace flowcharts and other difficult software tools with an efficient approach to software production
4. provide a natural transition from high levels of logic abstraction into detailed code
5. facilitate program logic documentation and maintenance.

Although not all PDLs accommodate support by tools, some provide listings indented according to the logic and program structures, cross-reference listings for names and subprograms, and detection of structure delimiters. In the past, many different classes of design aids have been referred to as PDLs. These include graphic approaches such as HIPO and structured flow charts; requirements oriented tools such as the Software Specification Language and the Problem Statement Language; mathematical representations of design such as Higher Order Software Specification and the P and V notations; and programming language-oriented tools such as the Caine, Farber and Gordon PDL, the IBM PDL, and the Program Design and Documentation Language. Most of these approaches can be eliminated from consideration as true PDLs, except for the classes involving graphic representation and programming language-oriented methods.

Both of these methods place primary emphasis on describing software algorithms or data. Programming language-oriented PDLs are a special class in themselves, in that they

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

are easily adapted to a computer-based development scheme. PDL descriptions can be easily entered and refined with a simple text editor. In fact, if the PDL is based on the implementations programming language, the source code can be created directly from the PDL description using the text editor.

The work that has been initiated by the IEEE working group on Ada as a PDL has focused on resolving the issues associated with using Ada as a program design language. Ada is a prime candidate for a PDL since it meets most, if not all, of our requirements for a PDL as stated previously. It is also of importance because of recent government direction to use Ada-based PDLs in responding to RFPs. The IEEE product, at the present time, specifies the features or characteristics of a design language that is based on the syntax and semantics of the Ada programming language (ANSI/MIL STD 1815A). A design language, as defined by the working group, is a textual language for the precise and concise expression of program design and one which provides a friendly vehicle for communicating and expressing software designs. The design language is a tool to be used throughout the life cycle of the product; it must be simple, human engineered, precise, verifiable, and supportive of existing program methodologies to the same extent that Ada supports these design concepts.

The current Ada as a PDL guide includes direction on the following major issues involved with program design:

- life cycle support
- methodology of support for life cycle
- features
- properties
- support mechanisms
- language issues
- development support environment
- human factors issues
- management issues
- PDL alternatives

The present product does not specify the following elements:

- a single Ada design language syntax
- the programming language or group of programming languages in which the system described in the design language text is to be implemented
- any system methodology to be adopted under the Ada design language
- the system or method by which design language text is represented, stored or processed

The selection of either a guideline, recommended practice, or standard is accomplished by means of a consensus of technical opinion within the IEEE working group.

This group is directing its efforts to avoid any damaging effects on present corporate investments in PDL design. However, the IEEE guideline will be directly influenced by the lessons learned from these developers and will hopefully bring together the wide scope of work in the PDL area. The availability of a PDL tool in the Ada Programming Support Environment will also foster development of DoD software throughout the life-cycle of the software, if such an end product can be realized in a timely fashion. (Some of the technical problems associated with the IEEE effort will be detailed throughout this report.)

Program Design Languages — Why a Common Ada PDL?

The major issues of modern software development stem from the costs of software development, use and maintenance. The growth of the software development process has shown a chaotic pattern from the beginning. Even now, after 25 years, we find little conformity in the specifics of software development. It is also clear that there are not enough trained software professionals to meet today's demand. And this situation is steadily growing worse. These issues have become increasingly clear in recent years, virtually crying out for an intelligent, planned approach to the problems. The United States Department of Defense, largely because of the visibility of its needs in this area, took the lead in the mid-1970s by sponsoring development of the Ada language, which directly addressed the major "software crisis" issues. The objectives of the Ada language are summarized here to illustrate the common thread of interest between the rationale for the Ada language and a common program design language based on Ada.

DoD initiated the Ada program to save taxpayer money through *standardization*. These savings will come from the portability of reuse of operational software, more effective use of support software (such as program design language), improved programmer productivity, and reduced software maintenance. There is little question that the entire software industry is in need of a modern, efficient, and highly portable system-implementation *language* and *toolset*. Technical arguments about which language is best really miss the point, for only *Ada* and its accepted *toolset* will benefit from DoDs investment in standards enforcement.

Traditionally, a Program Design Language (PDL) has been a means for program description and recording. It is now believed that a PDL need not be limited to these two activities. The scope of the PDL should be expanded to include correctness assessment and possibly the reusability of designs. Ada's strong typing, packaging mechanism for interface definition, and scope and visibility rules provide for increased checkability of design. Therefore, the increased analysis provided by an analyzer of a rigorously defined PDL based on

Ada should result in both increased reliability and maintainability of delivered systems, while fostering a superb means of communicating the design process. The incorporation of increased analysis of a PDL emphasizes the incremental validation or at least verification of a design during

the design process. The need for such aid need not be repeated here; it is obvious that early detection of design errors in the software life cycles is critical to schedule and cost constraints.

Ada is new and relatively untested as a PDL, but nevertheless meets most of the requirements of a design language. Although Ada programs are not particularly easy to write, the complexity of the language exists largely to enforce good programming practices.

The major progam design features supported by Ada include:

- packages
- subprograms
- generics
- tasks and task types
- exception handling
- comments
- pragmas
- types
- stubs

These features are extremely important to the support of the design process, but it should be noted that it is the total collection of these features and their interaction that provides for potential improvement in the existing design process. In the interest of brevity, only a few of these support mechanisms will be explained here.

The concept of the Ada package is thought to be the language's principal contribution to the programming science. Packages permit a user to encapsulate a group of logically related entities. Through the use of two package components (the specification and body), packages directly support the software principles of data abstraction, information hiding, modularity, and localization. Programmers can apply these principles in other languages, but Ada packages encourage and *enforce* these principles. Since the specification and body may be compiled separately, it becomes an easy matter to create the specification early in the software design process and then later to add the body as details about the low level operations are specified. Possibly most important, Ada packages aid in the process of controlling the complexity of software solutions by providing a mechanism with which to physically partition related entities into a logical groupings.

Ada's strong typing mechanism allows a user to define a set of values that objects may assume as well as the set of operations that may be performed on them.

The effects of strong typing enable both domain and operation checking at compile time, rather than at execution time. The goals of strong typing for a language apply directly to the design language also. These goals include factoring of properties, abstraction, and reliability. The interested reader may read of these goals directly from the DoD's *Rationale For The Design of the Green Programming Language*.

The generic is a reusable structure with an abstract parameter list in its definition. It can be viewed as an extension of the familiar "macro" concept. Generics provide a general facility for establishing translation time parameters for program units, thereby promoting reusability. The concept of reusability portability promoted by generics is valuable for many reasons, including:

1. tested generic programs are stable and reuseable
2. they provide the concept of an Ada components industry
3. generic programs need never be redesigned or retested.

All of these features illustrate their value in a design process — designs tend to become more stable as the use of tested and stable components are used as the basis for design.

Other features of Ada that support design include exception handling and tasking. Exception handling provides a complete description of a software system under error conditions as well as the system's response to these error conditions. It encourages a designer to define and to handle error conditions. The Ada tasking model, including rendezvous, task elaboration and activation, and allocators, provides a natural means by which real-time systems can be described.

Tasks can be viewed as independent, concurrent operations that communicate with each other by passing messages for real-time applications. Ada provides this strong facility for multi-tasking or for logically parallel threads of execution that can cooperate in a controlled manner.

Last, but not least, is the Ada commenting mechanism which provides an indispensable means for adding annotations to the language. Extensions provided by the comment mechanism have the advantage that an Ada compiler can be used to process the design language text; the design documentation can also be combined with its implementation and conveniently updated during maintenance and design. In summary, the use of English within an Ada PDL provides representation of high-level design information that can be later refined to a more detailed description. There is, however, a substantial amount of debate remaining on both the substance of allowable comments and their syntax. This issue will be addressed in the "Outstanding Issues" section of the paper.

This short overview has illustrated how the same features that make Ada so desirable as a language also enforce its choice as a design language. Ada is very rigorous; therefore using it in the early phases of the life cycle provides the capability of enforcing analysis and design compliance at a time when the most costly errors are propagated. As far as possible, the system architecture should be described using the constructs provided by Ada instead of the less cohesive form provided by comments.

There is a real danger that highly detailed commentary may actually lull the reader into the dreaded "tar pits" of having code that does not match the intent of the programmer — that is, a design that can be interpreted in more than one

way and is termed ambiguous or imprecise. Since Ada is machine analyzable, the syntax and static semantics specified by the language rules can be checked and analyzed by an Ada compiler alone. A stronger statement along these lines will illustrate that an Ada PDL also provides early prototyping, dynamic semantic checking and automatic simulation since it is an executable language.

The final solution to the basic problems of the software crisis lies in applying modern software methodologies, such as PDLs, that are supported by a higher-order language, such as Ada, that encourages and enforces these principles. The coupling of Ada with an Ada-based PDL offers the software industry a significant inroad into the solution of the software crisis and its inherent problems.

Outstanding Issues Involving Ada-Based PDLs

It can be safely said that the current PDL efforts using Ada as a base language vary in the degree of rigor with which they use Ada. They vary in form from Ada with comments (where the semantics and non-procedural description are included in the comments) to Ada intensive descriptions. Although there is clearly no agreement, as yet, on the exact form of a unique Ada-based PDL, there is general enthusiasm and agreement that key elements of the Ada language directly support program design.

Advocates for using a subset of Ada as a PDL usually endorse the inclusion of English descriptive phrases in the PDL descriptions. The syntax and semantics for the English descriptions can vary greatly, thereby loosening the rigor and increasing the possibility of ambiguity. The exclusion of features from the subsets is also a touchy affair, for possibly the very features eliminated in the subset may be required for a particular design.

Another area of concern is the use of annotations. Annotations usually fall into two distinct categories: those that support a methodology, and those that support a design tool. The Ada PDL descriptions with annotations generally try to stay within the bounds of Ada syntax so that the PDL descriptions can be compiled. The annotations are generally forms of Ada comments and have semantic restrictions relative to other Ada language elements. PDLs described with annotations for tools are also specified such that the annotations are in Ada comments. Processing can then be accomplished by another tool as well as by an Ada compiler.

To summarize the IEEE working group's consensus at the present time, the major issues facing Ada as a PDL revolve around the use of Ada extensions. The alternatives for augmenting Ada with constructs fall into two distinct classes: those that are compatible with Ada compilers (comments and pragmas), and those that are not (pseudo-code and free text). Extensions that use comments and/or pragmas have the advantage that an Ada compiler can be used to process the design language text, and that the design documentation can be combined with its implementation. Although pseudo-code or natural language may have its place, such a language will require special tools and will present an added burden to the overall problem of software management.

In a design language that is to be compatible with Ada compilers, comments provide an indispensable mechanism for adding new constructs to the language. Two forms of comments are possible:

- structured - identified by special characters to highlight the comment as belonging to the formal structure of the PDL
- unstructured - used for natural language explanation of statements made in Ada

Associated with these issues are the semantic rules indicating which constructs, if any, are allowed after the identification of a comment. Presented in a different manner, should the Ada PDL exactly correspond to Ada syntax and semantics at all levels of design or not? That is, can the comment fields be followed by text which in itself is Ada?

Or should it be fully conformant to Ada syntax? This very issue is being discussed and analyzed with possible resolution occurring at the meeting to be held in April. Proponents of the rigorous approach to Ada extensions contend that anything that is inserted in the comment field resembling Ada can be accomplished in the spirit of Ada via another Ada mechanism such as procedures or packages. Others contend that enforcing such rigor increases the level of effort to develop the design, modify the design and maintain the design.

The present product of the group is a draft guideline that has evolved over a period of two years. It can be driven to a recommended practice or even a trial standard, depending on the group consensus. It is presently intended to be used as a document that provides useful information during the process of evaluating a design language and its associated tools. It is also useful to developers of design languages and tools in evaluating issues such as features to be included, tool support, life cycle support, and management concerns and practice.

Conclusions

The Ada programming language provides a means for bridging the gap in software development methodology. Ada, by means of introducing formalized constructs such as packages, generics, concurrent tasks, exceptions, and separate program unit specifications, provides design representation. The use of Ada as a PDL is not only a realizable goal, but one that has been achieved by a number of organizations at this time.

The IEEE Ada as a PDL working group has been chartered to generate, at the very least, a guideline document for Ada-based PDL(s). The derivation of a guideline by the IEEE working group will add to the momentum of the Ada effort and should help ensure both the acceptance of the Ada language and its efficient use as a learning mechanism.

ADA DESIGN LANGUAGE CONCERNS

T. Kaye Grau and Edward R. Comer

Harris Corporation
Melbourne, Florida

Abstract

This paper presents Ada Design Language concerns regarding [illegible] with regard to: [illegible] information exchange [illegible] use of an Ada DL to [illegible] the Ada language through [illegible] and annotation; and [illegible] software [illegible], and Ada Design [illegible] made of the relative [illegible] and of the obstacles to [illegible] standard.

Index Terms - Program Design Language (PDL), Ada DL, Ada design language, specification language, Ada-based methodology.

Introduction

The use of Ada to specify design information has been recognized by the United States Department of Defense (DoD) and by industry, especially those that work on government contracts. The DoD considers Ada as the first step in solving the much quoted software crisis. A large part of the DoD's software crisis is the soaring cost of maintaining software; standardizing on Ada as an implementation language is an initial attempt to cut those maintenance costs. However, maintenance of a program is time-consuming and difficult without accurate design and requirements documentation. At this point, each contractor uses their own methodology, specification languages, and tools for generating the design and requirements documentation.

Some government contracts require the delivery of the development tools with the finished system; others do not. As a result of this lack of standardization, software engineers trying to maintain DoD's software must deal with many different methodologies and specification languages, sometimes without the accompanying tools to assist them.

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

Byron is a trademark of Intermetrics, Inc.

On the other hand, most government contractors have proprietary methodologies, specification languages, and tools which give them a competitive edge in bidding on government contracts. The primary goal of most methodologies is to minimize cost by improving productivity. Checks and balances are levied on the government contractors and their methodologies by the Military Standards and Data Item Descriptions (DID's) specified in the contracts. However, the standards and DID's generally define the information content of the deliverables but not the methods, formats including specification languages, or tools that are to be used to develop the deliverables.

Within this environment, the use of Ada as a Design Language (DL) is currently being included in some government contracts as a requirement. However, there is no standard for the use of Ada as a Design Language. As a result, many people who work for the government, industry, and universities have studied, proposed guidelines, and used Ada as a DL. This paper samples from current research and reports key concerns which have become apparent in efforts to accomplish any level of standardization of the usage of Ada as a DL.

Historical Perspective

Within a few short years of the definition of a Program Design Language (PDL) by Caine and Gordon in 1975 [1], PDL usage had become an accepted, if not preferred, software development practice. The origin of an Ada-based PDL, or Ada Design Language (DL), dates back to 1981 [2]. Yet three years later, Ada DL's are still shrouded in controversy and debate.

Over this three year period, there have been many serious efforts addressing Ada DL's. By early 1982, at least four DoD contractors (IBM, Harris, TRW, and Norden) had defined and were applying an Ada DL to software developments. In May 1982, the *IEEE Working Group on Ada as a PDL* was organized to address using Ada as a PDL. Judging the Ada DL issue to be too volatile to attempt a standard, the IEEE Working Group set about to develop a guideline rather than a standard. Concurrent with organization of the IEEE effort, a PDL/Ada Subcommittee (now the Design Subcommittee) of the ACM AdaTEC was announced. This subcommittee has since sponsored numerous presentations, panel discussions and "Birds of a Feather" gatherings on the subject.

Also in May 1982, the Naval Avionics Center awarded a study to SofTech for an "Ada Programming Design Language Survey" (N00163-82-C-0030). The resulting report in October of that year recommended that "the Navy not adopt a single Ada PDL but rather promote the use of an Ada-based PDL and adopt guidelines for development of PDL's." [3] This effort also determined that there was "no consensus on what an Ada PDL should consist of or how it should be used." [3]

Two years of debate since the flurry of activity in May 1982 has not resulted in an Ada DL standard being generated, nor even approved. Guidelines are not emerging from the IEEE and Navy efforts (though work is continuing). The growing list of Ada DL dialects has grown to include those of at least sixteen corporations and half a dozen universities. Dozens of papers have been published on the subject, elaborating, but rarely introducing new issues.

Most blame these disappointing results on a lack of maturity, indicating that "much more experience in the use of Ada-based PDL's is needed..." [4] This paper reflects the authors' concern over the seemingly slow progress in the standardization of Ada DL's and examines key language issues, focusing on areas of community divergence. Finally the paper will examine the underlying reasons for the divergence and test the hypothesis that Ada DL's are still immature.

Definition of an Ada DL

A single, concise definition of the phrase "Ada Design Language" has not been accepted by the Ada community. Since Ada Design Language (DL) is a merger of two languages used by software engineers, Ada and PDL, a definition of "Ada DL" can be determined by examining the definitions of Ada and PDL.

Ada is a computer language which has been developed by the United States Department of Defense. It is defined by the Ada Language Reference Manual [4]. Druffel has identified the qualities of Ada which support good software engineering practices: "Since Ada was designed by software engineers who intended to use the language, it is not surprising to find a number of Ada features which support modern software engineering practices. Specifically, Ada provides for structured programming, strong data typing, separate compilation, information hiding, data abstraction, encapsulation, separation of specification from implementation, separation of logical and physical concerns, and readability. Not surprisingly, software engineers are discovering that Ada provides for natural expression of design." [5]

PDL as defined by Caine and Gordon is a "'pidgin' language in that it uses the vocabulary of one language (e.g., English) and the overall syntax of another (i.e., a structured programming language)." [6] Pressman in his book on software engineering adds to this definition the following statement: "The difference between PDL and a real high-level programming language lies in the use of narrative text (e.g., English) embedded directly within PDL statements." [7]

One can therefore conclude that the definition of Ada DL should be a language that combines the vocabulary of English and the overall syntax of Ada to provide a means for software engineers to communicate their design ideas.

Herein lies the problem. The term "Ada DL" has been interpreted by many individuals and companies to produce a widely varying set of Ada DL's which range from very English-like to exactly Ada. Many of the Ada DL's have been used for the development of deliverable code and have proven their effectiveness. Several companies have developed or purchased tools to support the usage of their Ada DL.

This article summarizes the different interpretations of the term "Ada DL" relative to several key issues: life cycle applicability of an Ada DL, information expressed by an Ada DL, relationship of an Ada DL to the Ada language, extension of the Ada language through structured commentary and annotation, and relationship between methodology and Ada DL.

Life Cycle Applicability of an Ada DL

The DoD has made a major investment in the development of the Ada language and therefore would like to encourage the application of Ada throughout the development life cycle rather than limiting its applicability to the implementation phase. The degree of Ada's applicability to system requirements, system design, preliminary software design, detailed software design, and hardware design is a controversial issue. The application of Ada early in the development of a system which will not be implemented in Ada has also been controversial.

Virtually everyone agrees that an Ada DL should be used for detailed design of software. Some feel that an Ada DL should be used strictly for detailed design: "...designs should be expressed in Ada when they have reached a stage where they will require no more restructuring, but only refinements. For the structured analysis and design methodology, this stage is achieved after structure charts have been developed, but not before." [8]

Some authors have felt that an Ada DL is also applicable to preliminary design: a DL may be used during early design to "relate the emerging architecture to the mission and performance requirements" which "may require dynamic as well as static checking to compare design to performance" which would result in "automated traceability to mission requirements and system parameters." [9]

ANNA, a language for ANNotating Ada programs, extends Ada to allow the formal specification of the intended behavior of Ada programs at all stages of program development. ANNA can be used "not only for formal verification but also for

specification of program parts during program design and development....Such specification may allow the simulation of interfaces at the development stage and provide the basis for a proof of correct use of a subprogram or package independent of and prior to implementation, as well as a proof of correct implementation." [10]

Ada-based languages were used by General Dynamics on a case study funded by the Army. Ada was used for the specification of requirements, design, and implementation of the system. In particular, the Ada Requirements Methodology (ARM) developed as part of the case study combined the use of data flow diagrams, a data dictionary, a logical data structure model, and an Ada-based structured English. They concluded that ARM could be "used to state and graphically illustrate system requirements (both functional and non-functional)....From experience gained in this project, the researchers felt that ARM could replace the old military A-specification document, which proved unsuitable in adequately documenting the message switch modified by this project." [11]

The use of Ada as a system design language particularly for embedded systems has been described by Wheeler. He concluded that the use of Ada as a system design language encourages designers to use current practices to develop better structures for their systems, and its subsequent use to implement the systems preserves those structures in the product. [12]

An Ada-based language has even been used to design hardware. A Universal Asynchronous Receiver Transmitter (UART) designed by SofTech using an Ada-based language resulted in the following conclusions: "Ada can describe hardware, Ada can do hardware design," and "Ada can specify interfaces without knowing the hardware software boundary." [13]

As a result of these experiences in applying an Ada-based language to various phases of the life cycle, the IEEE Working Group on Ada as a PDL has concluded that the primary phases of the development life cycle which an Ada DL is intended to support are system design, software requirements, and software design. The use of Ada DL's for other purposes such as specification of system and hardware requirements is not ruled out. [14]

Virtually everyone agrees that an Ada DL should be used when the implementation language is Ada. The use of an Ada DL with other implementations languages has been supported by Hart: "Transliterations of Ada-based design (rather than a design expressed in syntactically correct Ada) into currently available languages will be easier because the design contains less syntactic construction which must be changed into the differing syntax of another language; a tailored Ada PDL could even incorporate some syntactic constructions of another implementation language." [15]

Problems which may arise when Ada is not the implementation language have been described by Alstad: "In many cases which may be expected to arise in practice, a feature of Ada used in a design may not map at all cleanly into the implementation language. This will probably lead to confusion or arbitrary choices during implementation, unless designers are prohibited from using that feature of Ada; in that case, it is questionable whether the PDL is still Ada." [16]

Implementation of an Ada-based DL in Jovial has been researched by Bein. He recognized that "certain Ada features do not translate 'nicely' into Jovial. Either the use of these features needs to be constrained, or the goal of close correspondence between design and implementation needs to be sacrificed to some extent." [17]

The IEEE Working Group on Ada as a PDL has made the following recommendation: "While an Ada DL must support the design of systems implemented in Ada, it should not preclude developments not in Ada. The Ada DL should support implementacions in other languages provided that proper user instructions are available. A Coding Standards document is recommended which describes in detail the recommended implementation for the various DL statements and constructs." [14]

In summary, the degree of applicability of Ada throughout all phases of the development life cycle is still being debated. Yet our research has shown that Ada-based specification languages including Ada DL's have been demonstrated to be applicable in most phases of the life cycle and with several implementation languages.

Information Expressed by an Ada DL

Historically, PDL's have expressed only algorithmic design information. Ada's influence and advances in software engineering have led to a broadening of the scope of information expressed in a DL. The complexity and size of software systems being designed currently require expressing the design of parts of the system in comprehensible design units rather than in a monolithic design. Design units stated in an Ada DL must contain two types of information: connectivity with other design units and a description of the design unit.

The connectivity among design units describes the interrelationships such as external data references and external procedure calls. Explicit expression of the coupling between design units supports evaluation of the complexity of the design. The two types of connectivity which must be expressed in design are horizontal connectivity among units at the same level of design elaboration and vertical connectivity between units at different levels of design elaboration. [18]

Horizontal connectivity among units at the same level of design elaboration summarizes references to externally defined data, externally defined process units (procedures, functions, and tasks), and other external interfaces such as interfaces with input/output devices and hardware interrupts. Based on this information, the coupling of this design unit with the rest of the system can be measured.

subset of a programming language while still retaining the concepts needed for design (rather than a programming) language." [1] In reality, there is very little conceptual differences between the purist view and this, particularly if one views coding as merely the final elaboration of a design, "replacing abstractions with target language statements." [1]

Whether by exclusion or convention, only certain Ada constructs are applicable to the early stages of design. The problem arises on determining which ones. The Navy survey found "a definite lack of consensus on the exclusion of any language feature." [22] Because they found that the "omitted features were small in number" it was concluded that "there is little advantage gained, either in encouraging design level documentation or in reducing the complexity of a PDL processor, by omitting these features." [2]

If one accepts that the subset issue is merely one of application, the next major issue arises on whether the Ada language alone is sufficient for all levels of design.

"Deviations from full-syntax Ada for design are mainly founded on the expectation that excessive syntax restrictions will not be accepted by the large veteran populations of software designers. This will be particularly true when those encumbrances (which support no role of software design) distract a designer's attention down to such a petty level of detail as to remove his perspective from the needed global and abstraction planes. Final design refinements and coding (which may become synonymous using Ada) are proper times for conversions of syntax simplifications into proper Ada, and can be achieved by trained coders without impacting basic, early-arrived-at design decisions." [15]

Indeed most current Ada PDL's provide capabilities to embed English narrative and mechanisms for extending the Ada language using annotations. Lindley reports: "The inclusion of English narrative is a key factor in the use of the PDL design rather than code. It is also important in providing the designer with the freedom to be creative. These advantages outweigh the fact that the inclusion of English narrative in situations where comments are not legal renders the PDL non-compilable." [22]

The primary issue regards the use of more classic structured English to describe the function of a program unit's body in place of legal Ada syntax. While not rigorous, this approach more closely follows the original intent of PDL's to describe a "level of design [which] can be understood by people other than designers." [6] "Annotations differ from English narrative in being an addition to legal Ada constructs rather than replacing them. Thus, they can be more formally defined in both syntax and semantics. They are important, as with English narrative, in encouraging design and in aiding the use of the PDL description as documentation. Since annotations are frequently implemented as specific forms of comments, they normally do not prevent the PDL from being processible as an Ada compiler; however, the compiler will not be able to check the validity of the annotations." [22]

Because the extensions and annotations to Ada represent a major point of contention, a more thorough discussion of structured commentary and annotations is provided in a following section.

Ada DL Compilation and Execution

Because of the potential automation advantage in using existing Ada language tools, the compilation and even execution of an Ada DL specification seems attractive. Compilability of Ada DL is currently the most commonly used contractual definition of Ada DL's. Compilability is certainly related to compatibility with Ada syntax and semantics but the advantages and disadvantages of compiling designs are still being discussed.

A requirement to be compilable can certainly be met by Ada DL's which use exactly (or a subset of) legal Ada syntax. "This directly does not preclude the use of Ada DL syntax which is not formal Ada, but merely forces all such instances where the DL departs from Ada to be coded as Ada comments." [14] Hence, a large number of those Ada DL's which extend Ada with commentary and annotations are indeed compilable.

"Execution of an Ada DL is indeed another matter. This means that upon conclusion of the design specification phase, a validated and verified prototype program already exists. Indeed, the two are largely one and the same." [20] While design execution provides the ultimate in validating software designs, opponents of execution of an Ada DL cite the human factors problems involved in developing a rigorous executable design. "There is a trade-off between ease of use and machine processibility. In order to be machine processible, information must be expressed formally in a syntax which may be unambiguously interpreted by the computer. Yet as a syntax becomes more rigorous and complete, the effort to learn and become proficient in the syntax increases." [14]

The authors have previously cautioned: "It is generally agreed that an intermediate design step prior to coding promotes more accurate problem analysis...would it not be just as foolhardy to directly code Ada from scratch (as it was with FORTRAN or PASCAL) without the benefit of this intermediate step?" [18]

In summary, it appears that the major issues regarding the relationship of an Ada DL to the Ada language are clear. Subsets of Ada in an Ada DL should not be defined, though conventions on usage may not make full use of the language. Extensions are also clearly needed. Compilation is recommended, though the level of interpretation and effectiveness of compiler output largely depends on other issues. More work is needed to assess Ada DL execution.

[illegible] connectivity between units of different levels of design abstraction summarizes parent child relationships. In top-down design, [illegible] design units are decomposed into lower-level, more detailed units. This decomposition is expressed in a hierarchical parent child type of connectivity. Whether the children are separate from the parent or nested within the parent [9] the parent child connectivity must still be expressed. In bottom-up design, detailed design units are synthesized together to form larger and larger units until the complete system is built. The child is designed before the parent but the parent child connectivity must still be stated. In either case, the traceability of requirements to design units is used to validate the requirements allocation which is inherently part of the parent child relationship.

The type of information needed to describe the design of the unit can be categorized as data, algorithms, and complementary information. The data encapsulated in and operated upon by the design unit must be defined by type and object declarations. The algorithms which define the step-by-step operations performed on the data must be expressed. Methodologies and standards require additional complementary information which must be stated for each design unit. The complementary information may include performance requirements, limitations, assumptions, assertions, state-machine representations, testing requirements, and any other pertinent information.

Review of currently available Ada DL's indicates that the scope of a traditional PDL has been broadened by the addition of information content. The information which must be expressed by an Ada DL includes not only algorithmic design but also data, complementary information required by methodologies plus connectivity information. The broadening of the scope of PDL's to the scope of current Ada DL's has resulted in confusion about terminology and applicability.

Relationship of Ada DL to the Ada Language

The Ada community has clearly diverged in generating a standard pidgin language merging Ada with English. The issue centers on how Ada-like or how English-like an Ada DL should be. There are two major issues:

a. intersection of the Ada DL with Ada syntax and semantics

b. ability to compile and even execute an Ada DL specification

The alternatives in each of these areas are discussed below.

Ada Syntax and Semantics

As depicted in Figure 1, the intersection of an Ada DL with the Ada language may take many forms. The purists who prescribe that the syntax and semantics of an Ada DL should be exactly Ada reflect a segment of the population who feel the Ada language already has sufficient capabilities for elaborating designs in a meaningful, readable manner. The Ada Language Reference Manual in fact states that "the syntax of the language avoids the use of encoded forms in favor of more English-like constructs" [4] "The significant advantage to this approach is as follows: by adhering to a prescribed set of conventions, the program's design can be subjected to verification from its earliest stages, and indeed, throughout its development." [20]

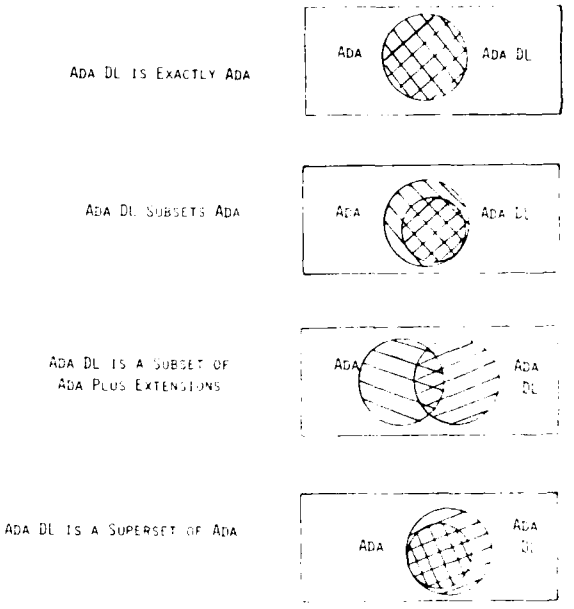


FIGURE 1

Ada DL and Ada Intersection Possibilities

Another group of authors feel that a subset of the Ada language is appropriate as a DL. "Probably the greatest advantage to a design language which is a subset of a target language is that maintenance of a design can easily (and probably must and should) be part of maintaining the actual program. Normal tools which support programming languages are available for checking the syntax of the design, for modifying the design, etc." [1] By using (almost) pure Ada as our design language we can enjoy many of the benefits provided by the Ada compiler and the Ada Language System (ALS), as well as integrating the DL program descriptions with other software development tools....Standard compiler options such as automatic indentation ('pretty printing'), cross reference listing, and Ada syntax checking are obvious benefits." [21]

It is questionable however whether every Ada construct is needed during design. Sammet, Waugh and Reiter report that in IBM's case "there was a clearcut decision to use Ada to obtain the dual benefits of having a design language which was a

Structured Commentary and Annotations

The IEEE working Group on Ada as a PDL has recognized that two types of comments are used to extend Ada. "These are unstructured comments and structured comments. Unstructured comments...can be used to indicate any information required in the design process without the need for a formal structure. These comments cannot normally be processed by a mechanical process. The structured comment is a much more formal structure and is identified by having an 'escape character' immediately following the Ada comment symbol '--', e.g., '--!', '--@', or '---'. The escape character has two functions. The first is to highlight this comment as belonging to the formal structure of the Ada DL and thus the information is easily obtainable by a reviewer or other people required to be aware of DL information. The second function is to indicate to DL tools that the comment is a structured Ada DL comment and that the tool is required to act on this information." [14]

Across the realm of existing Ada DL's there is a wide variety of structured commentary and annotations defined. These serve to extend the Ada language to better serve the design process in the following areas:

a. deferred design features

b. keywords for structured English descriptions

c. representation of complementary design details

d. formal assertions and annotations

e. tool directives

The following subsections discuss the various issues and approaches involved for the various structured commentary mechanisms.

Deferred Design Details

Privitera suggests: "In design, and even coding, there are often many decisions that one would like to postpone. Devices for expressing deferred decisions are used when Ada would normally require us to express more than we actually know. The opposite situation can also occur! We may know more than Ada allows us to express, or at least express conveniently. In these situations we usually rely upon comments. However, there are situations that recur with such regularity that they should be handled by uniform methods." [8]

Gabber proposes to handle deferred design through the addition of a new lexical element called "escape" to the Ada syntax. "The escape lexical element consists of any free text enclosed in curly brackets..It can replace almost any Ada syntactic entities, except program structure keywords (like PROCEDURE, WHILE, IF, END, etc.), program unit names and parameter lists. As the design progresses, the user refines the crude design of earlier stages by replacing escapes by more detailed constructs. At the end of the design process, when all escapes are refined to Ada code, we have the equivalent of full syntax Ada PDL." [23] Similar approaches to embed English phrases using the "{ }" notation may be found in many of the Ada DL's defined in the industry today.

Privitera identifies the need to defer design decisions on types, assignments, processing and addresses. He proposes the addition of "certain 'TBD' declarations to the package STANDARD (hence making them globally visible)." [8] Specifically, these include TYPE TBD, VALUE TBD, PROCESSION TBD AND ADDRESS TBD. This convention provides for such decisions to be deferred while still being legal Ada.

Keywords

Keywords are defined to augment Ada with two primary mechanisms: to allow the use of structured English body descriptions and to label important sections of text and Ada declarations. Of those Ada DL's which prescribe the use of structured English, appropriate Ada keywords are combined with English narrative in the classical pidgin language. These naturally include Ada block constructs such as:

```
begin ... end;
if ... then ... end if;
while ... loop ... end loop;
for ... loop ... end loop;
case ... end case;
```

Additionally, application-specific keywords (e.g., SORT, DISPLAY, POLL, TRANSFORM) may be defined in many strucutred English approaches. The Harris Ada PDL [18] formalizes these keywords to be action verbs of the form:

```
-- verb NOUN/OBJECT (optional modifiers);
```

Structured English is defined in an Ada DL either as a commentary (as with Harris' above) or by extending the Ada syntax with additional BNF definitions. Norden's NPDL/Ada defines "an ENGLISH_EXPRESSION which parallels Ada's EXPRESSION and an ENGLISH STATEMENT which is a SIMPLE STATEMENT alternative." [2]

Both Intermetrics' Byron and Harris' Ada PDL define keywords to label important sections of a design specification. Byron's keywords, labeled directives, are recognized by the presence of the following:

```
--! (keyword): (text)
```

The (text) part of a directive includes any text appearing on the same line as the keyword, as well as Byron text from subsequent lines." [24] The content of the text is defined by the keyword used to identify it. Twelve Byron directives are defined including ALGORITHM, EFFECTS, ERRORS, INVARIANTS, MODIFIES, NOTES, OVERVIEW, RAISES, REQUIRES AND TUNING.

Because the Harris Ada PDL defines all commentary in a rigid structure, special delimiters are not necessary to distinguish between forms of annotation. Keywords are defined to "efficiently organize the specification information while being Ada language compatible in both syntax and ordering." [18] Here, the keywords serve to organize subsections of both commentary and Ada language definition. Harris specification keywords include labeling of PROGRAM REFERENCES, DATA REFERENCES, EXTERNAL INTERFACES, SOFTWARE UNIT, IDENTIFICATION, DATA TYPE and OBJECT DEFINITIONS, and EXCEPTION DECLARATIONS. Other keywords define subsections to enhance the definition to serve as an on-line Unit Development Folder, including PROGRAMMING LANGUAGE, TIMING AND SIZING INFORMATION, SPECIAL TEST REQUIREMENTS and COMPLIANCE.

Complementary Design Details

In addition to satisfying a set of functional requirements, a software design must satisfy numerous other complementary requirements as the design is elaborated or tracing design elements or details back to the source requirements.

Privitera, for instance, stated that "requirements tracing is an important part of design validation, but Ada includes no mechanism for relating system modules to the requirements they are meant to satisfy." [8] As a result, many Ada DL's include provisions to support the allocation and traceability of requirements. The IEEE Working Group identified several areas of potential impact:

a. Performance which includes critical timeliness, frequencies, capacities, utilizations and limits.

b. Fault Tolerance which includes error detection, diagnosis, handling, backup and recovery, reliability and redundancy.

c. Security which includes multi-level security constraints, set/use access restrictions, breach detection and handling.

d. Distribtuion which includes geographical distribution of processing, data storage and access.

e. Adaptation which defines the need to provide flexibility for environment operation or user adaptation.

f. Quality including those requirements relating to usability, integrity, efficiency, correctness, reliability, maintainability, portability, testability, flexibility.

Formal Assertions and Annotations

Alstadt noted: "Ada does not include a method of making assertions. When intelligently interspersed with algorithmic statements, assertions about the state of the computation or about the state of the external world can clarify the function carried out by a processing segment. Furthermore, assertions can state the effect of a processing segment not yet designed; this ability is surely an aid to a modular design methodology. This problem may be restated as the inability of Ada to specify requirements at a low level." [16]

ANNA (ANNotated Ada) by Krieg-Bruckner and Luckham, is such a "proposed extension to Ada to include facilities for formally specifying the behavior of Ada programs (or portions thereof) at all stages of program development. Formal comments in ANNA consist of virtual Ada text and annotations... the inclusion of Ada text as formal comments--called virtual Ada text--gives a powerful comment facility without affecting the execution behavior of the underlying Ada program." [10]

ANNA defines Ada extensions in the form of a formal first order annotation language. The language includes:

a. declarative annotations (for variables, subtypes, subprograms, and packages)

b. statement annotations

c. exception annotations

d. visibility annotations

e. state and state transition annotations

Both virtual Ada text and annotations are entered in the form of structured Ada commentary.

IBM's PDL/Ada uses a similar annotation for state machine representation and for specifying the relationship of state variables.

Tool Directives

Virtually all Ada DL's prescribe the use of Ada DL support tools in addition to an Ada compiler. As a result, the need often arises to provide a mechanism to embed tool directives within the Ada DL specification (similar to pragmas in Ada). Candidate impacts to the Ada DL syntax to support automation are:

a. Inclusion of start/end delimiters.

b. Special embedded tool directives.

c. Formatting directives inline.

d. Special format requirements on Ada DL presentation.

For example, Byron defines a set of flags to provide directions to the Byron analyzer. "Byron constructs are distinguished by the prefix "--!". To the Ada compiler, this is a comment; to the Byron processor, it indicates something of interest. Flags take the form of a single special character immediately following the prefix. They include "--!>" and "--!<" for code extraction, "--!-" and "--!+" to control output formatting.[24]

The Navy study found that "no set of annotations was the same and rarely did two sets provide the same function in a PDL description." [2] The need for extensions and annotations is clear, but no convergence on a standard set of extensions seems possible due to influence by the host methodology.

Relationship Between Methodology and Ada DL

Design Languages have historically been methodology independent. Current research indicates that this may not be the case for Ada DL's. Various relationships between methodology and Ada DL's have been explored by different authors.

Privitera has clearly stated that Ada itself is not a methodology: "In our experience, Ada does not satisfy the requirements of a methodology. It certainly provides no insight on how problems are to be analyzed and, while there is guidance of a sort in the desire to use Ada's system structuring features effectively, Ada alone says nothing about how to structutre a system hierarchically, only that hierarchy can be expressed; it says nothing about where we should look to find our system modules, only that, once found, they may be conveniently written as program units; and it says nothing about what parts of a module belong in its interface and what parts in its implementation, only that, once identified, these parts can be kept separate." [8]

METHODMAN clearly states that a methodology is independent of the implementation language: "Indeed, many of the requirements for a software development methodology are largely independent of the target programming language." [25]

What then is an Ada-based methodology? The one characteristic that clearly identifies a methodology as Ada-based is the use of an Ada-based specification language for recording design and/or requirements information. Because of the close relationship between a methodology and its associated specification languages, the methodology may have a notable impact upon the syntax and semantics of the specification languages. The dependency of a specification language (Ada DL in particular) upon its associated methodology has been recognized by several authors.

For example, IBM's PDL/ADA was designed to support their state-machine based methodology as shown by the following quote: "The prime technical focus of the work has been to replace an existing design language and notation which supports a specific design methodology with a design language based on Ada without impacting the methodology." [1]

Clarke et al proposed a nest-free design methodology using the Ada package feature. They summarized their recommendation in the following statement: "We contend that a nest-free program organization also improves the readability of Ada programs and facilitates program development. Using packages and context specification to express a program unit's relationships, both to other program units and to data objects, results in a program organization in which program units can be arranged in any desired order." [19] This approach certainly impacts the specification of horizontal connectivity of design units.

All of the previously discussed types of information which must be describable by an Ada DL may be affected by the host methodology. Vertical connectivity is dependent upon the methodology's technique for partitioning the system into design units and whether a top-down or bottom-up approach is used. Horizontal connectivity information is impacted by the methodology's choice between nest-free and nested program structures. Data descriptions and algorithmic descriptions may be impacted by a recommended style (e.g., naming conventions for data types) and by the means for deferring design information (e.g., the use of { } as discussed earlier). The complementary information typically contains very methodology-dependent information such as IBM's state-machine representations. A methodology which supports stepwise refinement may require the DL to support the use of English statements early in the design which are progressively refined into syntactically correct Ada.

The Navy survey concluded: "Many of the differences in the definitions of a PDL came from each individual's or each company's appraoch to program design." As a result, the following impact on the DL was recognized: "For a highly structured, tightly integrated methodology, changes to the PDL may well affect the entire process so that the design of the PDL is to a large extent a result of decisions made about other parts of the software design process." [2] Due to the increased scope of Ada DL's, the DL is no longer independent of the methodology. In fact, the DL may be strongly dependent upon its host methodology.

Conclusions

There is clearly an industry divergence on the Ada design language issue which is perhaps even widening after three years of intense study and debate. This paper has investigated major language concerns in five areas; our conclusions are as follows:

LIFE CYCLE APPLICABILITY: There have been sufficient investigations to demonstrate the feasibility of applying an Ada DL over the entire life cycle. Widespread acceptance of this evidence is due to a lack of maturity in applying Ada DL's early in the life cycle.

INFORMATION EXPRESSED IN AN ADA DL: The continued confusion in this area is due to a fundamental resistance to change. The rescoping and redefinition of a DL has naturally occurred. There is no technical basis for significant debate.

RELATIONSHIP TO ADA: Ada subset debate is strictly one of application with no inherent language concerns. The need for compilability has been clearly demonstrated; differences are primarily motiva-

ted by a resistance to change. Most Ada DL definitions which are not compilable could use other proven mechanisms which would provide equivalent extensions without precluding compilation. Maturity of application is an issue regarding execution of an Ada DL.

STRUCTURED COMMENTARY AND ANNOTATION: Extensions of this form are clearly needed. The current diversity in approaches is largely due to differing methodologies. Clearly the detail to which much structured commentary is developed indicates considerable maturity -- not immaturity. A level of convention could be standardized which would allow methodology-specific extensions.

METHODOLOGY RELATIONSHIP: A large amount of resistance to an Ada DL standard is due to the methodology dependencies in the various approaches. An Ada DL standard does have the potential to impact one's way of doing business. Yet other successful specification standards have been developed which have not been unduly constraining. Objectiveness could result in a workable standard being developed.

The authors conclude that the hypothesis that Ada DL's are immature is largely false (save the specific issues identified above). The primary problem lies in a basic resistance to change and in methodology sensitivity. If anything, our maturity in methodologies has hindered standardization of an Ada DL.

Bogdan, representing the DoD's point of view, has stated: "From a government point of view, it is imperative that we have one standard." [26] The authors agree that an Ada DL standard is not only feasible, but overdue.

Acknowledgments

The authors would like to acknowledge the many sound technical contributions by numerous individuals on the topic of Ada Design Languages which has lead to a conclusion that Ada DL technology is more mature than previously recognized.

The opinions expressed herein are solely those of the authors and do not necessarily represent the views of Harris Corporation; the IEEE, the IEEE Working Group on Ada as a PDL, its participants or sponsors; the ACM, AdaTEC, AdaTEC Design Methodology Subcommittee, its participants or sponsors; the Department of Defense, Ada Joint Program Office; or others who might chose to raise exception.

References

[1] S. H. Caine and E. K. Gordon, "PDL--A Tool for Software Design," Proceedings of the National Computer Comference, AFIPS Press, 1975, pp. 168, 169, 271.

[2] J. E. Sammet, D. W. Waugh, R. W. Reiter, Jr., "PDL/Ada--A Design Language Based on Ada," Ada Letters, Volume II, Number 3, Nov/Dec 1982, pp. II-3.19, II-3.21--22.

[3] "Ada Programming Design Language Survey, Final Report," Naval Avionics Center, October 1982, pp. 1-1, 5-1, 6-1, 6-3.

[4] J. S. Kerner, "The Purpose of a Working Group on Ada as a PDL," Ada Letters, Volume II, Number 4, Jan/Feb 1983, pp. II-4.12, 13.

[5] Military Standard Ada Programming Language, ANSI/MIL-STD-1815A, U.S. Department of Defense, January 22, 1983.

[6] L. E. Druffel, "The Potential Effects of Ada on Software Engineering in the 1980's," Software Engineering Notes, Vol. 7, No. 3, July 1982, p. 5.

[7] R. S. pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill Book Company, 1982, p. 253.

[8] Dr. J. P. Privitera, "Ada Design Language for the Strucutred Design Methodology," Proceedings of the AdaTEC Conference on Ada, Oct. 1982, pp. 77, 78, 89.

[9] M. S. Gerhardt, "Description Languages Throughout the System Life Cycle," Notes of IEEE Working Group on Ada as a PDL, January 1983.

[10] B. K. Bruckner, D.C. Luckham, "ANNA: Towards a Language for Annotating Ada Programs," SIGPLAN Notices, Volume 15, Number 11, Nov. 1980, pp. 128, 129.

[11] H. C. Ferguson, M. B. Patrick, "Use of Ada in System Design: A Case Study " General Dynamics Data Systems Division, Ft. Worth, Tx., 1982, pp. 3, 10.

[12] T. J. Wheeler, "Embedded System Design with Ada as the System Design Language," Journal of Systems and Software, Volume 2, Number 1, Feb. 1981, pp. 11-21.

[13] C. Ausnit, "Ada Software Design Methods Formulation--Case Study Example," AdaTEC Meeting, Boston, June, 1982, p. 7.

[14] "Ada as a DL (Draft)," IEEE Working Group on Ada as a PDL, October 1983, pp.4--9, 11.

[15] H. Hart, "Ada for Design: An Approach for Transitioning Industry Software Developers," Ada Letters,, Volume II, Number 1, July/Aug 1982, pp. II-1.55--56.

[16] J. P. Alstad, "Problems With Ada as a Program Design Language: A Position Paper," Ada Letters, Volume II, Number 6, May/June 1983, p. II-6.51.

[17] E. Bein, "Ada Design, Jovial Implementation," Ada Letters, volume III, Number 4, Jan/Feb 84, p. III-4.62.

[18] J. K. Grau, E. R. Comer, H. Krasner and P. B. Dyson, Ada Process Description Language

Guide, Harris Corporation, Melbourne, Fl., March 1982.

[19] L. A. Clarke, J. C. Wileden, A. L. Wolf, "Nesting in Ada Programs is for the Birds," SIGPLAN Notices, Dec. 1980, p. 144.

[20] M. W. Masters, J. J. Kuchinski, "Software Design Prototyping Using Ada," Ada Letters, Volume II, Number 4, Jan/Feb 83, pp. II-4.70, II-4.74.

[21] P. G. Anderson, "A Design Language Based on Ada," Rochester Institute of Technology, May 17, 1982, pp. 8--10, 13.

[22] L. M. Lindley, "Ada Program Design Language Survey Update," Ada Letters, Volume II, Number 4, Jan/Feb 1983, p. II-4.62.

[23] E. Gabber, "The Middle Way Approach for Ada Based PDL Syntax," Ada Letters, Volume II, Number 4, Jan/Feb 1983, p II-4.65.

[24] M. Gordon, "The Byron(tm) Program Design Language," Ada Letters, Volume II, Nubmer 4, Jan/Feb 83, pp. II-4.76--77.

[25] A. I. Wasserman, P. Freeman, "Ada Methodologies: Concepts and Requirements," DoD Ada Joint Programming Office, Nov. 1982.

[26] W. R. Bogdan, "Ada Program Design Language Issues," Naval Air Development Center, 1983.

BIOGRAPHICAL INFORMATION

[illegible] received a B.S. degree in Mathematics from [illegible] Missouri State University in [illegible], and [illegible] from the University of [illegible] at Rolla. From 1978 to [illegible] was a teaching and research assistant at the University of Missouri at Rolla where [illegible] worked on a vehicle routing system. From 1979 to [illegible], she was a lecturer/professor in the Computer Science Department at the University of Central Florida. Since [illegible], she has been employed by Harris [illegible] in Melbourne, Florida.

She is the primary author of the Harris Ada Process Description Language Guide and has participated in both ACM AdaTEC and the IEEE Working Group on Ada as a PDL. She has participated in the creation of Harris' Progressive Project Document (PPD) and in definition of Harris' Tools for the Automated Development of Software (TADS). She is a major contributor in the development of Harris' Integrated SOftware METhodology (ISOMET). Currently, she is Group Leader of the Harris Methodology Group and is responsible for assisting software systems to projects with the customization and application of ISOMET, PPD, and TADS. She has recently been selected editor of Ada Letters, a bi-monthly publication of ACM AdaTEC.

Edward R. Comer received a B.S. degree in Mathematics and a M.S. in Computer Science and Applied Mathematics from the Florida Institute of Technology.

He currently leads a Software Technology Section within Harris' Software Operations, performing current research in advanced software methodologies and developing a highly automated software engineering environment. The Section provides a wide range of software technology support to development projects throughout the Harris Government Systems Sector.

Mr. Comer was the key individual in the development of Harris' Integrated SOftware METhodology (ISOMET). He is credited with inventing the Progressive Project Document (PPD) concept, Harris' cookbook approach to software development. He is a principal author of the corporation's Ada Process Description Language Guide, the first of its kind in the industry. Mr. Comer has been involved in the development of software management standards, including a sector-wide Computer Program Development Plan.

Mr. Comer has project development experience with various microprocessor and minicomputer applications in real-time systems. He has participated in several computer system design projects, contributing in areas of distributed architecture design, database organization, and modular software development. From 1979 to 1980, he was Group Leader of Modeling and Simulation Group, where he pioneered advanced simulation techniques applied to computer systems. This group's work was accomplished on both discrete and continuous systems using a variety of simulation languages.

SEEDING THE ADA SOFTWARE COMPONENTS INDUSTRY

Dr. Ken Bowles

TeleSoft
10639 Roselle St., San Diego CA 92121

Summary

The principal aim of the Ada effort is economic - particularly the enhancement of designer/programmer productivity in all parts of the software life-cycle. A shift in system design practice to widespread use of off-the-shelf large scale Ada software components would result in productivity gains exceeding a factor of ten - far more than likely to result from use of productivity enhancing software tools. To achieve widespread use of off-the-shelf Ada components requires establishment of a software components industry, and a shift in attitudes about education of system designers to use Ada. This paper reviews progress to date.

Rationale

Ada Objectives

Readers of these proceedings are familiar with the principal aims of the Ada language development effort. Those aims generally relate to improvement of productivity in all aspects of planning, development, installation, and maintenance of software intended to be embedded in systems with life-cycles measured in years or decades. Even the objective to make possible greater reliability of the embedded system software translates into a productivity aim, since improved reliability implies greater designer/maintainer effort, and greater effort by those who manage the designers and maintainers.

The design of the Ada language itself addresses especially the development of systems large enough to require the attention of teams of tens, hundreds or thousands of designers. This objective led to the PACKAGE construct of Ada, and to related constructs which make possible the factoring of complex designs into modules that are nearly independent of each other. The better the factoring, the better is the possibility to minimize the high overhead of coordination and communication among lead-design groups within a large design team.

The promotion of the Ada language by its principal sponsor, the U.S. Department of Defense, has emphasized standardization in the interest of reduced duplication of effort. Ada is unusual in that a standard has been approved before the appearance of widely accepted production compilers for the full language. By promoting industry-wide use of Ada, the DoD is also helping to reduce duplication of training. This should help to ensure that designers, managers, and supporting programmers can all talk with each other on the same terms.

Component-Based Design

Jean Ichbiah, principal designer of Ada, has said repeatedly that Ada was designed to become the basis of a new software components industry. He refers to the introduction of this concept by M.D.McIlroy at the NATO conference in Garmisch in 1968. To date, the components industry objective seems largely to have been ignored in DoD planning and promotion of the language.

The concept of component-based design of a large system can perhaps best be understood through analogy with electronic design as it is practiced today, and as it has evolved over the last 25 years. In the early 60's, most electronic equipment was constructed from scratch using discrete components such as resistors, capacitors, and transistors. The concept of plug-in modules had been introduced with printed circuit boards, but there were no accepted standards on interconnection of circuit boards made by different manufacturers. The benefits of interconnection standards were, of course, understood by the industry, as illustrated by the standards on electron tube models and pinout conventions.

Since the early 70's, it has become possible to package increasingly complex logic in large scale integrated components. Defacto standards have emerged for the interconnection of these components because of the dominance of a small number of microprocessor designs. Many complex, but commonly needed, logic functions are today available in the form of integrated circuit chips costing only a few dollars apiece. As an example, a designer needing to provide serial communications in a system would have spent 7 months on pre-production design, and perhaps $1000 per copy, on a suitable communications module. Today, most designers specify an off-the-shelf UART costing less than $5 per copy. Similar shifts to use of off-the-shelf LSI components has been experienced in connection with a wide variety of generic algorithms used widely in hardware designs.

The shift in practice of hardware designers to increasing use of large scale hardware components has resulted in drastically improved productivity of the hardware designers over 25 years. It is probably conservative to say that today's designers of digital electronics are at least ten times more productive than those of the mid 60's.

In the United States and Europe of today, software system design practices are closer to the electronic design practices of the 60's than those of the 80's. Even a programmer practiced in the use of Ada is most likely to create from scratch all modules used in a large system. We can roughly equate the level of integration associated with a single Ada language statement with the integration level associated with discrete electronic components such as resistors, capacitors, and transistors. To be sure, Ada is new and the older ability and investment have made creation of modest large-scale components difficult at best.

We are now beginning to hear of recent successes in Japan resulting from large scale use of inventories of off-the-shelf software components. In two unpublished reports, productivity gains ranging from 5:1 to 10:1 have been reported from Japanese "software factories". In one case an inventory of more than 5,000 re-usable components is in use, and more than 9% percent of each new major product design consists of reused components.

With evidence from several distinct sources that order-of-magnitude productivity gains are possible, it would seem that the package orientation of Ada would lead to growth of an Ada software components industry. Some detractors of the idea point out that programmers tend to favor the "not invented here" concept, preferring to create a new design rather than using one created by someone else. While this may be true today, it is worth noting that electronic designers behaved in the same way 20 years ago. Their shift to preferred use of off-the-shelf integrated components is a result of the unavoidable economic benefit thereby obtained.

Progress

Ada Standard & Validation

With formal approval of the Ada standard in 1987, shortly followed by formal validation of three compilers, the Ada program has taken a major step toward maturity. Strong administration of DoD policies requiring the use of compilers that have been validated should help to assure the portability of Ada programs among various processors and run-time systems.

Unfortunately, many designers who have started to work with early Ada implementations are coming to realize that the language standard leaves much to the imagination - particularly in aspects related to real-time applications. For example, a real-time system may well require asynchronous responses to external events, yet the standard permits validation of compilers which support only synchronous responses. To be sure, there are substantial differences between operational requirements on a large mainframe machine and those on a small machine that might be used for real-time control. Perhaps the formal validation suite needs to be extended to cover optional tests for operational characteristics typically encountered in real-time work.

An unfortunate conclusion is that merely correct use of the Ada language in writing a large application system program may not be sufficient to assure the portability of that program. Common module interconnection conventions or standards, whether mandated or evolved on a defacto basis, will also be needed.

[illegible]

[illegible] industry

[illegible] of large scale integrated circuit hardware components is not a new phenomenon. It took years for engineers to learn about the benefits of using off-the-shelf components to accomplish those tasks. Software engineering previously learned would have to be accomplished using independent methods.

For a designer to take advantage of an off-the-shelf component in a new project several requirements must be met:

-- The designer must be aware that the component is available off-the-shelf. In general, this means that the possibility must be kept that components exist off-the-shelf to cover the majority of needs now in the software system.

-- The cost of using the component in the project (system cost not project cost) must be less than the cost of developing, testing, documenting, and maintaining the component, including [illegible] project costs, ...

-- The designer must have an economic incentive to use an off-the-shelf component rather than one created from scratch. The incentive might come from faster completion of the project, greater freedom from errors in the completed product, higher performance of the off-the-shelf component because of greater design effort, ...

It will be seen that these considerations exist in common both for software and hardware design projects.

Failure of these points led to the [illegible] that successful application of any technology requires [illegible]-wide availability of a [illegible] inventory larger than some critical size. Only in that way will the probability of finding a sufficient component to [illegible] the designer's concern offset the cost in that way is it likely that a [illegible] effort will be expended to [illegible] of components that designers will be able to [illegible] their own duplicating [illegible] efforts.

There are, then, some relationships between software components and hardware components. Whereas the copying of silicon only components typically requires a large capital investment, the cost of copying the source program text of an Ada component may be very small. On the other hand, the assurance that a component is free from errors often requires a much larger investment than expended on initial design of the component. Whereas the source of an Ada component may have to be sold to assure portability, the suite of tests used to assure correct operation may be held confidential and not delivered to the designer's customers.

This writer and colleagues have been involved in an effort to establish a publications and distribution enterprise to make available an [illegible] inventory of Ada components. To date, progress has been very slow in part there have been too few groups already using Ada to justify the economically large initial investment. The Ada market is now emerging rapidly, and the distribution of Ada components should become commercially viable within another year or two.

Educating Lead Designers

Ada is not only a well defined computer programming language. It is also the focal point of a large scale community attempting collective use of new methods of program design, and new methods of managing program specification, design, implementation, and maintenance.

There is now a body of experience in attempts to re-educate working system programmers and designers to use Ada. In general, it is found that assimilation of the SYNTAX of Ada is not difficult and can be accomplished within a few weeks of "full-time-equivalent" effort. However, the usual result is that the learner goes back to methods of program design appropriate in the language culture in which most of the experience was gained. For example, experienced Fortran programmers are likely to write Fortran programs using Ada syntax - rather than to design Ada programs as Ada is intended to be used.

To [illegible] advantage of the strong productivity benefits of component-based design using Ada will require more effort in education. The method of education that has been found to work

*[illegible] of large "case-study" [illegible] In addition to the [illegible] syntax, the learner is presented *[illegible] case-study program at least several thousand lines long - i.e long enough to demonstrate non-trivial use of the relevant design concepts. The learner is asked to make relatively limited changes in the case-study program as learning exercises. These exercises are similar to the maintenance tasks assigned to most programmers during the life cycle of a large program.

[illegible] of learning enough about the case-study to make the assigned changes in a competent way requires several FTE months of learner effort. However, the typical result is that the learner emerges familiar with the design methods for which Ada is intended to be used. A well designed case-study will be composed using various off-the-shelf Ada components potentially usable in numerous program applications.

It is possible to minimize the time a learner must spend in going through the maintenance/modification exercises by providing instructor assistance and [illegible] automated learning materials. Normally, early stages of the learner's exposure to the case-study involve conventional presentation methods such as structured walkthroughs. Later stages evolve into design seminars in which the learners can see how similar components and design methods might apply to their own design problems.

It does not appear necessary to make the large investment for training of all the personnel assigned to a development project using Ada. In general, design responsibility is assigned to a small cohesive team of lead-designers. Other programmers and supporting designers flesh out the details of the final product, but do not strongly influence the significant aspects of the design. The lead-designers supervise the work of the supporting programmers and designers. Therefore, the key to training a large development team to make effective use of Ada may be to concentrate on case-study training of the cohesive team of lead-designers. It should then suffice to subject the supporting staff to more limited instruction as an introduction to use of Ada.

ECONOMIC, SOCIAL, AND LEGAL ASPECTS OF SOFTWARE IN THE FUTURE

Irv Feldman

Jersey City State College, Jersey City, NJ

Software has caused many changes in our everyday lives. It will continue to affect the economic, social, and legal systems in many ways. In some cases restructuring of these systems will be required; thereby creating dislocations. There are many problems to be solved and most do not have easy solutions. Ada and the techniques it uses, will definitely play a part in helping to find these solutions.

Computers have been a part of our lives for forty years. What changes have these "electronic brains",(to use an anachronism), caused? How will our lives be affected in the future?

Software has been responsible for most of the changes that have taken place. It will continue to change our economic, social, and legal systems in the future. In this paper, I shall attempt to discuss the ways in which our economic, social, and legal systems have been and will continue to be affected.

With the advent of the first practical computer, the need for software designed to perform certain functions arose. As time passed software, in the form of operating systems, assumed more and more of the routine functions previously performed by people. At the same time computers were increasing in power and decreasing in size and cost.

Somewhere along the line, high-level languages were developed. These languages allowed more people to learn programming. These high-level languages made possible the development of applications software.

Applications software, for business, has become an important force in the software marketplace. Unfortunately, they are still machine specific. Despite the efforts of the American National Standards Institute, a program written in Cobol '74, for a [illegible] will not [illegible]

another manufacturer's computer. True, there are methods of getting around this problem. However, the methods can be expensive. Usually, the expense does not justify the method.

Ada offers the opportunity for software to finally become machine independent. This can only reduce the present high cost of business programs. At last, economies of scale can be utilized. Instead of producing several hundred or several thousand copies of a program, we may see production runs of millions, thereby appreciably reducing the development cost per unit.

Software must reflect the fact that most businesses are different. Program modules, which can be easily modified to reflect these differences, will allow users to use their computer equipment more efficiently. This, of course, will keep the cost of software within reasonable bounds.

Large companies can afford to spend vast sums of money on sophisticated hardware/software systems, but smaller companies cannot. Smaller businesses will be the primary beneficiaries of decreasing software cost.

Small and medium-sized businesses have been the sources of technological innovation and job creation for a number of years. This has been necessary to allow them to compete with large businesses. Increasingly sophisticated applications software, at moderate costs, will provide better information for management decisions. This information will allow them to use their limited, available resources to their best advantage.

At the same time as software costs decrease through economies of scale, we will also be better able to take advantage of the presently unused capabilities of today's hardware. This problem has occurred because hardware technology is some twenty to twenty-five years more advanced than software.

We are only able to use about 20% of a computer's available capacity.

Software advances will allow us to use more of this capacity. This increase in usable capacity will mean that firms can use their existing computer equipment much longer without outgrowing it.

Social Effects

What effects will all of this have on our social structure? We are presently going through a period of social and economic dislocation unlike any since the beginning of the Industrial Revolution. Because we are shifting from a smokestack to a service economy, we have a problem with unemployment. Robots are taking over the production line. The American worker is faced with the problem of skills obsolescence. This means that we will have a large number of unemployed workers. Solving this problem requires retraining of unemployed people, and the redesigning of our educational system.

Businesses will need people whose education stresses reading, writing, arithmetic, and computer literacy. Computer education, starting in the primary grades and extending through college, will increasingly become available and necessary for success. These opportunities must be made available to all children and not just those who live in more economically stable school districts.

New educational methods involving the computer as a tool must be developed. A uniform software methodology will enable this to be done. At present, we have a hodge-podge of methods and special-expert-setting policy. This is caused by differences in the computer languages presently being used. Some languages are structured, some are not, some languages stress concepts, some stress syntax, some languages are easy to use, some are cumbersome. This morass causes confusion. A computer language should be easy to learn. After all, learning a computer language should be the same as learning any other foreign language.

The electronic cottage is with us. Computing may soon be a thing of the past as work can and will be performed at home with communication devices. Will we become a nation of homebodies never needing to leave home for work, entertainment, shopping or other pursuits? This possibility is not without its problems and the complexities of these.

What form will this communication take? Will we see Isaac Asimov's or George Orwell's future? Will communication be by videophone? Robots will substitute for human workers. The technological capabilities will exist but people will still have to meet with and touch one another. The art of communication will remain important in the future as it is today.

George Orwell foresaw a world in which a language called Newspeak was spoken. This language consisted of two vocabularies: the "A" vocabulary was used in normal, everyday conversation by everyone; the "B" vocabulary was used by those who shared the same political or social views. What George Orwell foresaw is not to come to pass. Our English version of Newspeak can be found in communications systems. In addition, the computer field contains a plethora of languages resulting in a war of babel. Each of these languages has its partisans and opponents who extoll its virtues and drawbacks. Ada will require that everyone use the language the same way. Controversy will still exist, (it should never be eliminated!), but it will be constructive.

Problems of Privacy

What about privacy? I feel that this issue is going to be discussed and argued over for many years to come. George Orwell's fateful year is here. Will the needs of our society and business applications require invasion of privacy? Yes! It has already come to pass. Data banks already contain financial information about many of us. The information is readily available and can be contested. The larger is that software can be created to conveniently "hide" information of a sensitive nature. We will never know it is being done.

The courts have yet to resolve the status of programs. Can copyright be defended? Are programs patentable? Who really knows? Certainly not the judges and lawyers. Software law will become a new growth field for ambitious young attorneys. Until these issues are settled, piracy of programs will continue to be a problem.

The Long John Silver of today steals programs because he or she either can not afford them or feels that they are not worth the price charged by the vendor. This trend will unfortunately continue. It is important that a new language allow some security measures to protect a potential author's investment of time and effort.

Will new programs be necessary in the future? Can we not simply modify existing programs? Some futurists say we will not need programmers. My question is: who will make the modifications? New programs are always needed, as the needs of businesses do not remain unchanged.

Ivy Sultan
[illegible]
[illegible]

[illegible] Professor Sultan received his [illegible] from [illegible] in 1968, he was employed in both profit and not-for-profit businesses in which he was involved with Computerized Accounting and Management Information [illegible]. Since 1980, he has held the academic rank of Assistant Professor of Business Administration at Jersey City State College. He has also taught various computer science courses at both the high school and [illegible] level.

OPERATING SYSTEM INTERFACE FOR ADA INSTRUCTORS

Donald C. Fuhr

Tuskegee Institute
Tuskegee, AL 36088

INTRODUCTION

Effective use or teaching of a programming language requires more than facility with the language itself. It is also necessary to deal with the computer as personified by the operating system. The LOGIN procedure is only the first handshake with the system. After a successful LOGIN, the system will just stare back stupidly from the CRT until one enters a command the system knows how to handle. The information regarding the options available at this point fills about six shelf-inches of reference books, and is completely understood by about the same proportion of users as the Ada Language Reference Manual. As with any language however, it is possible rather quickly to learn to enable one to get around without embarrassment. Hopefully, the following information will assist users in dealing with the Digital VAX/VMS operating system.

DIGITAL COMMAND LANGUAGE

General

As the dollar sign blinks back from the tube, the system is looking for a Digital Command Language (DCL) command. The user may invoke one of the editors to begin keying in a program, or to modify an existing one. One may also invoke one of the utilities to perform various predefined operations. One may even write a program in DCL. A few of the DCL commands everyone needs to know in order to properly manage files and directories are now described.

Help

When all else fails, this command provides access to most of the on-line documentation regarding DCL. The first screen is a list of all the commands and utilities. The prompt "Topic?" allows the user to explore any of the listed topics by typing in the one desired. "Subtopic?" prompts lead the user as deeply as desired into the details available. When one has seen enough, successive Returns will lead back to the dollar sign. Each of the utilities has a similar Help facility included within it.

Show

This command allows the user to find what is going on in the system or obtain a bearing if lost in a file structure. It is used with one of a large number of parameters to accomplish this task.

Show Time-Causes the system to display the current system time. This should be the same as the wall clock time, but, for various reasons, often is not.

Show Default-Displays the name of the current default directory. This allows the user to pinpoint his current position in the file structure.

Show Devices-Displays a list of all devices recognized by the system and whether or not they are on-line or allocated to a process. It also shows how many 512-byte blocks of storage are free on a disk, and whether or not a tape drive is mounted.

Show Process-Displays various information about your terminal session. Such information as elapsed time, CPU time, priority, and privileges is available.

Show Protection-Displays the level of access protection in effect for a file, or that which will be afforded any files created by the process.

Show Quota-If disk quotas are in effect on the disk in use, this command will show how many blocks of storage are allowed the process, and how many are currently charged. This is the official number tracked by the system, including temporary and lost files not appearing in the directory.

Show System-Produces a display of resource consumption information about all processes currently running. If the system is providing slow response, this is one of the tools used to find out which process is dominating the system. This display can also be used to watch the progress of a batch job.

Show Users-Produces a display of all interactive users currently logged into the system and which terminal they are using.

Set

This command enables a user to change various attributes of the process or files as desired. The most useful of the SET parameters available to the normal user are as follows:

Set Password-A user may change his password at any time, without reference to or permission

from anyone. This should always be done immediately after receiving a new account or any have been compromised. After invoking this command, the system asks for the old password, the new password, and a repeat of the new password. None of the passwords are echoed on the screen, and the new one becomes effective immediately. The password is not available in clear text to anyone, even the system manager.

Set Default-Allows the user to change the default directory at will. If a user has a directory structure including several sub-directories, this eliminates the necessity of including the directory name when typing in a filename. Programs are often written so that the default directory must be the same as the directory containing the input and/or output files.

Set Protection-Allows the user to restrict or allow access to any files owned by the process. If the command $SET PROTECTION/DEFAULT is given, all files subsequently created by the process will receive the level of protection indicated.

Rename

This command allows the user to rename any file or group of files owned by the process. It takes the form: $RENAME old file name new file name.

Delete

Allows the user to delete a file or group of files from the directory. If the form $DELETE/LOG is used, the system displays the name of each file deleted. If the form $DELETE/CONFIRM is used, the system will display the name of each file before it is deleted and ask for a Yes or No from the user as to whether it should really be deleted. This command is made more powerful by use of a "wildcard" character, the asterisk (*). If one wants to delete (or do any other applicable operation to) a group of files having some part of the filename in common, the asterisk is substituted for the common part, allowing the entire group of files to be dealt with using only one command. For example, the command $DELETE TEST.*;* will delete all versions and all types of files named TEST.

Purge

This command deletes all but the highest numbered (most recent) version of the files given as a parameter, or in the default directory if no parameter is given. If one wants to keep more than one version, the form $PURGE/KEEP=n may be used, where "n" signifies the number of versions to be kept.

SYSTEM MESSAGES

In the process of working with the VMS system, the user will receive system messages from time to time. They always take the same form, and are usually understandable, but sometimes they cause more concern than necessary. They take the form %FACILITY-L-IDENT, TEXT where the Facility is the utility, component, or program which caused the message. L is the level of severity which may be S (Success), I (Information), W (Warning), E (error) or F (Fatal). The higher the severity level of the message, the greater the probability that the action was not completed or completed incorrectly.
IDENT is an abbreviation of the message text.
TEXT is the full message. Every VAX facility should have a System Messages and Recovery Procedures Manual to assist in decoding system messages.

LOGICAL NAMES AND SYMBOLS

Logical Names

This concept is a form of information hiding applied by the VMS developers to system device names. By consistent use of logical names, the user may access a file or program anywhere in the system (if allowed by file protection) without a need to know on which actual device it is stored. The system manager may thus, with no impact on the user, move directories from one disk to another merely by redefining the logical name of the disk. If this capability did not exist, every user program would have to be re-written if the user directory were moved. Logical names are established by using various forms of the $DEFINE or $ASSIGN DCL commands.

Symbols

Most users find long command lines difficult to type in without error, and the system insists on a complete retyping if an error is committed. This is particularly frustrating when the command line is used frequently. A symbol can be defined as a shorthand expression for the command line, allowing it to be invoked by typing just a few characters. For example, $CAT is much easier than $DIRECTORY/SIZE/DATE/PROTECTION if a more complete directory listing is desired. A symbol can also be defined to execute an entire command procedure.

Login Command Procedure

The most convenient way to define useful symbols and logical names is by using a login command procedure. All VAX installations use a system-wide login command procedure to establish local symbols, etc. Among other things, the system login procedure looks in the user directory for a file called LOGIN.COM and executes it if it is found. Any user may create a personal LOGIN.COM which contains any instructions to the system that are desired to be executed at login time. An additional password or any shorthand symbols may be included to provide a customized environment.

SYSTEM SECURITY

User Authorization File

Each user has one record in the User Authorization File (UAF) which contains the Username, password, privileges, resource quotas, default directory, and all other user-unique information that governs the actions the user may

execute in the system. From a system security viewpoint, the key elements are the Username, password, and User Identification Code (UIC), all of which are assigned by the system manager when the record is created.

Username-This must be an alphanumeric character string no more than 9 characters long beginning with a letter. No two users may have the same username.

Password-This must be an alphanumeric character string no more than 31 characters long. For best security, it should be at least 8 characters long, and should not be easily relatable to any widely-known attributes of the user. The password never appears in clear text in the system and is not echoed on the screen when entered. The system manager can change passwords, but cannot read them. The user can (and should) change his or her own password at will. The correct username/password combination is required in order for a user to gain access to the system.

User Identification Code-This is a 6-digit octal code which the system assigns to a user process based on the UIC contained in the user's UAF record. The system also assigns the UIC to all files created by the user. The UIC takes the form (ggg,mmm) where the first three digits denotes the Group in which the user was placed by the system manager, and the last three digits denote the user's Member number within the Group. Both sets of three digits may take values from 000 through 377. Groups 000 through 010 are reserved for the various System accounts and automatically bestow the special privileges necessary to perform system management tasks. The other Groups and all Member numbers have no special significance except that which a local system manager may assign.

File Protection

File protection is defined in terms of the actions which the owner of a file permits other users (or himself) to take with regard to the file. The possible actions are: Read, Write, Execute, and Delete. The user may allow or prohibit any of these actions by any user of the system, and has complete control over this attribute of file. The classes of users with reference to a particular file are: System, Owner, Group or World. A particular user or process is placed in one of these categories by UIC as follows: A System user is one whose Group number is 000 through 010. The Owner is the one whose UIC exactly matches that of the file. A Group user is one whose Group number is the same as that of the file. A World user is anyone in the system. The default protection provided by VMS is S:RWED, O:RWED, G:RE, W:(no access), which means that the Owner and System users may Read, Write, Execute, or Delete the file, Group users may only Read or Execute the file, and no others have any access. The owner of a file may change this protection at any time using the $SET PROTECTION command and may change the default protection for the remainder of the terminal session by using $SET PROTECTION/DEFAULT. The latter command is a good candidate for inclusion in a LOGIN.COM file for the security-minded user.

PROCESS SCHEDULING

In any time-sharing system, the ideal situation is when no user of the system is affected by the other users. In the real world, this is true as long as the system is not heavily loaded. When the overall demand for system resources approaches the capacity of these resources, everyone can see the result. One's process may slow down, or may appear to stop entirely. There is very little that an individual user can do about this, and not much more that the system manager can do. It is worthwhile, however, for the user to know something about how the VMS operating system directs traffic among all the users in order to understand what is happening.

Process Types

A process is the basic entity that the Scheduler works on in apportioning access to the CPU and that the Job Controller moves in and out of memory. Besides system processes, there are basically three types of processes, each of which is handled differently.

Real Time Processes-A process that is normally started when the system is booted and runs continuously, this type is used for such tasks as data acquisition from laboratory instrumentation which demand instant response when the data is ready without the necessity for waiting for another process to complete. Real time processes are normally few in number and required very small amounts of CPU time. They have a minimal impact on other users of the system.

Interactive Processes-The system creates a process for each interactive user at login time; it continues until logout, regardless of what the user does. It is possible for processes to "spawn" subprocesses, but this action will not be discussed here. The process is assigned all the quotes, priorities, and privileges contained in the user's UAF record. Interactive processes are characterized by large amounts of terminal I/O compared to the amount of CPU time consumed. They are usually the greatest source of system contention.

Batch Processes-If a process involves executing an image (executable program) that requires large amounts of CPU time or clock time, it is normally run as a batch process. Started by means of the DCL $SUBMIT command, it runs to completion without further human intervention. All I/O operations must be to or from disk files. Batch processes are queued to run in sequence, and are normally set up to run in such a fashion that they use only resources not required by interactive jobs. They thus take longer to run than they would interactively, but they have much less impact on other system users. It is possible to establish multiple queues and run multiple jobs from each queue, but this must be done with caution, since too many batch jobs will interfere with each other just as inter-

Priority Scheme

Each process has a base priority for access to the CPU. For normal processes, the priorities range from a low of 0 to a high of 15. For real-time processes, the priorities range on up to 31. In a normal system, interactive processes (terminal sessions) have a base priority of 4 and batch processes have a base priority of 3. DCL system processes normally have a higher base priority.

Process Scheduling

Each process that is ready to use the CPU is queued in a first-in-first-out (FIFO) queue by priority. When the process currently using the CPU relinquishes it, the Scheduler checks all the priority queues and assigns the first process in the highest priority queue to the CPU. Once a process begins to use CPU, it continues until it needs to do an input/output operation or cannot continue for some other reasons, or until its allocated time (Quantum) has expired. Quantum, a time interval on the order of a few milli-seconds, is the maximum time any given process can use the CPU at one time. This precludes a process which does heavy computation from monopolizing the CPU. When a process uses up its quantum, it is queued at the end of the queue for its priority. Quantum is a system parameter which can be changed by the system manager, but only with great care.

Priority Promotion

The major problem with a simple round-robin scheduling procedure is that most processes spend more time doing input/output than they do computing, and they must gain access to the CPU at a particular time governed by the input/output device they are using. Otherwise, a character being read from a disk, for example, would be lost. VMS gives these "I/O bound" processes a priority promotion so that they may use the CPU when needed. This does not impact other users to a perceptible degree because an I/O operation requires only a tiny fraction of a quantum. It is thus possible for a process having a base priority of 4 to have an instantaneous priority as high as 9 while it is doing I/O operations, reverting to 4 or 5 when it is doing mostly computation.

Process States

If one executes a $SHOW SYSTEM command, among other things displayed is the "State" of each process. This provides a snapshot of the activity of the various processes and a clue to the overall system load. There are nine possible states in which a process may be:

CUR-There is always exactly one "Current" process, that is, the process currently using the CPU. Since the CPU is required to produce the display, the process calling the display will always appear as the Current process.

COM-When a process has everything it needs to compute except access to the CPU, it is "Computable". Any process indicated as COM is in a queue and will be scheduled to use the CPU as outlined above. If too many processes are Computable, the CPU is at or near saturation, and no more users can be supported.

LEF-When a process initiates an I/O operation or some other action that must be completed before the process can go on, a Local Event Flag is turned on until the action is completed. The process is indicated as being in Local Event Flag Wait state during this time. An I/O bound process spends an overwhelming majority of its time in this state.

PFW-Occasionally a process requires another page to be brought into its working set and must wait for it. It is put in to Page Fault Wait state during this time. See the Memory Management section for a more complete discussion of this topic.

HIB-A process not showing any activity for an extended length of time may be put into a Hibernate state by the Scheduler.

COMO, LEFO, HIBO-When memory becomes nearly saturated, one or more processes must be "Outswapped" from memory to the disk. When this occurs, one of these three states will be assigned depending on what the process was doing at the time it was swapped out. A more complete discussion of swapping will be found in the Memory Management section.

MWAIT-On rare occasions, a process will need a resource that is not available because it is being monopolized by some other process(es) which will not release it. This is called a Miscellaneous Resource Wait and almost always signals that the system is about to lock up, if it has not already done so. Often, the only way to clear this condition is to shut the system down and reboot it.

MEMORY MANAGEMENT

Virtual Memory

The VAX is a "Virtual Memory" machine. This means to the user that a program need not fit into the amount of main memory available in order to run. In fact, if the program will fit onto the disk(s) available, it can be run without any special scheduling action by the programmer. The Scheduler program of the operating system will take care of moving pieces of the program in and out of memory as needed. This does not mean, however, that the programmer can forget entirely about memory considerations. Memory availability is the single most important factor in the performance of an application or of a particular mix of applications. A basic understanding of memory management will enable the user to help the system manager achieve better performance of the application and of the system as a whole.

Blocks

All program segments and data are moved between disk storage and memory in blocks of 512 bytes each. In main memory, this quantity is referred to as a "page". Most measurements of

storage capacity are given in number of blocks or pages.

Working Set

The most important concept of memory management, Working Set is the amount of main memory being used by a process at any given time. The operating system automatically adjusts a process's working set within established limits according to the needs of the process and the other activity on the system. The system manager can easily control the working set limits at an overall system level or at an individual process level.

WSMAX-This is a system parameter that controls the maximum working set that any process may have. Individual processes may be allowed a smaller working set, but they may never exceed WSMAX.

WSDEF-This UAF quota establishes the working set that an individual process starts with.

WSQUOTA-This UAF entry establishes the maximum working set that a process may have without regard to any other processes on the system. The user is guaranteed to be able to use this much memory if it is needed.

WSEXTEND-This UAF entry establishes the maximum working set available to a user if sufficient memory resources are available. It allows a user to "borrow" memory if needed as long as the system is lightly loaded.

Paging vs Swapping

Paging-When a process needs a program segment or data not currently in its working set, the system executes a "Page Fault" and brings in another page from either the disk or from the bottom of the Free Page List or Modified Page List in memory. If a process is at one of the size limits on working set, the page which has been in the working set longest will be pushed out to make room. It will be put onto the top of the Free or Modified Page List depending on whether it has been modified since being brought in from the disk. If that page is still in the List when the program needs it again, it can be obtained very quickly. If not, a much longer disk I/O operation is required.

Swapping-If more processes are in memory than it will hold, and they are all using their WSQUOTA so the system cannot reduce their working sets, one or more of them must be Outswapped to the disk. The system will normally select the least active processes to be swapped until the ones left in memory can get enough memory to function.

To Page or to Swap?-A system manager is tempted to simply give everyone all the memory they need, avoiding the memory management issue entirely. This might be workable in a system with a large amount of memory and users working with FORTRAN which will run nicely in 200-250 pages of memory. Even at that, when one knows that VMS needs 750-1000 pages of memory, it is easy to see that a 4-Mbyte machine (8000 pages of memory) will accommodate only about 28-32 users without going into paging or swapping. When most of the users are working in Pascal or Ada, needing 500-1000 or more pages of memory, the impossibility of giving everyone all the memory they want can be seen. The choice then becomes one of paging or swapping. If working sets are allowed to grow very large, an individual process can work more efficiently as long as it is in memory, but if it needs to be outswapped, the entire process must be moved, consuming great amounts of system resources with no productivity except to free up memory. Furthermore, when the process gets to the head of the CPU queue, it must be completely transferred back from the disk, consuming more unproductive system resources. Swapping is generally undesirable if it can be avoided. If, on the other hand, working sets are limited too much, the system will spend so much time paging that no productive work gets done. Generally, a good system manager will experiment with working sets to obtain the best compromise between paging and swapping. If this still does not produce acceptable performance, the only remaining solutions are to limit the number of processes allowed to run at any one time or to buy more memory

SYSTEM UTILITIES

All systems have a library of programs called utilities which exist for the convenience of the system manager or system users. Many of them are integral parts of the VMS operating system software, some are written locally, and many are obtained through the Digital Equipment Computer User's Society (DECUS). Following are brief descriptions of a few of the standard utilities:

Mail

The Mail utility allows a user to send messages to another user or list of users on the system. If the addressee is currently logged into the system, a message announcing the arrival of a mailgram appears on his screen. If the addressee is not logged in at the time of transmission, the announcement appears during the next login. To invoke this utility, the command is $MAIL, after which the utility gives a prompt for another command. At this point, typing HELP will retrieve the on-line documentation explaining the various MAIL functions.

Phone

This utility allows two users to carry on a conversation between their terminals much as they would on the telephone. It is invoked with the command $PHONE, after which the normal HELP facility can be accessed. In the interest of user-friendliness, the commands resemble those used on the telephone: Dial, Answer, Hangup, etc. The Phone utility should be used with consideration. When a user dials another, the announcement of the call flashes on the called party's screen every 10 seconds or so until the call is answered or the caller stops

the ring. This can be most irritating if the called party is in the middle of something requiring concentration. It might be worth knowing that if a user executes the command $SET TERMINAL/NOBROADCAST, the announcement messages (and all others) will be inhibited.

Monitor

This utility provides another way of finding out what is going on in the system. It is invoked as a DCL command with a wide variety of parameters and qualifiers. Each command produces a terminal display giving a composite view of the activity of concern. A sequence to investigate the cause of a system slowdown might proceed as follows: 1) Enter $MONITOR STATES to find out if processes are being bottlenecked waiting for the CPU as indicated by a large number in COM or COMO state. 2) Enter $MONITOR PROCESS/TOPCPU to find out which users are monopolizing the CPU. 3) If the CPU doesn't seem to be the problem, enter $MONITOR PROCESS/TOPFAULT to find out if many processes are doing an excessive amount of page faulting, and which ones they are. Monitor can be used to check a great many other system conditions. $HELP MONITOR will show the other options.

Backup

This is the utility used by the system manager to take a "dump" of the disk periodically to insure file integrity and restoreability. In some systems, the user is responsible for backing up his files, in which case detailed instructions should be available. Backup is also the only way to copy files from one disk to another in a multi-disk system. Otherwise, the normal user needs to know that a properly managed backup scheme will ensure that at least one copy of every file exists on tape, and that it can be found in a few minutes if necessary.

Accounting

This utility is of very little use to the normal user, because almost all the functions require high privilege to look at the records of other users. It is worth while, however, to know that the system manager can use this utility to obtain detailed resource consumption data for any user or group of users for any length of time desired. This is useful to someone responsible for managing a group of users or to a user in a system in which resource utilization is the basis for billing.

CONCLUSION

This has been a brief discussion of some features of the VAX/VMS operating system that do not appear in the Primer, but which someone doing extensive work on a VAX might find useful. It was not intended to be an exhaustive treatment, but to provide pointers to potentially useful functions and aid in understanding some of the performance-determining actions of the system. Syntax of the commands and other options not discussed here can be found in the Help library or in the complete system documentation. Broad knowledge of these topics help any user to work more efficiently and participate effectively in the overall management of the system for the benefit of all users.

REFERENCE

1. Digital Equipment Corporation, VAX/VMS Primer, Maynard, MA, May 1982.

2. Digital Equipment Corporation, VAX/VMS Command Language Reference Manual, Maynard, MA May 1982.

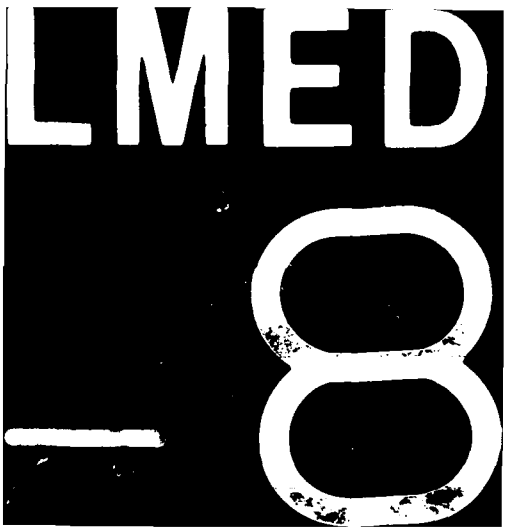
3. Digital Equipment Corporation, VAX/VMS System Manager's Guide, Maynard, MA May 1982.

4. Digital Equipment Corpoation, VAX/VMS Utilities Reference Manual, Maynard, MA, May 1982.

AUTHOR

Donald C. Fuhr, Director of Computer Services, Tuskegee Institute, Alabama 36088. Received BS degree in Electrical Engineering from Oregon State University in 1961, MS degree in Engineering Management from the University of Alaska in 1973. Retired from the U.S. Air Force with the rank of Major in 1981 after 20 years in various communications, system development, and data communications positions. Has been a VAX/VMS system manager for 2 years. Attended the U.S. Army-sponsored Ada Education and Training Summer Program in 1983 Is Associate Principal Investigator for Tuskegee Institute's Ada Education and Research Program.

AUTHORS INDEX


LMED
8