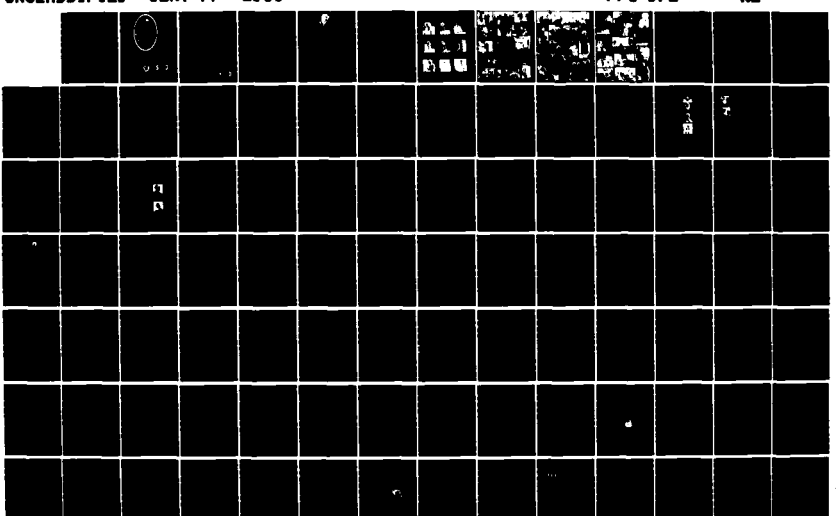
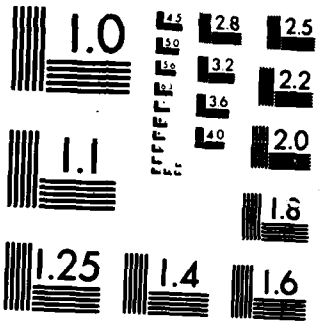
AD-A164 338 PROCEEDINGS OF THE ANNUAL NATIONAL CONFERENCE ON ADA 1/3
(TRADEMARK) TECHNOLO.. (U) ARMY
COMMUNICATIONS-ELECTRONICS COMMAND FORT MONMOUTH NJ
UNCLASSIFIED CENT .. 1985 F/G 9/2 NL








# Proceedings of the 3rd Annual National Conference on ® Ada Technology

## March 20, 21, 1985







**SPONSORED BY U.S. ARMY CENTER FOR TACTICAL COMPUTER SYSTEMS**

**FORT MONMOUTH, NEW JERSEY**

**Host College — Prairie View A&M University, Prairie View, TX**

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

# PROCEEDINGS OF 3RD ANNUAL NATIONAL CONFERENCE ON Ada® TECHNOLOGY

Sponsored By
U.S. Army Center for Tactical Computer Systems
(CENTACS) Fort Monmouth, New Jersey

Host College
Prairie View A&M University
Prairie View, Texas

Hyatt Regency Houston
Houston, Texas
March 20-21, 1985

Approved for Public Release: Distribution Unlimited

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

# 3rd Annual National Conference on Ada Technology

## CONFERENCE COMMITTEE

Elmer F. Godwin, Director, GEF Associates (201) 741-8864
Melissa Herrera, Assistant, U.S. Army CECOM (201) 544-2980
Michael Danko, RCA, Morristown, N.J.
Frank T. Hawkins, Prairie View A&M University, Prairie View, TX.
Charlene Hayden, GTE Communication Systems Div., Needham, MA.
Arthur M. Jones, Morehouse College, Atlanta, GA.
Joseph E. Kernan, U.S. Army CECOM, Ft. Monmouth, N.J.
Kurth Krause, Intermetrics, Inc., Huntington Beach, CA.
Benjamin Martin, Atlanta University, Atlanta, GA.
Isabel Muennichow, TRW, Redondo Beach, CA.
W. M. Murray, General Dynamics, DSD, St. Louis, MO.
Serge Paul-Emile, Digital Equipment Corp., Concord, MA.
John W. Roberts, The BDM Corp., Norfolk, VA.
Richard Simpson, SOFTECH, Inc., Waltham, MA.
Joan Sterling, Hampton Institute, Hampton, VA.
Ken Taormina, Teledyne Brown, Tinton Falls, N.J.
Paul Wolfgang, Computer Science Corp., Moorestown, N.J.

## TECHNICAL SESSIONS

**Wednesday, March 20, 1985**

| | | |
|---|---|---|
| 9:00 am | Session I | 1) Panel Discussion—Technology Sharing and Standardization vs Profitability: Are the Two Compatible? |
| 2:00 pm | Session II | 2) Ada Education: Strategies and Heuristics |
| 2:00 pm | Session III | 3) Ada Program Support, Tools and Techniques |

**Thursday, March 21, 1985**

| | | |
|---|---|---|
| 9:00 am | Session II | Continued—Ada Education: Strategies and Heuristics |
| 9:00 am | Session V | 4) Ada Methodology: Strategies and Techniques |
| 10:00 am | Session IV | 5) Ada Research: New Dimensions, New Directions |
| 2:30 pm | Session VI | 6) Ada Project Management: Decision Support |
| 2:30 pm | Session VIII | 7) Ada Applications: Implementation Issues and Project Results |

## PAPERS

Responsibility for the contents included in each paper rests upon the authors and 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. Request for individual copies of papers should be addressed to the authors.

## MESSAGE FROM DIRECTOR

On behalf of the sponsor, the U.S. Army Center for Tactical Computer Systems (CENTACS), the conference committee, and the host college, Prairie View A&M University, welcome to the 3rd Annual National Conference on Ada Technology. In general, last year's conference was considered a success, with an attendance of over 240, which included representatives from 32 colleges, 18 government agencies, and 58 U.S. companies.

The Committee is quite pleased and excited over this year's technical program, which includes a tutorial session, and six technical sessions. The tutorial session, "Technology Sharing and Standardization vs Profitability: Are the Two Compatible?," with its distinguished panel members should be of extreme interest to many attendees in view of the trend toward the use and introduction of Ada into new design, management, and engineering practices.

This annual conference provides the format and the opportunity to fuse together many disciplines that are considered essential for promoting and accelerating the distribution of Ada knowledge, from the realm of the software technologist to the realm of the system engineer and the software practitioner.

The 4th Annual National Conference on Ada Technology (1986) will be held at the Hyatt Regency Hotel, Atlanta, Georgia on the 19th and 20th of March 1986. The host college will be Atlanta University, Atlanta, Georgia.

The committee solicits the support of all members of the Ada family. The future success of the conference will depend upon the continued support provided by many individual organizations, government agencies, and participating colleges. Your comments and suggestions for improving the conference are welcomed.

Elmer F. Godwin
Director, Ada Conference

# Proceedings

## Bound—Available at Fort Monmouth

### 3rd Annual National Conference on Ada Technology

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

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

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

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

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

Include title, year, and AD Number:
2nd Annual Conference on Ada Technology 1984 -AD A142403.

**Highlights of**
**the 2nd Annual National Conference on Ada Technology**
**March 27-28, 1984**
**Sheraton Inn/Holiday Inn**
**Hampton, Virgina**

*Greetings*

Mr. James E. Schell, U.S. Army, Director of CENTACS, Ft. Monmouth, NJ, Sponsor

Dr. Martha Dawson, Vice President of Academic Affairs, Hampton Institute (Host College), Hampton, VA

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

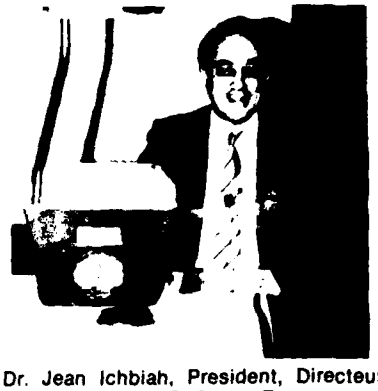
*Guest speakers from government, academia, and industry*

Dr. Mark Epstein, Office of the Asst. Secretary of the Army (RDA), Washington, DC

GOVERNMENT

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

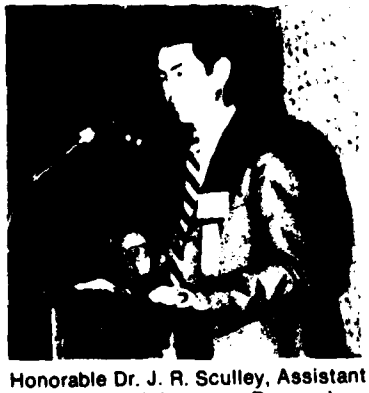
ACADEMIA



Dr. Jean Ichbiah, President, Directeur General, ALSYS Inc., France

INDUSTRY

*Banquet Guest Speaker*



Honorable Dr. J. R. Sculley, Assistant Secretary of the Army, Research, Development, and Acquisition

*Luncheon Guest Speakers*

Lieutenant General Emmett G. Paige, Jr., Commanding General, USA Information Systems Command, Ft. Huachuca, AZ

Brigadier General Alan Salisbury, Commanding General, US Army Information Systems, Software Support Command, Ft. Belvoir, VA


19 84

# CONTRIBUTORS

General Research
McLean, Virginia

Intermetrics, Inc.
Huntington Beach, California

SOFTECH, Inc.
Waltham, Massachusetts

TRW Electronics & Defense Sector
Redondo Beach, California

# TABLE OF CONTENTS

**Imperial Room West**

SESSION V: ADA METHODOLOGY: STRATEGIES AND TECHNIQUES

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



**THURSDAY, MARCH 21, 1985—2:30-5:30 PM**

**Imperial Room East**

SESSION VI: ADA PROJECT MANAGEMENT: DECISION SUPPORT

*Chairperson*: Joseph E. Kernan, CENTACS, Ft. Monmouth, NJ



**Imperial Room West**

SESSION VII: ADA APPLICATIONS: IMPLEMENTATION ISSUES AND PROJECT RESULTS

*Chairperson*: Kurth Krause, Intermetrics, Inc., Huntington Beach, CA

# TWO OPEN ENDED CASE STUDIES FOR ADA TRAINING
# A REAL-TIME PROCESS MONITOR
# AND AN
# AIRLINES RESERVATIONS SYSTEM

Philip Goldstein, Philip Caverly, Morteza Aabdollah

Ada Technology Center
Jersey City State College

## Abstract

This paper describes two case studies that have been developed at the Jersey City State College Ada Technology Center. The topics chosen are a Real-Time Monitoring System and an Airlines Reservation System. The case studies are educational in nature and are intended as a sequel for students who have taken the Introductory Professional Level Ada Course developed earlier by the Center. Both efforts were supported by the CENTACS division of Fort Monmouth.

## Introduction

This work was undertaken as part of a contract awarded by CENTACS, Fort Monmouth to the Ada Technology Center at Jersey City State College to continue the development of earlier curriculum material developed at the Center. Under a previous contract with CENTACS, the Center developed two introductory Ada courses designed to introduce programmers to Ada. One of these courses, called the Professional Course, was designed for engineers, scientists and advanced level programmers. The project reported here involves the development of two case studies as extensions of the Professional Course. The purpose is not to provide complete programs and documentation, but rather to develop a practical and theoretical framework with which students could apply the principles of software engineering in setting up, analyzing and solving the problems. The specific problems chosen are a Real-Time Monitoring System and an Airlines Reservations System. The problems are stated, a partial analysis is given, a top-down approach to the solution is developed in reasonable detail, and portions of the program are produced. There is, however, no guarantee that the solution and analysis are optimal in any sense of the word. The problems are open ended, as are real world problems, and students must provide some of their own analysis and design in order to complete the case studies. The problems may also be extended in various ways, and students are encouraged to develop extensions of their own. The specific topics for the case studies were chosen because they are typical of the kinds of projects Ada programmers will be working on, and because tasking is used in both studies. Students who complete the case studies will have gained some significant insights on how real problems are solved.

## General

Case studies are important educational tools because they give students some insight into real situations. There are two aspects to every case study: (1) The analysis of the real problem that has to be solved; (2) The synthesis and design of the solution.
In doing the analysis, students must decide what is important, what is secondary, what can be ignored on a first cut. A methodology needs to be chosen for the design phase of the case studies. Ada supports many different methodologies such as SADT, SREM, Parna's Method, the Jackson Method and Object Oriented Design. These are explored for possible use in the design. There are a wealth of practical applications using the above methodologies in the General Dynamics/Softec Case Studies Report using Ada Software Design Methods Formulation(1).

Next, a method for analyzing the design of each case study is required. Each case study makes use of concurrent

processes and thus involves Ada's tasking facilities. Several methods are being studied. One method uses finite state machines to analyze concurrent processes. An application of finite state machines in this area is given in reference (2). Petri Net theory can also be used to analyse concurrent processes. It is important that the student developing the case studies be able to take full advantage of the software engineering tools supported by Ada and not rely on limited experience in dealing with concurrency.

## Real-Time Process Monitoring System

This problem was chosen because process monitoring and control is a key feature of most embedded systems and of many laboratory computer applications. It is thus important for students to obtain some exposure to this field. The student objectives for this case study include:

(1) To become familiar with important concepts in real-time process management.

(2) To understand the need for tasking in this application.

(3) To develop requirements for the system.

(4) To implement a system in Ada.

This problem is approached at different levels of complexity. At its most basic level, a general purpose monitoring system(GPS) will, at regular time intervals, sample the value of some physical quantities. Sensors are used to transduce the value of the physical quantity to a voltage, which is then measured by an Analog to Digital Converter(ADC). Often, the GPS will log the data acquired onto a log device (a recorder or a file), and it can be setup so that a measured value falling outside of preset limits will sound an alarm. For simplicity we initially assume that only one physical quantity is being measured and that this quantity is temperature. The student is to develop a program that does the following:

1. Read the temperature sensor at regular time intervals and place the data and the time at which it was collected into a file.
   a. If a dangerous condition arises (temperature out of bounds) an alarm must sound.
   b. If the sensor malfunctions (fails to report), an alarm must sound.
2. Permit entry of commands from the terminal. Typical commands might be:
   a. Quit (Shut down system).
   b. Display most recent measurements.
   c. Display measurement history.
   d. Temporarily disable system.
   e. Change interval between measurements.
   f. Sound alarm.

In developing a strategy for solving this problem, the student will note that tasking is required; a sequential program cannot be used. One reason is that when a program unit performs an input operation (GET from terminal), that unit suspends and cannot perform any other activities until the input request is satisfied. Having thus decided to use tasking, the student must then decide on what tasks are needed and how they should communicate with each other. First, we need a TERMINAL_IO task to permit communication between user and system. It is also clear that we need a MEASUREMENT_TASK to interrogate the temperature sensor. Since this is to be done at regular intervals, a timer task is needed to "wake up" the measurement task. Also needed are a task(s) to log acquired data to the log file and also to read data from this file. In order to simplify communication between tasks, most tasks communicate only with a COORDINATOR task. (Students are asked to consider alternate approaches to the use of the COORDINATOR task). A simplified block diagram showing the tasks and communication channels is given in Figure 1.

Since writing of code is the last step in the process, students must clearly indicate the purposes of each task. First, they set up a table such as shown in Figure 2 which provides a brief set of the requirements of every task. Next, they set up a detailed specification sheet for each task such as shown in Figure 3 for task MEASUREMENT_TASK where the parameters and purpose of each enty are specified. Next, students begin to write Ada code for the tasks. Figure 4 shows a first stab at coding the measurement task. At this stage it is not intended to be executable code. Note that the task contains a SHUT_DOWN entry so that it can terminate in an orderly fashion. Here, when the SHUT_DOWN entry is invoked, it causes an exit from the loop so that the task reaches its end

statement. Notice also that the task contains a call to a procedure called MEASURE_TEMP which we assume is available on the system either as a separate compilation unit or is in an appropriate package. In our environment using Ada/Ed on a VAX we did not have any access to actual interface equipment or sensors. Hence the student has to simulate the external environment in some manner, possibly by storing the data in a file.

Testing of the program to see that it is "bug free" and does what it is supposed to is an important part of the case study. Students must develop strategies for determinig that the program works properly, although complete checking may not be possible.

The case study can be extended in various ways. For example, there might be more than one temperature sensor, or there might be a set of different types sensors, or some element of control of some external device could also be included.

## Airline Reservations System

The design of an airline reservation system requires that students design a data base and solve the passenger list update problem. The data base structure is related to the type of searches and updates that can be made by the reservations agent. A reservations agent should be able to:

(1) Determine if one can go from CityA to CityB.

(2) Determine if there are seats on Flight XXX on date YYY.

(3) Add passengers to Flight XXX.

(4) Delete passengers from Flight XXX.

Files should be structured so that file searches can be minimized.

The passenger list update problem is illustrated by the following situation. On behalf of a customer, reservations agent RA1 looks up seat availability on Flight 937 and learns that there is one seat left. While RA1's customer is mulling things over, another reservations agent, RA2, requests a seat on Flight 937. How should this situation be handled? If the program does not enforce limits on seat sales, the flight will be overbooked. On the other hand, suppose the program holds off RA2's request until RA1 either reserves a seat or decides to cancel his request? What happens if RA1 goes to lunch and forgets to close out his request? Is RA2 to be left waiting indefinitely? Clearly these situations need to be addressed.

Student objectives for these case studies include:

(1) To develop requirements for the system - what can the reservations agent do?

(2) To design the database.

(3) To design the required tasks.

(4) To implement a system in Ada.

Here also, the problem can be approached in different levels of complexity. First, we might consider that there is only one reservations agent, but this is too simplistic. Hence, we consider two reservations agents as the minimum number. Ideally, each agent should use a different terminal, but it is also possible to run the systems using one terminal and simulate two agents. Initially, we can limit all flights to the same day. Further complexity can be introduced by:

1. Allowing managerial personnel to cancel flights.

2. Storing flight information and passenger lists for more than one day.

## Designing the Front End Task

A front end task is required to service each terminal. The front end task acquires input from the reservations agent, and if a legal request has been input, the request is then dispatched to another task that will then perform the requisite action. The student must consider whether to make the front end command driven or menu driven.

## Summary

The case studies will help the student bridge the gap between small text book programs and real world problems. The topics chosen for the case studies are representative of the type that will be encountered in an industrial enviroment, but are specified at a level that the student can model and simulate in Ada using modern software engineering techniques: methodologies, methods of analysis, charts, diagrams and mathematical analysis.

## References


(1) D.L. Parnas. "On the criteria to be used in decomposing systems into modules." Technical Report, Department of Computer Science, Crnegie-Mellon University. Pittsburg, PA., 1971.

M.A. Jackson. "Principles of Program Design." Academic Press. 1975.

G. Booch. "Software Engineering with Ada." Benjamin/Cummings. 1983.

E. Yourdan and L. Constantine. "Structured Design: Fundamentals of a Discipline of Computer Program and System Design." Prentice-Hall. 1979.

"An Introduction to SADT (TM)." Softec Inc. Walthan, MA. Document 9022-78, Feb 1976.

(2) R.J.A. Buhr. "System Design with Ada." Prentice-Hall. 1984.


## 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

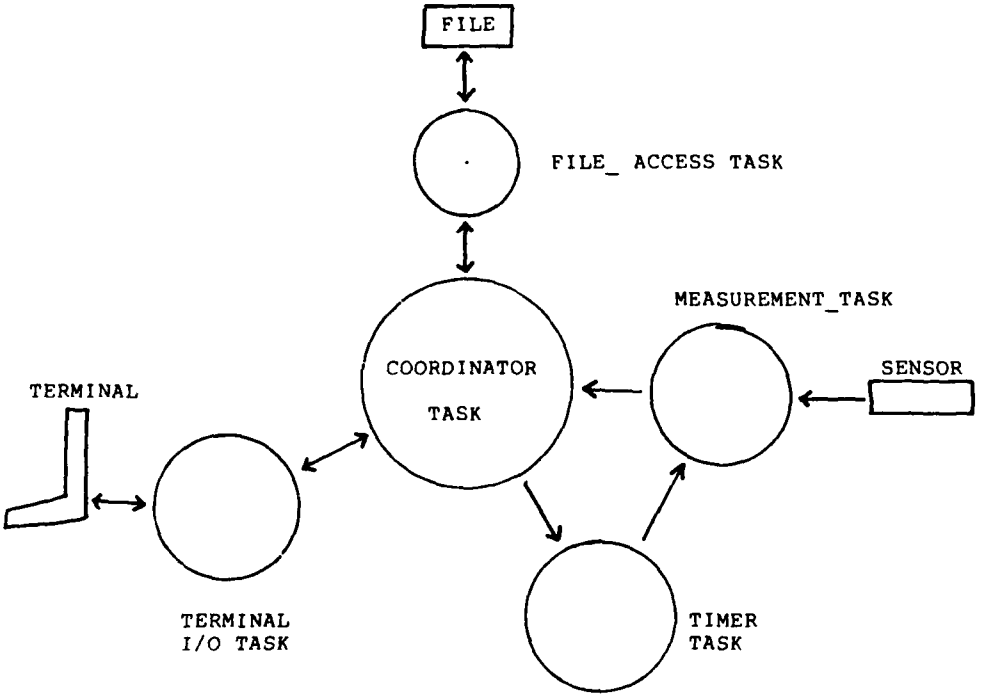

Philip Goldstein is Professor of Computer Science at Jersey City State College, and a member of the Ada Technology Center at the College. He teaches courses in microcomputers, Computer Organization and Computer Graphics. He has extensive experience in the use and development of real-time systems for 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


Mr. Aabdollah is President of M & M Computer Systems Consulting Inc., a company specializing in consulting services to the utility industry in the area of design, specialization, and implementation of real-time computer based systems.

Mr. Aabdollah has held Technical, Supervisory and Project Management positions in the area of real-time control systems for the past 16 years.


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




-- Figure 1 --

| Task Name | Purposes |
|---|---|
| TERMINAL_IO | 1. To intercept keyboard entries and determine their validity. Valid keyboard commands are sent to the coordinator task for further processing.<br>2. Print messages. |
| COORDINATOR | To coordinate the activities of the other tasks. In case of sensor malfunction or dangerous condition, sounds alarm. |
| MEASUREMENT_TASK | To obtain the data from the ADC (measure the temp) and send the data to COORDINATOR task. This task is normally suspended. |
| TIMER | Makes an entry call to MEASUREMENT_TASK at regular intervals. |
| FILE_ACCESS | To send and receive data from the disk file. |

-- Figure 2 --

Task Name: MEASUREMENT_TASK

Requirements: To obtain data from the ADC (measure the temp) and send the data to COORDINATOR task. This task is normaly suspended.

| Entry Name | Purpose | Parameter (name, mode, type) |
|---|---|---|
| WAKE_UP | Causes this task to obtain a temperature value from the ADC. Then the time is obtained and this data is sent to the COORDINATOR task.* | None |
| SHUT_DOWN | Results in normal termination of this task. | None |

* We shall assume that this task calls a procedure named MEASURE_TEMP which reads the voltage signal produced by the ADC and then converts it to a temperature value. The details would be system dependent and need not concern us.

-- Figure 3 --

```
task MEASUREMENT_TASK is          --First version not intended to
     entry WAKE_UP;               --be compilable code.
     entry SHUT_DOWN;
end MEASUREMENT_TASK;

task body MEASUREMENT_TASK is


     TEMP : TEMPERATURE;  --Global type
     DATE : TIME;

begin
     loop
        select
             accept WAKE_UP do
                  MEASURE_TEMP (TEMP);
                  DATE := CLOCK;     --Get current time.
                  COORDINATOR.GET_MEASURED_DATA (TEMP,DATE);
                     --This is an entry call to coordinator.
             end WAKE_UP;
        or
             accept SHUT_DOWN do
                  exit;
             end SHUT_DOWN;
        end select;
     end loop;

end MEASUREMENT_TASK;
```

-- Figure 4 --

# ADA AS A PRIMARY LANGUAGE IN A LARGE UNIVERSITY ENVIRONMENT

H. Evans, W. A. Greene, J. Niño, W. Patterson, D. Rudd & J. Thomas

The Ada Implementation Group at the University of New Orleans
Department of Computer Science, University of New Orleans
New Orleans, LA

ABSTRACT: Ada has been implemented as the primary teaching and programming language in the Department of Computer Science at the University of New Orleans. The UNO Ada Implementation Group addresses a number of issues involved in Ada implementation in a large university environment, including the differences brought about in introductory courses through the use of Ada.

## INTRODUCTION

This paper will be divided into four parts:

(a) Results of the implementation of Ada as the primary programming language in the first Computer Science course;

(b) Curriculum development and modification using Ada as the primary programming language in the second Computer Science course;

(c) Current use of Ada in other Computer Science courses;

(d) Proposed use of Ada in other Computer Science courses.

## RESULTS OF THE IMPLEMENTATION OF ADA IN CS1

The University of New Orleans (UNO) is on a semester system, with most Computer Science courses having three hours of lecture per week, and offering three credits to the student.

In August of 1984, CSCI 1583 was implemented at UNO, a course whose catalog description is as follows [UNO84]:

> CSCI 1583 - Introduction to Computer Science. An introduction to computer science and programming using a procedure-oriented language. Emphasizes algorithm design, verification, and analysis. Intended primarily for computer science majors. (Prerequisites: Concurrent registration in Calculus or Discrete Structures.)

This course in considered by us to be the equivalent of the CS1 course proposed in [Rust79] and indeed to have most of the features of the revised CS1 proposal in [Koff84].

The course is directed towards students intending to major in Computer Science, and students majoring in other fields are normally directed towards one of the other entry-level courses; however, students majoring in other disciplines may also enrol in CSCI 1583 if they so choose.

Because the course is directed to Computer Science majors, there is an attempt to teach the introductory material in a more profound way.

Four sections of the course were offered (by four of the authors of this article) to more than 120 students in the Fall 1984 Semester. (Four sections are also being offered in the Spring Semester.) The textbook used is [Pric84].

The university computing facilities consist of a VAX 11-780 cluster with four processors; however, in the Fall 1984 Semester, an Ada compiler (Telesoft Ada Version 1.3 for VMS)

[Tele83] was present on only one host; and, for all intents and purposes, the student users were physically connected to that host and the system served the users as though it were a single stand-alone processor.

A concern expressed frequently prior to the beginning of the semester by various members of our team was whether or not the system would be capable of handling the demand on resources imposed on it --- for, in addition to the student use of Ada, most other Computer Science courses requiring significant amounts of programming (typically in Pascal, FORTRAN 77, and VAX-11 Assembler) used the same processor.

All four sections of CSCI 1503 were viewed by the Department of Computer Science and AIGUNO as experimental; consequently, there was an allowance made for variation in the choice of topics taught in the course.

It should be pointed out that, in previous years, the introductory Computer Science course at UNO was taught using Pascal as a primary language. Consequently, in most cases, the syllabus for the previous introductory course was used as a starting point.

## SYNTHESIZED CSCI 1503 TOPICS

The Computer and Its Components
Memory - How Types Are Represented
Types in a High-Level Language (The Ada Philosophy)
Expressions and Assignments
Data Types Are Objects and Manipulation of Objects
Control Statements: if, case (plus short circuit)
Loops: Conditional Loop
Middle Exit Loop
Definite Loop (for)
Subprograms : procedures, functions
Introduction to a Class of Problems Adaptable to Recursion
Arrays (Uni- and Multi-Dimensional) --- Unconstrained
Records (not variant records)
Subtypes and Derived Types
Enumeration
Overloading

The structure of Ada seems to encourage the presentation of certain of these topics from a more profound perspective than previously. As an example, in our previous courses, discussion of I-O was limited to line-oriented I-O, and Ada forced us into a discussion of stream-oriented I-O, and the overloading of the get and put operations. Both discussions seemed to be well-received by students, thus leading to a more sophisticated understanding of I-O at this level.

As a second example, Ada seems to demystify the concept of a subprogram. Having in, out, and in out clarifies considerably the roles of parameters, and the entire discussion of "call by value" and "call by reference" can be, and was, omitted.

A third example arises in the teaching of arrays. The Ada view of array types as being intrinsically unconstrained, with the fixed range of positions declared with the variable. This seemed to provide a significant advantage in the understanding of arrays as compared to the comparable subject matter taught using Pascal.

One of the projected benefits of Ada is the consistency that the language imposes because of its early standardization. Consequently, we asked ourselves whether or not differences between the language and the compiler forced us into any "patchwork" in the course.

Three instances were reported among the four sections of the course:

1. Direct conversion between long integer and float types was not supported;

2. There was no program library;

3. Enumerated I-O was not supported.

**System Performance Issues:**

As mentioned above, concern was expressed prior to the beginning of the semester about the likelihood of system failure because of the demand on resources imposed by so many Ada users.

In general, this did not become a problem. Ada compiles on our system were typically much slower than Pascal compiles, for example, but the difference did not seem to bother students. (It did, however, bother some advanced students, as will be noted later.)

System failure could be traced to the Ada compiler a few times, but each time this was traced to peculiarities in the communication between the compiler and the operating system, rather than failure because of an overload on the system.

**Other problems**

Virtually all of the other problems encountered in the implementation of CSCI 1583 can be traced to weaknesses of Version 1.3 of the Telesoft Ada compiler.

Version 1.3 has now been replaced by Version 2.1, a version certified by Telesoft and submitted for validation. It appears that most, if not all, of these problems have been resolved in Version 2.1; thus we do not expect to re-encounter these situations.

We are also of the opinion that Version 1.3 of Telesoft Ada would lead to very serious problems in the implementation of second and third courses in Computer Science. Thus Version 2.1 would seem to be the minimal version required for a university environment primary programming language.

A catalog of the weaknesses of Telesoft Ada Version 1.3 that are likely to surface in an introductory course can be found in [Evan85], thus they will not be repeated here.

**Student Performance**

CSCI 1583 experienced an overall 50% dropout rate. This should be analyzed in the context of two considerations: 1) UNO has an open admissions policy, and students admitted to the university may freely choose their major; 2) the predecessor course to CSCI 1583 had (and has) a comparable dropout rate.

**Conclusions**

For the introductory course in Computer Science, it is not only a reasonable proposal to use Ada (as has been suggested in [Ruge83] and [Koff84]), but it is feasible in a large university environment, using available machines and compilers. Further, the use of Ada in the introductory course seems to leave students no worse off, and probably better off, than similar students studying Computer Science using Pascal.

## CURRICULUM DEVELOPMENT AND MODIFICATION FOR CS2

At UNO, the second course in Computer Science (CSCI 2120) has an objective the introduction of many software engineering concepts, and the attempt to understand through programming the difference between small and large programs. The catalog description of the course reads:

> CSCI 2120 - Structure of Algorithms. Prerequisites: Computer Science 1583 and either credit in [Discrete Structures] or concurrent registration in [Discrete Structures] and credit in [Calculus 1]. A second course in programming with procedure-oriented languages. Introduces fundamental concepts needed for the construction and analysis of effective algorithms, and applies these ideas to the modular development of large programs.

An outline of the actual topics taught (using Pascal as the programing language) in Fall 1984 is as follows (quoted from a course handout prepared by W. Patterson):

"This course may be described as falling into three major divisions:

"1. Advanced Pascal -- a study in greater depth of the features of the computer programming language Pascal; including such additional features as scalar types, the case statement, records, files, sets, and pointers.

"2. A study of good programming technique. This part of the course analyzes program style, design, testing, program verification. Design issues are examined in some detail; top-down structured design is studied.

"3. Algorithms -- following from the study of top-down design, specific classes of algorithms such as recursion and backtracking are studied, along with case studies of classic large-scale problems.

"Textbooks

"1. Standard Pascal - User Reference Manual, Doug Cooper, W.W. Norton, 1983.

"2. Problem Solving & Computer Programming, Peter Gregono & Sharon H. Nelson, Addison-Wesley, 1982.

"References

"1. Pascal, Nell Dale & David Orshalick, Heath, 1983.

"2. Pascal User Manual and Report (2nd ed.), Kathleen Jensen & Niklaus Wirth, Springer, 1974.

"3. Advanced Programming and Problem Solving With Pascal, G.M. Schneider & S.C. Bruell, Wiley, 1981.

"4. The Elements of Programming Style, Brian Kernighan & P.J. Plauger, McGraw-Hill, 1978.

"5. Program Style, Design, Efficiency, Debugging, and Testing, Dennie Van Tassel, Prentice-Hall, 1978."

In Spring 1985, CSCI 2120 is being taught using Ada. A number of modifications to the course content have been made possible and desirable because of the use of Ada.

The first part of the course remains an introduction to language features. In particular, variant records, strings, pointers, separate compilation, packages, file I-O, and tasking are introduced.

The discussion of program style remains comparable.

The discussion of program design changes fundamentally because of the language support for packages, generics, and separate compilation.

The development of algorithms, and the discussion of program correctness is comparable to the earlier course.

There are currently two sections of CSCI 2120 taught using Ada. (A third section uses Pascal, for reasons of transition and also to serve majors from other disciplines requiring an introductory courses in Pascal.)

## CURRENT USE OF ADA IN OTHER COURSES

Three advanced courses currently use Ada:

CSCI 4990 is a topics course that will be taught a number of times, during this period of transition. It is an advanced introduction to Ada for senior level Computer Science students. It provides a great deal of programming experience as well as a greater insight into the language than is possible in the programming languages course (where the design principles of

many languages are studied).

For example, one topic discussed, with corresponding programming assignments, involved tasking --- a feature not supported in the other languages available on our computer system.

An interesting insight from a CSCI 4990 student (whose primary programming language had been Pascal) was the following: "Ada compiles are so slow, that now I only compile as a last resort."

The comment was offered as a criticism of Ada and/or Ada compiler(s), yet it seems clear that the end result was to cause the student to be a great deal more careful about his design and coding. It is not proposed that compilers should be made deliberately slower in order to frustrate their use as *de facto* text editors; nevertheless, it seems that frequency of compilation should be considered as an issue in the measurement of productivity in the software life cycle.

Of course, the separate compilation (even of specification and body) additionally lend support to the argument for the efficiency of Ada.

A second advanced course to use Ada is CSCI 4501, Programming Language Design. In this course, Ada is discussed only in the context of its design principles; since many other languages are also discussed, little time is spent analyzing Ada in depth.

The third course to use Ada currently is CSCI 4401, Operating Systems I, where Ada examples have been used to demonstrate the concepts of *rendezvous* and tasking in operating system design.

## PROPOSED USE OF ADA IN OTHER COMPUTER SCIENCE COURSES

Since for all students in future years, Ada will be their primary language, Ada will probably be chosen in courses where the language of implementation of programs is left to the student.

CSCI 2125, Data Structures, will be taught using Ada, beginning next semester, as a reference language (which means that Ada examples will be given, but that the students will be free to choose a language for the implementation of their programs).

Our senior level course in Software Design, is expected to be offered in 1985-86, and will use Ada.

Other courses that may use Ada in the future are Data Communications and Networks, Operating Systems II, and Systems Programming.

## OVERALL CONCLUSIONS

At this early stage, it is too soon to judge the overall success of our experiment, however, it is generally felt that the level of implementation to date of an Ada-based curriculum has been successful, and we remained optimistic (although not in an unqualified fashion) about the prognosis for the later stages of our curriculum development and modification.

## REFERENCES

[Ada 83] Reference Manual for the Ada Programming Language, United States Department of Defense, Washington 1983.

[Auge83] Augenstein, Moshe, Aaron Tenenbaum, and Gerald Weiss, Selecting a Primary Programming Language for a Computer Science Curriculum: PL/1, Pascal and Ada, ACM SIGCSE Bulletin, vol. 15, no. 1, February 1983, 148-153.

[Aust79] Austing, Richard H., et al, eds., Recommendations for the Undergraduate Program in Computer Science, Communications of the ACM, vol. 22, no. 3, March 1979, 147-166.

[Barn84] Barnes, J.G.P., Programming in Ada, 2nd ed., Addison-Wesley, Reading, 1984.

[Evan85] Evans, Howard, and Wayne Patterson, Implementing Ada as the Primary Programming Language, ACM SIGCSE Bulletin, vol. 17, no. 1, March 1985.

[Koff84] Koffman, Elliot B., Philip L. Miller, and Caroline E. Wardie, Recommended Curriculum for CS1, 1984, Communications of the ACM, vol. 27, no. 10, October 1984, 998-1001.

[Pric84] Price, David, Introduction to Ada, Prentice-Hall, New York, 1984.

[Tele83] Telesoft-Ada Compiler User's Manual, Telesoft, San Diego, May 1983.

[UNO 84] University of New Orleans Computer Science Department, Curriculum in Computer Science, New Orleans, August 1984.

---

University of New Orleans
New Orleans, Louisiana
January 10, 1985

**AUTHORS**

Dr. Howard Evans is an Assistant Professor of Computer Science at the University of New Orleans. He received his doctorate from Tulane University. His research interests are in compiler construction.

Dr. William Greene is an Assistant Professor of Computer Science at the University of New Orleans. He received his doctorate from Tulane University. His research interests are in the area of the analysis of algorithms.

Dr. Jaime Niño is an Assistant Professor of Computer Science at the University of New Orleans. He received his doctorate from Tulane University. His research interests are in the area of semantics.

Dr. Wayne Patterson is an Associate Professor of Computer Science at the University of New Orleans. He received his doctorate from the University of Michigan. His research interests are in the area of cryptography.

Mr. James N. Thomas is an Instructor of Computer Science at the University of New Orleans. He received his bachelor's degree from the University of New Orleans. His research interests are in the areas of operating systems and networks.

Dr. David Rudd is an Associate Professor of Computer Science at the University of New Orleans. He received his doctorate from the University of Miami. His research interests are in the area of Ada technology.

# TEACHING ADA FROM A CONCEPTUAL VIEWPOINT

J. J. Buoni and E. S. Santos

Department of Mathematical and Computer Sciences
Youngstown State University
Youngstown, Ohio 44555

Over the past fiew years in many Academic programs, the Ada programming language has been introduced in advance courses as a vehicle for instruction in presenting the principles of software engineering. There are several other approaches to the introduction of the Ada programming language in advanced courses. The purpose of this paper is to introduce the approach used by the authors over the past three years; that is, to introduce Ada in the "Programming Language Design" framework.

Background:

The following paper offers an alternative view to the instruction of Ada at the advanced undergraduate level which departs from the traditional "Software Engineering" approach.

Introduction:

The external enviroments of a program during its execution may be termed its operating enviroment. Batch-processing, interactive and embedded systems are three different types of operating enviroments whose different requirements have an important influence on the language design. It is not unreasonable that programming languages are designed with different designs. Over the past few years in many Academic programs, the Ada programming language has been introduced in advanced courses as a vehicle for instruction in presenting the principles of software engineering. There aresevera1 other approaches to the introduction of the Ada programming language in advanced courses. The purpose of this paper is to introduce the approach used by the authors over the past three years; that is, to introduce Ada in the "Programming Language Design" framework. Few programmers ever think of themselves as language designers, yet any program has a user interface that in fact is a form of programming language. The user interface consists of the commands and date formats that are provided for the user to communicate with the program. The designer of the user interface for a large program such as a text editor, an operating system, or a graphics package must be concerned with many of the same issues that are present in the design of a general-purpose programming language. The aspect of program design is often simplified if the programmer is familiar with a variety of constructs and implementation methods from ordinary programming languages. The approach taken by the authors is to start with the overall language design principles, study them in relative isolation and then seek examples of these principles in Ada and other programming languages. The course entitled "Programming Language Structures" has followed this approach over the past three years because we believe that it is only by understanding the basic underlying concepts that meaningful comparisons may be drawn between Ada and other programming languages and only then will Ada be fully understood. Central to this theme is the text by Ledgard and Marcotty entitled "Programming Language Landscape" which has been used in our Programming Language Structures course for most of the past three years and which has recently been supplemented by the Ada Language Reference Manual5. This paper presents the experiences of the authors in the instruction of Ada in the above setting over the past few years. Central to this course is the comparison between PL/I (a batch processing language) and Ada (an embedded systems language). Developed in the early 60's it is not surprising that PL/I would not stack up well against Ada. But in an educational enviroment, it serves as a model for a language which is at the opposite spectrum of Ada; yet in some sense, Ada may be thought of as a logical completion of PL/I. This comparison becomes necessary in the Programming Languages course as taught at Youngstown State University, since the principle vehicle of instruction has been PL/I and not Pascal.

Scope:

PL/I and Ada are both block structured languages. The essentials of block structure is a system of program units that delimit the region of program text amd a method for specifying the names that belong

to these regions. The conventional rules of lexical scoping which one may attribute to Algol 60 may be summarized as follows[3]:
1.) The scope of a declaration includes the block in which it occurs but excludes any block surrounding it.
2.) The scope of a declaration includes any block contained within the block in which the declaration occurs but excludes any contained block in which the same identifier is redeclared.
These basic rules are quite complicated when applied to Ada [5]. Yet to what extent do these rules hold exactly in Ada and PL/I is of much importance. PL/I offers an escape to these rules with its External declaration concept allowing one to introduce them into selected choices of separately compile procedures.

On the otherhand, Ada's complex rules for managing the name space[1] leads one to differentiate between Scope and Visibility Rules. The scope of an entity is the region of a program where its declaration has effect and the visibility of an entity defines where its name may be seen. In general, the scope of an indentifier starts at the point where the indentifier is declared and extends to the block that contains the declaration. Similar to PL/I, Ada offers a mechanism to escape these rules with the package concept[2,5]. Since the scope of the entities with the same identifier may overlap as a result of overloading, the term 'visibility' has been added to the vocabulary which informally means that the visibility of an entity defines where its name may be seen. In all cases, an entity is visible only within its scope[5].

PL/I Example 1:

```
P:procedure;
  declare A,B;
  Q:procedure;
    declare B;
  end Q;
end P;
```

By the traditional rules of scoping, procedure Q's variable B creates a hole in the visibility of procedure P's variable B. Hence, P.B (notation borrowed from Ada) is not known in the procedure Q. Ada has rectified this situation.

Ada Example 1:

```
procedure P is
  A,B: float;
  procedure Q is
    A: integer;
  begin --
  end P;
```

Similar to PL/I, the variable A defined in P is not directly visible in Q; however, its scope includes all of Q. One is able to use selected component notation P.A to obtain access to P's variable A. It may be worth mentioning that this qualified name mechanism does to some extent exist in languages such that one may be able to access the components of a record i.e. in PL/I. Hence, one sees how Ada has filled a void which existed in the Algol 60 scope rules.

Parameter Passing:

In the study of Programming Languages, one usually mentions five types of parameter passing. They are pass by name, reference, result, value, and value-result. PL/I supports pass by reference. Ada on the other hand with its IN, OUT, and IN-OUT parameter passing mechanism supports what at first appears to be a form of pass by value, result and value-result but hides the actual implementation with stern warnings when refering to a parameter whose type is an array, record, or task type. An implementation may achieve the results of IN, OUT and IN-OUT effects by copy or by reference. However, the language does not define which of these mechanisms are to be used for parameter passing nor whether different calls to the same subprogram require one to use the same mechanism. The execution of a program is erroneous if its effect depends on which mechanism is selected by the implementation. Such "Information Hiding" of the implementation is not surprising to Ada and may be considered an analogue of the manner in which the multi-dimensional Array implementation is hidden.

Also, Ada has taken the attitude that side effects are somewhat immoral. Thus it requires parameters passed to functions have mode IN only.

Control Structures:

In programming Language Structures one encounters the definition of RE(n) structure which is composed of basic actions, if-then-else, and loop constructs together with an exit statements of the form ext(i) where i is any integer between 1 and n and any group of the above statements is also a basic action. That Ada supports RE(n) structures but not L-structures (L structures contain unlimited goto's) is not surprising. Recall that Ada's exit statement, defined as[5]:

```
exit::=exit[loop-name][when condition]
```

allows several nested loops to be exited and Ada's goto statement[5]

```
goto-statement::=goto label-name;
```

requires that the innermost sequence of statements which encloses the target statement must also enclose the goto statement.

Exception Handlers:

Exception handling have been classified in two categories:

1.) Those that return (unless otherwise directed) from the error handler to the vicinity of the error raising statement. PL/I provides this feature in many of its handlers.
2.) Those that Do NOT return to a vicinity of the statement that raised the exception. Ada falls into this category.

In studying the exception handling capability of both Ada and PL/I one is left captive by the power of Ada in this respect. Exceptions in PL/I are used as a normal programming technique. In contrast, exceptions in the Ada language are intended specifically for handling errors and limiting conditions. In both languages, execution of the normal part of a program is suspended when an exception occurs; execution of an exception handler (if any) is initiated. It is at this point that Ada and PL/I severely disagree. In Ada the program block unit terminates because Ada considers exception handlers as the logical completion of the block unit, while PL/I may take one of three alternatives unless otherwise directed, i.e. return to the statement in which the error was raised (on Conversion), return to the statement after the statement that raised the error (on endfile), or return control to the operating system (on error). PL/I seems to have been the first language to provide elaborate exception handling facilities; however, they are not uniformly treated. Both PL/I and Ada propagate errors to the next level if no exception is specified and then proceed according to their respective rules. Consider the following implementation of a merge sort of two sorted files.

Ada Example 2:

```
with TEXT-IO;use TEXT-IO;
procedure main is
   FILEA,FILEB,FILEC:file-type;
   type DATE is array (positive range
   1..20) of integer;
   A1,B1,C1 : integer;
   ALOGIC,BLOGIC : boolean;
   package MY-INTEGER-IO is new
   INTEGER-IO(integer);
   use MY-INTEGER-IO;
begin
   create(FILEC,OUT-FILE,'FILEB.DAT');
   open(FILEB,IN-FILE,'FILEB.DAT');
   open(FILEA,IN-FILE,'FILEA.DAT');
   get(FILEA,A1);
   get(FILEB,B1);
   begin--inner block
      loop
         ALOGIC:=false; BLOGIC:=false;
         if A1 >= B1 then
               ALOGIC:=true;
               put(FILEC,A1);
               get(FILEA,A1);
                     else
               BLOGIC:=true;
               put(FILEC,B1);
               get(FILEB,B1);
         end if;
      end loop;
exception
   when end-error =>
   loop
      if ALOGIC then
            put(FILEC,B1);
            get(FILEB,B1);
                else
            put(FILEC,A1);
            get(FILEA,A1);
      end if;
   end loop;
    end; -- inner
  exception
     when end-error=>close(FILEC);
end main;
```

While in PL/I the same type of program would be:

PL/I Example 2:

```
MERGE:proc options(main);
      dcl (FILEA,FFILEB,FILEC) file record
      sequential, (A1,B1) char(80);
      dcl (AEOF,BEOF) bit(1) init('1'b);
      on endfile(FILEA) begin;
         AEOF='0'b;
         on endfile(FILEB) BEOF='0'b;
         read file(FILEB)into(B1);
         do while(BEOF);
            write file(FILEC)from(B1);
            read file(FILEB) into(B1);
         end;
      end;
      on endfile(FILEB) begin;
         BEOF='0'b;
         on endfile(FILEA) AEOF='0'b;
         read file(FILEA)into(A1);
         do while(AEOF);
            write file(FILEC)from(A1);
            read file(FILEA) into(A1);
         end:
      end;
      open file(FILEA) input, file(FILEB)
      input, file(FILEC) output;
      read file(FILEA)into(A1);
      read file(FILEB)into(B1);
      AEOF='1'b; BEOF='1'b;
      do while(AEOF&BEOF);
         if A1>B1 then do;
            write file(FILEC)from(B1);
            read file(FILEB)into(B1);
         end;
                        else do;
            write file(FILEC)from(A1):
            read file(FILEA)into(A1);
         end;
      end;
      close file(FILEA),file(FILEB),
      file(FILEC);
      end MERGE;
```

Type checking:

Given the fact that modern programming language design theorist now seem to

we agreed that typed languages are to be preferred, Ada with its strong typing, offers a refreshing alternative to PL/I's fifteen pages of conversion rules[6].

Ada's strong typing ensure that discriminants always have a value. This is especially different from Pascal. In Pascal, variant records may have the following type of declaration which lead many to claim that Pascal is not a strongly type language.

Pascal Example:

```
...some declarations omitted...
type VISA=(PERMANENT, TEMPORARY,VISITING);
NEWPERSON=
 record
  NAME:WORDS:
  AGE:YEARS;
  PRESENT:WORKWEEK;
 case CITIZEN:boolean of
      true:(PENSIONNO:integer);
      false:(STATUS:VISA;PASSPORTNO:
      integer)
 end;
var FERGUSON,SMITH:NEWPERSON;
...some assignments....
FERGUSON.CITIZEN:=false;
FERGUSON.STATUS:=VISITING;
SMITH.CITIZEN:=true;
SMITH.PENSIONNO:=2361;
```

whereas the assignment

```
FERGUSON.PENSIONNO:=87431;
```

would be illegal and cause a runtime error. The same example in Ada would be the following:

Ada Example 3:

```
type VISA is(PERMANENT,TEMPORARY,VISITING);
type NEWPERSON (CITIZEN:boolean) is
     record
         NAME:WORDS;
         AGE:YEARS;
         PRESENT:WORKWEEK;
         case CITIZEN is
              when true =>
                  PENSIONNO:integer;
              when false=>
                  STATUS:VISA;
                  PASPORTNO:integer;
        end case;
     end record;

...sample declarations...

SMITH:NEWPERSON(true);
FERGUSON:NEWPERSON(false);
```

Structured Programming:

Ada and PL/I both contain ample structured statements. However programming style encouraged is different because PL/I contains an UNTIL clause and also encourages the use of goto's in exception handlers.

References:


[1.] Booch,G.,"Software Engineering with Ada" Benjamin/Cummings(1983).
[2.] Evans,A.,"A Comparison of Programming Languages: Ada Pascal and C" in Comparing & Assessing Programming Languages Ada C Pascal" edited by A. Feuer and Narain Gehani, Prentice Hall (1984).
[3.] Ledgard H. and Marcotty M., "The Programming Language Landscape", SRA (1981).
[4.] Pratt T.W. "Programming Languages", ANSI/MIL-STD-1815A, (1983).
[5.] Reference Manual: Ada programming language, ANSI/MIL-STD-1815A, (1983).
[6.] Reference Manual: PL/I checkout and Optimizer Compiler, IBM Program Product, (1976).



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



E. S. Santos, Professor
Youngstown State University
Youngstown, Ohio 44555

# ADA AND THE BUSINESS SCHOOL CURRICULUM

Diane M. Fischer

Department of Business Computer Information Systems
and Quantitative Methods
Hofstra University, Hempstead, NY 11550

Abstract

The current curriculum for business computing is examined in view of including Ada. Constraints are discussed. These include accreditation requirements and school-wide requirements for a Bachelor of Business Administration degree. DPMA and ACM model curriculum are considered. Ada is compared with popular business languages, COBOL, BASIC and FORTRAN. Suggestions for including Ada in particular courses are given. Topics in Ada especially relevant to a first course are noted, as is the background material needed to teach Ada. The business market is investigated in terms of programming needs. The university is considered both as leader and follower vis-a-vis this market. Trends in computing are noted and suggestions given for the use of Ada.

### 1. Business Computing Curriculum Accreditation Requirements

To discuss the inclusion of Ada in the Business Computer Information Systems (Business Computing) curriculum, one must look at constraints on the curriculum. Hofstra University's School of Business holds accreditation from the American Assembly of Collegiate Schools of Business (AACSB) for both the undergraduate and graduate programs. The Business Computing curriculum for a bachelor's degree in business administration (B.B.A.) has been specially designed to meet AACSB requirements.

Presently a business student needs 125 credits to graduate, of which 62 must be in designated liberal arts areas. In addition, a student has 39 required credits of Accounting, Business Law, Finance, Quantitative Methods, Management, General Business and Marketing. This leaves the 24 credits (eight courses) which distinguish the Business Computing major from other B.B.A. degrees. Of these, six are required courses and two are electives within the department.

The introductory course, required of all business school students, devotes one-third of its content to programming in BASIC and the remainder to computer literacy and business applications. COBOL, the main business language, is taught in a two semester sequence. These serve as prerequisites for the advanced courses of systems analysis and design, management information systems and equipment selection. The two department electives are chosen from six courses covering the following subjects: FORTRAN, a collection of several languages, simulation, minicomputers and microcomputers, reading and research, and work experience in an internship. There are no further electives, unless a student takes more than 125 credits to graduate.

The School of Liberal Arts contains a Computer Science department. Business Computing offers a computer science minor consisting of a set of required courses to be substituted for a designated set of otherwise required liberal arts courses.

### 2. Possible Ada Courses

With this background, one can address the question of how Ada can be included in the Business Computing curriculum. Such a course would have to be included among the eight major courses or taken as part of a computer science minor. To relegate the teaching of Ada to a Computer Science minor would effectively remove it as a possible course for the majority of majors, who do not minor in computer science. It would remove the course from Business Computing control. The course would be taught from a Computer Science viewpoint which is more technical and less business-oriented than Business Computing. This also assumes the Computer Science department would be willing to offer this course regularly. Ada has been taught by Computer Science on an experimental basis, but their main language is PL/I.

Ada for business applications would best be taught in the Business Computing department. Options include substituting Ada for one of the languages presently offered and adding another course to the department electives. These electives are typically offered once a year and have minimal enrollment. Adding another course would spread the students thinner and place further burden on a heavy faculty teaching load. Faculty teach three to four courses per semester. The faculty are often understaffed because AACSB requires them to have appropriate terminal degrees and because competition for Ph.D.'s in Business Computing is heavy. The option of adding an Ada course is not viable. The only remaining possibility is to substitute Ada for a language presently being offered. This is currently being done. In the past year, Ada has been included as one of the languages offered in the comparative languages course.

The department has recently decided to collapse the two FORTRAN and languages electives into one course. It will be offered each semester and will cover one of Ada, FORTRAN or Pascal each time. Students will be permitted to take the course more than once. This change reflects the increasing importance of Ada, but the course is an elective and majors can graduate without it.

The remaining possibilities are to substitute Ada for either BASIC or COBOL, the languages offered in the two required language courses. The format for the introductory course is constrained by AACSB requirements. It may be possible to teach a subset of Ada to non-technical students to give them a general understanding of programming and to cover the basic program structures. The success of Ada in an introductory course would depend on the level of the students. BASIC is easy to teach and one of the easiest languages to learn. However, even this simple programming component is very difficult for many students. It does not seem reasonable to offer a five week introduction to Ada as a first language in that course.

The two-semester COBOL course is a standard for business computing. AACSB does not specifically require that COBOL be taught. Instead, for these and the remaining major courses, AACSB requires a basic understanding of the concepts of Management Information Systems and of computer applications. This is very general. But inherent in their guidelines is the assumption that there exists a wide range of business applications written in the chosen language. Any substitution for COBOL would have to be justified in terms of what is used in the business community. No major change is forseen in the near future.

While not binding, the DPMA model curriculum for undergraduate education in Computer Information Systems has served as a guide for course offerings.[4] This model specifies COBOL but emphasizes that the model curriculum is a 'living' document open to change. ACM curriculum recommendations for Information Systems does not specify any language.[1] Both guides suggest a wide range of computer science techniques for which no one existing language is adequate. File handling is best taught with COBOL. Sorting algorithms, hashing, stacks, queues and trees are better taught with a scientific programming language like Ada.

Any course offering in Ada will require computer support. This includes a working compiler, manuals and textbooks.[2] There are Ada compilers available for a few machines. DEC has announced an Ada compiler prevalidated for the VAX 11/780 running the VMS operating system. Hofstra runs this compiler on its VAX 11/782. The Business Computing department has a subset of Ada running on IBM PC computers. Manuals and textbooks are scarce, but the situation is improving. A colleague is currently under contract to write an Ada and business applications text.

The newly chosen elective business course in Ada or other major languages will be open to business students majoring in other departments. These students must have had BASIC, but not necessarily COBOL. The course will cover notation, scalar types, control structures of selection and repetition, one dimensional arrays, linear records, and simple functions and procedures. An overview of other language features such as packages, private types, tasks, exceptions and generic program units may be given.[7, 10]

Thus far, this paper has focused on how Ada can be included in an AACSB accredited business school curriculum. Ada can and is being offered as a language supplemental to COBOL. While it is possible that the roles of Ada and COBOL could be reversed in the future, there would have to be a large amount of business programming done in Ada to justify this. The following section considers Ada's place in the business computing market.

### 3. Ada For Business Computing

Ada was designed for programming large-scale, real-time embedded systems. As noted, it includes facilities for abstract data types, multitasking, generic program units and real-time constraints. Its package feature that allows separate compilation has lead to new ideas regarding the

place of nesting in program design. A structured programming language, it was targeted for replacing Pascal and FORTRAN for real time applications, not for commercial applications. However, the power of the language lends itself to a broader range of applications.

In the past it has been easy to classify programming applications as either scientific or data processing. FORTRAN covered the former and COBOL dominated the latter. Modern applications cannot be as easily classified. An example is computer-aided manufacturing. To facilitate this, we will see the roles of FORTRAN and COBOL diminishing and more versatile languages being used.[7]

There are several factors which might limit widespread acceptance of Ada in the business computing field. A serious one is inertia. COBOL began in 1960. An ANSI-Standard COBOL was issued in 1974. COBOL 80 is in the process of being made the latest national standard, facilitated by its acceptance by the international standards committee. There are compilers and manuals available for this latest version of COBOL. The VAX under VMS runs one at Hofstra. Regardless of the availability of two much more powerful versions of COBOL, the great majority of the business community is still using COBOL 60. A discussion at a COBOL session at NCC 84 indicated that companies are loathe to update their COBOL programs. It seems clear that they will be even less anxious to scrap their programs entirely to convert to Ada.

What might soften this resistance is the availability of conversion packages from COBOL to Ada. Whether it makes sense to spend the man power on a system to translate probably poor patched, unstructured COBOL code into Ada is a question beyond the scope of this paper. A better idea is the provision of interfaces between COBOL and Ada programs which would make such a transition less painful.

In any case, for Ada to be more palatable to the business market, sophisticated I/O and file handling mechanisms need to be made available. These will probably come in the form of Ada packages which can do COBOL-type file manipulation. COBOL was designed to provide output for business applications. Ada will have to provide similar output to challange COBOL seriously within the business community.

One must consider the business community as a whole and not simply from the view of information systems. Hofstra's School of Business can serve as both a microcosm and a source of this community. The traditional academic lines are Finance, Management, Marketing, and Business Law. Professors in these disciplines rarely are interested in learning high level programming languages. They want user-friendly systems that are easy to learn to manipulate and that can easily interact with one another.

Popular business programs include VISICALC, a financial spread sheet; MINITAB and SPSS, statistical packages; LINDO, a linear programming package; SHAZAM, a time-series package. These are the so-called problem-oriented languages. Faculty use these types of programs in classroom teaching and in research. Business students learn these prograns and the techniques of using them. They enter the business community with the knowledge that they can use computers without having to master the arts of computer programming. Articles on future trends[9] emphasize attempts to make computers easier to use for non D.P. professionals. The market for Apple's MacIntosh computer indicates the power of easy to use computing systems. This market will increase as more software is available. Ada, as a complex and technical language, is not going to meet this demand directly. This is not its primary purpose. If Ada can be used as a language behind such packages, it will find a large business market willing to accept it. If it does, it will be even more necessary to teach Ada to business students so they can direct the development of specialized business applications in their business careers.

In summary, this paper has briefly considered how to include Ada in a business school curriculum and future acceptance of Ada in the business community as justification for such inclusion.

**References:**


(1) ACM Curricula Recommendations for Information Systems. Volume II, 1983.

(2) ACM Position on Standardization of the Ada Language. Commun. ACM 25.2 (Feb 82) 118-120.

(3) Ada: Past, Present Future: An Interview with Jean Ichbiah, Principal Designer of Ada. Commun. ACM 27.10 (Oct 84) 990-997.

(4) DPMA Model Curriculum for Undergraduate Computer Information Systems Education 1981.

(5) Hoare, C.A.R. The Emperor's Old Clothes. 1980 Turing Award Lecture, Commun. ACM 24.2 (Feb 81) 75-83.

(6) Ledgard, H.F. and Singer, A. Scaling Down Ada. Commun. ACM 25.2 (Feb 82) 121-125.

(7) Smedema, C.H., Medema, P. Boasson, M. The Programming Languages Pascal Modula CHILL Ada. Prentice-Hall Internationsl, New Jersey, 1983.

(8) Tompkims, H.E. In Defense of Teaching Structured COBOL as Computer Science. SIGPLAN Notices 18.4 (April 83) 86-94.

(9) Traub, J.F., ed. Quo Vadimus: Computer Science in a Decade. Commun. ACM 24.6 (June81) 351-369.

(10) Wegner, P. Self-Assessment Procedure VIII. Commun. ACM 24.10 (Oct 81) 647-678.

(11) Wichman, B.A. Is Ada Too Big? A Designer Answers the Critics. Commun. ACM 27.2 (Feb 84) 98-103.

# ACADEMIC IMPLICATIONS OF ADA IN INDUSTRY

Joan M. Sterling

Hampton University
Department of Mathematics and Computer Science
Hampton, Virginia 23668

As knowledge about the Ada language becomes more widespread, the number of industrial organizations using Ada increased accordingly. More companies are using Ada and considering Ada, as both a specifications and implementation language than ever before. As an exchange faculty member during the summer of 1984, I had the opportunity to observe some industrial requirements, with respect to Ada, for software development. This paper is therefore a discussion of Ada's present position in various industrial applications, some future industrial requirements and how Ada can best be utilized to fill those requirements, as well as some suggestions on how industry, colleges, and universities can work together to produce a sufficient number of individuals trained in all aspects of Ada. These individuals could then help meet the growing demand created by the United States government and the industrial world for Ada qualified people. The supply of Ada trained individuals is low compared to the very high demand.

## *INTRODUCTION*

The annual U.S. Army sponsored Faculty Research and Enhancement Program took place during the period June 10, 1984 through July 17, 1984 at Tuskegee Institute in Tuskegee, Alabama. I was an attending professor from Hampton University. Although I had previous encounters with Ada through two previous seminars (one 3 day seminar and one 4 day seminar), I did not really understand the syntax or real purpose of the language until completion of the summer course. The attainment of knowledge from the instructors, course materials, and various speakers who shared their experience with the participants of the Ada Summer Research Program has made me an Ada advocate. I am using that knowledge to help me teach two Ada courses; An Introduction to Ada and Advanced Ada Programming. The advanced course incorporates many software engineering practices.

### First Industrial Encounter

Upon completion of the Ada Summer Research Program at Tuskegee Institute, I became an exchange faculty member at one of the major industrial companies which handle software contracts, especially large contracts for the Federal government. The Federal government has issued a mandate which had two deadlines concerning software were January 1, 1984 and July 1, 1984. The January 1, 1984 deadline concerned the use of Ada for any and all new software being developed for defense mission-critical applications entering Advanced Development. The July 1, 1984 deadline concerned the use of Ada for any and all software entering Full-Scale Engineering Development. Other types of programs were encouraged to use Ada as soon as and whenever possible. Although the Department of Defense reset the deadlines, the use of Ada was and is presently a governmental mandate. Therefore, this particular major industrial company was working towards the development of software with Ada so that they could maintain their governmental contracts.

The first concern, for this and other companies I have talked with was the procurement of a validated Ada compiler which was compatible with their particular computer system. Several companies offered Ada compilers but they were almost never delivered in completed form by the contracted time. Partial compilers were usually delivered and updated over a period of time until the completed software package could be delivered. The partial compilers could compile programs composed of the Pascal subset of the Ada language: However, these compiler packages did not have the source code necessary to implement some of the more dynamic and interesting facilities of the language, such as, enumeration types, generics, and tasking. Consequently, this particular company was getting a slow start with its effort towards meeting the governmental mandate.

The lack of a complete compiler did not hinder the writing of objectives and the creation of small Ada source programs to test the aspects of a compiler as the pieces arrived.(1) I participated in this facet of the "move towards Ada". Two types of code were of interest to this industrial organization: regular program source code and code to monitor the system efficiency as Ada programs were compiled or executed. The regular program code was used to check the following and more: to determine whether error messages occurred when they should; to determine if various programs actually did turn control over to the system; to determine if the attributes really gave the required information; to determine if Ada's implementation of generics, packages, and tasking

(1) See sample objectives and source code.

were really the long awaited boost to software engineering that they appeared to be; and last but not least, to determine the real software engineering worth of the package. Packages were of special interest since it appeared that existing library facilities could really be enhanced and/or updated with their use.

The second type of code was system oriented code to monitor the computer's actions and reactions as Ada code was compiled or executed. Of special interest was how much of the system's power was going to be required to implement such features as tasking, generics, unbounded arrays,and pragmas. This aspect of the testing went beyond my level of expertise with Ada because I had only dealt with Ada from the programmer-software engineer side. Nevertheless, this aspect of the testing held a particular interest.

Educators teach languages and test for error situations, but very often the slowness of the system is taken as an ordinary daily occurrence. Industry on the other hand should not and cannot tolerate unnecessary slowness because some degree of efficiency is necessary.

Several programs were written in Ada to be used on an IBM P.C. computer. The memory of the IBM P.C. was increased to 512K initially to eliminate as many memory problems as possible before they might occur. The IMB P.C.'s compiler presented a problem because it was only partially complete thereby being incapable of testing those Ada facilities of most interest. Those features one could test with the system, for example: a minor program such as a prime number generator, took hours to execute. From the time periods involved in executing "small" programs with the micro, one could draw the conclusion that if the programs being compiled had contained the main software design features of Ada, the small system would have come to a virtual standstill or crashed.

Several months later, I had the opportunity to interact with other micros through the use of Ada and the responses were very similar to those mentioned above. It appears that the time has yet to arrive when the micro and Ada will be compatible.

Industry is presently experiencing a shortage of personnel capable of programming in Ada. A plausible reason for this shortage is that there exists only a few individuals capable of utilizing the language's capabilities to its fullest extent. Most of industry's beginning Ada programmers are experienced in Fortran and Cobol programming. Therefore, they tend to produce code in Ada which looks like Fortran or Cobol source code, respectively. However, after working with Ada for a length of time, they start to use the more advanced features of the language. If Ada's potential, as a software engineering language, is to be realized by industry, then the number of capable Ada programmers must be increased dramatically. The question is: How is this feat to be accomplished?

## Academic Outlook

Several institutions around the country are trying to bridge the gap between the number of Ada programmers presently available and the number of Ada programmers needed to meet the demands of industry. These demands were as a direct result of the mandate of the Department of Defense pertaining to specific areas where Ada should become the only language used to develop necessary software. The academic world is, however, experiencing some of the same problems which exist in the industrial world: a shortage of competent teachers (programmers) of the Ada language and most importantly the lack of a validated Ada compiler. A few institutions have the Ada Ed interpreter distributed by New York University which does the job but is slow. The slowness of compilation and execution speeds are especially noticeable if several students are trying to compile and/or execute the Ada programs at the same time. Therefore there is no mystery as to why students tend to shy away from both introductory and advanced courses in Ada programming and Ada software engineering. This trend tends to defeat a common goal of industry and academics; to increase the number of Ada educated computer scientists.

As in industry, partially completed Ada compilers are available to the world of academics. The frustrations associated with not being able to use the "power house" structures of the language, so that individuals themselves can determine what they consider the full worth of the language, takes its toll on student enrollment. Rumors travel fast among students about a course or its teacher. Ada student enrollment is on the rise at those schools where it is being offered, but because of the reasons mentioned earlier Ada's enrollment lags behind those of such languages as Fortran, Pascal, and Cobol. Inspite of the major discouraging factors involved with an Ada education, there is hope for the future for Ada in academics as well as in industry.

## Academic/Industrial Common Efforts

There are several Ada user's groups presently in existence, for example: the ACM group, the group at the University of Houston at Clearlake and some of them are involved with experiments which are attempting to test and evaluate Ada in various working environments. At present it appears that no major governmental agency, like NASA for example, is attempting any full scale software development with Ada. Yet the interest in the language and its capabilities is very much alive and well. Some NASA locations are experimenting with such systems as the NTELL 432, TeleSoft, and Rom.

Presently code is being generated for software to test Ada's feasibility for a number of areas. Some questions which need to be considered are the following.

Could it make the running of a space station easier than it presently is or harder? Would flight programs become more manageable with Ada's capabilities as a resource, especially tasking? If Ada was used in a distributed network environment and one of the members of the network failed, could tasking recover the situation? Would satellite and other methods of communications benefit from its seemingly robust nature? Is it really true that several groups of programmers, each

group in a different location, can actually create segments of code for given modules of a relatively large project? Could the groups meet after an allotted period of time and turn the modules into a working program in 25 to 50 percent less time than would be the normal time period by today's standards? Is Ada really the portable language the world hopes and believes it to be? Is the ability to separately compile parts of a program worth the overload? Can a group create software in a particular area, in-house and then offer that software to the world with minimum difficulty?

Each of these questions needs an answer, if one of is to really become knowledgeable in the Ada world. The world of academics is doing its share to answer some of the questions raised above and to supply competent Ada programmers to the industrial/governmental world at the same time. Not only is Ada being taught as a class at a rapidly growing number of institutions, these institutions are obtaining grants and proposals to perform specific experiments that could indeed give Ada that needed boost to remove doubt from the minds of many non-believers. The University of Houston at Clearlake, the University of Virginia, the University of Maryland, Hampton University, Old Dominion University, and many others in other parts of the U.S. involved with Ada ranging from the beginning teaching stages to the more complex areas of creating software under proposals and contract with various governmental agencies. For instance, the University of Virginia is working with Ada in a distributed network environment, the University of Houston and Maryland are doing some work to evaluate the language, and Hampton University is trying to develop some mathematical packages.

Many industrial organizations,who have little or no governmental contracts at the present time, are staying away from Ada. They considered it a dying cause even before its birth. However, those organizations who do a great deal of contractual work for the government must turn to developing software using the language if they are to maintain their status with the government. They are willing to work with the world of academia to obtain Ada qualified and software engineers. Several suggestions have been offered: 1) Do not teach the language from the syntax point of view; 2) teach good software development methodologies, such as various design approaches, style of development, and usage of portability and reuse capabilities during development; 3) interact more with various industrial organizations so that a student will be the recipient of a broad-based education thereby effectively increasing his worth to any organization to which he may belong.

Many academic institutions are trying to comply with industry's requests pertaining to computer training. Academic institutions, however, also need some assistance from industry. They need incentives to offer potential Ada students, such as workable equipment, good salaries, and good working environments upon completion of a course of study which included Ada training. To a certain extent, industry provides for some of these needs. Quite often students discover that a company offers a high salary if a potential employee has been trained in the area of Ada. Sometimes they offer free system maintenance for a period of time to a school which is struggling to implement a program in computer science independent of an offering in Ada programming. A faculty exchange program is also maintained to allow both academia and industry to keep abreast of each other's needs.

Various industrial organizations offer short courses in Ada which encompass programming and software engineering. Others such as Digital Equipment Corporation are in the process of offering such courses. They also allow employees time during their regular work day to take courses. Very often they reimburse tuition and fees to those employees who take courses on their own time. Industry is working towards eliminating the shortage of Ada educated individuals.

The number of Ada qualified individuals is on the rise, but the number is not yet remotely close to the large number needed in the industrial world, if the Department of Defense's mandate is to be met in the near future. Education and industry must do more hand-in-hand work if Ada's future goals are to be met. Exchange programs which allows institutions, which do not have any Ada computing facilities, to access certain company accounts in order to execute student programs might be implemented in a large scale across the country. The duality of benefit would be the following: The institution would benefit because it could teach Ada with the excess cost of acquiring an Ada compiler system not being present. The company would benefit because individuals would be trained in the use of Ada for programming and software engineering. Academic institutions could hold special courses for employees whose organizations needed Ada for its company's governmental contractual stability. These courses would also be available to companies who aspire to obtain governmental contracts but do not presently create software for the government.

In conclusion, software engineering with Ada is increasing in volume, however, it has not reached a level encompassing current needs.

The following are samples of objectives and source code created for micro compiler testing and standard compiler testing.

Exceptions

Specify "exceptional" situations which arise during program execution. These situations are usually caused by an error in the program.

Some exceptions are predefined and others can be declared by the user. A programmer can raise an exception.

The list of predefined exceptions is as follows:

```
constraint_error      numeric_error
program_error         storage_error
tasking_error.
```

Note: It is possible to use the pragma "SUPPRESS" to stop the raising of the above mentioned exceptions.

Constraint_error:

specifies that an error message be given if some entities go out of bounds previously set.

Test Objectives:

1. Check that each of the following causes the correct error message to be printed: a) attempt to violate a range constraint; b) attempt to violate an index constraint; c) attempt to violate a discriminant constraint; d) attempt to use a record component that does not exist for the current discriminant values. 2. Check that a null access value will cause the error in each of the following cases: a) attempt to use a selected component of the null object; b) attempt to use an indexed component of the null object; c) attempt to use an attribute of the object; d) attempt to use a slice of the object.

Numeric_error:

Raised when the result of a numeric operation exceeds the implemented range of some real type.

Test Objectives:

1. Check that each of the following causes the raising of this error message: a) data underflow; b) data overflow; c) divide-by-zero.

Program_error:

Raised upon an attempt to make use of a unit whose body has not yet been elaborated.

Test Objectives:

1. Check that a raise occurs automatically: a) call to a subprogram; b) attempt to activate a task; c) attempt to elaborate a generic instantiation. 2. Check that this situation occurs when an exit from a function is attempted without using a "return" statement. 3. Check that a raise occurs during the execution of a selective wait that has no else part and all alternatives are closed. 4. Check that this is caused during an erroneous execution. 5. Check that an incorrect order dependency causes this error.

Storage_error:

Raised when storage allocated to a task or for a collection is exhausted during the execution of an allocator.

Test Objectives:

1. Check that both of the following are causes of this error: a) when dynamic storage allocated to a task is exceeded; b) when the space available for the collection of allocated objects is exhausted.

Tasking_error

Occurs during inter_task communication.

Test Objectives:

1. Check that the abnormal termination of a server task causes this error in the "caller" task. 2. Check that abnormal abortion of the caller task does not raise an error.

PRAGMAS:

Pragmas are used to convey information to the compiler. They are allowed after a semicolon delimiter but not within a formal part or discriminant part. The list of predefined pragmas are as follows:

| | | |
|---|---|---|
| Controlled | Elaborate | Inline |
| Interface | List | Memory-size |
| Optimize | Pack | Page |
| Priority | Shared | Storage-unit |
| Suppress | System-name | |

The pragmas memory-size, storage-unit, and system deal primarily with areas which will not be touched upon by this group of tests.

Controlled:

Specifies that automatic storage reclamation must not be performed for objects designated by values of a given access type.

Test Objectives:

Syntax:
1. Check that non-simple names of access types are not allowed. 2. Check that multiple arguments are not allowed. 3. Check that this pragma is only allowed within the declarative part or package specification which contains the declaration of the access type. 4. Check that derived types cannot be used. 5. Check that the pragma declaration must occur after the declaration of the given access type.

Semantic:
1. Check that storage reclamation does not occur during the duration of this program. 2. Check that in the absence of controlled automatic storage reclamation follows the system default method.

Elaborate:

Specifies that the corresponding library unit body must be elaborated before the compilation unit.

Test Objectives:

Syntax:
1. Check that only simple names denoting library units can be used as arguments. 2. Check that this pragma is only allowed to immediately follow the context clause of a compilation unit. 3. Check that each argument must be the simple name of a library unit mentioned by the context clause. 4. Check that both single and multiple arguments are allowed.
Semantic:

1. Check that the components of the argument list are elaborated. 2. Check that an error occurs if the necessary units are not elaborated.

Inline:

Specifies that subprogram bodies should be expanded inline at each call whenever possible. (In the case of generics, this applies to instantiations.)

Test Objectives:

Syntax:

1. Check that only names of subprograms or the name of a generic subprogram can be used as an argument. 2. Check that this pragma is only allowed in one of the following three places: a) at the declarative place of a declarative part of a program; b) at the place of a declarative item in a package specification; c) after a library unit in a compilation but before any subsequent compilation unit.

Semantic:
1. Check that the code for the members of the argument list is actually expanded at the place of each and every call. 2. Check that no arguments are passed, etc. as in the usual way that subprogram calls are handled. 3. Check that without inline, the expansion does not occur.

Interface:

Allows a subprogram in another language to be called by an Ada program provided all communication is achieved via parameters and function results.

Test Objectives:

Syntax:
1. Check that a language name and a subprogram are the only allowable arguments. Check that this pragma is allowed at the place of a declarative item and must apply to a subprogram declared by an earlier declarative item. 3. Check that the above-mentioned declarative item and pragma must belong to the same declarative part or specification. 4. Check that this pragma must appear after the subprogram declaration and before any subsequent compilation unit when dealing with a library unit.

Semantic:
1. Check the system to determine how the acquiring of the subprogram in a second language is achieved. 2. Check the system to determine how the return to the Ada language is achieved.

List:

Takes one of the identifiers ON or OFF as a single argument. If a compiler listing is occurring then the use of OFF will cause it to cease until ON causes it to restart the printing of the given file.

Test Objectives:

Syntax:
1. Check that it can be placed after a semicolon delimiter but not within a formal part or discriminant part. 2. Check that this pragma can be placed anywhere that a syntactic category whose name ends with declaration, statement, clause, or alternative can be positioned.
Semantic:
1. Check that a listing of a compilation is suspended or continued until a list pragma with the opposite argument is encountered.

Memory-size:

Takes a numeric literal as the single argument. It is only allowed at the start of a compilation and only before the first compilation unit in a library. It is associated with the package for the "SYSTEM". This pragma will not be dealt with by this series of tests.

Optimize:

Specifies whether time or space is the primary optimization criterion.

Test Objectives:

Syntax:
1. Check that this pragma cannot appear any place in a program other than the declarative part of a block. 2. Check that optimize only applies to the block or body enclosing the declarative part to which it belongs. 3. Check that the only arguments allowed are either "time" or "space".

Semantic:
1. Check that the system time becomes shorter versus default time for the same amount of code.
2. Check that the amount of space allowed for a block of code changes if optimize uses space as an argument.

Pack:

Specifies that storage minimization should be the main criterion when selecting the representation of the type being considered.

Test Objectives:

Syntax:
1. Check that a record or array type are the only possibilities as arguments. 2. Check that this pragma can only occur in the declarative part, package specification, or task specification.
3. Check that any use of a representation attribute of the packed entity must appear after the pragma declaration.

Semantic:
1. Check memory allocated for the designated code to determine whether the amount of space is actually less than the default amount. 2. Check that the amount of memory will correspond to the default value if pack is not applied.

Page

Specifies that the program text which follows the pragma should start on a new page if the compiler is currently producing a listing.

Test Objectives:

Syntax:
1. Check that this pragma can occur where any other pragma can occur.

Semantic:
1. Check that all text following the call to this pragma does go to a new page if the compiler is listing a program.

Priority:

Specifies the priority of the task in which it occurs or it specifies the priority of a main program.

Test Objectives:

Syntax:
1. Check that the only arguments are static expressions of the predefined integer subtype priority. 2. Check that multiple arguments are not allowed. 3. Check that this unit can only appear: a) within the specification of a task unit; b) immediately within the outermost declarative part of a main program.

Semantic:

1. Given several tasks of unequal priority, check that the tasks are selected in order of highest priority to lowest.

Shared:

Specifies a variable which must be shared by more than one task. Every read or update of the variable is a synchronization point for that variable.

Test Objectives:

Syntax:
1. Check that only one argument is allowed.
2. Check that the variable involved is of type scalar or access type. 3. Check that the variable declaration must be followed immediately by this pragma in the same declarative part or package specification.

Semantic:
1. Check that the reading and direct updating of allowable variables is an indivisible operation.

Storage-unit:

Takes a numeric literal as its single argument. It is allowed at the start of a compilation, before the first compilation unit. This pragma will not be dealt with by this series of tests.

Suppress:

Specifies that errors associated with the identifier in the argument list will not be checked for, unless it is too costly to suppress the checks. I the identifier is followed by some entity to be worked upon, then those errors associated with the entity will be suppressed.

Test Objectives:

Syntax:
1. Check that the identifier of a check is the only necessary argument. 2. Check that along with the identifier, the following may also be present: the name of either an object, a type or subtype, a subprogram, a task unit, or a generic unit.
3. Check that this entity is only allowed either immediately within a declarative part or immediately within a package specification. 4. Check that the only allowed form is with a name that denotes an entity (or several overloaded subprograms) declared immediately within the package specification. (Providing this pragma is to be used in a package specification.)

Semantic:
1. Check that the suppress extends from the place of the pragma declaration to the end of the declarative region associated with the innermost enclosing block statement or program unit.
2. Check that the suppression extends to the end of the scope of the named entity when working with a package specification. 3. Check that when an optional name is given (name of one of the entities in objective 2.) the suppress applies only for:
a) operations on the named object or on all objects of the base type of a named type or subtype;
b) calls of a named subprogram; c) activations of tasks of the named task type; d) instantiations of the given generic unit.

System-name:

Specifies system changes. Takes an enumeration literal as the single argument and is only allowed at the beginning of a compilation. In the case of libraries it must appear before the first compilation unit only. This pragma will not be dealt with in this series of tests. It deals with the "SYSTEM".

The following is a listing of programs to be used to test various aspects of an Ada compiler. Those programs with code for separate compilations, packages, pragmas, and tasking were not used on micros. Several of the programs in part/and or total are by Young [5]. Booch[2] and Wiener[4].

```
--Use_interface is a procedure which uses the
package interface.
--Some computations are performed by Sqrt and exp.
--
--Created August 22, 1984
--
With Text_io,interface; Use text_io;
Procedure Use_interface is
Num,Ans1,Ans2:float;
package flt_io is new float_io(float); use flt_io;
Inter_in,Inter_out:  file_type;
```

```
    begin --Use_interface
         Open(Inter_in,in_infile,"Interi.dat");
         Create(Inter_out,out_file,"Intero.ans");
         Set_input(inter_in);
         Set_output(inter_out);
         Put_line("Output answers created by a
             Fortran Subprogram");
         Put_line("----------------------------");
         New_line(2);
         Put_line("Num     Ans1      Ans2      ");
         Ans1 :=Interface.Sqrt(num);
         Ans2 :=Interface.Exp (num);
         New_line;
         Put (Num,5);
         Put (Ans1,5);
         Put (Ans2,5);
         New_line;
         Close(inter_in);
         Close(inter_out);

         end Use_interface;

-- Package interface displays the use of pragma
interface.  It allows Ada to make use of a sub-
program in a different language.  The language
used here is Fortran.
-- Created August 22, 1984

--
Package Interface is

-- Ada specifications of Sqrt and Exp

    function Sqrt(x: Float) return Float;
    function Exp (x: Float) return Float;

private

--The Fortran Subprograms

    pragma Interface(Fortran,Sqrt);
    pragma Interface(Fortran,Exp);

End Interface;

--  Program except tests two exceptions and the
pragmas suppress page, and list.Numeric_error and
Constraint_error are to be tested.
--
--  Messages will be printed if either of the
errors is encountered.
--  The pragmas page and list will also be checked
by this program.
--
--
--  Extreme data values are required to get
numeric error to appear.  A divide-by-zero and a
number larger than the maximum system float value
will cause numeric error.
--  Use a=0 and a=0.00001 or some equally likely
candidate.

--  Suppress and its parameters explain what
errors are being suppressed.

    With text_io; Use text_io;
    Procedure Exceptions is

      Pragma Suppress(Division_check);
      Pragma Suppress(range_check, on = >temp);

--Determine whether Suppress belongs here or inside
of procedure roots.

subtype Non_negative_real is float range 0.0..Float
'last;
subtype Pos_real is float range Float'small..float'
last;

package flt_io is new float_io(float); Use flt_io;

a,b,c,r1,r2:float;

Pragma List(off);

function Sqrt (x_value: in Non_negative_real;
    eps: in Pos_real:=0.001) return float is

--
--
--Only floating point numbers greater than 0.0 are
accepted.
--Positive square root is returned with the speci-
fied accuracy
--
--the approximation method is used to find the
square root.
--
old_value:float;   --kth approximation
new_value:float;   --f+1st approximation

begin
      old_value:0.0;
      new_value:=x_value/2.0   --initial guess


      while abs(new_value-old_value)> eps
         loop
           old_value:=new_value;
           new_value:=0.5* (old_value + x_value/
             old_value);
         end loop;
      return new_value;
end Sqrt;

Pragma List (ON);
Pragma Page;

Procedure Roots (A,B,C:  float;  R1,R2: out float)
is
--
Pragma Suppress (division_check);
Pragma Suppress (range_check, on temp);

--Roots actually finds the square root.
--Sqrt is called from this procedure.
--

      temp:float;
begin --roots

temp:=sqrt (b*b-4.0*a*c);
r1:=(-b+temp)/(2.0*a);
r2:=(-b-temp)/(2.0*a);

exception
```

```
when numeric_error =>
    put("Numeric_error");
    new_line;
    put("overflow or divide by zero");
    new_line;
when constraint_error =>
    put("Constraint_error"):
    new_line;
    put(" B*B-4*A*C is negative");
    new_line;
end roots;

Pragma Page;

begin --exceptions
    get(a);
    get(c);
    Roots(a,b,c,r1,r2);
    put("a= ");
    put(a);
    put(" ");
    put("b= ");
    put(b);
    put(" ");
    put("c= ");
    put(c);


    put("r1= ");
    put(r1);
    put("r2= ");
    put(r2);
    new_line;

end exceptions;
```

Acknowledgements:

Valuable comments and information were received from the following people for this paper: Allen R. Crawley of Information Development and Applications, Inc. (Ideas) Virginia Beach, Virginia; Duvan Luong, IBM Corporation, Gaithersburg, Maryland; Susan Voigt of NASA Langley Research Center, Hampton, Virginia; Iona Black of Hampton University, Hampton, Virginia.

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

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

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

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

Joan M. Sterling is a lecturer of Mathematics and Computer Science at Hampton University in Hampton, Virginia 23668. Ms. Sterling is a graduate of the College of William and Mary and Old Dominion University where she received a Master's Degree in Mathematics and a Master's Degree in Computer Science respectively.

# QUEUE_MANAGER, A Useful Example For Teaching Ada

David Rudd

Department of Computer Science
University of New Orleans
New Orleans, LA 70148

Summary

It is the author's belief that clear, well-constructed examples play a major role in teaching the concepts and methods of computer programming. The purpose of this paper is to describe one such example (actually a progression of related examples) that the author has found to be a valuable and useful pedagogical tool in an Ada course. The example consists of a package which can be used in queue management -- first a single queue with a specified number of integer entries, then an arbitrary number of queues each with a (possibly) different number of integer entries, and finally an arbitrary number of queues each with a (possibly) different number of entries of an arbitrary type.

## 1. Introduction.

It is the author's belief that good examples play a major role in computer science education. It is fine to discuss a general philosophical framework for a concept, why it is important, and rules and methods for implementing it; but there is no substitute for a clear, non-trivial example. Student appreciation and comprehension are greatly aided by illustrative examples of concepts.

Since the package is the major new feature of the Ada programming language, it is especially desirable to incorporate packages into important examples and homework assignments in an Ada course. The purpose of this paper is to describe a series of packages of increasing complexity and utility which can be used to illustrate such major features of Ada as exceptions, generics, private types, unconstrained arrays, and variant records. The packages are presented as progressively more powerful solutions to the general problem of managing queues.

## 2. The Original Problem.

In order to couch the problem in a reasonably realistic setting, we consider an office with 10 telephone lines numbered 1 through 10. Callers are all trying to reach an agent in order to have their questions answered. We assume that only one person can talk to the agent at a time; the others (if any) remain on hold.

The operator who initially answers the calls places callers in a first-come-first-served queue and connects the caller at the head of the queue to the agent when a signal is received from the agent. At any time, the operator wishes to be able to do the following:

1) place a caller at the tail of the queue
2) remove any entry from the queue
3) connect the entry at the head of the queue to the agent and have that entry removed from the queue
4) list the entries in the queue, in order.

## 3. QUEUE_MANAGER_1.

The first assignment is to write a package which can be used for the original problem. A possible solution is given below.

```
--      The purpose of this package is to manage
--  a queue of integers from 1 through MAX_SIZE.
--  The queue will be represented as an array
--  with 1 the index for the head and COUNT the
--  index for the tail.  The package can be used
--  to perform the following operations:
--
--      1) insert an entry at the tail of the
--         queue
--      2) remove any entry from the queue
--      3) display the entry at the head of the
--         queue, then remove that entry
--      4) list all entries in the queue in
--         ascending order.
--
with TEXT_IO; use TEXT_IO;
package QUEUE_MANAGER_1 is
   procedure INSERT (X : in INTEGER);
   procedure REMOVE (N : in INTEGER);
   procedure CONNECT;
   procedure LIST;
end QUEUE_MANAGER_1;
```

```
package body QUEUE_MANAGER_1 is

   package INT_IO is new INTEGER_IO(INTEGER);
     use INT_IO;

   MAX_SIZE  : constant INTEGER := 10;
   COUNT     : INTEGER range 0..MAX_SIZE := 0;
   Q         : array (1..MAX_SIZE) of INTEGER;

   procedure INSERT (X : in INTEGER) is
   begin
      COUNT     := COUNT + 1;
      Q(COUNT)  := X;
   end INSERT;

   procedure REMOVE (N : in INTEGER) is
   begin
      for I in N .. COUNT - 1 loop
         Q(I) := Q(I + 1);
      end loop;
      COUNT : = COUNT - 1;
   end REMOVE;

   procedure CONNECT is
   begin
      PUT_LINE("head of queue is "); PUT(Q(1));
      REMOVE(1);
   end CONNECT;

   procedure LIST is
   begin
      for I in 1 .. COUNT loop
         PUT(Q(I)); NEW_LINE;
      end loop,
   end LIST;

end QUEUE_MANAGER_1;
```

The students might be presented with this particular solution for purposes of discussion. Here are some suggestions for such a discussion.

1) What is accomplished by declaring the array object Q in the body of the package? (If it were declared in the specification, the integrity of the package might be compromised. For example, a user program would be able to automatically insert phone number 2 at the head of the queue each time it rang, or not allow number 5 into the queue at all.)

2) What happens if you attempt to insert into a full queue or remove from an empty queue? How can exceptions be used to improve the robustness of the package?

3) Supposed it is decided to change the representation of the queue from an array to a linked list. What changes, if any, would have to made in the package specification, body, and user program?

4) The package could be made more complete by including a procedure to present the user with a menu of choices. Then the statement part of the user program could consist of only a single statement -- a call to the menu procedure. Is this a good idea? Show how to incorportate such a procedure into the package.

5) Since this package is going to be used by human beings, who are by their very nature error prone, it is important to provide some means for recovery from the inevitable input errors (e.g. typing the letter "l" instead of the number 1). What feature of Ada gives the package the means to recover from such errors? (This is a good way to see the need for exceptions such as DATA ERROR.)

4. QUEUE_MANAGER_2.

We are now ready to consider making our QUEUE_MANAGER more powerful, and in so doing we introduce some additional Ada features.

We can imagine that our customer is pleased with the package QUEUE_MANAGER_1, so much so in fact that he wants to use it to manage other queues, still of integers but with different lengths. Package QUEUE_MANAGER_2 is to provide this additional capability. A possible solution is given below.

```
--     The purpose of this package is to
-- manage an arbitrary number of queues of
-- integers, each with a (possibly) different
-- size.  Each queue Q will be represented as
-- an array with 1 the index for the head and
-- Q.COUNT the index for the tail.
--
--     The package can be used to perform the
-- same 4 operations as QUEUE_MANAGER_1.
--
with TEXT_IO; use TEXT_IO;
package QUEUE_MANAGER_2 is

 type QUEUE (NUMBER_OF_ENTRIES : INTEGER)
   is limited private;
 -- QUEUE is a discriminated private type with
 -- discriminant NUMBER_OF_ENTRIES

procedure INSERT  (Q : in out QUEUE;
   X : in INTEGER);
procedure REMOVE  (Q : in out QUEUE;
   N : in INTEGER);
procedue CONNECT  (Q : in out QUEUE);

procedure LIST    (Q : in QUEUE);

   private
      type QUEUE_DATA is array (INTEGER range
        <>) of INTEGER;
      type QUEUE (NUMBER_OF_ENTRIES: INTEGER) is
         record
            QUE      : QUEUE_DATA (1..NUMBER_OF_
                          ENTRIES);
            COUNT    : INTEGER := 0;
         end record;

   end QUEUE_MANAGER_2;
```

# MODELING ADA TASKS — AN INITIAL SURVEY

R.M. Blasewitz and M.J. Gagliardi

RCA Government Systems Division
Missile and Surface Radar

## Abstract

This paper presents the results of a preliminary investigation into techniques and methodologies that support the representation of real-time system designs in Ada. It represents an overview of some widely disseminated methods, including Buhr diagrams, Petri Nets, PDL code and flow charts. The objective of the research is to derive a means of communicating real-time processes in a suitable fashion across the life cycle of the software product. The paper also discusses some of the problems encountered with the research due to the present state of Ada tools and compilers.

## Introduction

The Ada* programming language provides a technique for expressing potential parallelism as an approach to solving the synchronization and communications problems of today's major real-time systems. The name given to programming notations and techniques for expressing potential parallelisms is "concurrent programming." Concurrent programming is important because it provides an abstract viewpoint from which to study parallelism without being buried by the details of a particular implementation. The ability to write concurrent programs is very desirable for a number of reasons:

1. Real-time systems, operating systems, data-base systems can be expressed in a convenient notation at a high level of abstraction.
2. Algorithms that cry out for concurrency are best expressed using language features that support and model concurrent events.
3. The complex reasoning involved in concurrency and execution time constraints can be made more user friendly and hence understandable.
4. Program execution time, efficiency and elegance can be greatly enhanced without pushing the state of the art of fourth-generation hardware.
5. A certain class of problems can be most easily and elegantly solved by parallel communication processes instead of the often-used sequential methodology.

The initial desire to use concurrent programming languages stemmed from attempts to write conceptually concise programs that reflected or mirrored the structure of an algorithm. However, current interest is probably largely motivated by the desire to take advantage of recent advances in the realm of computer architecture. These advantages materialize in many ways, namely:

1. More computing power per device per dollar is being realized today.
2. Computing facilities offered by microcomputers or computers on a chip rival those of larger minicomputers and main frames.
3. Benefits of highly parallel hardware architectures and concepts are accruing through support of concurrent operations in an efficient and understandable manner. These benefits are now of concern in the scientific community.

Although concurrent languages offer aid to the programmer in abstracting the functional features of a program from the implementation of an algorithm, the real (or unreal) "art" of designing parallel programs is still undeveloped because we lack formalism and understanding of parallel programs. To further complicate this matter, there are very few, if any, acceptable methodologies or practices in current use to clearly and concisely represent real-time or parallel program design. Although many methods have been proposed, few have gained wide acceptance within the software development community. Is the problem due to the complexity of software designs, languages, concurrent programming knowledge, or to a lack of acceptance of new programming concepts and paradigms? Or is it a combination, subset or superset of those reasons? Obviously there is no universal answer to this question. A survey of existing methodologies or practices in this broad arena leaves one more bewildered than one would expect. The picture becomes even more fuzzy when Ada enters as a possible candidate for real-time program development. The potential in all these techniques is clearly very high, but we cannot expect instant solutions for embedded or large-scale computer program developments in the real-time programming community.

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

Where does Ada fit into this section of the software universe? It is generally agreed that earlier languages such as PL/1, Algol 68, Concurrent Pascal, Modula or even Euclid have offered only rudimentary facilities for concurrent programming. From a programming language viewpoint, Hoare's proposal to use the rendezvous concept as the basis for concurrent programming was a major advance.

Hoare defined a concurrent program as a collection of sequential programs that can execute in parallel — all cooperating to implement a common objective. These sequential programs or processes interact by first synchronizing and then exchanging information. Synchronization and communication are viewed as an integral activity that is called the rendezvous. This leads us to Ada, whose concurrency facilities are based on Hoare's ideas with modifications and additions to deal with the realities of hardware and with other practical concerns such as error handling and program development. Ada is the first major general-purpose programming language to provide high-level concurrent programming facilities based on the rendezvous concept. Ada and its facilities are elegant and easy to use, but are untested and untried to any significant degree. Only time and practical experience will lead to a detailed evaluation of Ada's capability in this area. This paper proposes to evaluate Ada's initial capability in this area and to offer a range of viewpoints for real-time design representation in Ada. However, this research is somewhat weakened by the lack of Ada compilers to support the concurrency features of Ada in total. As such, these observations are certain to undergo perturbations as more efficient Ada compilers become available.

The methodology employed by the authors centered around the following considerations:

1. The basis of the study will be Ada in its present state (compiler state).
2. Various approaches to representing real-time or parallel program execution will be examined against conventional methods.
3. Ada's usefulness in the concurrency programming domain will be assessed, with emphasis given to its ability to implement algorithms in real-time.
4. Using the Ada designs as a basis, a real-time representation will be presented that combines the present working knowledge of presently available real-time design representations.

The recommendations and conclusions regarding Ada as a real-time systems language rest almost entirely on design experience gained from an operating system project that included the modification of a vendor-supplied Ada compiler for translation from VAX 11/780 code to Nebula code. Although Ada is the first language to use the rendezvous concept, it is the opinion of the authors that the compiler technology has not given Ada the chance it deserves. Very few of the present Ada compilers implement full tasking (Ada's concurrency facility) in a manner conducive to a fair and conclusive study of performance or efficiency. Nontheless, other aspects or issues of Ada can be evaluated, such as expressive power, user friendliness, concurrency capability, and scheduling mechanism capability.

Coupled with this look at Ada, initial conclusions are drawn regarding capabilities for real-time programming representations using Ada as the implementation language.

An Ada program that uses tasking may suffer from significant portability problems (due to instruction execution times from machine to machine), but this aspect of its capability was not considered a major issue in evaluating its present capabilities as a real-time language. It is also assumed that the current lack of full-capability Ada compilers (not to say production-grade compilers) will soon be alleviated.

## II. Major Approaches to Representing Parallel Program Execution

This section examines a number of the six noted major techniques presently used to represent or communicate the behavior of parallel processing systems. Included in this set are:

1. Petri nets
2. Buhr diagrams
3. Flow charts
4. PDL code
5. State graphs
6. Object-oriented design
7. Functional decomposition techniques
8. Mathematical notations

Although all of these techniques play an important role, the ability to represent design through good, clear diagrams appears to be the best communication tool. When systems are modified, clear diagrams are an essential aid to maintenance and allow changes to be made with a better understanding of the consequential effects of that change. Obviously, mathematical notations are important, but somehow lack the power of noise-free communication made available by clear diagrams, charts, etc.

The first technique investigated is the Petri Net, developed by the German Scientist Carl Adam Petri in the early 1960s to study and model communicating parallel automation. These nets have a mathematical side and a graphical, intuitive side. As such, these nets offer a possible means of clarifying the abstract concepts of parallel programming.

A Petri Net is a directed graph that contains two kinds of nodes: place nodes, and transition nodes. Place nodes are represented by circles and transition nodes by means of bars, small black boxes, or rectangles with statements in them. Figure 1 illustrates a simple Petri Net with four place nodes and three transition nodes.

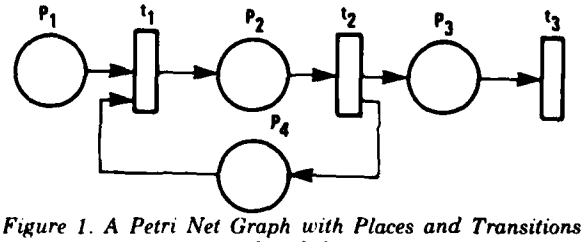



*Figure 1. A Petri Net Graph with Places and Transitions Labeled.*

Before we can examine how Petri Nets model parallel processes, let us clear up some basics about their operation. Arcs that connect places and transitions are called directed arcs. In general, a transition has input places and output places; a place is an input place of a transition if that place has an arc directed at its transition. For example, place $P_4$ is one of the input places for transition $t_1$ and one of the output places for $t_2$. These input and output places can be mathematically represented as:

$I(t_1) = \{P_1, P_4\}$
$I(t_2) = \{P_2\}$
$I(t_3) = \{P_3\}$

where $I(t_i)$ = Input place for transition $t_i$
$\{P_i, P_j\}$ is the set $P_i, P_j$ at the input or output places

and

$O(t_1) = \{P_2\}$
$O(t_2) = \{P_3, P_4\}$
$O(t_3) = \{\}$

where $O(t_i)$ = Output place for transition $t_i$

Petri Nets can also be marked by placing tokens, represented as small dots, in the net place nodes. The Petri Net executes by firing its enabled transitions. A valid firing situation is defined as one in which each input place must have a token in it.

Therefore, the structure and marking of a Petri Net determines its execution. When the firing occurs, the changes are marked by placing the input tokens into each of the transitions output places. Consider Figure 2, which is a marked version of Figure 1.

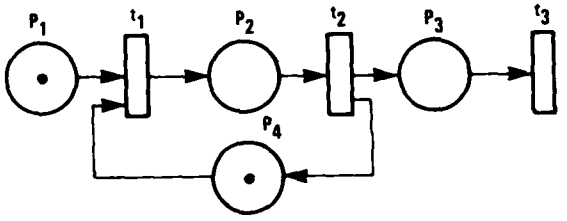



*Figure 2. A Marked Petri Net Graph.*

Now assume that enabled transition $t_1$ in Figure 2 fires. The new marking is then illustrated by Figure 3.

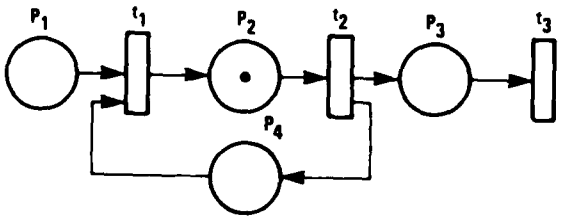



*Figure 3. A Fired Petri Net.*

If we again fire the enabled transitions, we have the Petri Net as represented in Figure 4. This sequence of events illustrates the basic working of the Petri Net. Our next concern is the actual use of these graphs in modeling computer programs.

To model the dynamic behavior of a system, the execution of a process is represented by the firing of the corresponding transition as illustrated. The changes in system state are

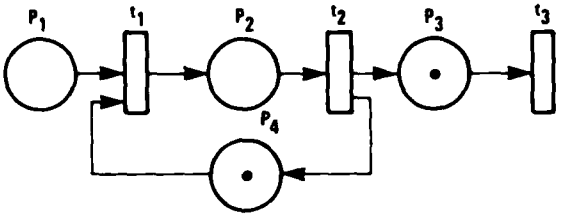



*Figure 4. The Second Firing of the Petri Net.*

represented by the movements of tokens in the net. Without getting into the mathematical representations afforded by Petri Nets, one can easily model sequences of statements by transitions, the points between actions by places, and the value of a program counter by the location of a Petri Net token. Before moving on to parallel program modeling, let us examine the use of Petri Nets in modeling sequential programs consisting of sequence of statements, conditional statements, and loop statements.

Consider the following partial Ada code for division with remainder:

```
quotient := 0 ;
numerator := x ;
denominator := y ;

while numerator >= y loop
        quotient := quotient + 1 ;
        numerator := numerator - denominator ;
end loop ;
```

The Petri Net representation for this code is illustrative of the elements of a sequential program. The sequential elements with their Petri Net representation are given in Figure 5.

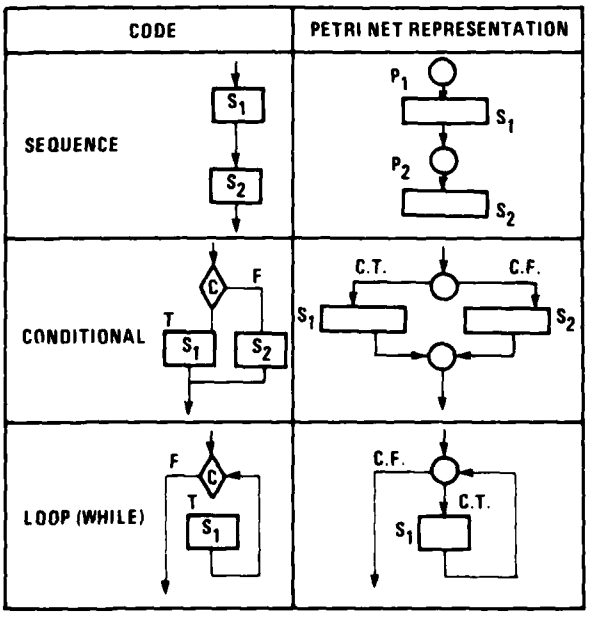



*Figure 5. Modeling Sequential Structured Elements With Petri Nets.*

The Flowchart for the Ada program is given in Figure 6 with the corresponding Petri Net given in Figure 7.

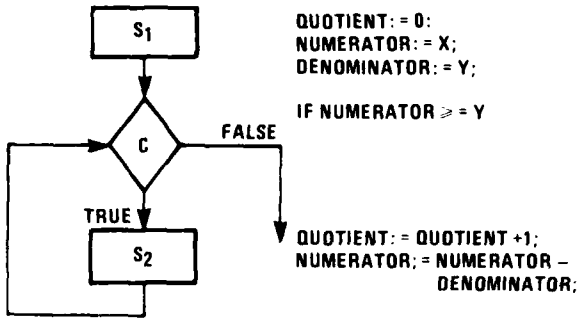



*Figure 6. Flowchart For Ada Division Program.*

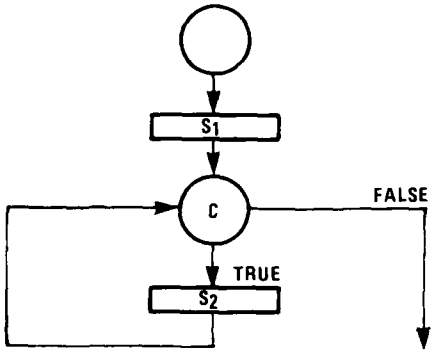



*Figure 7. Petri Net For Ada Division Program.*

We have described classical Petri Nets up to this point. We can now extend the classical Petri Net with a mechanism for "zero testing" a place; this mechanism is called the inhibitor arc. An inhibitor arc from a place $P_i$ to a transition $t_i$ terminates with a small circle, rather than an arrowhead. Figure 8 illustrates the use of an inhibitor arc.

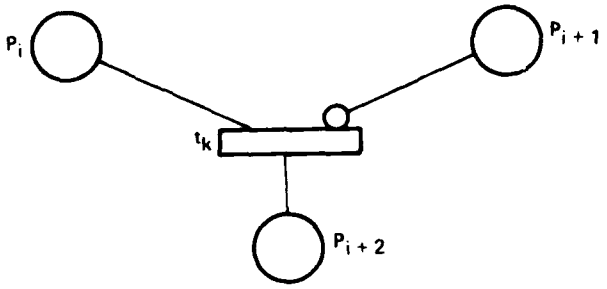



*Figure 8. The Inhibitor Arc Extension For Zero Testing.*

For $t_k$ to fire there must be at least one token in $P_i$ and zero tokens in $P_{i+1}$. Thus the transition $t_k$ tests place $P_{i+1}$ for zero. Simply, a transition cannot fire unless its inhibitor arc place is empty. This type of scheme can be used to model concurrent programs rather simply. To represent concurrent activities no new Petri Net mechanisms need be introduced. To spawn new paths of control, a fork or cobegin is used, as shown in Figure 9.

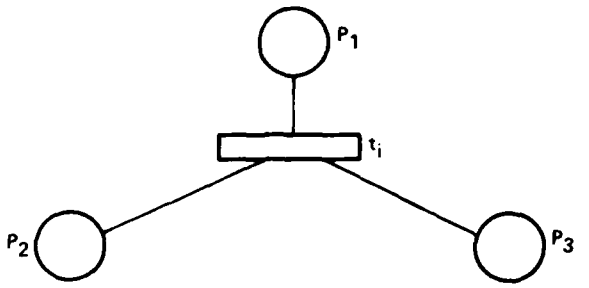



*Figure 9. The Petri Net Cobegin or Fork (one path in, two paths out).*

Using this mechanism we can represent two or more parallel paths, where each path operates independently. Figure 10 illustrates a possible Petri Net for parallel activities and precedence, since the completion of activities represented by $t_1, t_2, t_6$ must precede the start of activity $t_4$.

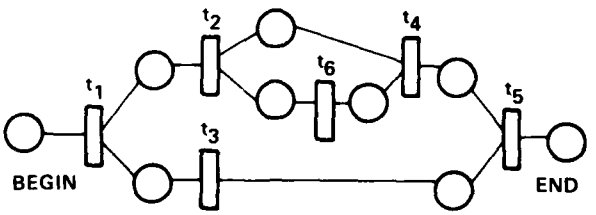



*Figure 10. Parallel Activities With Precedence For Petri Net Graphs.*

Obviously, Figure 10 illustrates a straightforward and simple parallel scheme. If we are concerned with shared resources or possible conflicts the Petri Net representations become more complex, but are still representable without deadlock. Of more importance is the case of concurrent tasks which need to communicate or synchronize with each other. In Ada this process occurs through the rendezvous. Figure 11 illustrates how synchronization can be achieved by rendezvous.

The Petri Net illustrates that Task B does not want to fire its $t_i$ and that task A does not want to fire its $t_2$ until task B has fired its $t_2$. If Task B reaches $P_{1b}$ before task A has fired $t_1$, task B will wait for task A. Task B detects that task A has fired $t_1$ by the presence of a token in $P_{3b}$. Looking at task B as a server task and task A as the client task, the transitions have the following meanings:

A $t_i$ = A requests service from B (A is suspended)

B $t_i$ = B accepts A's request for service (rendezvous is initiated)

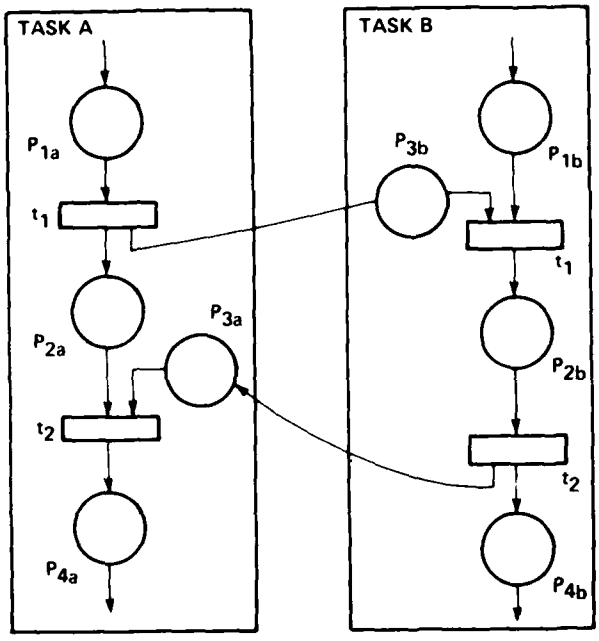



*Figure 11. The Ada Rendezvous Mechanism.*

B $t_2$ = B finishes performing the service for A (rendezvous terminated)

A $t_2$ = A resumes operation after its rendezvous with B

This has been a quick and simplistic look at the modeling capability of Petri Nets. More detailed information is included in citations 15, 12, 5, 24, 25 in the bibliography.

The second description technique presented here is the method proposed by R. J. A. Buhr in his book *Systems Design with Ada*. His objective was to provide a design-oriented introduction to Ada and to present a useful, graphical design notation. His methodology is intended to be:

1. an aid to conceptualizing the organization of a system in Ada terms
2. an approach to communicating design approaches and decisions
3. a basis for computer-aided design of systems, using Ada as the specification and/or implementation language.

The description techniques presented also include concurrency representations and cover solutions to the basic problems of material exclusion, synchronization, scheduling and deadlock. Only superficial knowledge of Ada is required to understand this brief introduction to the pictorial conventions concerned with concurrency. However, one should be somewhat familiar with the concept of an Ada task and its rendezvous mechanism.

Buhr presents a set of notations for use in representing Ada concurrency, shown here in Figure 12.

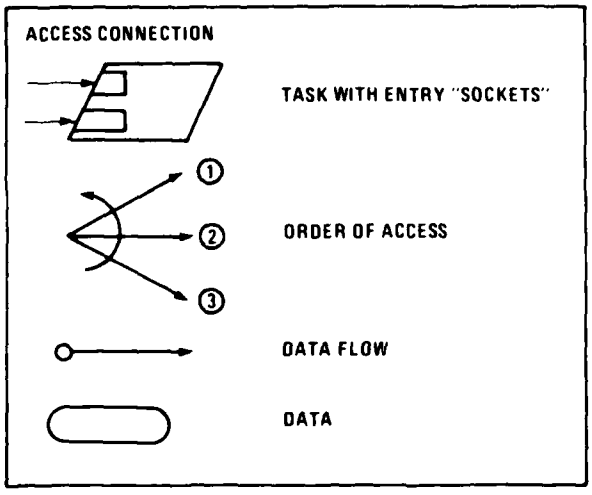



*Figure 12. Basic Pictorial Conventions for Concurrency Features.*

As shown, pockets of tasks, known as entries in Ada, behave like any other interface functionally, but have exclusivity and timing concerns. The rendezvous mechanism requires the calling task to meet with the accepting task, then wait while the accepting task services the call. If the accepting task is busy, then it cannot accept a new call. The new callers are then placed in a queue associated with that particular entry, which ensures mutually exclusive processing of entry calls from different tasks. The basic symbols given in Figure 12 are not sufficient for all purposes and Buhr recommends additional symbols (as shown in Figure 13).

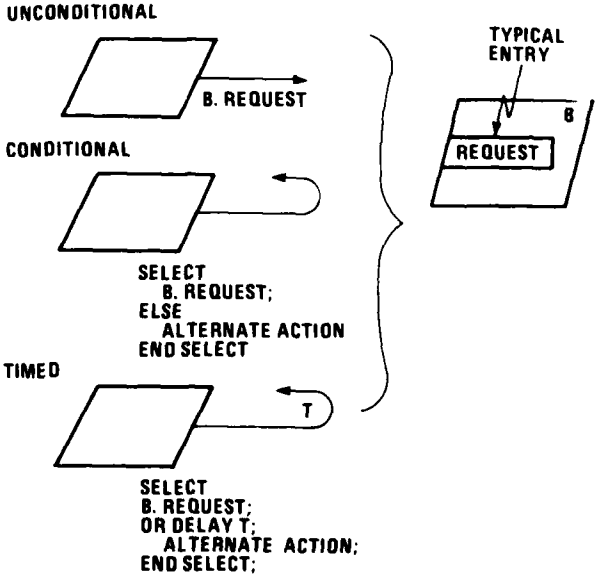



*Figure 13. Structure Graph Symbols Expanded for Various Types of Entry Calls.*

To further expand the symbology, entries that are accepted in a particular order are illustrated as in Figure 14.

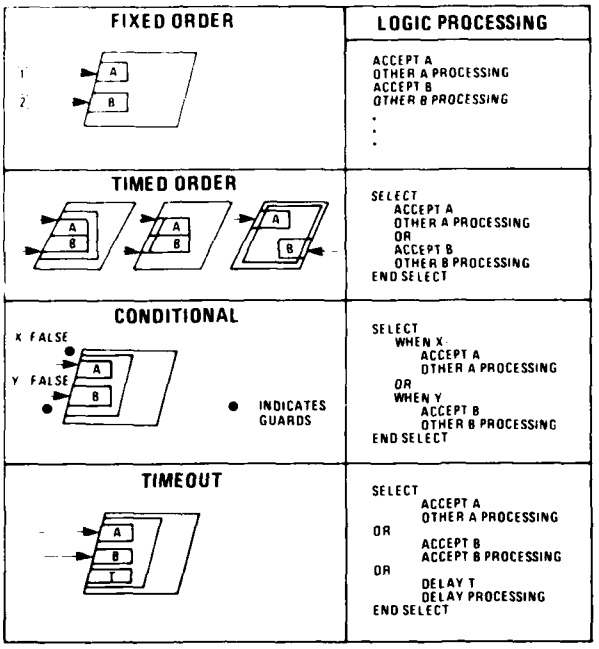



*Figure 14. Fixed, Timed, Conditional, and Timeout Conditional Entries.*

To illustrate the nature of intertask communication and the use of the rendezvous mechanism, consider as an example a simple buffer (Figure 15). There is a consumer task, a buffer task and a producer task. The buffer task has entries for read and write to be used by the consumer and producer.

The symbols indicate the flow of data, the entries and mutual exclusion of the two actions (read/write). To illustrate the rendezvous mechanism, Buhr diagrams provide the direction passed between the tasks as illustrated.

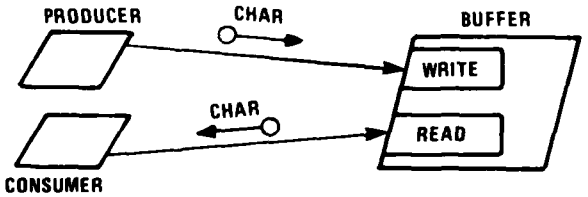



*Figure 15. A Basic Buffer With Producer and Consumer Tasks Shown Pictorially.*

To further illustrate the grace and simplicity of Buhr diagrams, we have selected as an example the interaction between tasks for a disk manager function. Using the symbology presented allows one to pictorially illustrate the concurrency of the system. Figure 16 illustrates the disk manager function using Buhr Diagrams. In this particular system, there are six major tasks with communications occurring between the two major tasks, Disk start and Disk complete.

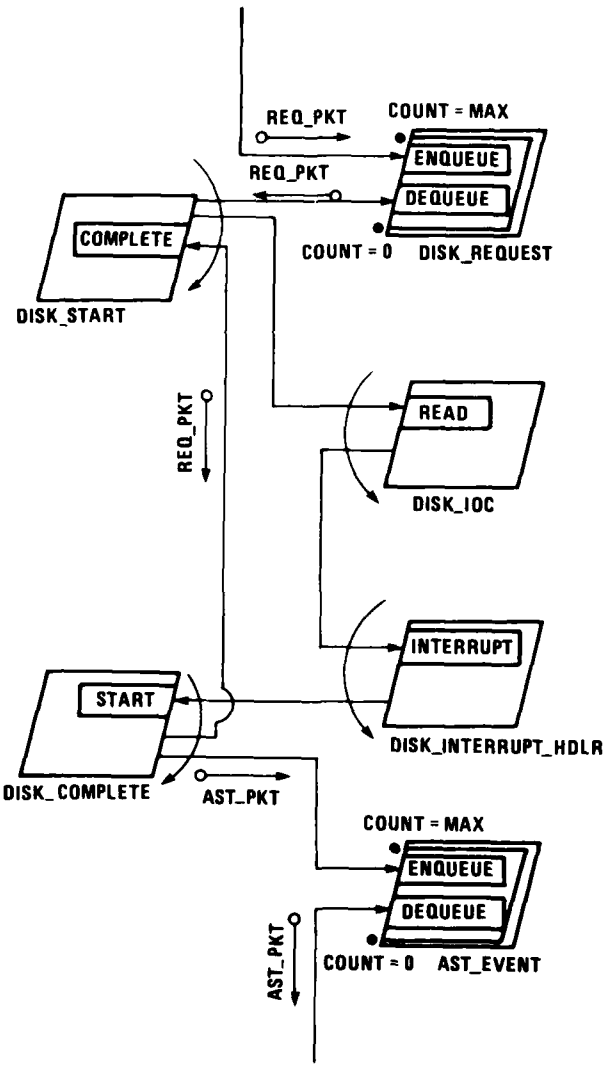



*Figure 16. The Disk Manager Function Illustrated Using Buhr Diagrams.*

Obviously we have only touched the surface of Buhr's methodology or symbology here, with the intent of avoiding too much detail too early. Detailed analysis of the methods are given in the following sections when Ada is formally introduced via examples.

The third methodology, examined in detail, is the state diagram approach. This type of approach is useful when entity tapes, variables or programs can be thought of as being in a given number of states. It has been used in the design of control program mechanisms, systems software and network protocols. The basis for this approach is that a finite-state machine is a hypothetical mechanism that can be in one of a discrete number of conditions or states. Events may cause it

to change its state. In this manner, a process can be represented as a collection of finite-state machines. This gives a precise way to conceptualize and draw complex processes and to check that all possible state transitions have been reconciled.

State transition diagrams are used to represent the behavior of finite-state machines. A finite-state machine is thought of as a black box that can be in one of a number of possible states as illustrated in Figure 17.

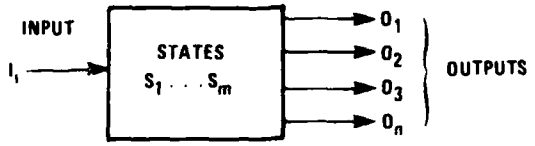



*Figure 17. A Finite-State Machine With Finite Inputs, Outputs and States.*

The finite set of input types is limited by allowing only one input to be active at any time. This is usually accomplished through a queuing mechanism which allows inputs to be handled one at a time. The state of the machine is a static discrete variable which can change only at the instant when an input is received. Algebraically we could represent the state machines behavior as:

$S(t_i) = f_1(S(t_i), I(t_i))$

$O(t_i) = f_2(S(t_i), I(t_i))$

where

$t_i$ = time that an input is received

$s(t_i)$ = state of the machine at $t_i$

$I(t_i)$ = input at time $t_i$

$f_1$ = function which dictates behavior of machine as a response to $S(t_i)$, $I(t_i)$

$S(t_{i+1})$ = state of the machine for $t_{i+1}$ or next state

$O(t_i)$ = function of $S(t_i)$, $I(t_i)$, dictating the output at time $t_i$

These two functions define a state machine. Using circles to represent the states and arrows as the transition mechanisms, outputs can be clearly illustrated as shown in Figure 18.

The arrows between the circles showing the state transitions are labeled with the input that stimulates the transition and the resulting output. Some inputs cause no state change as is illustrated by state 5 receiving an $I_1$ input. Sometimes more than one input can cause the same transition, as between states 3 and 4. Obviously the drawing of Figure 18 is useful for representing machine behavior, but it does leave some possibilities unanswered, such as what occurs when a state receives an input that has no transition arrow? In order to complete the description of a system in this manner, a state transition matrix can be drawn as shown in Table I.

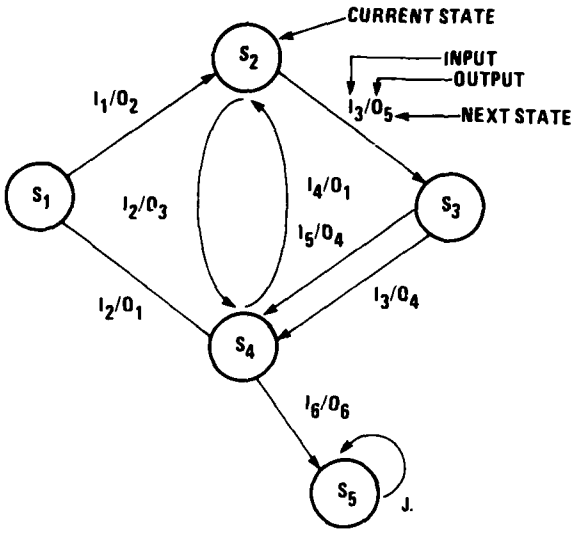



*Figure 18. Example Of A Finite-State Machine Transition Diagram Illustrating Outputs and State Changes.*

TABLE I. STATE TRANSITION MATRIX WHICH CONTAINS THE INFORMATION ILLUSTRATED BY THE STATE DIAGRAM

| INPUT | STATE | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | NEXT STATE |
| $I_1$ | $2/O_2$ | – | – | – | 5/ | |
| $I_2$ | – | $4/O_3$ | – | $1/O_1$ | – | OUTPUT (IF ANY) |
| $I_3$ | – | $3/O_5$ | $4/O_4$ | – | – | |
| $I_4$ | – | – | – | $2/O_1$ | – | |
| $I_5$ | – | – | $4/O_4$ | – | – | |
| $I_6$ | – | – | – | $5/O_6$ | – | |

Applying these state mechanisms is rather straightforward if one considers the various concurrent actions as states with the transitions occurring as needed. As the concurrency becomes more complex, however, or as the transitions among a number of states become large, these diagrams become cluttered and confusing. A possible approach in this case is to use the so-called fence diagrams, where states are shown as vertical bars and transitions are horizontal lines linked to the states by arrowheads. Another possible reason for avoiding these charts is that they appear to be data flow diagrams, which can confuse their true meaning. However, they are useful in showing the multiple states possible for entity

types in data base systems. They are also useful for illustrating the behavior of systems with multiple inputs, complex processing and synchronization requirements. They therefore have their place in representing computer program design, although fine tradeoffs would be required to determine their overall usefulness in representing complex concurrent events found in real-time systems.

The sections following include a description called Ada Program Design Language (PDL) representation. PDLs present an elegant possibility in the consideration of design representations. They have certainly taken a step forward within the software development environment presently and are becoming increasingly popular as design representation tools and analyzers. The reasons for the popularity of Ada-based PDLs include:

1. the power of the Ada programming language is utilized in the design process
2. communication is enhanced by using the same language notation throughout the life cycle
3. various levels of design detail can be represented and focused upon
4. a mechanism is provided for supporting the transition of Ada based software engineering practice.

Using the work accomplished by the IEEE working group in this area as the major reference [26, 27] allows us to summarize Ada's power as a PDL: "Ada provides constructs which support modularity, abstraction, information hiding, concurrent processing, generics, exceptions, strong typing, and data description. These are many of the features required as a design language." The PDL examples illustrated in this paper comply with Ada syntax and semantics, and are used to illustrate the essence of the method and the robustness offered by Ada.

Other possible representation approaches include Object Oriented Design, Flow Charts, Structured Analysis Concepts (data flow design, data structure design), Functional Decomposition, and Programming Calculus.

Many claims have been made about the different strategies for designing software. For functional decomposition, the proponents have largely said "it is a good design, for sure." For data flow design methods, they have said "this design is better than yours. Let me tell you why." For data structure design methods, the claim is that "mine is right, the others are wrong." In the programming calculus, the contention is that "Program A is probably correct, and the others are unproven." All of this leaves an area for innovation in the area of program design. If one restricts the design methodology to real-time design, the above methodologies fall even harder. The current state of the art was represented schematically by Johnson in the form of Figure 19.

The design of real-time systems for the future will place some requirements on a complete methodology, not just the design representation chosen. These demands could include:

1. a rational procedure for partitioning and modeling the problem

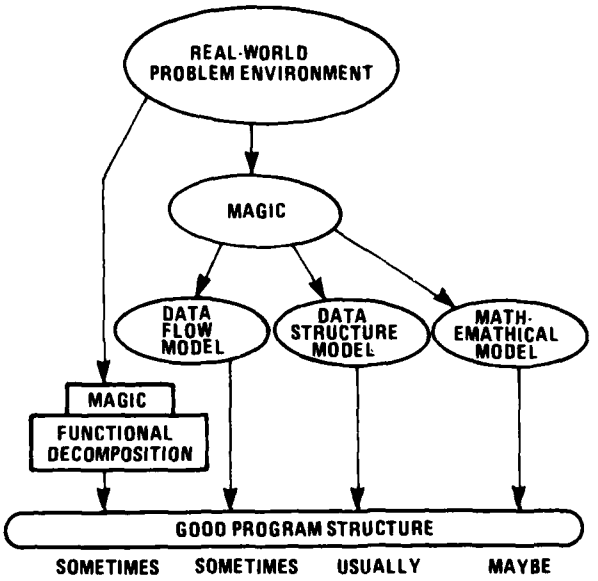



*Figure 19. Current State of the Art in Using Design Strategies.*

2. consistent designs as a resultant output (not dependent on personnel)
3. accommodation for partitioning of the design process
4. correctness of individual design steps guaranteeing the correctness of the final combination
5. opportunity for innovation in the algorithmic stage, but controlled during the entire design process.

The only certainty in all of the literature on design is that we agree that we are not there yet. There is still too much magic involved in the design of software, whether it be real time or not.

## III. Ada's Approach To Parallel Programming

In order to write programs that are concurrent or parallel in nature, the programmer needs a way of specifying which processes are conceptually concurrent. In Ada, the language construct used to group such sequences of actions is known as a task. Tasks are entities that operate in parallel. There is concurrent execution of two or more threads of control. However, the concurrency may be *actual*, as in the case of systems configurations that utilize multiple processors, or *apparent*, as in a multiprogramming environment with interleaved execution or a single processor. Declaring a unit to be a task provides only logical concurrency; the language cannot assure any assumptions about the efficiency or execution speed of these tasks. The operating system in conjunction with the routine support system has the responsibility for scheduling different tasks and for allocating any resources they may need.

In Ada, tasks represent independent program units that can execute at their own pace and are essentially isolated unless the programmer specifies explicit synchronization points.

The synchronization point serves as a place where one of the affected tasks can "wait" for the other task process to rendezvous with it. This synchronization mechanism delineates the code which is to be executed while the tasks are attempting synchronization. In particular, most tasks can be broadly categorized as either servers or requesters. Usually, requesters are the active elements of a system of cooperative tasks. They are servers to accomplish certain defined functions. Servers are generally passive, reacting only to the external requests generated by requesters. The Ada tasking mechanism permits the user to define server/requester relationships clearly and concisely.

Tasks have three main purposes:

1. they may model or control co-existing objects in the problem domain

2. they may serve as controlling or synchronizing agents, providing the effects of semaphores and locks, buffers, monitors, schedulers, controlling access to shared resources, or synchronizing the actions of otherwise independent tasks

3. they may serve to define concurrent algorithms for more efficient execution in a multiprocessor architecture.

In Ada, tasks call upon the programmer to decompose a problem into a manageable group of independent threads of control. Without any regard for the physical representation, the abstraction of a solution with many tasks is a natural one, drawn directly from our understanding of the problem space. The key in using tasks is being able to represent real-time or real-world parallel activities within a level of our solution space. This is not a minor step, for many programmers/designers are very comfortable with sequential solutions to problems, but are not at ease with concurrent solutions to problems that could utilize concurrent actions. As a matter of fact, we could state more directly that the art of designing parallel programs is underdeveloped because we do not understand or perceive parallelisms clearly and we have little knowledge to communicate about these designs.

The major topics examined in this section include task specifications, rendezvous mechanisms, control mechanisms for the rendezvous, and task types and families.

This section assumes that the reader is somewhat acquainted with Ada and its basic constructs. Since tasks are a unique feature of Ada and unlike any other feature found in high-level languages, they merit some examination and explanation.

## Task Specifications

Tasks are program units that may operate in parallel with other program units. A task specification, like a package specification, defines the interface which other related program components use to interact with the task. The interface consists of entry declarations that are similar to the subprogram declarations in a package specification. Some simple examples of an Ada Task Specification are illustrated below.

```
task Buffer is  -- task with entries
        entry Put (in_buffer: in message) ;
        entry Take (from_buffer: out message) ;
end Buffer ;

--              or

task Buffer is  -- task without entries
end Buffer ;

--              or

task Buffer ;   -- task without entries
```

As illustrated above, some tasks have entries. An entry of a task can be called by other tasks. A task accepts a call of one of its entries by executing an accept statement for the entry. Synchronization is achieved by the rendezvous mechanism and is discussed in the next section. The model offered by Ada is based on Hoare's Communicating Sequential Processes, in which parallel processes synchronize and communicate by means of input and output statements.

## The Ada Rendezvous Mechanism

Synchronization between two tasks occurs when the task issuing an entry call and the task accepting an entry call establish a rendezvous. The two tasks communicate with each other during the rendezvous. Entries are also the primary means of communicating between tasks. To illustrate this mechanism, an example is given which is based on a creator/server of messages. The creator writes the messages and provides it to the server, who transmits it to some other location. Since the creator is providing the service, it must have an entry into its process by which it can accept a message. This can be accomplished as follows:

```
task Creator is

        entry Transmitor (m: in message) ;

end Creator ;
```

The task specification establishes the interface to the services provided by the task body. The entry declaration is much like a procedural declaration. It has the same format and may have in, out, and in/out parameters. The corresponding task body defines the processing to be done by task.

including an accept to fulfill the entry specification. The body for the above task specification is given by:

```
task body Creator is
        output_message : message ;

        procedure Transmit (any_message: in message) is
                null ;
        end Transmit ;

begin
    loop
        accept Transmitor (m: in message) do
                output_message := m ;
        end Transmitor ;
        transmit (output_message) ;
    end loop ;
end Creator ;

-- The specification and body of the Server task is given below.

task Server ;

task body Server is
        text : message ;

        procedure write (text_out: out message) is
                null ;
        end write ;

begin
    loop
        write (text) ;
        Creator.Transmitor (text) ;
    end loop ;
end Server ;
```

The call in the above code is given by

creator.transmitor (text);

and looks like a procedure call. However, the major difference is that server and creator are operating in parallel. This implies that the rendezvous does not occur until a task is suspended. It will then wait for the called task to reach the accept statement. If the task providing the entry reaches the accept statement first, it waits until the entry is called.

When both conditions have been satisfied, the tasks are synchronized and the information is passed via the parameter list. In summary, the rendezvous brings together what had been two independent threads of control into a single synchronized thread of control. In this manner, the Ada rendezvous becomes the mechanism for task coordination and for sharing information. It should also be noted that the rendezvous mechanism has an asymmetric nature, since:

1. the calling task must know the name of the accepting task as well as the specification of the entry point
2. the task providing the entries and accepts is essentially passive; it provides a service to any task that knows how to call it
3. the accepting task does not know the name of the caller
4. a task providing entries may have a number of tasks queued waiting for service at a number of different entry points.

This asymmetry allows us to distinguish between active and passive tasks. Passive tasks provide services through entries and accepts. Active tasks use the services provided by issuing entry calls. These active tasks are similar to application tasks that may use the services provided by a real-time operating system. Passive tasks have the characteristics of operating systems and require considerable skill to design and to implement.

## Control Mechanisms For The Rendezvous

Each of the two types of tasks illustrated previously (the calling task and the called task) has a mechanism for controlling the rendezvous. For the called task, it is the selective wait, and for the calling task it is the conditional and timed entry calls. Each of these mechanisms will be described briefly.

The selective wait statement is very useful when it is necessary to react to externally changing conditions that make it necessary to accept entry calls in an arbitrary order. The selective wait statement permits the programmer to define several accept alternative actions to be selected. Since the selected wait contains accept statements, it may appear only in the body of a task. An example of a selective wait is given below for the example illustrated previously

```
task body New_Message is
        receptacle : message ;
        new_message : boolean := false ;

begin
    loop
        select
                accept Transmitor (m: in message) do
                        null ;
                end Transmitor ;
                new_message := true ;
        or
                when new_message =>
                        accept Take (m: out message) do
                                null ;
                        end Take ;
                        new_message := false ;
        end select ;
    end loop ;
end New_Message ;
```

The when clause stipulates that the condition new message must be true for the take entry to be available for rendezvous. The operations on new message ensure the correct ordering of rendezvous.

The calling task has essentially two mechanisms to allow it to control the conditions under which a rendezvous may occur. One control mechanism is to issue an entry call only if a rendezvous is immediately available. This is the Ada conditional entry call. The following code illustrates the above concept:

```
select
        message_receiver.transmit (text) ;
                .
                .       optional sequence of statements
                .
else
                .
                .       do alternative action
                .       (could be null)
end select ;
```

The result is that the rendezvous will occur only if no other entry calls are queued for message receiver transmit. If the rendezvous cannot take place, the alternative action is executed.

The second control mechanism allows the calling task to enter the queue for an entry. If the rendezvous does not occur within a specified time, the calling task leaves the queue and continues execution. This is the Ada timed entry call. For example,

```
select
        message_receiver.transmit (text) ;
                .
                .       optional sequence of statements
                .
else
        delay 10.0      -- delay 10.0 seconds
                .
                .       alternative optional sequence of
                .       statements
end select ;
```

If the rendezvous occurs within 10.0 seconds, the rendezvous task will participate in the rendezvous, execute the optional sequence of statements, and then exit the select statement. If no rendezvous occurs within 10.0 seconds, the alternative optional sequence of statements will be executed.

### Task Types and Families

Ada allows the definition of task types for declaring multiple tasks of similar nature, in the same manner as generics are allowed for subprograms and packages. It is also possible to have a family of entries in which each entry of the family is to accomplish a similar function. Task types facilitate the declaration of similar tasks, since several tasks can be declared collectively in an array or individually. The declaration of a task type is syntactically similar to the declaration of a task, the only difference being the presence of the keyword type in the task specification. For example:

```
task type Do_something is
        entry Do_this ;
        entry Do_that ;
        entry Do_everything ;
end Do_something ;
```

The declaration

DS1, DS2: Dosomething;

declares that two tasks become active just prior to execution of the first statement of the subprogram or package in which they are declared. Arrays where elements are tasks are declared just like arrays with other types of elements for each element of array DS declared as

DS: array (index) of Dosomething;

is a task.

Task types are also like limited private types. Objects of task types are constants and cannot be assigned to or compared for equality. Tasks can be passed as parameters; the actual parameter and the corresponding formal parameter designate the same task for all parameter modes. If an application needs to create tasks dynamically, then access types must be used. For example, consider access type *IndexDoSomething* is access Dosomething; and variable AnotherIndex declared as

AnotherIndex:Dosomething;

A task can also be created dynamically by calling the allocator as illustrated by the statement

AnotherIndex:=new Dosomething;

Allocated tasks become active when allocated, and must have terminated or be ready to terminate when the scope of the block, subprogram or task in which the access type is declared is about to be vacated. Otherwise, Ada prevents vacating the scope section.

We have only briefly described tasks and some of their more important features. It is obvious that the subject of tasks and their potential use is a complicated matter. However, despite their complexity, tasks are an important and necessary concept. The control of concurrent processes is a necessity in real-time systems and Ada provides this feature at a high level of abstraction within the higher level language.

## IV. Illustrative Examples of Parallel Program Representations Using Ada

The ability to represent parallel or concurrent designs is essential in communicating a designer's approach to solving the real time systems problems of today. A sound knowledge of Ada's tasking programming constructs will enable designers to cope with real-time systems at the coding level. However, the understanding of design representation is enhanced through pictorial representations when possible. This section does not propose to answer all the questions about Ada and concurrent processing, but does make some pointed recommendations about design representations and Ada. The key to success in this area is the ability to produce Ada designs that are understandable to a broad spectrum of interested parties. Eventually a concerned organization will derive its own specialized or tailored graphical notation for representing concurrent designs using Ada.

This paper includes four illustrative examples of parallel programming representations. These examples will be described using each type of representation: the Petri Net, the Buhr Diagram, PDL, and the Flow Chart.

There will be annotated code for each example. The representations will be compared with each other through these examples, thus exposing the strengths and weaknesses of each.

It is the belief of the authors that the information derived from state diagrams, namely state transition tables, can be derived from Petri Nets. This is accomplished by placing tokens in the nodes of Petri Nets for every configuration of the Petri Net, firing the applicable transitions and marking the state transitions that occurred in the state transition table. Therefore, the following examples will not contain state diagrams as a method of representation.

The first example is a simple tasking program introduced by J. G. P. Barnes. The example problem is to consider a family going shopping to buy ingredients for a meal. The Ada code for this example is shown in Figure 20. Suppose they need fish, salad, and wine. Provided there are three people in the family, a simple solution may be implemented. The solution is sequential in the sense that the family must pick up the items (in parallel), agree to meet at a central location (near cashier), then pay for the items. The procedure Pick Up Items is invoked, the three tasks execute in parallel and the procedure cannot return until all the tasks have terminated i.e., all the ingredients have been found and the family has met at the central location.

```
procedure Shopping is

        procedure Pick_Up_Items is

                task Get_Salad ;
                task Get_Wine ;
                task Get_Fish ;

                task body Get_Salad is

                        begin
--                              Find and take salad.
                                null ;

                        end Get_Salad ;

                task body Get_Wine is

                        begin
--                              Find and take wine.
                                null ;

                        end Get_Wine ;

                task body Get_Fish is

                        begin
--                              Find and take fish.
                                null ;

                        end Get_Fish ;

        begin
                null ;

        end Pick_Up_Items ;

        procedure Pay_For_Items is

        begin
--              Give money to cashier
                null ;

        end Pay_For_Items ;

begin
        Pick_Up_Items ;
        Pay_For_Items ;

end Shopping ;
```

*Figure 20. Ada code for the Shopping Program.*

The flowchart for this program, shown in Figure 21, is straightforward. The three subtasks (denoted by the parallel program with the double stripes on the sides) are contained in the procedure Pick Up Items (denoted by square box). Control is passed outside of Pick Up Items when all the subtasks have terminated.

The PDL code for this example problem, shown in Figure 22, illustrates the similarities with straight Ada code.

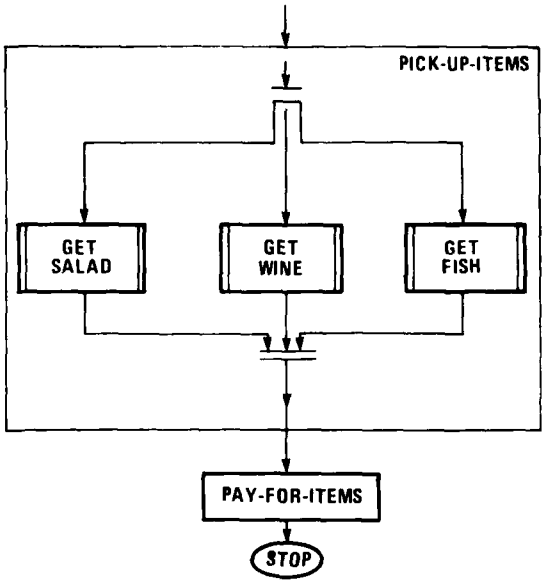



*Figure 21. Flow chart for the Shopping Program.*

```
procedure Shopping is
        procedure Pick_Up_Items is
                task Get_Salad is
                        Find and take salad.
                end Get_Salad
                task Get_Wine is
                        Find and take wine.
                end Get_Wine
                task Get_Fish is
                        Find and take fish.
                end Get_Fish
        end Pick_Up_Items
        procedure Pay_For_Items is
                Give money to cashier.
        end Pay_For_Items
begin
        Pick_Up_Items
        Pay_For_Items
end Shopping
```

*Figure 22. PDL code for the Shopping Program.*

The Petri Net for this example is also rather straightforward, as illustrated in Figure 23. The firing of transition $T_1$ occurs only after the three tasks have terminated. This example is a simple high-level solution to the original problem with no data passed.

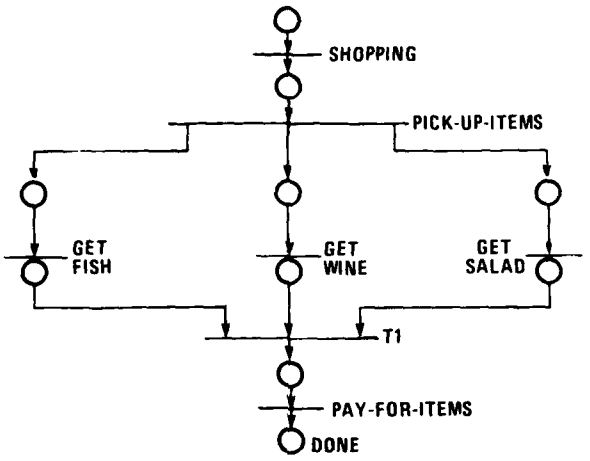



*Figure 23. Petri Net for the Shopping Program.*

The flow chart and Petri Net representations could be sufficient methodologies in this case. However, the Buhr diagram for this case, shown in Figure 24, illustrates the structure of the program and the sequence of procedure calls which adds a level of information not given by the others. Since there is no data being passed, this representation does not have a distinct and noticeable advantage over the other methods.

After reviewing the first example it appears that for a high-level solution or a high abstraction process, flow charts and Petri Nets are sufficient to represent the solutions with respect to the central flow. Since there was no communication between the program elements, the Buhr diagram did not appear to provide any more insight into the program than the other representation. However, representations are needed for lower level solutions and inter-module communication, which need to be included in the representation. This will be shown in the next example, which takes us one step lower into the design.

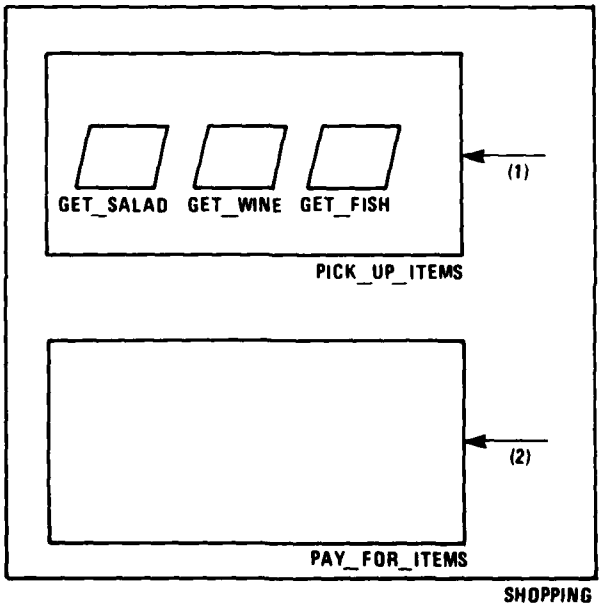



*Figure 24. Buhr Diagram for the Shopping Program.*

The second example is taken from the book *Studies in Ada Style*[28]. The program is a console drive for a PDP-11 and is discussed in detail in the reference. In short, the functions performed by this program are buffering of requests for the device, ensuring the integrity and validity of these requests and fielding interrupts from the hardware.

A central problem in this example is implementing a synchronous process with the synchronous mechanism of rendezvous provided by Ada. This is accomplished using three explicit tasks that monitor requests from the program, interrupts from the input and output devices. All three tasks communicate through shared queues as illustrated in the Ada coding example in Figure 25.

The flow chart for the terminal driver is shown in Figure 26. This example shows that several Ada semantic constructs were not included in this representation, such as select statements, exceptions, and terminations. The select and termination constructs were added with little effort, with major problems occurring with the exception constructs. The data structures in this example (I/O buffer, hardware devices, characters) cannot be represented in a reasonable fashion because of the inability of flowcharts to graphically show data structures and data flow. The flow chart is also weak in depicting the relationship between the interrupt handler and the device driver.

The PDL code for this problem (Figure 27) is considerably simpler than the equivalent Ada code structure.

The Petri Net for this example (Figure 28) contains all the major Ada tasking semantic constructs to model this program properly. The notations used conform to the methodology representations offered by G. Cherry's *Parallel Programming in ANSI Standard Ada*[5], and appear to be adequate for this design. The weakness of the Petri Net representation is its inability to relay data transfers among design elements.

The Buhr diagrams now tend to give more useful information graphically to enhance communications. For example, the Buhr diagram for the terminal drive example, shown in Figure 29, shows the structure of the terminal driver package and its subtasks. The data flow into and out of the package is depicted as well as the data exchanged between the subtasks. The Buhr diagram clearly illustrates Device Reader task receiving the input character from the keyboard hardware through its interrupt handler. The interrupt handler then deposits this character into the input buffer. The Read Character entry for the device driver task can then return the input character from the input buffer to process the call for this entry. Similarly the Device Writer is also clearly annotated. This example is not overly complicated and the Buhr diagram fits well on one page. However, for more complicated examples, Buhr diagrams begin to crowd the page and lose their readability. This example shows relatively easily the advantage Buhr diagrams have over other representation methods: namely, the control flow and the

```
package Terminal_Driver_Package is

        task Terminal_Driver is
                entry Read_Character(C : out Character) ;
                entry Write_Character(C : out Character) ;
                entry Reset ;
                entry Shut_Down ;
        end Terminal_Driver ;

end Terminal_Driver_Package ;


with Queue_Package, Low_Level_IO ;
use Low_Level_IO ;


package body Terminal_Driver_Package is

        task body Terminal_Driver is

                -- Group all of the machine dependent constants together

                Console_Input_Vector : constant := 8#60# ;
                Console_Output_Vector : constant := 8#64# ;
                Enable_Interrupts : Integer := 8#100# ;
                Write_Time_Out : constant Duration := 0.5 ;
                Number_of_Lines : constant := 2 ;
                Line_Length : constant := 132 ;

                task type Device_Reader is
                        entry Interrupt ;
                        entry Start_Up_Done ;
                        for Interrupt use at Console_Input_Vector ;
                end Device_Reader ;

                task type Device_Writer is
                        entry Interrupt ;
                        entry Start_Up_Done ;
                        for Interrupt use at Console_Output_Vector ;
                end Device_Writer ;

                package Char_Queue_Package is new Queue_Package(Character) ;
                use Char_Queue_Package ;

                type Driver_State_Block is
                        record
                                Input_Char_Buffer, Output_Char_Buffer :
                                  Blocking_Queue(Number_of_Lines*Line_Length);
                                Cur_Reader : Device_Reader ;
                                Cur_Writer : Device_Writer ;
                        end record ;

                type Ref_to_Block is access Driver_State_Block ;
                Cur_State : Ref_to_Block ;

                task body Device_Reader is
                        temp_input : Character ;
                begin
                        accept Start_Up_Done ;
                        Send_Control(Console_Keyboard_Control,
                                                Enable_Interrupts) ;
                loop
                        accept Interrupt do
                                Receive_Control(Console_Keyboard_Data,
                                                Temp_Input) ;
                        end Interrupt ;
                        Append(Cur_State.Input.Char_Buffer,
                                                        Temp_Input);
                end loop ;
        end Device_Reader ;

        task body Device_Writer is
                temp_output : Character ;
        begin
                accept Start_Up_Done ;
                Send_Control(Console_Printer_Control,
                                                Enable_Interrupts) ;
                accept Interrupt ;
                loop
                        Remove(Cur_State.Output_Char_Buffer,
                                                        Temp_Output) ;
                        Send_Control(Console_Printer_Data,
                                                        Temp_Output) ;
                        select
                                accept Interrupt ;
                        or
                                delay Write_Time_Out ;
                        end select ;
                end loop ;
        end Device_Writer ;
```

*Figure 25. Ada code for the Terminal Driver Package.*

```
                procedure Shut_Down_Old is
                        raise Cur_State.Cur_Reader'FAILURE ;
                        raise Cur_State.Cur_Reader'FAILURE ;
                        Destroy_Queue(Cur_State.Input_Char_Buffer) ;
                        Destroy_Queue(Cur_State.Output_Char_Buffer) ;
                end Shut_Down_Old ;

                procedure Start_Up is
                        Cur_State := new Driver_State_Block ;
                        Init_Queue(Cur_State.Input_Char_Buffer) ;
                        Init_Queue(Cur_State.Output_Char_Buffer) ;
                        Cur_State.Cur_Reader.Start_Up_Done ;
                        Cur_State.Cur_Writer.Start_Up_Done ;
                end Start_Up ;

        begin
                Start_Up ;

                loop
                        select
                                accept Read_Character(C : out Character) do
                                        Remove(Cur_State.Input_Char_Buffer,
                                                                        C) ;
                                end Read_Character ;
                        or
                                accept Write_Character(C : out Character) do
                                        Append(Cur_State.Output_Char_Buffer,
                                                                        C) ;
                                end Write_Character ;

                        or
                                accept Reset do
                                        Shut_Down_Old ;
                                        Start_Up ;
                                end Reset ;
                        or
                                accept Shut_Down ;
                                Shut_Down_Old ;
                                exit ;
                        or
                                terminate ;
                        end select ;
                end loop ;
        exception
                when Terminal_Driver'FAILUER =>
                        Shut_Down_Old ;

        end Terminal_Driver ;

end Terminal_Driver_Package ;
```

*Figure 25. Ada code for the Terminal Driver Package. (Continued)*

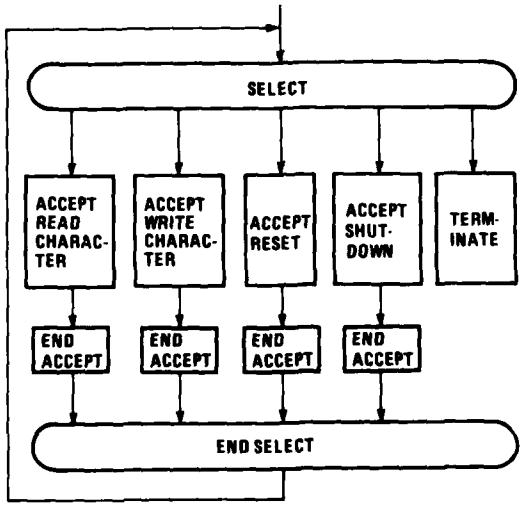



*Figure 26. Flow Chart for the Terminal Driver Package.*

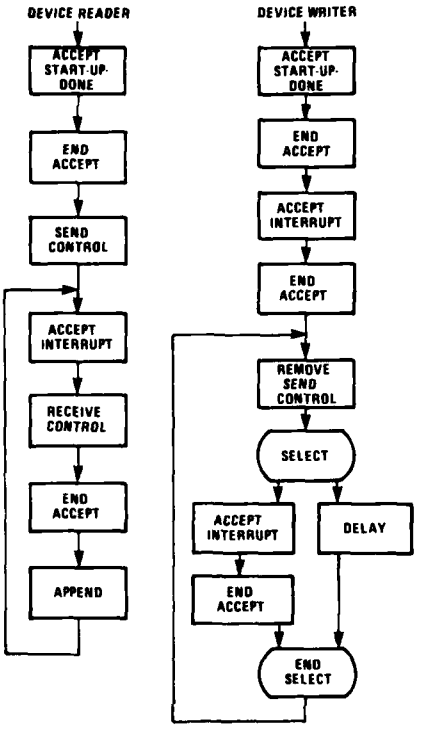



*Figure 26. Flow Chart for the Terminal Driver Package. (Continued)*

```
package Terminal_Driver_Package is

        task Terminal_Driver is

                task Device_Reader is
                        send control sequence to terminal
                        loop
                                wait for interrupt
                                append character to input buffer
                        end loop
                end Device_Reader

                task Device_Writer is
                        send control sequence to printer
                        wait for interrupt
                        loop
                                remove character from output buffer
                                send character to printer
                                select
                                        wait for interrupt
                                or
                                        time out printer
                                end select
                        end loop
                end Device_Writer

                procedure Shut_Down_Old is
                        destroy reader queue
                        destroy writer queue
                end Shut_Down_Old

                procedure Start_Up is
                        create reader queue
                        create writer queue
                end Start_Up

        begin Terminal_Driver

            loop
                select
                        accept Read_Character (c: out character) do
                                remove character from input buffer
                        end Read_Character ;
                or
                        accept Write_Character (c: in character) do
                                append character to output buffer
                        end Write_Character ;
                or
                        accept Reset do
                                Shut_Down_Old
                                Start_Up
                        end Reset
                or
                        accept Shut_Down
                        Shut_Down_Old
                        exit
                end select
            end loop
        end Terminal_Driver
end Terminal_Driver_Package
```

*Figure 27. PDL Code for the Terminal Driver Package.*

data flow are both clearly evident and identified. The notation may be a little rigorous in the sense that all the Ada tasking semantics must be depicted and many notations are therefore needed, but that can be reduced according to need or complexity.

The final example is a scheduling algorithm problem which is again a step up in concurrent complexity. The timing diagram illustrated in Figure 30 depicts the timing requirements of this problem.

In summary form, Module A must communicate with Module B every 20 microseconds. Module C must communicate with Module A every 40 microseconds. One possible solution is to implement a scheduler task which first delays 20 ms and signals Module C to communicate with Module A. The scheduler then delays another 20 ms, signals Module A to communicate with Module B, then signals Module C to communicate with Module A. This process is repeated indefinitely.

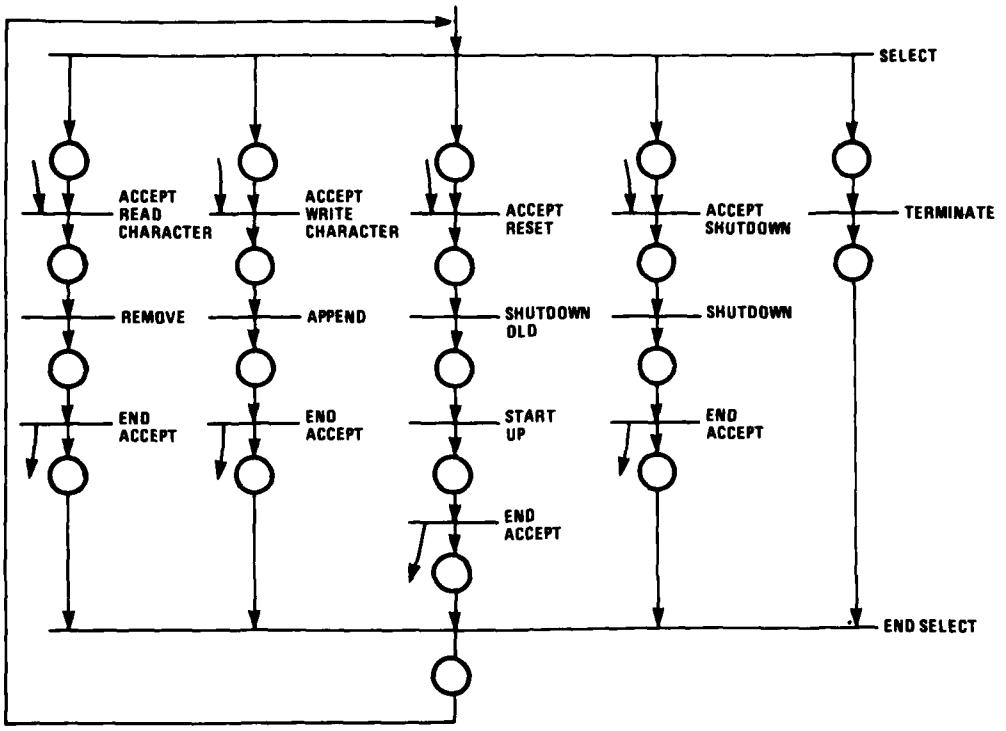



*Figure 28. Petri Net for the Terminal Driver Package.*

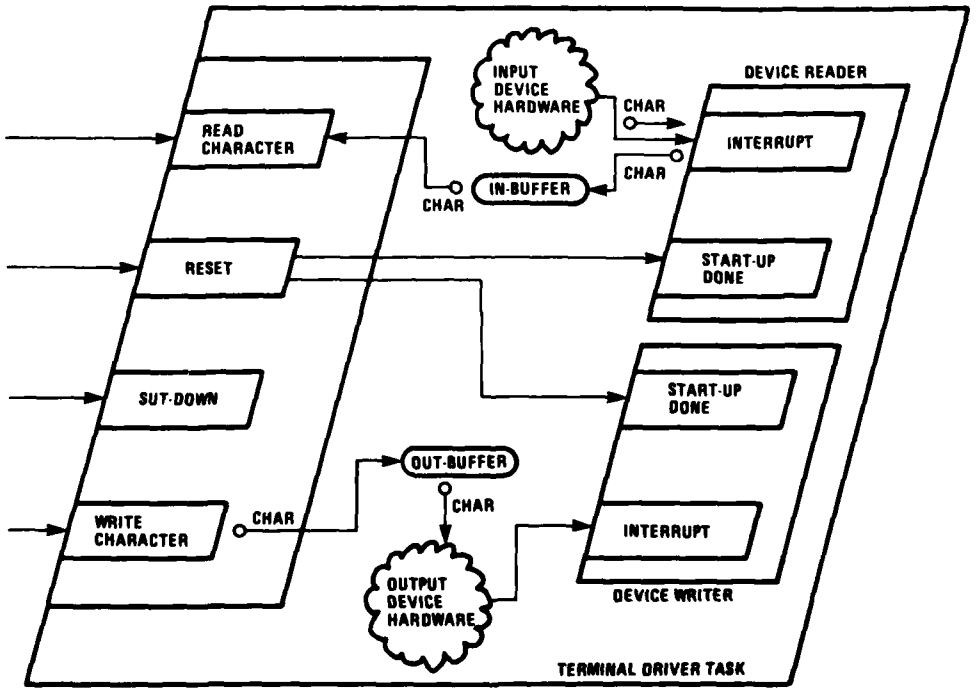



*Figure 29. Buhr Diagram for the Terminal Driver Package.*

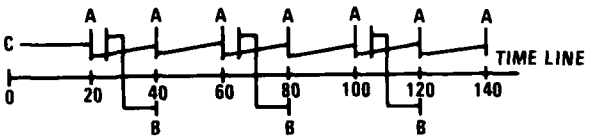



*Figure 30. Timing Diagram for the Scheduling Algorithm.*

The Ada code, flow chart, PDL, Petri Net and Buhr diagram for this solution are shown in Figures 31 through 35.

```
package Init_2 is

        task Scheduler is
                entry Start ;
        end Scheduler ;

        task A is
                entry Signal ;
                entry Comm ;
        end A ;

        task B is
                entry Comm ;
        end B ;

        task C is
                entry Signal ;
                entry Comm ;
        end C ;
```

```
------------------------------
        task body Scheduler is

        begin
                accept Start ;

                loop

--                      delay 20 ms ;
                        C.Signal ;

--                      delay 20 ms ;
                        A.Signal ;
                        C.Signal ;

                end loop ;

        end Scheduler ;
----------------------
        task body A is
        begin
                loop
                        select
                                accept Signal ;
                                B.Comm ;
                        or
                                accept Comm ;
                        end select ;
                end loop ;
        end A ;
----------------------
        task body B is
        begin
                loop
                        accept Comm ;
                end loop ;
        end B ;
----------------------
        task body C is
        begin
                loop
                        accept Signal ;
                        A.Comm ;
                end loop ;
        end C ;
-----
begin
        Scheduler.Start ;
end Init_2 ;
```

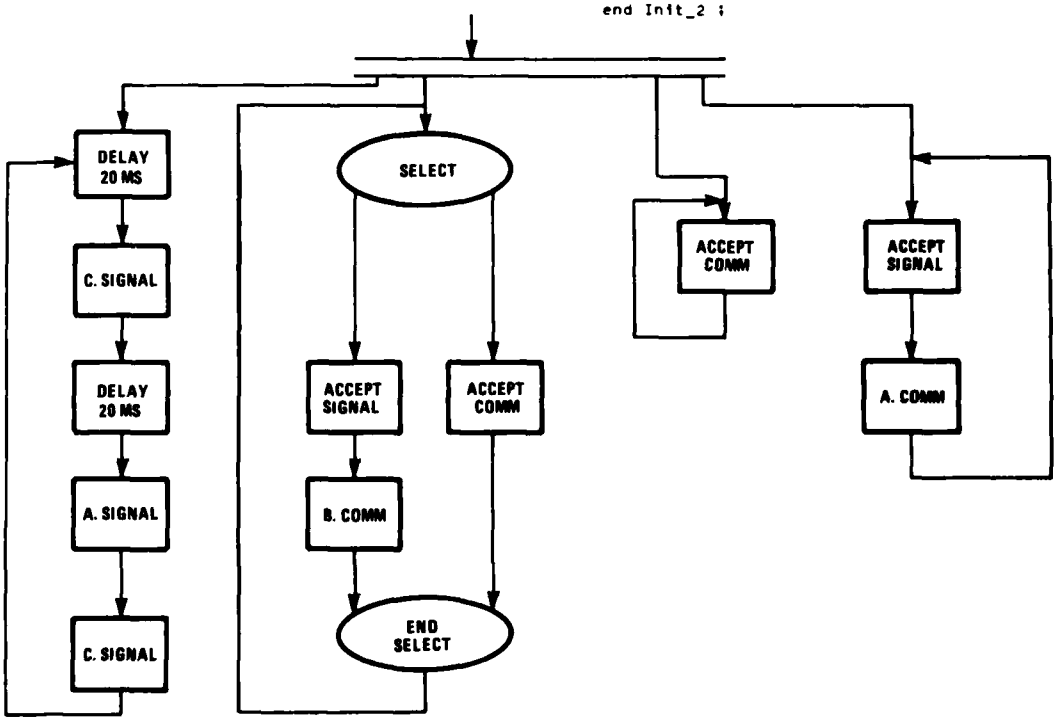



*Figure 32. Flow Chart for the Scheduling Algorithm (1).*

```
package Init_2 is
        task Scheduler is
                accept Start
                loop
                        delay 20 ms
                        C.Signal
                        delay 20 ms
                        A.Signal
                        C.Signal
                end loop
        end Scheduler
        task A is
                loop
                        select
                                accept Signal
                                B.Comm
                        or
                                accept Comm
                        end select
                end loop
        end A
        task B is
                loop
                        accept Comm
                end loop
        end B
        task C is
                loop
                        accept Signal
                        A.Comm
                end loop
        end C
begin
        Scheduler.Start
end Init_2
```

*Figure 33. PDL Code for the Scheduling Algorithm (1).*

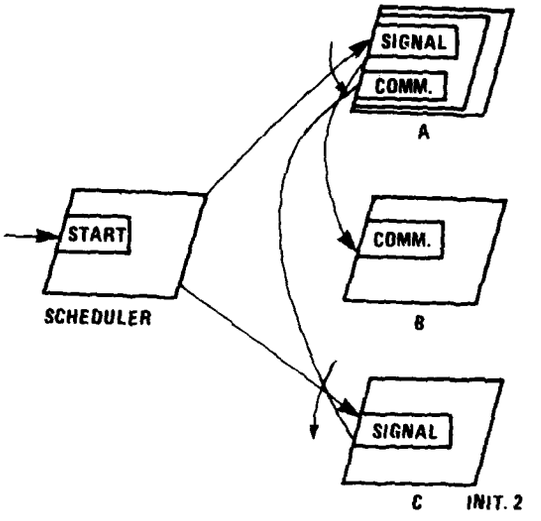



*Figure 35. Buhr Diagram for the Scheduling Algorithm (1).*

Another possible solution is for the modules to implement the scheduling algorithm themselves instead of relying upon the Ada scheduling capability. That is, Module B loops indefinitely, first delaying 20 ms, then making an entry call to communicate with Module A. Module A is also looping indefinitely on a select statement. One alternative of the select is to delay 40 ms, then communicate with Module B. The other alternative is to accept communication entry calls from Module C. This solution is workable so long as the delay does not

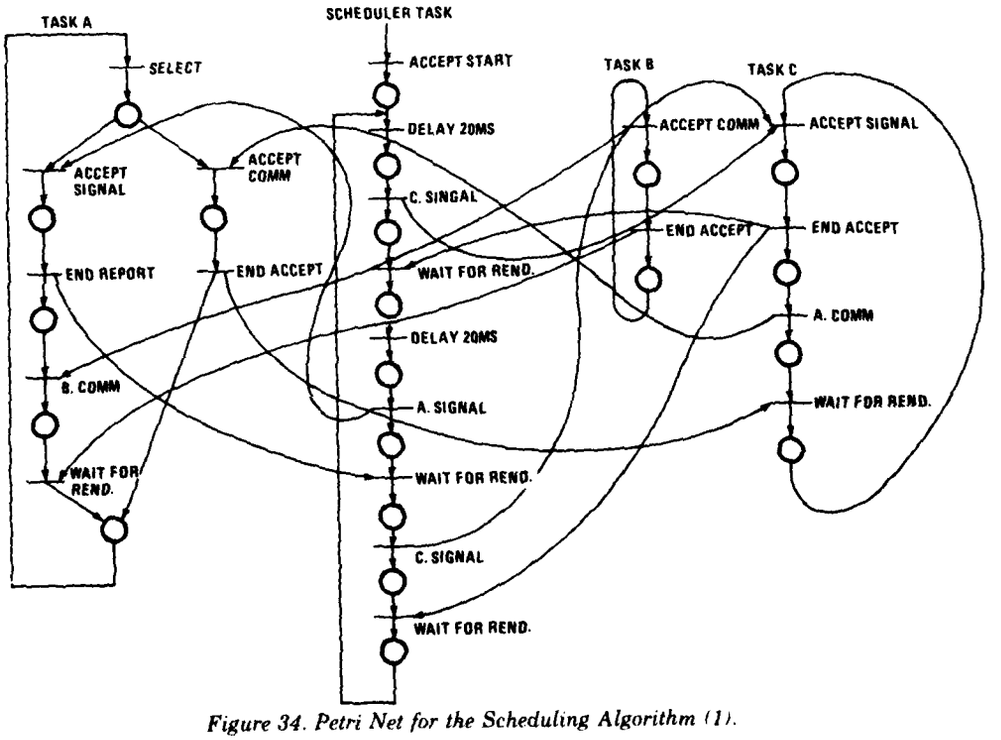



*Figure 34. Petri Net for the Scheduling Algorithm (1).*

exceed the delay duration specified. However, this is not currently guaranteed by Ada. The Ada code, flow chart, PDL, Petri Net and Buhr diagram for this solution are shown in Figures 36 through 40.

The first solution suffers from the same delay statement problem; that is, it suffers from the fact that after a rendezvous has occurred in Ada, the task to execute next is not specified.

In summary, Buhr diagrams are particularly useful in describing tasks and their interventions. Data flow between tasks (through the Ada rendezvous) is included in this pictorial notation as well as control flow. This basic set of notations is sufficient for describing the full set of Ada tasking semantics with a few exceptions such as task termination and dynamic task creation.

```
package Init_3 is

        task A is
                entry Comm ;
        end A ;

        task B is
                entry Comm ;
        end B ;

        task C ;

----------------------
        task body A is
        begin
                loop
                        select
--                              delay 40 ms ;
                                B.Comm ;
                        or
                                accept Comm ;
                        end select ;
                end loop ;
        end A ;

----------------------
        task body B is
        begin
                loop
                        accept Comm ;
                end loop ;
        end B ;

----------------------
        task body C is
        begin
                loop
--                      delay 20 ms ;
                        A.Comm ;
                end loop ;
        end C ;

-----
begin
        null ;
end Init_3 ;
```

*Figure 36. Ada Code for the Scheduling Algorithm (2).*

However, Buhr points out in his book that procedural access is only through packages. This restriction must be removed in order for Buhr diagrams to handle tasks in subunits. This is the case when a server task is needed that loops indefinitely, accepting entry calls to its service routines.

Buhr diagrams can be used throughout the life cycle of software, starting from top-level design to unit coding. The fact that manual updating of Buhr diagrams may be tedious is forcing the issue of automated Buhr diagram processing. At this time Buhr is doing extensive research in this area and is expected to have a product for accomplishing some automation of his notation.

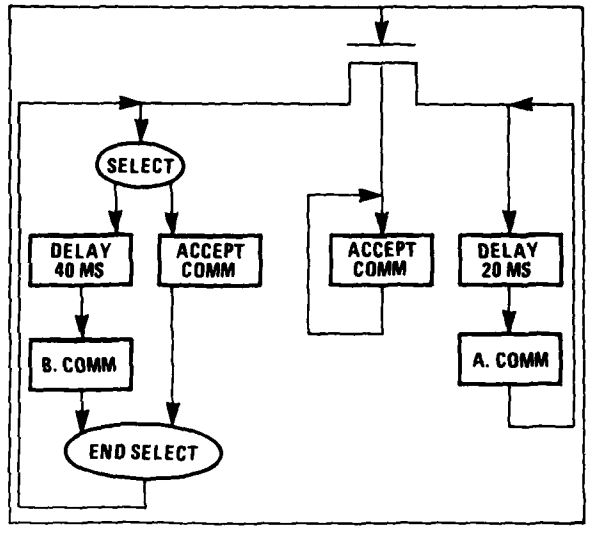



*Figure 37. Flow Chart for the Scheduling Algorithm (2).*

```
package Init_3 is

        task A is
        begin
                loop
                        select
                                delay 40 ms
                                B.Comm
                        or
                                accept Comm
                        end select
                end loop
        end A

        task B is
        begin
                loop
                        accept Comm
                end loop
        end B

        task C is
        begin
                loop
                        delay 20 ms
                        A.Comm
                end loop
        end C

end Init_3
```

*Figure 38. PDL Code for the Scheduling Algorithm (2).*

## V. Conclusions

Concurrent or real-time systems are, by definition, systems whose proper functioning is dependent upon time-critical events. High-level language implementations that exhibit large overheads will usually make the language unsuitable for real-time programming. Many of the features found in

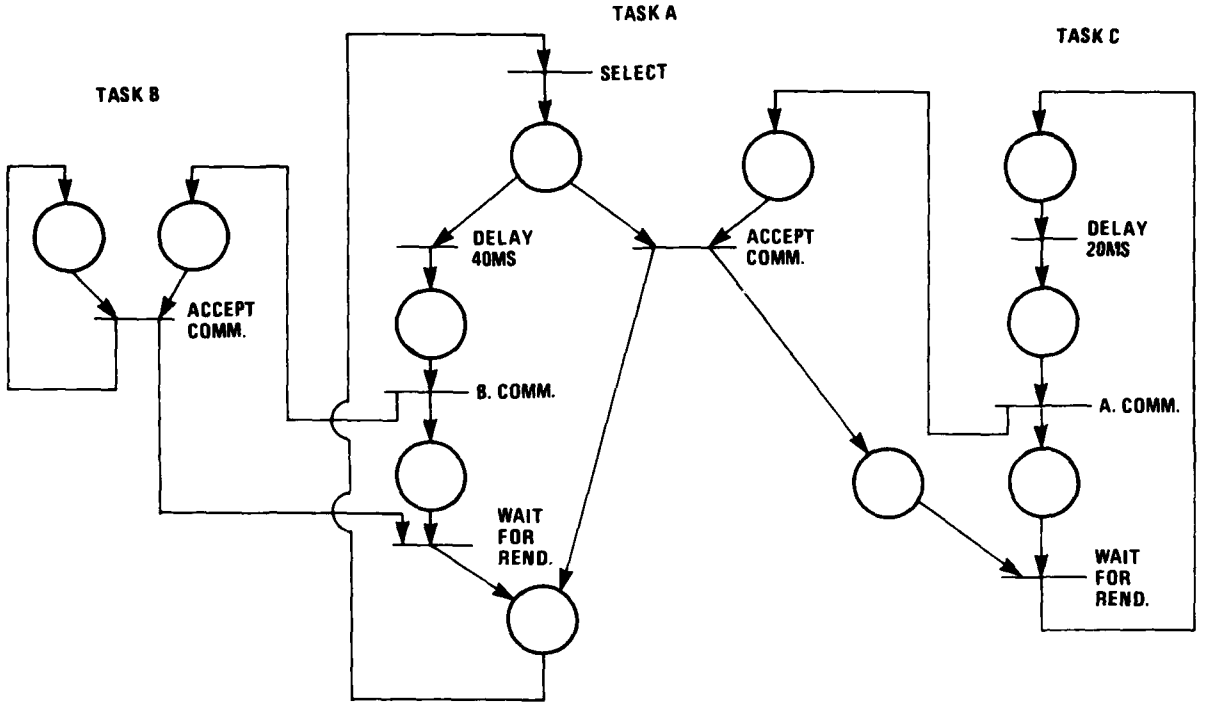



*Figure 39. Petri Net for the Scheduling Algorithm (2).*

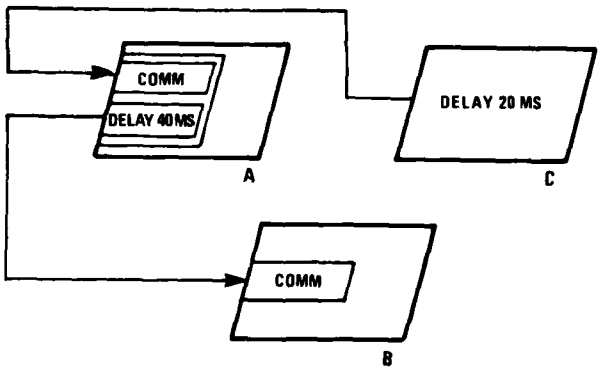



*Figure 40. Buhr Diagram for the Scheduling Algorithm (2).*

Ada, while adequate for operating system design and general applications, do not lend themselves easily to efficient implementation. In particular, dynamic task creation and deletion is worrisome for a number of reasons. One major worry is the lack of traceability and hence the increased ambiguity of design. Other concerns in real-time systems could be the complexity and overhead associated with these features. Obviously as the complexity of the real-time system grows, the number of tasks may grow proportionately, as will the possible need for message buffers (not provided by Ada) and hence the overhead. It would also appear that to diminish ambiguity, programmers must have better control of task scheduling policy. Other areas of concern involve task handling of interrupts and scheduling of I/O requests where fast response to external events is often essential to the proper functioning and viability of the system.

On the positive side, Ada has an impressive number of process control structures for real-time implementations. There are also mechanisms for awaiting any of several messages and for the non-deterministic selection among messages. However, as pointed out previously, there are no means for direct discrimination among arriving messages, nor is there a mechanism for sending messages that can be received by any of many identical servers.

Although this paper addressed a means of graphically denoting designs in Ada, it also pointed out the shortcomings of present compiler technology to support the acceptance of Ada. This immaturity (or lack) of Ada compilers has forced a retrenchment from Ada and, in particular, from its more esoteric features (generics, tasking, exception-handling). In concert with this drawback is the very visible absence of Ada software engineering tools, graphical or otherwise. The Ada market has not matured to the point where Ada tools and environments have been developed to aid rather than hinder large program development. With respect to real-time systems, which usually (if not always) require sophisticated tools and design methodologies, Ada seems to be in its infancy.

The coupling of the Ada mission with its cry for environments has forced software engineers to implement systems in a manner that is less than desirable and unsupported by tools, including efficient compilers and translators. Accordingly, any strong criticisms of Ada at this stage of its maturation may well be unfounded and too severe. It seems likely that the normal evolution of compilers and associated tool sets will reveal the *real* ability of Ada to meet the demands of real-time systems. Indeed, it appears, at present, that the support mechanisms as examined in this paper may well surpass the technology made available in Ada. Although general in nature, the concepts behind a graphical notation seem both viable and worthwhile in the long run. These concepts, supported by the sophisticated automated tools under development (e.g., Buhr's methodology) show real promise of leading the technology front.

## Bibliography


(1) Ada Language Reference Manual, ANSI-MIL-STD 1815A.

(2) Gehani, Narain, *Ada Concurrent Programming*, Prentice Hall, Inc., Englewood Cliffs, NJ 08632, 1984.

(3) M. Ben-Ari, *Principles of Concurrent Programming*, Prentice Hall International, 1982.

(4) Buhr, R., *System Design with Ada*, Prentice Hall, 1984.

(5) Cherry, G. W., *Parallel Programming In ANSI Standard Ada*,, Reston Publishing Co., Inc., Reston, VA, 1984.

(6) Booch, G., *Software Engineering With Ada*, Benjamin Cummings Series in Computer Science, 1983.

(7) Barnes, J. G. P., *Programming In Ada*, (second edition), Addison-Wesley Publishing Company, 1984.

(8) Schumate, K., *Understanding Ada*, Harper & Row Publishers, NY, 1984.

(9) Nissen, J., and Wallis, P., *Portability and Style In Ada*, Cambridge University Press, 1984.

(10) Olsen, E. W., and Whitechill, S. B., *Ada For Programmers*, Reston Publishing Co., Inc., Reston, VA, 1983.

(11) Bergland, G. D., "A Guided Tour of Program Design Methodologies, Bell Telephone Labs," *IEEE Computer*, Oct. 1981.

(12) Ramamoorthy, C. V., and Ho, G. S., "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 5, Sept. 1980.

(13) Habermann, A. N., and Nassi, I. R., "Efficient Implementation of Ada Tasks," Dept. of Computer Science, Carnegie-Mellon University, Pitts., PA, Jan 1980. CMU-CS-80-103.

(14) Karp, R. M., and Miller, R. E., "Parallel Program Schemata," *Journal of Computer and System Science*, 3,147-195, 1969.

(15) Hehner, E. C. R., "On The Design of Concurrent Programs," *Information*, Vol. 18, No. 4, Nov 1980.

(16) Petri, C. A., 'Communication Disciplines, Computing Systems Design," *Proceedings of the Joint IBY and U. Of Newcastle Up On Tyne Seminar*, 1977.

(17) Wirth, N., "Toward A Discipline of Real Time Programming," *Communications of the ACM*, Vol. 20, No. 8, Aug. 1977.

(18) Habermann, A. N., "Synchronization of Communicating Processes," *Communications of the ACM*, Vol. 15, No. 3, 1972.

(19) Andrews, G. R., "Synchronizing Resources," *ACM transactions on Programming Languages & Systems*, Vol. 3, No. 4, Oct 1981, pp. 405-430.

(20) Cohen, N. H., "Parallel Quicksort: An Exploration of Concurrent Programming In Ada," *Ada letters* Vol. II, No. 2, Sept-Oct 1982.

(21) Mayoh, B. H., "Parallelism in Ada: Program Design and Meaning," *Proceedings of the 4th Collogue Internationsl sur la Programmation*, edited by Robinet, Springer-Verlag, 1980.

(22) Mahzoub, A., "Some Comments on Ada As A Real Time Programming Language," *Sigplan Notices*, V16, N2 (Feb. 1981).

(23) Eventoff, W., Harvey, D. & Price, R. J., "The Rendezvous and Monitor Concepts: Is There An Efficiency Difference?" *Communications of the ACM*, 1980.

(24) Agerwala, T., "Putting Petri Nets To Work," *IEEE Computer*, Dec 1979.

(25) Peterson, J. L., "Petri Nets," *Computing Surveys*, Vol. 9, No. 3, Sept. 1977.

(26) Blasewitz, R. M., "Ada As A PDL, Have The Major Design Issues Been Addressed and Answered?" *Proceedings of the 2nd Annual Conference on Ada Technology*, 1984.

(27) "IEEE Draft Recommended Practice on the Use of Ada as a Program Design Language," 1983, 1984, 1985 minutes, notes, R. Blasewitz, Chairperson.

(28) P. Hibbard, A. Hisgon, J. Rosenberg, M. Shaw, M. Sherman, *Studies in Ada Style*, Springer-Verlag, 1981.

(29) Peters, L. J., Software Design: *Methods & Techniques*, Yourdon Press, 1981.

(30) King, D., *Current Practices in Software Development*, Yourdon Press, 1984.

# Ada SUMMER SEMINAR--TEACHING THE TEACHERS

Dr. M. Susan Richman
The Pennsylvania State University, Capitol Campus

Dr. James M. Shoaf
North Carolina Central University

Mr. Donald C. Fuhr
Tuskegee Institute

## Abstract

The Ada Curriculum Development Seminar was organized and conducted over a six-week period by a team of three professors. It was designed to consider various aspects (organizational, curricular, and laboratory) of an intensive program and to evaluate them for applicability within an academic environment. Both industrial and academic methods and materials were used, as well as various video-based media. Detailed questionnaires captured participant opinions regarding the effectiveness and acceptability of each method of presentation. Various system management techniques were also tested in an effort to arrive at an optimum support environment for such a seminar. The results of these tests and evaluations are intended to be used in the design of future intensive programs as well as by the participants in the planning of Ada courses in their own schools.

## Introduction

### Program Background

The Ada Curriculum Development Seminar held at Tuskegee Institute, Alabama from June 10 through July 20, 1984 had its roots in three years of previous similar programs. These programs were sponsored by the U.S. Army Center for Tactical Computer Systems (CENTACS), and were held at Ft. Monmouth, NJ during the summers of 1981, 1982, and 1983. The objectives of all these programs were to propagate the Ada Programming Language into college and university computer science curricula by providing an intensive learning experience for faculty members. That the tradition is growing is shown by the fact that all three of the professional staff of this Seminar were participants in the 1983 program, and most of the guest speakers were involved in some way with one or more of the previous efforts.

### Seminar Contract

Funding support for the Seminar was provided by CENTACS via an existing contract between the Electronic Devices and Technology Laboratory at Ft. Monmouth, NJ and the Southeastern Center for Electrical Engineering Education (SCEEE) of St. Cloud, Florida. SCEEE, in turn, negotiated a subcontract with Tuskegee Institute, providing a fixed level of funding for the local administration of the Program. SCEEE provided direct appointments to the participants and paid them directly.

### Seminar Objectives

The objectives for the Ada Curriculum Development Seminar as stated in the subcontract were:

1. To provide graduate level Ada language instruction to qualified college faculty.

2. To experiment with methods of instruction for Ada for later potential use within academic institutions.

3. To explore curriculum and techniques issues providing insight and recommendations for the introduction of Ada into college level computer science and/or engineering curricula.

4. To explore potential uses of Ada as a hardware design language.

5. To encourage the inclusion of Ada instruction in those colleges represented by the participants. The participants will perform a review at the conclusion of the experimental program. The format will also include short presentations by Ada experts from Industry, Government, and Academia.

### Participants

Attending the Seminar were thirteen participants from seven different institutions. All but two of the participants possessed a doctoral degree, but their backgrounds varied widely. Some had one or more degrees in computer science with commensurate experience. Others had had no prior experience with higher order programming languages. This diversity in backgrounds caused a number of difficulties in presenting the seminar and reduced our ability to make it as effective as we would have desired it to be.

## Planning Model

NOTE: This is a synopsis of a document written before the Seminar as an attempt to capture our thoughts at that time regarding program philosophy. It is included here as an outline of what we intended to accomplish.

The fundamental premise behind our planning for this Seminar is that college students and, therefore, college faculty need a different approach to the Ada programming language from that which is appropriate for working programmers. This premise is based on the following observations:

The vast majority of Ada training courses for industry are only a few days in length, and do not always include hands-on exercises. Essentially all the information must be presented by the instructor, with very little outside reading or assimilation time for the students. We believe that this leads to shallow learning of syntax and semantics, with little understanding of the theoretical basis for proper system design using the language. We believe this approach is not appropriate for teaching the language as a design tool.

College courses, on the other hand, emphasize individual study and research in conjunction with lecture presentations. The result is that college students are taught to apply the language as a tool for problem solving and to draw inferences from this activity as to what new applications may be developed. We believe that college faculty should be taught in the same way.

We intend to test this hypothesis by teaching selected topics such as Generics by both methods and evaluating the group's reactions.

We believe it is important in teaching Ada to college faculty to take advantage of the varied backgrounds of the participants. This can be done by relaying questions to members who may be able to answer them, by having members give presentations, by having them help one another with programming exercises, and other similar techniques.

We believe that evaluation of student progress is an essential element of all training, but particularly in the college environment where grades must be reported. We intend to explore the opinions of the group regarding the impact of Ada on the preparation and grading of examinations, and to discuss broader issues of effective evaluation of programming progress.

We believe that one of the most important ideas to get across in teaching Ada is the concept of software maintenance and how Ada simplifies it. We intend to highlight this feature by requiring the participants to modify existing code under several different conditions.

We intend to gather a large amount of data, via various questionnaires, on the seminar participants' opinions regarding the various pedagogical issues that come up.

## Instructional Activities

### Course Texts

Observations: As is traditional in a college course, texts were chosen based on the instructor's assessment of their appropriateness for the audience. They were followed fairly closely in the order of presentation of topics and supplemented as considered necessary by the instructor. In this program the primary texts were:

An Introduction to Ada,
S.J. Young, Ellis Horwood Ltd, 1983
Software Engineering with Ada,
G.Booch, B.Cummings, 1983

Young is primarily a language text, while Booch emphasizes software engineering. The two texts complement each other nicely. Reading assignments to supplement the lectures were included in the outline for the seminar; however, we determined as the seminar progressed that the assignments were not generally completed.

A third document which was invaluable to the instructor, and to those students who made use of the library copies available, was the "Rationale for the Design of the Ada Programming Language".

Recommendations: There is no single text currently available that would be adequate for any similar program. However, new texts are continually being written and should be reviewed in planning for future seminars. Until a single adequate text appears, the combination we used is a workable alternative.

It is vital in a concentrated program with limited time for digestion of new concepts that students receive as many exposures as possible. Readings, reinforced by the lectures and the laboratory exercises, provide a firm foundation for further study and use of the language. The participants must be convinced of the necessity of completing the assignments on schedule.

### Use of the Ada Language Reference Manual

Observations: The Ada Language Reference Manual (ANSI/MIL-STD-1815A), the only completely reliable source of Ada information, is a vital student reference. It is essential that students become familiar with it as soon as possible. However, learning to use the Reference Manual is a non-trivial task.

In the beginning of the seminar, the students answered questions of the form "What would happen if....?" by actually writing a small code segment and trying it on the system. However, as the questions and the corresponding test programs grew more complex (and computer time became more critical), the Reference Manual became the favored source of information for the students. At first, they found the Manual to be intimidating, difficult to read, and not much of an aid in understanding. However, eventually they discovered that the information WAS accessible even if, after looking

in five different places for the critical section, one had to read that section three times before comprehending it.

Many of the excellent examples in the Reference Manual were used as illustrations for various lecture topics. An understanding of the structure of the example provides a context into which the student can fit the syntax and language rules, making them more reasonable and understandable.

It was further seen that the Reference Manual was the best source (not only comprehensive, but also quite readable) of information on Input/Output. By the end of the course most of the participants had concluded that the Reference Manual was indispensible for programming in Ada.

Recommendation: A copy of the Reference Manual for the Ada Programming Language should be provided for each student. Through whatever means are possible they should be encouraged to become familiar with its style and learn to use it as a source of information and as their final authority on questions about the language.

Classroom Library

Observations: Located in the classroom was a fairly extensive collection of reference materials for use by the students. These materials included numerous language texts, Ada reference books, reference materials for VAX/VMS and the EDT editor, and some of the periodical literature relating to current activities in Ada and future Ada conferences. The participants were thus able to evaluate many texts for appropriateness for use in Ada courses projected for their schools.

Also available for use by the students were various commercially produced video resources on Ada. These included: (1) a videotaped course "Programming in Ada" presented by Ichbiah, et. al. in 1980; (2) "The World of Ada Part II", and (3) "Ada Overview", a PLATO CAI course on a CDC 110 microcomputer. These were accessible to the participants in their free time and were used to reinforce and amplify other presentations.

Order of Presentation of Topics

Observations: Since Ada is more than just another programming language, in order to provide the proper setting for the seminar, the lecture of the first day was devoted to the background of why and how Ada came into being. This was, perhaps, the one topic that NOBODY felt should be moved to another position in the syllabus. A language overview was then given, followed by an exhaustive treatment of each of the components that go to make up an Ada program. The objective was to build a complete set of tools beginning with Lexical Elements and progressing through Data Types, Data Structures, and Subprograms to allow coding of increasingly more complex programs. This was essentially the order in which the topics are treated in the SofTech course L202, Basic Ada Programming, which we were using as an experiment in methodology.

Various circumstances caused us to deviate from the originally-planned order. The most important of these were the need to enable the participants to write programs earlier and the need to run compilations and executions in batch mode which necessitated earlier discussion of File_IO.

Recommendations: In setting up the order of presentations in a programming language course there are several competing and conflicting aims which must be balanced. Firstly, there is the pedagogical aim of not overwhelming the student with new information to absorb -- with the consequent risk of brain shut-down. In addition to learning to write the code, the student must learn a great deal of information in order to interact with the computer system. He or she must learn the log-in sequence, operating system commands, file creation and manipulation, reactions to unexpected system responses, interfacing between files and programs, creation and use of program libraries, and commands for compilation and execution. Secondly, however, it is essential that the student begin writing code and testing it on the system as early in the course as possible. These two goals are in strong conflict. The best solution is to reduce, in whatever ways possible, the amount of information the student must learn in order to get a meaningful response from the system to his Ada code.

Since the fundamental Ada concept of Packages allows programmers to use tools without necessarily knowing all of the details involved in the implementation, we recommend beginning the course with Packages and having the students write programs which use packages previously designed and made available to them. This tactic will also reinforce the principles of abstraction which are so vital to Ada. An overview of all the data types available in Ada should come next, with just enough detail to allow students to write programs involving simple data structures. Input/Output incantations for all the different data types should be provided at this point with only minimal explanation of the actual mechanics of IO. Subprograms and control structures will then allow development of solutions to some significant programming exercises. The details of structured data types, such as multidimensional or unconstrained arrays and discriminated records, can and should be postponed until the students have had an opportunity to work with the simpler forms.

Lecture/Laboratory Daily Format:

Observations: Throughout the seminar we experimented with lengths and times of lectures. In the beginning of the seminar the morning was devoted to a lecture presentation and the afternoon to laboratory work. For a time, the ratio of lecture to lab time actually increased because of our efforts to stay on schedule. It soon became clear that there was insufficient laboratory time for the students to practice using the language features. As a result, much of the detail which was covered during the lecture was not fully absorbed and retained. The ultimately most successful mode was that of two or three lecture

SCHEDULE---PROGRAMMING EXERCISES

| | Day 1 | Day 2 | Day 3 | Day 4 | Day 5 |
|---|---|---|---|---|---|
| Week 1 | | Ex 1,2 | Ex 3,4 | Ex 5 | Ex 6 |
| Week 2 | Ex 7 | Ex 8 | Ex 9 | | |
| Week 3 | Ex 10 | | | Ex 11 | Ex 12 |
| Week 4 | speaker | | holiday | | |

SCHEDULE---PROJECT

| | Day 1 | Day 2 | Day 3 | Day 4 | Day 5 |
|---|---|---|---|---|---|
| Week 4 | speaker | | holiday | | project dist. |
| Week 5 | Ex 11F speaker - - | team project development | Ex 11F speaker | | Ex 11F speaker - - |
| Week 6 | speaker | speaker | speaker review guid. | presen. guid. | project pres. |

Ex 11F = a group of two people finish Ex. 11

Environment

Each computer user required an orientation to the working environment, plus specialized information on environmental aspects when deemed necessary. The environment consisted of the computer system, the text editor, and the Ada compiler.

Background system information was divided into three parts. In addition to a basic introduction to the system, there was a special presentation on running batch jobs, plus many short presentations on individual system features (called VMS Minutes).

The first-day environmental orientation began with the computer interface step--the local login procedure followed by the VAX user-account initialization. Afterwards came an introduction to VAX/VMS commands for handling files. The information presented had three parts: the notion of file directories, a set of eight basic commands, and a small set of practice exercises. The practice exercises included steps preparatory to running a simple Ada program.

Information for submitting batch jobs for compilation and execution came at the end of the first week. The remaining system information was presented in the format of VMS Minutes. Each of these short presentations by the system manager usually centered on one useful system feature.

Information on the NYU Ada/Ed compiler was given in two parts. A short introduction to compilation and execution of Ada programs was the second part of the first laboratory session. Only a minimum background was presented, because of the diversity and bulk of information given at the first-day orientation laboratory. In the second week, the laboratory director discussed Ada/Ed program libraries.

The VAX EDT editor was the tool used for source code entry. On the first-day laboratory the basic operations of this editor were presented. The laboratory director followed this on day two with information on more advanced features of the editor--information that was found to be not very worthwhile.

Programming Exercises

The twelve programming exercises were graduated in difficulty and based on most-recently-presented Ada features from the lectures. As minor changes occurred in the lecture topics, corresponding changes were immediately incorporated into the exercises. Also, the last six of the exercises were structured so as to serve as preparation for the team projects. The Ada or environment areas featured in the exercises are shown in the table below.

PROGRAMMING EXERCISE FEATURES

| Exercise | Ada or Environment Features |
|---|---|
| 1 | VMS commands, basic EDT operations, basic Ada/Ed operation |
| 2 | IO Package instantiation for integers, string output |
| 3 | FOR loops, local subprograms, type conversions |
| 4 | Same as Exercise 3, plus constants and emphasis on modifiable code |
| 5 | IF and CASE statements, enumeration types, integer ranges |
| 6 | FOR loop, arrays, local subprograms, overloaded GET and PUT |
| 7 | file I/O, VMS files, WHILE loop |
| 8 | packages, Ada/Ed libraries, function calls, WHILE loop, file I/O |
| 9 | records, enumeration types, packages |
| 10 | variant records, integer ranges, exceptions |
| 11 | access types, packages, tasks |
| 12 | generic subprograms with a formal generic subprogram parameter |

periods of approximately one hour each, interleaved throughout the day with laboratory periods. This provided a better mode for retention and comprehension of the information presented in the lectures and also resulted in more efficient use of the lab time, since batch compilations and executions could be going on during the lectures.

Methodology in Presentation of Topics

Observations: In teaching Ada, the methodology used is closely related to the topic sequence, particularly the issue of "top-down" vs. "bottom-up". In the use of the SofTech materials we followed the bottom-up approach, treating each topic in great detail, then progressing to the next topic. It became clear as the course progressed that not even a class consisting of Ph.Ds can absorb great amounts of detail after hearing it only once, or sometimes even twice. It was evident by the questions asked and the programs written that not all of the concepts were grasped and appreciated right away, not even by the most experienced computer scientists in the group. It is often necessary to hear something several times, preferably in different ways, before it makes a lasting impression. For this reason we found it advisable to change our approach to provide a variety of exposures to the basic information: lecture, discussion, in-class exercises, laboratory exercises, reading assignments, and videotape presentations all contributed to the assimilation of the material.

Recommendations: The course should begin with an overall view of the Software Crisis, Software Engineering, and the History of the Ada Programming Language. Sending the participants advance materials to read on these topics may make this presentation more effective. The course should progress from there into an overview of the language covering, without much detail, program structure, the different data types, and control structures. One might supply students with a package or packages and have them write program(s) using the facilities in the packages. It seems best to introduce new topics by giving simple examples, giving the basic facts, reinforcing these by means of exercises whenever possible, and using the spiral approach of returning to previous topics with new perspectives and insights.

In place of exhaustive detail, it is better to explain the "why" and the "how" of the concepts. It is reasonable to expect the students to locate in the Reference Manual, and in other reference sources, some of the detail which they will need in order to complete their programming assignments. The instructor should encourage the independent use of videotaped lectures and other course materials for multiple exposures to topics as needed.
One should supply IO_Incantations as soon as they are needed for programming. There is no need to postpone I/O operations until the students have the background to fully appreciate generic packages and Text_IO. The use of limited-function, already-instantiated "Easy_IO" packages is at best a stop-gap measure, and at worst misleading and confusing.

Terminal access for each student during the lecture periods can be a two-edged sword. While useful for illustration and as a teaching tool, terminals can also be distracting. If terminals are available in the classroom, the capability of deactivating them would, at times, be valuable.

Guest Lecturer Program

A valuable component of the seminar, the guest lecturer program was designed for experts in various aspects of the Ada world to share some of their expertise with the participants. The eight speakers provided valuable insights and were well received by the participants. This program should be part of any similar seminar, and should be expanded if seminar constraints allow.

## Laboratory Activities

Overview

The laboratory activities consisted of three interrelated parts: an introduction to the programming environment, the solution of twelve graduated programming exercises, and a team project. These activities were synchronized with the corresponding lecture presentations. Together the lecture and lab work gave the seminar participants a fairly thorough background in Ada. The particular schedule and choice of topics were selected on a day-to-day basis because of changes made in the original organization of seminar topics. This offered the advantage of using the latest Ada features from the lectures in a laboratory exercise.

The first four weeks were devoted to the programming exercises and environmental introduction, while the last two weeks were spent on the team projects. A simple schedule for each part of the laboratory activities is given below. Following paragraphs supply details for the scheduled activities indicated.

SCHEDULE---ENVIRONMENT

| | Day 1 | Day 2 | Day 3 | Day 4 | Day 5 |
|---|---|---|---|---|---|
| Week 1 | Ada/Ed EDT | EDT | * | * | VMS |
| Week 2 | * | Ada/Ed | Ada/Ed | * | * |
| Week 3 | * | * | * | * | * |

* = VAX/VMS Minutes

Of the participants in the seminar, six completed all of the exercises, two completed eleven exercises (with Exercise 11 80% finished), and the other four completed 7-9 exercises. These results reflect the varied programming background of the participants, some loss of programming time for guest speaker presentations, and the above average difficulty of Exercise 11.

Team Project

The team project was intended as a small software engineering experience in which the key Ada features of packages, subprograms and tasks are used as tools. The seminar group was divided into 4 groups of approximately equal talent. Each team was given some general software engineering guidelines in addition to the project problem description. The particular problem consisted of simulating the operation of an airport with two runways, controlled by one air traffic controller.

The project guidelines and problem description were distributed on the last day of week 4. At that time the majority of participants were in the final stages of Exercise 11, having finished Exercise 12 earlier. Project planning began immediately while Exercise 11 was being completed. During week 5, the teams made progress reports. At the beginning of week 6, it was decided to give the teams more work time, postponing the project solution and review presentations until the last day of week 6.

The project problem was based very heavily on the Ada features used in Programming Exercises 9, 11, and 12. Exercise 9 presented the basic data structure with input and output. Exercise 11 presented a chance to become familiar with access types in setting up a queue and with tasks involving a third-party task that manages a shared data resource. Exercise 12 provided an opportunity to use a generic compilation unit. The project brought all of these concepts together in a team setting. The two major constraints of the problem were a generic queue package for setting up plane arrival and departure queues and the concurrent processing of planes on the runways and in the two queues.

Each team encountered three major hurdles in its work: setting up the basic plane flight data structures and accompanying I/O operations, setting up the generic queue facility using access types, and the tasking component for controlling the accessibility of the queues. By the end of day 4 of the final week, one team had overcome all three hurdles. The other teams were involved with the last hurdle, making it possible for each team project solution to be realistically reviewed by another team.

The review process involved each team passing its completed code to another team. The second team was to implement a change in specifications in the code. The objective was to demonstrate the ease with which good Ada code can be analyzed and the ease of localizing the effect of modifications.

The project solution and review presentations on the last day showed several creative and workable approaches to the project problem. Each team followed the instructions to present a hierarchical chart of program units to aid in explaining the overall nature of their project design. The main differences occurred in the division of the tasking aspect of the problem. Some of the main issues raised about the project solution development were minor confusion on the initial requirements, completion of the design occurring in the midst of the major coding, and that ease of design development corresponded to a good working knowledge of the tools available. The modification presentations mentioned at least one aspect in each project solution in which some Ada feature would have made the proposed modification easier. Each review team mentioned that modifications would sometimes require considerable searching through the code to insure complete correctness of modifications. Each team, from both a design and modification viewpoint, learned much about the significance of software maintenance considerations.

Conclusions

Overall the main goal of the laboratory activities was accomplished: first-hand experience in using the powerful main features of Ada, in a team setting. The secondary goals of completing the team project, getting familiar with all the advanced Ada topics, and gaining skill in writing easily modifiable code were reached with varying degrees of success. A few modifications in the seminar structure would improve the chances of achieving these secondary goals:

1) adding two weeks to the length of the seminar in order to schedule speakers well before the project development period and to allow for adequate exposure to advanced Ada features;

2) providing an additional orientation session to the environment for novices, less information on the advanced editor features, and sample code with or without compilation errors;

3) requiring more programming exercises of short length so as to provide better feature exposure and to emphasize abstraction and software engineering through the use of interesting package problems;

4) providing more guidance and background on key aspects of software engineering relating to the team project--better preparatory exercises, background information on efficient ways to develop software components and test them, and tips on using Ada and other tools in program design.

## Computer System Support

### System Configuration

Hardware: Computer support for the seminar was provided by a dedicated VAX-11/780 which contained 8 MBytes of memory, 512 MBytes of disk storage, and 40 ports. The 16 classroom terminals were connected to the computer through a MICOM Micro600 Port Selector. This hardware configuration proved to be adequate for this size program, and probably for a group of up to 20 participants.

Software: Initially, the seminar used Version 1.1 of the NYU AdaEd interpreter running under VMS. This worked well enough, notwithstanding its well-known slowness and the various documented bugs. Later, we used Version 1.4, with only moderate gains in speed, but much more reliable performance. Along with Version 1.4 came a faster parser, which provided exceptionally quick syntax checking. AdaED supported the seminar well, but required constant monitoring to prevent unacceptable system performance degradation.

User quotas and privileges: Most user authorizations in effect were the defaults provided by VMS. Exceptions are as follows:

> A 1300-page (650 KByte) memory allocation was necessary to allow interactive Ada jobs to run without overwhelming the system with page faults.
>
> It was necessary to increase the PGFLQUOTA to 13000 from the default 10000 value in order to allow Version 1.4 to run correctly.
>
> All users were given GROUP privilege in order to allow them to stop their own batch jobs when they behaved erratically.
>
> Each user was given Read access to the system Accounting data file and instruction on how and why to use it as a last resort in troubleshooting a problem which cannot be isolated between VMS, Ada, SetL, and programmer logic error.

### System parameters

Interactive operation: The first day in lab, in order to test the limits of the system, we deliberately had all 13 participants start an interactive compilation at once. After thrashing for about 2 1/2 hours, the jobs finished and the system recovered. We concluded that the system would not support 13 jobs needing the same resource at the same time, but if the jobs were started in a random fashion as would normally be the case, the system would survive. This proved to be true for the rest of the seminar. The problem with interactive work is that the user's terminal is disabled for the 15-30 minutes or more that a compilation takes, preventing work on another compilation unit or exercise while waiting for the first to complete. A partial solution is to perform compilations in batch mode. This releases the terminal as soon as the system accepts the job.

Batch operation: To strike the best balance between memory limitations and CPU contention, we operated with three batch jobs running simultaneously from one queue, allocated 2500 pages (1.25 MBytes) of memory each. Running fewer than three jobs uses memory inefficiently; running more causes a bottleneck at the CPU.

### Command Procedures

Since this was in large part a research environment, we realized that we would need ways of changing the operating parameters of the system quickly. To this end, several command procedures were developed before and during the seminar to make this easy and reliable.

ADAUAFCHG.COM: Used to modify entries (such as quotas and privileges) in all group User Authorization File records with a simple, interactive procedure.

ADAQSTOP.COM: Used to change the operational parameters of the batch queues as needed.

COPYFILES.COM: Used by the faculty to send files containing useful examples, lab assignments, and other information to all participants.

COPYLOGIN.COM: Used to add entries to each participant's LOGIN.COM file when the function was needed, but the explanation would not have been understood.

MAIL distribution files: Three files for mass mailing were made available to all: one containing only the Usernames of the staff, one containing only the Usernames of the class, and one containing all Usernames in the program. These were used extensively for broadcast communication among all three groupings.

## Seminar Logistics

### Classroom/laboratory facilities

A dedicated classroom was provided for the program. One terminal was provided for each participant, one each for the Lab Director and Academic Director, and one spare. The system printer was remoted into the classroom for hardcopy output. A coffee bar and break area were set up in one corner of the room, and videotape machines and demonstration microcomputers in another corner. Two overhead projectors were available. Access to the classroom was provided from 8:00 am until 11:00 pm Monday through Friday, with weekend hours as requested by the participants. The room was locked whenever it was unoccupied, protecting the equipment and enabling participants to leave materials there if desired. The facility was deemed adequate except for some temperature control problems.

## Budget

The budget total of $62,000 for Tuskegee Institute covered faculty salaries and expenses, computer operation and maintenance, seminar logistics, review and coordination meetings, and overhead. Actual expenses were within budget, although the distribution of expenses among budget elements was considerably different from the original estimates.

## Seminar Staff

A total of six people performed various tasks in support of the seminar. The Academic Director was the primary instructor; thus, the primary selection criterion was lengthy teaching experience, including the teaching of Ada. The Lab Director developed and administered the programming exercises; thus, the primary qualifications were facility with the language and ability to work well with people. The Systems and Logistics Director's job was to handle all system actions and logistics arrangements. The qualifications for this job were primarily managerial. Due to the critical importance of good computer support, it is essential that this person occupy a position of authority with respect to the computer system and the people who directly operate it. Other personnel involved were a technical specialist who operated the system, a secretary who performed the clerical support, and a statistician who designed and analyzed the participant questionnaires.

## Analysis of Participant Responses

The seminar participants were asked to provide written reactions to the program in four different formats: 1) A daily questionnaire during the lecture phase; 2) A mid-point questionnaire at the end of the third week; 3) A questionnaire each day on which a guest speaker appeared; 4) A final questionnaire at the end of the program.

## Daily Questionnaires

These surveys were used to keep a daily pulse on the participant reactions to the seminar. They were used to determine the need for changes in approach or sequence as the program progressed. Because of this utilization, they reflected opinions on specific topics, and showed a steady improvement throughout the seminar.

## Mid-course Survey

This questionnaire was administered to capture the thoughts of the group at the point at which they had been exposed to essentially the entire language, but before they had an opportunity to work extensively with it. We tried to quantify the comments we had received informally regarding topic order and lecture/lab time proportions. The quantified responses to this mid-course survey indicate that, in most areas, the program was successful up to that point. There were no major problems identified, at least none that had not been addressed by the staff. There were no signs of real dissatisfaction from anyone.

## Speaker Critiques

A questionnaire was distributed on each day when a guest speaker was scheduled. This was not so much intended as a speaker rating, but as a measure of the value of the presentation to the program. Summarized results from these questionnaires are as follows:

> The group overwhelmingly rated the guest presentations as understandable and useful, appropriate for the program and for the time in the program, not too technical, and worth the time allocated. Even the somewhat less dynamic speakers were rated high. Many of the group, while approving the speakers, also complained verbally about not being able to work on the team project as much as they would have liked. This would seem to make a good case for a longer program.

## Final Program Survey

The end-of-seminar questionnaire was designed for somewhat more free-form responses. There were some numeric categories, but most of the questions required write-in answers. The group, as had been the case throughout the program, was most reluctant to write much. We will use some of the specific comments as indicators of possible improvement areas, but they are not amenable to statistical analysis. Those numeric results that are significant are as follows:

> The participants were asked to rate themselves on 13 background and academic-related factors compared to the others in the program. The results showed a very high correlation between their opinion of their computer-related background and Ada ability and the opinions of the staff regarding their relative performance in the program.
>
> The seminar was rated overall as being very worthwhile by 92% of the group. They were completely ambivalent about whether it should be restricted to particular skill levels. All of them felt the seminar was well organized.
>
> 92% of the group agreed that the lab assignments greatly aided understanding of Ada. This response from a group of academics is strong evidence that adequate lab work is very important to Ada instructional programs.
>
> All of the group rated the instruction excellent to outstanding in being well-organized, presented clearly, not too general or theoretical, and informative and useful. 83% said it was aimed high enough, but one participant strongly disagreed with that premise.
>
> The ratings for the instructors were similar to those for instruction. All of the group rated them excellent to outstanding on competence, attentiveness and understanding, being well-suited for the course, and effective use of audio-visual aids.

The participants rated the individual lab assignments as generally being about right for complexity, tending toward the complex. They reported having problems understanding Labs 10, 11, and 12, but gave the others good marks for understandability.

VAX/VMS information was rated excellent to outstanding by 83-100% of the group as being necessary, adequate, and timely. They were undecided as to whether it would have been better in one chunk. EDT Editor information was rated as necessary and adequate by all of the group.

The computer was found by 75% of the group to have provided adequate support and to be easy to learn to use.

All of the group rated the guest speakers overall as valuable, a welcome change of pace, and experts in the field. A slightly smaller number, 83%, felt that they did not detract from the lab work. The vocal comments on this matter were not reflected in the questionnaires.

The seminar was judged to have been worth the time by all of the group. 83% felt it should be at least 6 weeks long, and 67% voted for 8 weeks. The seminar format was highly approved over other media such as videotape. The group was undecided as to whether the seminar was too challenging or ambitious, but they said it lived up to their expectations.

The end-of-program survey results reflect overall satisfaction with the program, as well as a desire on the part of the participants to provide constructive criticism in order to improve it. The results are in general agreement with our assessment of the program's strong and weak points, and with our perception of the group's reactions.

## Conclusions and Recommendations

### Program Length and Composition

The six-week length of this seminar was minimally adequate. We covered essentially the entire language, but there was not enough time to adequately cover the advanced topics. Most of the participants needed more laboratory time in order to become really fluent with the language.

We recommend that the seminar be scheduled for eight weeks. The first three weeks of this time should be spent on a basic coverage of the language together with many lab exercises illustrating the specific language features. The next two weeks should be used to expand upon the advanced features, emphasize graduated exercises using these advanced features, and include most of the guest speakers. The final three weeks should be allotted for the team software development project. We strongly believe that this project is essential to giving the participants insight into the full power and utility of the language.

### Contract Timing

We strongly recommend that the fun[illegible]ng source, contract mechanism, and host site(s) be identified as early as possible, and that the contracts be finalized not later than mid-January. This would allow selection of participants by mid-March (before they have other summer commitments), and sufficient time for pre-seminar preparation.

### Academic Considerations

Our experience leads us to believe that Ada should be presented using a "top-down" approach, but not by the definition normally applied to that term. A language overview is important at the beginning, but it must include enough detail so that students can start to write programs very early. One should take advantage of Ada's information-hiding features to allow students to write small programs using packages already in existence without fully understanding how they work. Then, as they gain more knowledge and experience, they can be assigned to develop the packages to support larger systems. It must always be remembered that an overview does not prepare one to write code, and an extremely detailed lecture does not illuminate the big picture. A view of the entire puzzle is needed, with just enough detail to enable the students to begin to assemble the pieces for themselves.

### Laboratory Considerations

Our experience reemphasized the need for adequate laboratory facilities and time to support the instructional effort. The facilities available for this seminar were deemed adequate by us as well as most of the participants. The final daily schedule of 1 hour of lecture followed by 2 hours of lab, repeated in the afternoon was the consensus choice by both participants and staff.

One area we believe should be improved is that of VMS/EDT familiarization. With the diversity of backgrounds among the participants, there were some with extensive VAX experience, and others with no appreciable experience with any computer. Even with a more homogeneous group, this could be a problem. Our recommended solution for this problem is to offer a pre-seminar tutorial the day before the official seminar opening for those who wish it.

### Group Size and Composition

The group size of 13 was easily accommodated for this program. The wide diversity in background among the participants caused some difficulties and necessitated some compromises in depth and speed of presentations. This reduced the benefits of the program in various ways for most of the group. A larger group with the same diversity would have been very difficult to handle. We believe that a more homogeneous group composition is very important to the overall success of the program. This should be achieved by stating a minimum qualification of fluency in a higher order programming language for applicants. If desired, a separate seminar should be offered for those learning Ada as a first language.

The feasible size of a seminar group depends largely on facility availability. Assuming a homogeneous group, we believe that 20 participants could be accommodated with the facilities at our disposal.

## Budget Considerations

The total budget allocation of $62,000 (not counting participant compensation) for this seminar was adequate, and would probably be adequate for another year at this location. Most of the planning factors are specific to location, and would vary depending on how much setup work needs to be done. It is importantthat the overall funding for the program include compensation for the participants. If a qualified pool of applicants is to be attracted, some form of income for the summer is necessary in order to attract those who might have other summer employment opportunities.

## Staff Requirements

Our complement of three direct professional staff plus three support staff was adequate. We believe that three is the minimum number of professional staff required to present an intensive seminar such as this. The heavy (and sometimes short notice) duplication requirements necessitate clerical support on nearly a full-time basis. The need for a system operator depends on local management structure. The need for a statistician depends on data analysis requirements.

## Seminar Results vs. Planning Model

The objective and subjective appraisals of the seminar support the hypothesis stated in the Planning Model: That academics learn better and are more satisfied with a program taught in the academic mode compared to an industrial-type program. The participants were not pleased with the SofTech approach of presenting all the detail in the lectures. When we changed to shorter, less detailed lectures supported by reading and practice they responded better, and seemed to learn more.

Several of the test methods we had intended to use fell victim to the shortage of time. We did not feel we could afford to teach the same topic twice using different methodology, particularly since the group so thoroughly disapproved of the SofTech approach. There was also strong resistance to examinations, at least the recording of scores. Therefore, our objective of research on testing issues was not met.

We were able to take much advantage of the superior experience of some members in helping the others, but we found that the slower students were reluctant to ask for help from the others.

The modifiability and reusability of good Ada code were made clear to the participants via exercises requiring modification of previously-written code, team exercises, and use of previous exercise results in the team project. These activities were very successful and gave the participants a good appreciation of these significant Ada features.

## Conclusion

This Ada Curriculum Development Seminar was deemed by all participants and staff to have been very successful. Our plans were generally effective, anticipating most problems that arose. We worked hard to ensure responsiveness to participants' concerns and made numerous in-progress adjustments as a result. We strongly recommend that seminars of this type be continued at this and other sites as the best way to prepare faculty to teach Ada effectively. We believe that, if government funding cannot be made available, various progressive industrial firms would be willing to do their part to assure the flow of well-educated computer science graduates fluent in the Ada language.

## References


1. Booch, Grady: Software Engineering with Ada, Benjamin/Cummings, 1983.

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

3. Ichbiah et al: Rationale for the Design of the Ada Programming Language, Draft for editorial review, Honeywell and Alsys, 1984.

4. SofTech, Inc: Course Notes, Basic Ada Programming (L202), U. S. Army (CENTACS), 1983.

5. Reference Manual for the Ada Programming Language, ANSI/MIL STD 1815A, 1983.


## Biographical data

M. Susan Richman is chairman of the Mathematical Sciences Program at the Capitol Campus of the Pennsylvania State University, Middletown, PA. Dr. Richman is a graduate of the University of California, Berkeley, and received the Ph.D. in Mathematics from the University of Aberdeen in Scotland.

James M. Shoaf is Associate Professor of Mathematics at North Carolina Central University, Durham, NC. He is a graduate of Pfeiffer College and received the Ph.D. in Applied Mathematics from North Carolina State University.

Donald C. Fuhr is Director of Computer Services at Tuskegee Institute, AL. He is a graduate of Oregon State University and received the M.S. degree in Engineering Management from the University of Alaska.

# Some Practical Experience in the Organization of a Library of Reuseable Ada* Units

Randal Leavitt

PRIOR Data Sciences Ltd., Nepean, Ontario

*Ada is a trademark of the US Government, Ada Joint Program Office

Abstract:

This paper summarizes the steps being taken by one real-time software firm to prepare for the new Ada marketplace. By concentrating on the creation of a library of reusable software we are beginning to use the new methods best suited for Ada programming. Tool acquisition costs must be carefully compared with potential long term savings during this transition period.

## Goals: What We Expect to Achieve

Our company is a relatively small software engineering firm that develops real-time systems. We take a very competitive approach to the marketplace, maintaining our lead position by employing exceptionally talented people. We do not have an excessive amount of money to invest in capital goods to support our work (ie, our offices are not the most elegant one possible), yet we are very much aware that we have to make some sizable expenditures to prepare for the coming Ada transformation. Ada will require a significant amount of "up front" work in order to be successfully used.

Our situation, therefore, is a typical one in the software industry, and we hope to contribute to progressive developments and also to learn as much as possible by discussing it openly.

We have decided to create a library of reusable Ada software to focus our Ada conversion, with the expectation that this will:

a. enhance our reputation as experts in the most advanced forms of software engineering,

b. lead to improvements in our software developments methods,

c. reduce our system development costs,

d. produce better quality systems for our customers, and

e. lead to the development of some marketable software products.

It is interesting to note that the very readable Ada language can contribute to reduced maintenance costs, but this does not directly benefit us since we are seldom required to maintain our delivered systems. The creation of a library will, however, give us an opportunity to take full advantage of the maintainability of Ada software. This kind of change may lead to a complete transformation in the way real-time systems are acquired and maintained in the future, with more of the maintenance work and responsibility for correct performance falling on the original developer.

## What We Have Accomplished So Far

A library of reusable software can be a very complicated system. Recognizing this, we have begun the development of our current library as a prototyping experiment. We expect to continue with this "build as you go" approach until we have a good understanding of our actual requirements. At that point a thorough revision or a new beginning may be called for. Consequently, we are adding to our library as fast as possible, responding to feedback about suggested improvements, and making notes about where we should be headed in the future. This approach to library development requires very little initial expense, and since it has already led us to a few mildly surprising conclusions we feel that it is the best strategy to follow at this time.

We began with perhaps the simplest possible definition of a library, namely a collection of Ada program unit source

files. Our library now includes about twenty-five different units covering:

a. mathematical functions packages,

b. complex arithmetic abstract type,

c. user interface procedures,

d. development tools such as prettyprinters and communications support programs,

e. discrete Fourier transform procedures, and

f. various utilities such as sorting and searching routines.

We then decided that each library unit must be accompanied by at least one demonstration or test program. Such a program may also require test data files. We have now added these to our library as well, and set up a small database showing how these files are related to each other. A library such as this very quickly develops its own complicated internal dependencies. The complex arithmetic types, for example, are used in the discrete Fourier transform package, and the user interface routines appear in many of the test programs.

We have also written a brief user's guide for the library. It lists the library contents under various categories, describes access procedures, and gives the style rules followed while writing the library entries. Each library unit has a standard header block, which is also explained in the user's guide. We expect this guide to be just as useful for those making new contributions to the library as it is for those drawing on the resource.

The next step was the announcement of a company policy requiring Ada project managers to have an early design review with the library manager for each project. This review will help guarantee that the library is used as much as possible, and that library enhancements will be made to meet real project work needs. We intend to establish the tradition of reuse right from the start with our Ada work.

Finally, as one of our accomplishments, we have recently made our first Ada software sale, using our library components to construct the delivered product. Based on this we are now setting up another database to keep track of which library units have been drawn out and used. This will allow us to properly distribute future updates and revisions.

## Some Preliminary Decisions

The process of actually attempting to put together a software library forces decisions to be made. These constitute the primary lessons learned from our protoytping exercise. We are recording them carefully for later analysis. So far they can be summarized as follows:

a. The library will only contain text files to keep its organization and maintenance as simple as possible. Users will take copies of the routines they want and compile them as needed.

b. A database is needed to keep track of all the relationships between the library elements. This ensures that revisions and updates are properly followed up on throughout the library.

c. There must be at least one test program for each library unit. These tests will be as self contained as possible so that simply running on will give a clear indication of whether or not the library is correct.

d. A common header block will be used for every Ada program unit in the library. The sections in this header will provide enough information to allow an *experienced programmer* to use or maintain the unit.

e. Ada project managers will be required to use the library extensively. They are also expected to identify areas needing enhancements.

f. Contributions to the library will not be written as part of any particular project's work. There must be no doubt about the ownership of the library contents.

g. A database will be used to record who has used the library resources so that future updates can be properly distributed.

h. Library program units will be written using the entire Ada language, even though we only have a subset compiler at present. Full compilers will soon be available.

## Findings Related to Reusable Ada Components

In the process of putting together our prototype Ada library we have noted the following interesting points:

a. Ada compilers must eliminate unreferenced subprograms, data objects, and unreachable code in executable program images. This material has to be included in library units to provide the needed generality, but often is not fully used in any single application. The lack of this feature in an Ada compiler creates a severe restraint for truly generalized library development.

b. Library generic units are very difficult to write. The requirements are often complex, vague, and conflicting. Generic units are seldom produced as a by-product of regular software development since the effort required to properly generalize them is usually significant. Since it seems to be practically impossible to get them right the first time some process of gradual introduction into the library may be needed. Specialists should be employed to create usable, reliable, and well documented generic units.

c. Generic units are also difficult to use, especially when they have many interrelated parameters. The parameter matching rules can be very subtle. Also, since a generic instantiation does not show the actual interface (ie, the procedure parameters or the package specification items), instantiated units are easily misused. These problems incline programmers away from generic units. Ada training courses should emphasize generic unit use. Generic unit documentation must include complete examples of instantiations.

d. The generic package is the most complete unit to use as the common library generic unit. Generic procedures, for example, cannot propogate exceptions to the calling program in any useful manner. This can be done if a generic package is provided to make both the exception and the subprogram names available. Since this approach must be taken to effectively create generic tasks, it simplifies the library if all generic units are implemented as generic packages.

e. Since the instantiation of some generic units can be troublesome, it seems to be a good idea to have some library units which are typical intstantiations of existing units. For example, a generic trigonometric functions package with a floating point type parameter could be instantiated as another library unit for the standard type FLOAT. This second unit can then be referenced as a simple package. This approach can be used to simplify the use of library entries.

f. A library must be much more than a collection of Ada program units. Test programs and data, specification documents, and user documentation must also be stored. Database facilities are needed to keep track of relationships between library elements, of users requiring updates, and of classifications and categories used to organize the library contents.

## Future Considerations

As we are building our current library we are beginning to formulate some ideas about the ideal system we would like to have.

First of all, our current project is loading the library from the bottom up. This does not appear to be the most profitable approach. Most programmers can produce low level building blocks quite easily, often with less effort than it takes to query a complicated library. What we seem to need is a library loaded from the top down. We are considering now if this might be feasible using an Ada PDL to document designs which we can store in the library. These standard designs may provide very good starting points for future development work.

It is also obvious that the interactions of our software engineers with the library system are similar to those that a specialist in any field will have with an expert system. An expert system front end may be very helpful to both those drawing from the library, and to those adding to it. It may also be feasible to have the library store only very generalized templates which are used to guide the output of a particular Ada program tailored to the user's stated requirements. The potential for productivity improvements appears to be very large using these methods.

Present day software development methodologies do not assume that software reuse is the fundamental operation; instead they are based on new development. This must change if we want our Ada library to be really beneficial. We have already begun the investigation of how to make these changes.

Many of the benefits of a software library can be realized with other programming languages as well. Whether we should have a different library for each language, or one Ada library with automated translators is currently an open question. Our

preference would be automated translation, but the cost of developing this is beyond our means at present.

Formal methods involving proofs of correctness are too expensive to be applied on project work. However, the cost equation changes if we consider library units which may exist for many years and which may be distributed to many systems which are very difficult to update. We anticipate the need to use much more formal and rigorous methods to build the solid core of a library.

Finally, data communictions must be included as an important feature of our library service. We must be able to fill up our library from external sources, and able to deliver to these sites as well. Without this capability we will not be able to load our library adequately. The principal difficulties in this area are those concerned with ownership and liabilities.

## Conclusions

Our initial attempts to create an Ada software library have been very exciting for us. There is good reason to believe that this project will radically change the way we do our work in the future, leading to new methods based on reuse and requiring much more formalism and precision. We expect that this will be a very satisfying environment to work in. On the other hand, the general lack of demand for Ada programming up to this point has made it difficult to justify major efforts for our library. We have had some minor successes, such as as initial delivery of a program based on library components, but in general the mood remains tentative.

We expect this situation to change now that the Ada compilers have finally arrived. Our response to a strong demand for Ada software will be based on our library and on our policy for software reuse. We feel that this is the best way for a company like ours to participate in the Ada culture.

## References


"Suggestions for Using and Organizing Libraries for Ada Program Development"
J.A. Goguen
SRI International
Menlo Park, CA
94Ø25
1983

"Automating Software Development: A Small Example"
S. Fickas
Symposium on Application and Assessment of Automated Software Development Tools.
1983

"Reusable Software Engineering: Concepts and Research Directions"
P. Freeman
Tutorial on Software Design Techniques
Fourth Edition
IEEE Catalogue Number EHO2Ø5-5
1983

"Some Practical Experience with a Software Quality Assurance Program"
G.G. Gustafson
R.J. Kerr
Communications of the ACM
Volume 25, Number 1
1982

"Contemporary Software Development Environments"
W.E. Howden
Communications of the ACM
Volume 25, Number 5
1982



Author: Randal Leavitt

PRIOR Data Sciences Ltd.
39 Highway 7
Nepean, Ontario
Canada
K2H 8R2

Telephone: 613 82Ø-7235


Mr. Leavitt is in charge of Ada application development at PRIOR. He is chairman of the Ada Working Group within the Canadian Standards Association, and has been actively involved in Ada evaluation, teaching, and consulting for several years. He has a B. Math degree from the University of Waterloo, and is an ACM and IEEE member. He also has a Certificate in Computer Programming from ICCP.

# DEBUGGING ADA[r] TASKING PROGRAMS[*]

Robert A. Conti

Digital Equipment Corporation

Symbolic debuggers permit a user to debug a compiled program in terms of the the original source code in which it was written. A symbolic debugger for Ada must also be able to cope with the multiple threads of execution represented by Ada tasks. This paper describes the typical kinds of bugs that a user of Ada tasks will encounter and lists a set of requirements for a symbolic debugger. Finally, the special commands and features that have been developed for VAX DEBUG (the VAX/VMS (tm) symbolic debugger) are presented.

## Introduction

VAX DEBUG is the multi-language debugger that executes under the VAX/VMS operating system. At the time of this writing, VAX DEBUG supports the Ada, Pascal, FORTRAN, COBOL, MACRO, C, PLI, RPG, and BLISS languages. Support for Ada was incorporated into the debugger concurrently with the development of the VAX Ada compiler. Much work had to be done because Ada is different in many ways from languages that the debugger previously supported. Special features were needed for Ada's tasks, packages, subunits, overloaded subprograms, attributes, and exceptions.

This paper discusses only those debugger features developed to support Ada tasking. The three major sections of this paper, respectively, describe the typical kinds of bugs and problems that occur when programming with Ada tasks, list requirements on the debugger, and describe the debugging commands that aim to satisfy the requirements.

## Typical Ada Task Debugging Problems

Clearly, it is important that a debugger be able to help with the most frequently occurring bugs. In this section we review what, in our experience, seem to be the more frequent kinds of tasking bugs and show how they generate requirements for the debugger. Note, we are not faulting the Ada language in any sense here -- tasking is just another language construct with its own characteristic set of bugs, just as infinite looping is a bug characteristic of while loops.

### Deadlocks

Ada tasks can deadlock in many ways. (In this paper we use the term deadlock loosely, to mean that one or more tasks of the program are waiting forever. Our use of the term is wider than its usual definition which is restricted to the existence of a "circular wait").

Probably the most common bug when using tasking is a deadlock initiated by an unanticipated exception. Ada rules require that a task that propagates an unhandled exception must first wait for

---

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

tm VAX and VMS are trademarks of Digital Equipment Corporation.

* The information in this paper is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation.

its dependent tasks, and then must terminate. It is often the case that a reading of the program indicates that the program is deadlock free, but the reader has overlooked the possibility that some exception can occur at run-time and cause deadlock.

A simple illustration of deadlock due to unanticipated exception follows. In this example, one task handles commands from the terminal, and another performs computations in parallel. As the program is written, there appears to be no deadlock. The procedure INTERACTIVE_SOLUTION waits at its end for both tasks to terminate. The terminal handler initializes for the problem, and then starts the background compute task by calling its entry START. After proceeding with further work, both tasks terminate and the procedure completes.

Unfortunately, if an exception occurs in the region labelled "initialize for problem", a deadlock will result. If such an exception were to occur, TERMINAL_HANDLER would propagate the exception and then terminate. Task BACKGROUND_COMPUTE, however, is programmed to wait unconditionally at the accept for entry START. The premature termination of TERMINAL_HANDLER prevents the expected call to that entry. Ada rules require that a procedure such as INTERACTIVE_SOLUTION must wait for its dependent tasks to terminate. Thus, if such an exception were to occur, the procedure will wait forever for BACKGROUND_COMPUTE to terminate, and BACKGROUND_COMPUTE will wait forever at its accept.

```
procedure INTERACTIVE_SOLUTION is

    task TERMINAL_HANDLER;
    task BACKGROUND_COMPUTE is
        entry START;
    end;

    task body TERMINAL_HANDLER is
    begin
        -- ... initialize for problem

        BACKGROUND_COMPUTE.START;

        -- ... talk to the user, etc.
    end;

    task body BACKGROUND_COMPUTE is
    begin
        accept START;
        -- ... do the work here
    end;

begin
    null;
end;
```

This example points out some Ada-specific debugging needs. First, the termination of a task by exception, while "simply another Ada rule", surely seems to be an unusual event. It would be nice if we could tell the debugger to let us know if this ever happens. Second, at the time of the deadlock, the program simply stalls with TERMNAL_HANDLER terminated, and BACKGROUND_COMPUTE and the caller of INTERACTIVE_SOLUTION in wait states. If the debugger could show us the state of these tasks (i.e. terminated, waiting, and waiting, respectively) it would help. If the debugger could do more, and show us the detailed reason for each wait, including the name of the entry (waiting for dependents, waiting at accept of START), that would even be better. It turns out that satisfying the simple needs of this example goes a long way to aid task debugging.

Another set of deadlocks arises from the Ada rule that propagation of an exception must wait for dependent tasks.

This is illustrated by the following program. In this example, a NUMERIC_ERROR exception is raised by the computation 3/Z, a divide by zero. As in the previous example, the exception leads to a deadlock by causing an entry call to be omitted.

```
procedure THIS_IS_MAIN_PROGRAM is

    task PARALLEL_COMPUTE is
        entry START;
    end;

    X : integer;
    Z : integer := 0;

    task body PARALLEL_COMPUTE is
    begin
        accept START;
        -- ... do work here
    end;


begin
    X := 3/Z; -- raises exception
    PARALLEL_COMPUTE.START;
end;
```

What is different in this case is that the exception never propagates. This is because Ada rules state that the exception must wait for the termination of all dependent tasks before propagating. The exception propagation is held up until task PARALLEL_COMPUTE terminates. This example points out a need to have the debugger show us when a task (in this case, the environment task that is automatically created to call the main program) is waiting for dependent tasks because of an exception.

Another kind of deadlock is associated with errors in calls to an entry which is a member of a family of entries. An entry in an family is used by specifying its index on the entry call and also on the accept. If the computations on each side of the rendezvous produce different values, a deadlock will result. The following fragment illustrates such a deadlock.

```
BASE_INDEX : integer := 1;

-- the call in task A:
B.SOME_FAMILY(BASE_INDEX + 2) (P)

-- the accept in task B:
accept SOME_FAMILY(BASE_INDEX + 3) (P)
```

No rendezvous can occur because the call is to entry family number 3, but the accept is for entry family member 4. This kind of bug can be hard to detect by reading the program if the index calculations are complicated. Therefore, another debugging need is the ability to inspect the index value for any task suspended at an entry call or accept for an entry in a family.

Another set of problems arises from select statements. Run-time calculations can be specified for the entry indices, when conditions, and delay statements of a select statement. In addition to previously mentioned deadlock because of wrong family indices, deadlock can also result from a condition being permanently computed as FALSE. Excessive delays can result if the delay expressions are incorrect. Ada-specific debugging help is needed here to make condition values and delay expressions readily available.

Another problem related to select statements is their non-deterministic nature. If several accept alternatives are open and callers are enqueued on each, the choice of which rendezvous to accept is arbitrary (up to the run-time system). When a task is waiting at a select, the debugger should provide a means of suspending execution before any statements of the chosen accept alternative are executed.

Other cases of deadlock arise from:

1. Entries in a task being called in the wrong order. For example, task A waits forever on a call to entry ONE, but task B executes an accept for entry TWO.

   Again, the debugger should display entry names when tasks are suspended at an entry call or accept.

2. A task not being programmed to terminate. Perhaps the task goes into an infinite loop. By Ada rules, the task that is its master cannot terminate.

   Here, about all we can ask the debugger to do is show when a task is waiting for its dependents and show the current statement being executed by any task.

3. Busy-waiting on a variable used as a flag that is to be set by a lower priority task, which never runs because some higher priority task is always ready to execute. This kind of deadlock is a bit more "dynamic" than others because tasks remain in compute states and don't suspend.

   The debugger should allow a running program to be interrupted asynchronously. It should allow one to find out what statement any given task is currently executing. It should also allow changing of priorities so a correction can be implemented and tried without the need to recompile.

## Other Tasking Problems

Other kinds of tasking problems are: non-repeatable execution, races, loss of access to a task, task starvation, stack overflow, and excessive context switching.

Non-repeatable execution can occur if tasks in the program execute delay statements, or do asynchronous I/O that depends on some external hardware device, or if time-slicing [see note below] is enabled. This can be especially pronounced while debugging if the execution of the debugger slows the program down relative to external events. The debugger should allow a user to force any execution order that might occur naturally. The user should be able to prevent a task from executing at any time (ignoring any asynchronous events directed at that task, or waiting for events to be delivered to some other task), and be able to switch control to any task that is

---

Note: Time-slicing is not required by the Ada language. VAX Ada by default uses FIFO scheduling, and provides a pragma TIME_SLICE to enable round-robin scheduling and specify the time quantum.

eligible to execute (allowing the task to acknowledge the asynchronous event). Combining these capabilities with being able to execute commands at breakpoints seems sufficient to force any desired execution order that can occur in practice.

A race is the accessing of an object that is shared between two tasks in the wrong order, or by both tasks concurrently, because the tasks are not properly synchronized. When there is only one physical processor, a suspected race might be verified if a user is able to change task priorities while debugging, thereby causing one task to execute in preference to the other. Alternatively, commands that force explicit task switching and suspension may be helpful. Another way to detect a race is to be able to set a "watchpoint" on the shared data. A watchpoint is a way for a debugger to be invoked on any attempt to read or write the data. The debugger could then show which task was executing at each reference.

Loss of access to a task can occur when one has declared an access type designating a task. After the program has assigned an access variable (pointer) to point to one task, the program may then assign it a different value. Ada rules state that the task can continue to execute even though the access variable has been reassigned, or the scope declaring the access variable has been left. Clearly, the user needs a way to name such tasks on debugging commands independent of the use of program variables. Some unique and universal way of naming tasks is needed.

Task starvation can occur when higher priority tasks prevent a lower priority task from gaining access to the processor for long periods. This can be detected if the debugger can find out how much "CPU time" a given task has received, or a count indicates how often a given task has run. Being able to change priorities would allow the starved task to execute.

When an implementation uses fixed length stacks for tasks, stack overflow can occur. The debugger should display the amount of stack space currently consumed by a task and how much total space is available. The debugger should also automatically monitor and report if stack overflow is imminent.

Excessive task switching may result from the way the program is designed. This can arise from the improper assignment of priorities, or specifying too short an interval for the scheduler time slice. This illustrates a need to maintain statistics such as the total number of context switches. Statistics can be used both to improve performance and to find subtle bugs.

## Requirements for the Debugger

A symbolic debugger for Ada must allow a user to both observe and modify program behavior. Ideally, to the maximum extent feasible, the simple act of observing a program should not also modify its behavior.

A symbolic debugger must, of course, offer many other features not directly related to tasking, such as the ability to display source program lines as the program executes, to use names of program objects rather than merely their addresses, to cope with overloaded subprograms, etc. Here, we shall concentrate only on debugging requirements pertaining to Ada tasks.

A proposed set of requirements on a debugger for Ada tasks follows. The debugger should:

1. Provide a way to uniquely identify tasks in debugging commands that is independent of variable names.

2. Display a detailed reason why a task is suspended.

   To merely show that a task is suspended, or to display the program counter is not as useful as to give a more detailed reason, such as, "suspended for an entry call", "suspended for a delay", etc.

3. Show the name of the entry and the index for a task suspended on an entry call or accept of an entry family.

4. Show the state of when conditions, delay values, entry indices, and entry names for a task waiting at a select statement.

5. Show the amount of stack currently consumed, and the amount of stack space available. Automatically watch for impending stack overflow. (This applies primarily to implementations that allocate a fixed amount of space for the stack at the time a task is created).

6. Provide a way to select, for display, a subset of all tasks in the program based on priority and scheduling state.

   In some applications, there may be hundreds of tasks in a program. Clearly, an easy way is needed to determine which tasks of a given priority are in the ready state (and thus might take control of the processor). It's also informative to know what tasks are terminated, suspended, etc.

7. Provide statistics on the number of tasking-related operations executed, such as context switches, entry calls, accepts, etc.

   Providing a report on the total number of context switches can help a user learn how modifications might increase or decrease overhead. These numbers are readily available to the tasking run-time system, but very difficult for the user to obtain otherwise.

8. Be able to restrict all but a chosen task or set of tasks from the processor.

   Putting tasks "on hold" allows one to debug a task or task set in isolation without interference from other tasks that might change the state of the program, or, worse, abort the task being debugged.

9. Be able to observe and modify any variable in any task, at any time.

10. Be able to control the scheduling discipline (round-robin or FIFO) while debugging.

11. Allow changing priorities.

12. Help the user cope with non-repeatability by allowing a user to generate all possible execution orderings.

    It must be possible to: place any task on hold, switch control to any ready task, set breakpoints, and execute debugging commands at breakpoints. These appear to be sufficient to reproduce any possible natural ordering.

    It must also be possible to disable round-robin scheduling.

    It is advisable to attempt to eliminate unnecessary sources of non-repeatability within the run-time system.

13. Provide a way to invoke the debugger on various unusual run-time events, such as termination of a task by unhandled exception.

    There should be a way for the user to tell the debugger to perform some action when one of these events occurs, such as halt the program.

14. Detect deadlocks when they occur, or upon demand.

15. Measure execution progress, such as CPU time, in each task.

## Debugging Commands

In attempting to satisfy the above requirements it was necessary to define only a handful of new commands (plus qualifiers) for VAX DEBUG.

A command SHOW TASK has been defined for the purpose of observing task states, and a command SET TASK has been defined in order to modify task states. The existing breakpoint and tracepoint commands (SET BREAK and SET TRACE) have been modified so that interesting tasking events can invoke the debugger. (NOTE: A breakpoint suspends program execution and causes the debugger to prompt the user for debugging commands. A tracepoint merely displays a message and continues execution. VAX DEBUG provides numerous additional options, including automatic execution of a command sequence when the event occurs).

## Preliminary Definitions

A user must be allowed to examine the variables of a task that is not currently executing. To provide such visibility, a command was defined to allow a user to make any task the default task for the debugger's commands that observe and modify variables. This task is called the visible task.

The task which is running on the processor is called the active task. When the debugger is invoked, the visible task is made the same as the active task. Using debugging commands, the user can change which task is visible (can be observed and

modified), and which task is active (will execute next).

Requirement 1 above points out the need to generate a unique way of referencing a task. To satisfy this, the Ada run-time system increments a counter each time a task is created by the program. A construct called a "task ID" is defined by VAX DEBUG. A task ID has the format %TASK n, where n is the count. A user can use a task ID, as well as a task object name, on any task debugging command.

The debugger has also defined some useful task-valued functions. The functions %ACTIVE_TASK, %VISIBLE_TASK, and %CALLER_TASK evaluate to the active task, visible task, and the calling task in a rendezvous, respectively. These are especially useful in conditionally executing debugger commands. To illustrate, the following command sets a breakpoint on line 10 such that the breakpoint is triggered only when the executing task is named DRIVER. Without this feature trying to debug a subprogram called by hundreds of tasks could be pretty tedious!

```
SET BREAK %line 10
      WHEN (%ACTIVE_TASK = DRIVER)
```

## The SHOW TASK Command

*The SHOW TASK command* has several qualifiers. Without qualifiers, it illustrates the current state of the visible task, as illustrated in Figure 1.

In the display of Figure 1,

| | |
|---|---|
| * | indicates the task is the active task. |
| task ID | is the unique ID for the task. |
| pri | is the task's current priority. |
| hold | indicates if the task has been placed on hold. |
| state | is the language-independent task state. This can assume the values RUN, READY, SUSP, TERM, CREA for running, ready, suspended, terminated, and created. |
| substate | is the Ada-specific state information. |
| task object | is the name of the task object in the program. |

The substate field can assume any one of a long list of values. A suffix [exc] and [abn] are appended to indicate if the state was obtained because of an exception or abort. The full list of substates and explanations follows:

| | |
|---|---|
| Abnormal | Task has been aborted. |
| Accept | Task is waiting at an accept statement. |
| Activating | Task is elaborating its declarative part. |
| Activating tasks | Task is waiting for tasks to finish activating. |
| Completed [abn] | Task is completed due to an abort statement, but not terminated. |
| Completed [exc] | Task is completed due to an unhandled exception but not terminated. |
| Completed | Task has completed normally. |
| Delay | Task is suspended at a delay statement. |
| Dependents | Task is waiting for dependent tasks to terminate. |
| Dependents [exc] | Task is waiting for dependent tasks because of an unhandled exception. |
| Entry call | Task is waiting at an entry call. |
| I/O or AST | Task is waiting for I/O completion or software interrupt. |
| Not yet activated | Task is waiting to be activated. |

```
SHOW TASK

   task id      pri hold state   substate          task object
 * %TASK 3       7       RUN                    EXAMPLE.PRODUCER
```

Figure 1. The SHOW TASK Display

| | |
|---|---|
| Select or delay | Task is waiting at a select statement with delay alternative. |
| Select or term. | Task is waiting at a select statement with terminate alternative. |
| Select | Task is waiting at a select statement with no else, delay or terminate alternative. |
| Shared resource | Task is waiting for some shared resource. |
| Terminated [abn] | Task terminated by abort. |
| Terminated [exc] | Task terminated by unhandled exception. |
| Terminated | Task terminated normally. |

A command qualifier /ALL can be used to obtain a brief display of all tasks currently in existence. This is illustrated in Figure 2.

The qualifier /FULL causes more detailed information to be displayed, as shown in Figure 3.

In Figure 3 we see detailed information about the task. Information is displayed about the task's waiting for a rendezvous, its type and creation, its task control block, and its stack usage. The rendezvous information satisfies requirements 3 and 4 regarding the display of details about entry indices, delay values, etc. For example, in Figure 3 we see that the entries in the select are named and entry index values are displayed.

Qualifiers /PRIORITY = n, /STATE = s, and /HOLD can be used separately or in combination. They restrict the set of tasks that will be displayed to only those that satisfy all these selection criteria. For example, SHOW TASK/PRI=7/STATE=READY/NOHOLD, displays all tasks of priority 7 that are in the READY state and have not been placed on HOLD (SET TASK/HOLD is discussed later).

The qualifier /STATISTICS changes the nature of the SHOW TASK display. Instead

```
SHOW TASK/ALL

   task id      pri hold state   substate          task object
   %TASK 1       7       SUSP  Dependents         121036
   %TASK 2       7       SUSP  Select or term.    EXAMPLE.WORKER
 * %TASK 3       7       RUN                      EXAMPLE.PRODUCER
```

Figure 2. The SHOW TASK/ALL Display

```
SHOW TASK/FULL %TASK 2

  task id      pri hold state   substate          task object
  %TASK 2       7       SUSP  Select or term.    EXAMPLE.WORKER

        Awaiting rendezvous at: select with terminate.
        The select has 4 arms.
        When    Alternative, 'VAL(index)     Do Part  Next Stmt
          true  PRIORITIZED_WORK(i), 4       0000061E 000006DF
          true  MORE_WORK                    00000624 000006DF
          false PRIORITIZED_WORK(i), 3       0000062A 000006DF
          true  Terminate

        Task type:       WORKER
        Created at PC:   EXAMPLE.%LINE 8
        Parent task:     %TASK 1
        Start PC:        EXAMPLE.WORKER$TASK_BODY
        Task control block:              Stack storage (bytes):
          Task value:    1104528           RESERVED_BYTES:        3072
          Entries:       13                TOP_GUARD_SIZE:        5120
          Size:          1598              STORAGE_SIZE:         30716
        Stack addresses:                   Bytes in use:           352
          Top address:   1155584
          Base address:  1186300           Total storage:        40506
```

Figure 3. The SHOW TASK/FULL Display

of displaying the state of a particular task, it displays global state information, most important of which is the number of context switches that have been performed. This is illustrated in Figure 4.

Other qualifiers are as follows:

| | |
|---|---|
| /TIME_SLICE | Displays the number of seconds in the round-robin scheduling interval. |
| /CALLS | Displays the name of each routine called by the task and the current line number in that routine. |

We have shown that the SHOW TASK command satisfies requirements 2 through 7.

## The SET TASK Command

The SET TASK command allows modifications of certain attributes of tasks.

The qualifier /ACTIVE switches the active task (the task that is currently executing). Like any VAX DEBUG command, it can be used in conjunction with a breakpoint command. The following command illustrates some of the power of VAX DEBUG. It sets a breakpoint on line 30. The breakpoint is honored only when the active task is Y. If so, a task switch is performed (from Y) to T3.

```
SET BREAK %line 30
   WHEN (%ACTIVE_TASK = Y)
      DO (SET TASK/ACTIVE T3)
```

The qualifier /VISIBLE is the one SET TASK command that doesn't really modify the behavior of the program. This command is used to make another task visible for debugging commands. For example, the following command sequence examines a variable VAR1 in task T3 while some unknown task is active, then it restores visibility back to the active task.

```
SET TASK/VISIBLE T3
EXAMINE VAR1
SET TASK/VISIBLE %ACTIVE_TASK
```

The qualifiers /HOLD and /NOHOLD allow one to place any or all tasks on hold. A task on hold will not be permitted to run. SET TASK/HOLD can be used to keep other tasks from interfering while debugging a particular task. Examples of its use are:

```
Put all tasks on hold:
SET TASK/HOLD/ALL

Release only the task numbered 4:
SET TASK/NOHOLD   %TASK 4
```

The qualifiers /PRIORITY and /RESTORE complement each other. To illustrate, the following command sequence sets the priority of task T to 8 and then restores its natural (declared) priority.

```
SET TASK/PRIORITY = 8 T
SET TASK/RESTORE      T
```

The /RESTORE qualifier eliminates the need to remember the task's original priority.

Other qualifiers for SET TASK are:

| | |
|---|---|
| /ABORT | Abort some task |
| /TIME_SLICE= t | Change the round-robin time-slice interval; a 0 value causes FIFO scheduling. |

The SET TASK command satifies requirements 8 through 12.

```
SHOW TASK/STATISTICS

task statistics
    Entry calls       = 2       Accepts = 2     Selects = 1
    Tasks activated   = 2       Tasks terminated   = 0
    ASTs delivered    = 7       Hibernations       = 5
    Locks tested      = 39      Locks that blocked = 12, 30%
    Total schedulings = 19
        Due to task activations                    = 1
        Due to suspended entry calls               = 2
        Due to suspended accepts                   = 2
        Due to suspended selects                   = 1
        Due to waiting for a DELAY                 = 5
        Due to scope exit awaiting dependents      = 1
        Due to delivery of an AST                  = 7
```

Figure 4. The SHOW TASK/STATISTICS Display

## Commands for Detecting Tasking Events

It is desired to invoke the debugger upon detection of events known only to the run-time system, such as termination of a task by unhandled exception.

One would like to be able to halt the program as well as pause it and execute commands. The existing command to halt the program is SET BREAK. For example, SET BREAK %line 10, halts the program before line 10 is executed. The command to pause the program is SET TRACE. The normal parameters to these commands are program addresses.

These commands have been modified by adding a /EVENT qualifier. The event qualifier allows the break or trace action to occur not when an address is reached, but when a particular run-time event occurs. For example, the following command causes the debugger to be invoked when any task terminates by unhandled exception. When the debugger is invoked by such an event, the command SHOW TASK is executed to display which task is terminating. Since a breakpoint is requested, the program halts after the display, and remains in the debugger.

```
SET BREAK/EVENT=EXCEPTION_TERMINATED
   DO (SHOW TASK)
```

There are numerous tasking run-time events that can be detected. The list of event names and definitions appears in table 1.

The /EVENT qualifier satifies requirement 13.

## Possible Future Commands

Of the requirements list, only requirements 14 (automatic detection of deadlocks) and 15 (measure task progress) have not been met by one of the commands described above. We hope to fill this gap in the future.

Deadlock detection can be either continuous or on-demand. With continuous detection, considerable run-time overhead is incurred to ensure that a deadlock is detected as soon as it occurs. With on-demand detection, a small amount of continuous overhead is incurred, but the user must specifically request an analysis for deadlocks.

| | |
|---|---|
| RENDEZVOUS_EXCEPTION | Triggers when an exception begins to propagate out of a rendezvous. |
| DEPENDENTS_EXCEPTION | Triggers when an exception causes a task to wait for dependent tasks in some scope. |
| TERMINATED | Triggers when a task is terminating, whether normally, by abort, or by exception. |
| EXCEPTION_TERMINATED | Triggers when a task is terminating due to an exception. |
| ABORT_TERMINATED | Triggers when a task is terminating due to an abort. |
| RUN | Triggers when a task is about to run. |
| PREEMPTED | Triggers when a task is being preempted from the RUN state. |
| ACTIVATED | Triggers when a task is going to run for the first time. |
| SUSPENDED | Triggers when a task is about to be suspended. |
| READY | Triggers when a task has become ready to run. |

TABLE 1. Event Names and definitions

Both forms could be implemented within the framework outlined earlier. The first command to follow would invoke the debugger when the next deadlock is detected. The second command would analyze the current program state and report all deadlocks.

```
SET BREAK/EVENT=DEADLOCK

SHOW TASK/DEADLOCK
```

Measuring task progress could require a relatively expensive operation be performed at each task switch. To minimize the impact of this monitoring on programs not needing such monitoring, a command is needed to enable or disable the monitoring. This could be done via a SET TASK/MONITOR=CPU_TIME. If this command were invoked for some task, the elapsed CPU time would appear in the SHOW TASK display.

## Implementation of the Commands

The run-time cost associated with these commands has been very small.

Much of the information displayed by SHOW TASK was already embedded in the tasking run-time system. A few additional instructions had to be added, for example, to specify a reason code for a task suspension. Most of the overhead, for example, converting binary to symbolic format, and much arithmetic, occurs only while debugging.

For SET TASK, it was necessary to add some tests of a few master flags, that mean for example, "DEBUG changed some scheduling info", or "some event is enabled". Only if the master flag is found to be set is more detailed code executed to fully analyze the particular situation.

## Conclusion

This paper has described typical bugs that occur when writing programs using Ada tasks. A set of requirements for a debugger of Ada tasks is listed. Finally, a set of debugging commands to address these requirements has been described. We have shown how a few simple commands can add tremendous power in the ability to debug Ada tasking programs.

## Acknowledgments

The author would like to acknowledge the contributions of other members of the VAX Ada compiler project and the VAX DEBUG project toward this effort.

## References

[1] Ada Programming Language, ANSI/MIL-STD-1815A, U.S. Government, 10 December 1980.

[2] Developing Ada Programs on VAX/VMS, Digital Equipment Corp., Maynard, Mass., 1985.

[3] VAX Ada Programmer's Run-Time Reference Manual, Digital Equipment Corp., Maynard, Mass., 1985.

[4] VAX Ada Language Reference Manual, Digital Equipment Corp., Maynard, Mass., 1985.

Robert A. Conti is a member of Digital Equipment Corporations's VAX Ada compiler development team. His responsibilities included the implementation of tasking, task debugging, and general Ada debugging. Prior to joining Digital, Mr. Conti worked at Westinghouse Electric Corporation on software for several military programs, most notably AWACS. He received the BS in Engineering from Case Western Reserve University, the MS in Electrical Engineering from Johns Hopkins University, and the MS in Computer Science from the University of Maryland. He is a member of the IEEE and ACM.

The author's address is: Digital Equipment Corporation, ZK2-3/N30, 110 Spitbrook Rd., Nashua, NH 03062.

# THE ADA* LANGUAGE SYSTEM

Dennis J. Turner

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

The Ada Language System (ALS) is an integrated, rehostable, retargetable and extensible programming environment for the Ada language. Significant benefits are expected to be derived through extensive use of the ALS as a common environment across Army Battlefield Automated Systems (BASs). This paper provides background on ALS activities to date, current status and future plans.

## Introduction

A great deal of attention has been given in recent years to the growing costs which can be directly associated with the proliferation of programming languages and computer hardware within Army Battlefield Automated Systems (BASs). However, there is another aspect of BAS proliferation which is considerably less visible but, is also a source of significant cost.

Every BAS, whether it includes a large mainframe computer or a deeply embedded microprocessor, has an associated support computer and a set of computer programs which collectively comprise the so-called "support system." The term support system is something of a misnomer because, while it would seem to refer to a post-deployment period, it actually refers to the entire cycle of a BAS.

Figure 1 illustrates the relationship between the support (or host) computer and the fieldable (or target) computer. Programmers use the support computers to prepare and integrate programs which will ultimately execute on the target computer. Program preparation can include text editing, language translation, linking, simulation and a variety of other activities including configuration management. The separation of activities across these systems is dictated by the different demands which are placed on the computers. Target computers are designed and configured to accomodate the needs of Army field users and of the mission environment. These needs are typically very different than those of programmers, who work in a laboratory environment and who require interactive terminals, large amounts of primary and secondary memory and very high speed peripheral devices.

Support systems typically are comprised of a general purpose commercial mainframe computer, an assortment of peripheral devices, a vendor supplied operating system and a collection of software (some vendor supplied, some government developed) which programmers use to develop application software for the target computer. The support programs, (such as text editors and language translators) which programmers use to develop applications software, are often referred to as "tools." Just as a carpenter uses a variety of tools to build a house, a programmer uses a variety of (software) tools to design and implement application programs.

The life cycle costs of a support system primarily are composed of:

a. The cost of maintaining the vendor supplied host computer hardware.
b. Costs for initial licenses and the recurring support costs for vendor supplied software (operating system and tools).
c. The cost to develop and maintain additional government required tools.

The life cycle costs of "b" and "c" can significantly exceed those of the mission program, depending on the extent of the tools and the size and complexity of the application.

An Army contractor typically chooses a support system as a function of the programming language, target computer and programming staff experience. With little Army constraint imposed on these selection factors, it is no wonder that we have found ourselves in a situation where nearly every BAS has a unique support system associated with it.

Even with the advent of the Ada programming language, there is little reason to expect a significant reduction in the proliferation of support systems, even though they may all share an Ada theme. In anticipation of the vast market across government, industry and academia, at least 35 U.S. and 15 foreign initiatives are now underway to produce Ada compilers and Ada support tools. Each of these initiatives represents a substantial

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

amount of software. For the most part, they will accomodate only a narrow range of computers and are considered to be proprietary in nature. Unless very deliberate steps are taken to constrain the choices that can be made for use on Army systems, it is likely that a large number of these very distinct compilers, tools and host computers could become associated with Army BAS (see Figure 2).

In anticipation of this difficulty, the U.S. Army Communications-Electronics Command (CECOM) set out to develop a solution.

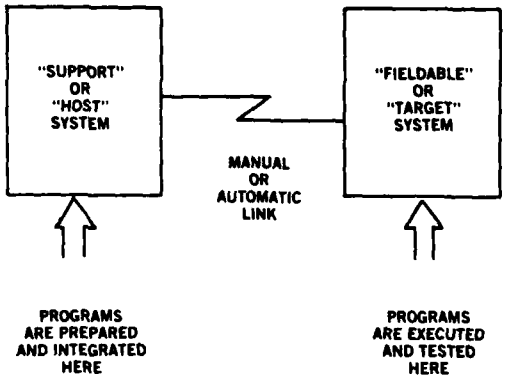



Figure 1

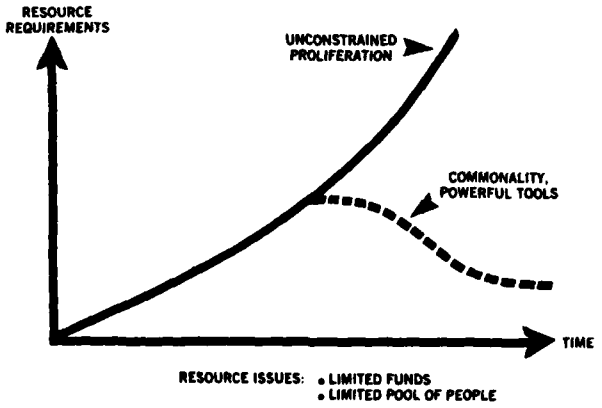



Figure 2. Support environment.

## Key Characteristics

In June of 1980, CECOM awarded a contract to Softech, Incorporated to develop an Ada Language System (ALS) which could satisfy three primary goals: reduce the proliferation of support environments for Army BASs; improve the productivity of programmers; and improve management control over the software life cycle.

The ALS has several important characteristics that serve as the basis for meeting these objectives: integrated environment, rehostable, retargetable and extensible.

Each of these characteristics now will be described in greater detail.

### Integrated Environment

The ALS is first and foremost an "environment." This is a commonly used term which refers to a support system which consists of a large variety of tools to assist programmers in a wide range of activities. Most of the current U.S. and foreign Ada initiatives seek to develop only a compiler. Tools beyond support of language translation typically are not included in these initiatives. The ALS includes a rich set of powerful tools which support activities such as command processing, data base management, language processing, program analysis, configuration control, text processing, file operations and other miscellaneous activities.

In order to appreciate totally he notion of an integrated environment, one must have had some experience with more traditional systems, where tool design has been approached in a manner that can best be characterized as ad hoc. In these systems tools ahve been developed independently of one another, with no common design philosophy or objectives. As a result, they are difficult to use, modify and expand.

Figure 3 illustrates the integrated nature of the ALS. Here, all tools communicate with the user through a common and friendly command language processor. "Inter-tool" communication occurs through a common data base and through standard interfaces. The result is a cooperating system of integrated tools which are easy to use and which can be modified and expanded in a straightforward manner.

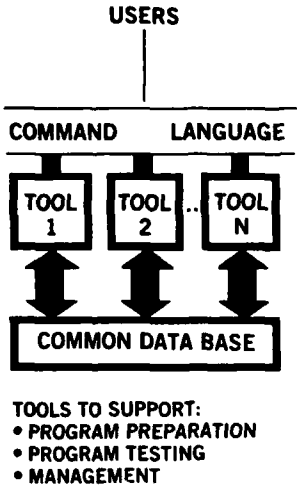



Figure 3. ALS characteristics: integrated.

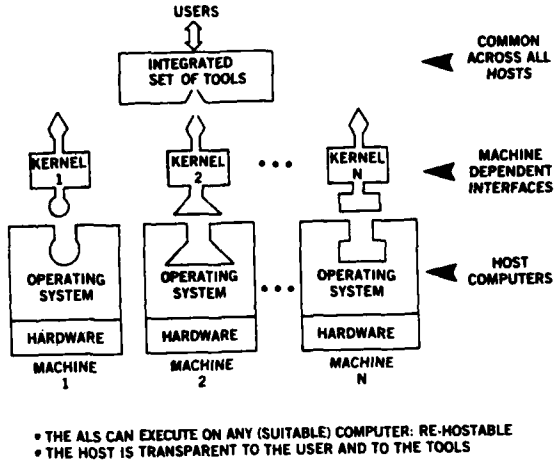



Figure 4. ALS characteristics: "rehostable."

Rehostable

A support environment which can be moved (transported) from one host computer to another with a minimum of difficulty is said to be "rehostable." The ALS achieves its rehostability through two primary characteristics. First, all of the tools are written in the Ada language and need only be recompiled for a new host. Secondly, as illustrated in Figure 4, the tools to not communicate directly with the host operating system. Instead, all tools communicate with a "kernel" which maps the tool interfaces into the services through the underlying operating system. This approach "decouples" the tools from any dependency on the operating system or the host hardware (computer).

In order to rehost the ALS, one only needs to implement a kernel for he new host. The tools (which represent the bulk of the system) can be moved without modification. A second advantage to this approach is that the underlying operating system and host hardware is transparent to the ALS users. Since users communicate only with ALS tools and not with the host operating system, they see the same interface, independent of what the host may be.

Rehostability is a particularly attractive feature of the ALS because it can be used to accommodate concerns for hardware competition and for hardware technology insertion.

Retargetable

Most support environments are developed to accommodate a particular target computer or, at most, a narrow family of target computers (typically representing the products of the associated vendor). A host environment which can accommodate an arbitrary set of target computers is said to be "retargetable." The ALS is such an environment.

As illustrated in Figure 5, the tools of the ALS can be divided into two categories: those that contain dependencies on the target computer (e.g., compiler, assembler, linker, debugger, etc.) and those that do not (e.g. text editor, configuration control tools, command processor, etc.).

The most significant target dependent tool is the Ada compiler itself. The compiler translates Ada source programs into the instructions that are understood by the target computer. In the ALS, the compiler has been deliberately designed to consist of two primary pieces. The first, called the "front-end," performs lexical and syntactic analysis and translates the Ada source code into an intermediate language representation called DIANA. The second piece, called the "back-end," takes the DIANA representation (which is independent of target computers) and performs semantic analysis, optimizations and ultimately produces a program which can be executed on the desired target computer.

Only the back-end of the compiler contains target computer dependencies. With this approach, it is possible to associate a single front-end with multiple back-ends, where each back-end is tailored to the characteristic of a particular target machine.

In addition to the basic structure of the compiler, all the target dependent tools have been carefully designed to isolate those dependencies and to place them in tables (as data) rather than in

algorithms (as code).

The design approach, taken in the ALS for retargeting, makes it possible to accommodate an arbitrary number of target computers. This is a necessary characteristic if the ALS is to succeed as a common environment.

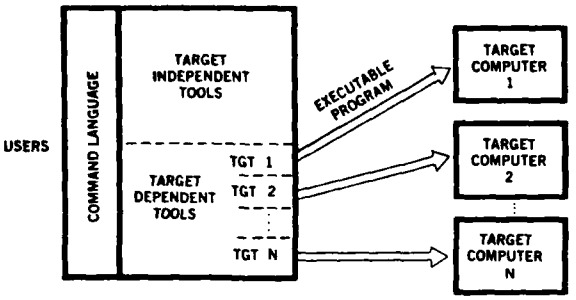
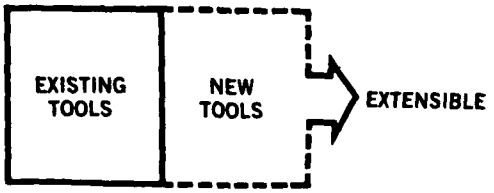



Figure 5. ALS characteristics: retargetable.

## Extensible

Extensibility, as depicted in Figure 6, refers to the ability of an environment to accept additonal tools as new requirements are identified and as technology advances. The extensible nature of the ALS is derived from the standard interfaces within which the tools function. Given a clear definition of how a tool must communicate with the user, another tool or the kernel (recall that the kernel maps tool requirements into the services provided by the host operating system), it is a very straightforward matter to either modify a tool or add a new one.

In this day of rapid technology advances, no support environment can hope to survive for very long if it cannot keep pace with those advances. The extensible nature of the ALS has been designed to meet this need.




- **INTERFACE REQUIREMENTS WELL DEFINED**
- **CAN BE WRITTEN EITHER IN ADA OR (POWERFUL) COMMAND LANGUAGE**

Figure 6. ALS characteristics: extensible.

## Recent Activities and Future Plans

The ALS is being developed to provide a comprehensive set of design and user documentation, in accordance with MIL-STDs 483, 490 and 1679. The system is government owned and written in Ada; Ada also has been used as a Program Design and Language (PDL). Training material, to include a System Administrator's course and a textbook, are also being produced.

One of the most significant statistics that can be associated with the system is the fact that it currently consists of over 500,000 lines of source code. This puts it "on par" with the software contained in some of the Army's major BASs. The size of the ALS is one of the more important reasons that the Army must begin to control the proliferation of support environments. We simply cannot afford to manage an arbitrary number of them.

Figure 7 illustrates the categories of tools which will exist within the ALS. There are currently some 75 distinct tools distributed across these categories.

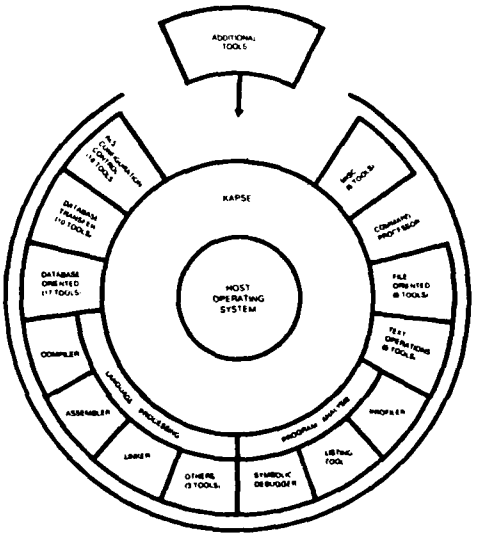



Figure 7. ALS components.

The ALS is currently hosted on a Digital Equipment Corporation (DEC) VAX-11/780 with the VMS operating system. The version of the compiler which is targetted to the host was validated by the Ada Joint Program Office (AJPO) in December 1984. Continuing development activities will produce targets for the Intel iAPX286 and a "bare" (no resident operating system) VAX by early 1986. Plans are also underway to retarget the ALS to the Motorola 68000 family. That effort is expected to be initiated before the end of 1985.

In the summer of 1983, an ALS Early Release Program was initiated to help insure the ultimate success of the product. Advertisements through the Commerce Business Daily offered an interim version of the ALS, including source code and documentation, to U.S. industry in exchange for commitments to pursue rehosting and/or retargeting activities. The response to this program was far greater than expected. Approximately 60 U.S. companies are participating in the program.

The advantages of the participating companies are obvious: they have received a significant amount of software at no cost and have an early opportunity to tailor it to their products. In this way they can establish a competitive position for future Ada based contracts.

There are several benefits for the Army. First, the Early Release Program will cause new hosts and targets to appear sooner than they would have otherwise. This can only help to accelerate the use of Ada in Army BASs. Secondly, it will establish a competitive base for future ALS extension activities. Thirdly, it is likely to focus some industry IR&D expenditures on the ALS.

There are a number of near-term activities which should be pursued to enhance the recently validated baseline system. First is the commitment which the Army has made to continued improvement of the performance of the system. Second is the need for continued development of new hosts and targets for the ALS. Third, a standard Ada program library needs to be developed and managed. This library will provide the mechanism for promoting the reuse of common Ada software. Another benefit from such a library would be that it could serve as a proving ground for functions (which have been monitored for stability and popularity) that are candidates for implementation in hardware (e.g. Very High Scale Integrated Circuits). Successful establishment of such an Ada package library involves both technical issues (how to specify such programs) and business concerns (how to motivate contractors to reuse software).

Other needs include:

a. the development of a more comprehensive set of environment tests;
b. the automation of training material;
c. the incorporation of intelligent work stations; and
d. technology to support distributed host and target considerations.

## Strategies for Use

An ideal strategy for the ALS would be one in which industry would be free to use any Ada support environment for BAS development purposes and then transition to the ALS for the post deployment period. This would maximize competition for development and still provide a common and affordable post deployment environment, where the impact of proliferation is felt the most. The obstacle to achieving this strategy is subtle but extremely important.

The difficulty has to do with the Ada compiler and other language processing tools which influence the performance characteristics of programs which ultimately execute in the Army mission environment. No two compilers translate a given source program in the same way. Further, no compiler is error free, and each compiler contains a different set of errors. It should not be difficult to see that a transition from one compiler to another will cause at least two problems:

a. Translated application programs will execute with a different performance characteristic.
b. Previously nonexistent errors will appear (because the source programs were designed to accomodate the errors of only the first compiler).

For these reasons a transition from one compiler to another after any major test could negate the conclusions derived from the test. An attempt to transition prior to testing defeats the whole purpose of pursuing a transition.

A technical solution to this problem does not currently exist. However, if certain technology advances were to occur, the tools which influence the performance of Army mission software could be approached in one of two ways:

a. One set which could "plug in" to any vendor's environment.
b. One set which could be accessed from any vendor's environment.

Until technology advances enough to provide these solutions, it would appear that the only affordable strategy is one which encourages the ALS for development and support of Army BAS. When sufficient justification exists, other environments could also be used but the preference would be clearly for the ALS.

## Request for Copies of the ALS

Request for copies of the Ada Language System, documentation or any of the Ada or ALS training material developed at CECOM should be directed to:

Commander, U.S. Army CECOM
ATTN: AMSEL-TCS-ADA
Fort Monmouth, NJ 07703-5204

## Summary

This paper has focused on the software support environments which are used to develop and maintain computer programs in Army BASs. Considerable proliferation and unnecessary costs have been incurred from past practices. In order to promote a greater degree of convergence in the future, the U.S. Army CECOM is developing the ALS.

Technology advances are required before it will be possible to exchange software across dissimilar support environments. Until those advances occur, the ALS is expected to become a common environment across Army systems.

## Biographical Sketch

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

# ADA IMPLEMENTATION IN A NON-ADA ENVIRONMENT

J. C. Helm

Ford Aerospace &
Communications Corporation

T. E. Cook

Ford Aerospace &
Communications Corporation

## ABSTRACT

At NASA's Johnson Space Center, existing software systems containing millions of lines of code are planned for continued use in projects that span the next decade. The software systems and tool sets that are under configuration control, are proven, reliable, and operable. For NASA, recovery of as many software systems as possible appears to be cost effective. Since Ada is mandated by the Department of Defense as the future language, and due to Ada's portability and life cycle cost reduction, it is being considered as a language for modification to existing software systems.

## INTRODUCTION

This paper investigates the feasibility of using Ada and newly developed Ada packages in existing non-Ada software environments. An existing group of software environments was identified. From these environments a feasibility criteria for implementing Ada code in a non-Ada environment were determined. The criteria consist of a set of parameters that were used to develop metrics. The metrics provided management with an Ada Decision Matrix to use in determining if new modules planned for integration into an existing non-Ada environment should be written in Ada.

Further, this paper demonstrates the need for interfacing Ada packages into existing non-Ada environments, which should result in interface or linkage mechanisms being provided as a part of the standard Ada tool set. The paper analyzes the interface mechanisms required for a selected software environment.

## TECHNICAL APPROACH

A systems analysis approach was taken to identify a set of decision parameters and establish a decision criteria. The particular decision analysis tool applied was the decision table or matrix (Reference 7). Two additional design concepts that were investigated before deciding on the decision matrix were the Decision Tree and Structured English methods.

The Decision Tree method diagrams conditions and actions sequentially showing the relationship and permissible action of each condition.

The Structured English method uses one of three basic types of statements to reach a decision. First, the Sequence structure is a series of single steps or actions necessary to reach a decision. Second, the Decision structure is used when two or more actions can occur depending on values specified for specific conditions. Third, the Iteration structure is used when certain activities are repeated while a given condition exists or until the condition occurs.

The decision analysis strategy and the decision matrix in particular was chosen because it identifies existing software and hardware conditions and suggests actions to be taken based on the conditions. The decision matrix establishes a decision criteria based on the actions and incorporates all the conditions to form a decision rule.

D. B. Baker developed an Ada Decision Matrix comprised of two parts, a worksheet, "Project Risk Potential in the Use of Ada", and a "Risk Priorities Matrix" (Reference 2). The decision matrix addressed three risk areas when considering the use of Ada for mission critical software: technical, acquisition and economic.

The decision matrix concept developed for this paper contains rows and columns that show the decision parameters and associated action statements. The action statements indicate selections to make when certain conditions exist. The matrix also contains action values and weights applied to the action statements. A decision rule is formed from the action values to establish a decision criteria.

To build the decision matrix the following steps were taken:

1) Determine the most relevant factors to be considered, that is, identify the condition statements for the decision parameters.

2) Determine the most feasible steps or activities that apply to each statement. These form the action statements.

3) Study the combination of action statements for each condition and assign appropriate weight.

4) Fill in the matrix with possible action statements.

5) Apply a rule with assumptions.

The decision matrix shown in figure 1 contains the condition statements, action statements, action values, and decision weights. The condition statement section identifies the relevant decision parameters. The action statement lists a set of events or selections that exist for each condition statement. The action values specify a range of choices and assigned weights that apply to each action statement for a condition. The decision weight column is filled in based on the value chosen from the action statement list.

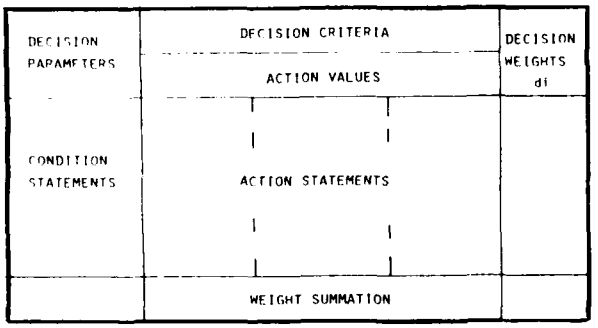



**Figure 1. Ada Decision Matrix - Form**

A decision rule is developed from the decision weight column. The formula for the decision is:

$$A = \frac{1}{n \times w} \sum_{i=1}^{n} d_i$$

where: n = number of decision parameters (12)

w = normalization weight factor (10)

d = decision weights

The decision rule plot is shown in Figure 2.

An assumption is if one or more of the decisions is zero, automatically reject Ada.

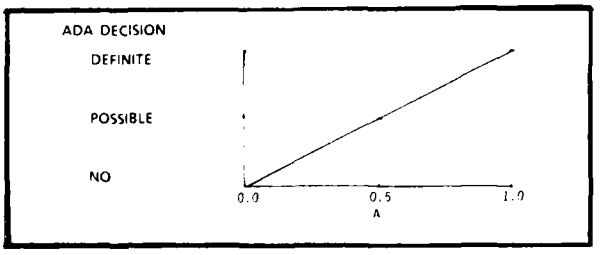



**Figure 2. Decision Rule Plot**

## ADA DECISION MATRIX

A systems analysis methodology using a decision matrix strategy was developed. To construct the matrix six NASA software environments were identified. Experts from each environment were interviewed and an on-site inspection made. Using the gathered information, a set of 12 decision parameters was established and used to construct the condition statements for the decision matrix. The decision parameters selected were:

- Existance of a validated Ada compiler to produce target code
- Coupling and cohesion of existing software environment
- Efficiency of compiler produced code in a real time environment
- Proven reliability of the compiler for [illegible] software environments
- Adaptability of the existing [illegible]
- Remaining software life cycle
- Independence of the software modification
- Potential reusability of the software
- Cost effectiveness of retraining
- Interface mechanisms
- Necessary or desired language features
- Other cost considerations

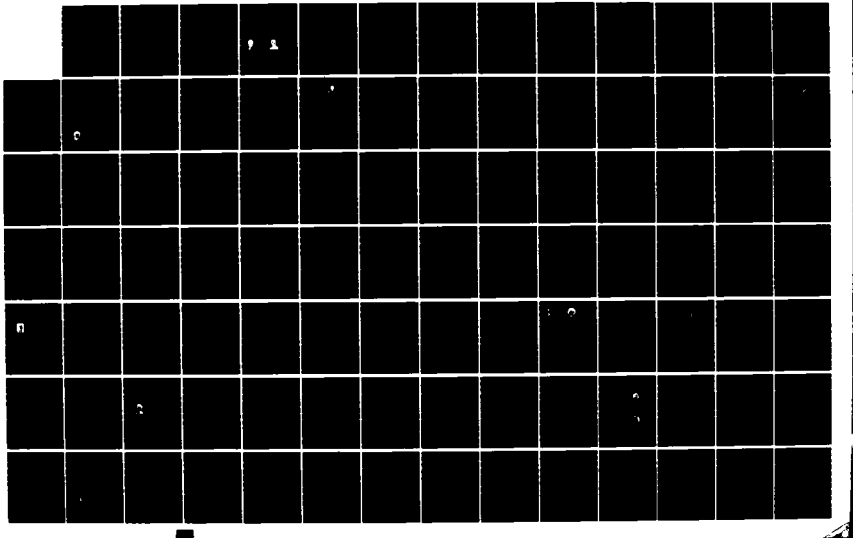
Each condition is further discussed to evaluate and clarify its potential contribution to the action statements and the decision rule.

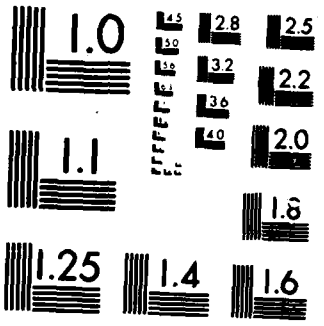
A major concern was the existance of a validated Ada compiler to produce target code. Several validated Ada compilers now exist that will allow formatting and transferring Ada program files from a host computer to a target compatible computer. This will not be a future concern due to the effort being expended by software vendors, however it must be researched.

The coupling and cohesion was a concern to existing FORTRAN software environments. Ada with its strong typing, abstraction, generic definitions and seperate compilation will simplify the previous problems encountered with module cohesion and modules coupled by common or global variables. Coupling is defined here as a measure of the strength of interconnection between one module and another. Common environment coupling is when two or more modules interact with a common data environment. Cohesion is the degree of functional relatedness of processing elements within a single module. The seven levels of cohesion are coincidental, logical, temporal, procedural, communicational, sequential, and functional (Reference 8). For existing software modules that are highly coupled it will be difficult to integrate new Ada modules. Also, if the existing software has no identifiable cohesive form, integrating an Ada module would not be practical

AD-A164 338 PROCEEDINGS OF THE ANNUAL NATIONAL CONFERENCE ON ADA (TRADEMARK) TECHNOLO.. (U) ARMY COMMUNICATIONS-ELECTRONICS COMMAND FORT MONMOUTH NJ CENT .. 1985 2/3

UNCLASSIFIED F/G 9/2 NL





In a real time environment analysis must be performed on the Ada compiled code to determine if the compiled code would degrade the efficiency of the computer. The following equation could be used:

$$Cm \times Ce + E < 100\% U$$

where:

Cm = Current machine utilization as a percent the machine is utilized for non-Ada

Ce = Compiler efficiency, a factor computed between Ada and non-Ada

E = Percent of CPU required for future expansion

U = Central processing unit utilization

Even though validated Ada compilers exist, the validation process does not prove the reliability of the compiler. The compiler could compile code that contains faults that go undetected during testing. Therefore, for life critical software environments the decision to use Ada should be based on its past performance.

Careful consideration must be given to the adaptability of the existing tool sets or a transition to Ada tools before making the decision to implement Ada. There are three catagories of tool sets most vendors have or will be developing. The three sets are data base control, application, and target development tools.

The Data Base Control Tools are divided into three areas, the Data Base Manager (DBM), the Configuration Control Management (CCM), and the Librarian. The DBM predefines data base premitives and allows definition of user premitives. It also provides services for creating, accessing, modifying, relating, and deleting all Ada development environment (ADE) data base objectives. The CCM provides control over the manipulation of ADE data base objects, including archiving and revision control services. The Librarian is responsible for controlling the logical grouping of objects comprising Ada library units and subunits, as well as controlling access to those objects.

Application development tools include the Editor, Formatter, Pretty Printer, File Maintainer, and Debugger. The Editor is used by programmers to enter Ada source text, as well as other textual materials; it must be capable of Ada-indenting and format control. The Formatter processes text files and reformats them into documentation files. The Pretty Printer prints Ada programs in a logical Ada format and highlights Ada reserve words, etc. The File Maintainer allows comparisons of object programs; text files and typeless files can each be compared. The Debugger provides a symbolic debugging facility to aid in testing Ada application programs.

Target development tools are configured to support specific target machines. The tools include the Ada compilers themselves, Runtime Support Packages, Assemblers, Object Linkers, and Exporters. The Ada compilers with unique code generators will (eventually) be available for target CPU's. A unique Runtime Support Package must be supplied for each target environment. Each target also requires its own Assembler. The assembler must be available as a cross development tool also.

The Object Importer is used to bring into the Ada development environment binary modules produced by other languages. FORTRAN 77 is probably the only language considered at this time. The Linker combines Ada-binary with Libraries and Runtime Support Packages to create Ada Program Files. The Exporter tools are responsible for formatting and transferring Ada Program Files from the host environment to the target environment.

There are no standard library packages defined for Ada other than those given in the language reference manual (Reference 1). Predefined packages must be supplied for standard math functions, statistical packages and common abstract data types. Special standard math packages will be required for particular applications similiar to scientific subroutine packages. For avionics applications matrix and quaternion math routines will need to be developed.

I/O packages are provided by means of predefined packages (chapter 14, Reference 1). The generic packages SEQUENTIAL_IO and DIRECT_IO define I/O operations applicable to files containing elements of a given type. Text I/O are supplied in the package TEXT_IO. The package IO_EXCEPTIONS defines the exceptions needed by the above three packages. A package LOW_LEVEL_IO is provided for direct control of peripheral devices.

For existing software systems, the remaining software life cycle must be considered. Large software systems, with no foreseeable need to upgrade, should be left intact and maintained until they become obsolete and can be phased out. Ada should be considered when the additional cost to implement in Ada is less than the life cycle cost savings over the remainder of an existing project.

Software for large systems is continuously changing due to design changes either in hardware or software. For independent software modifications, Ada permits program units to be subdivided into units that can be modified, coded, checked out, integrated, and documented (Reference 3). The Ada software modifications must conform to the basic program units independent of the software in the system. Otherwise further consideration must be given to the logical grouping and the heirarchial compilation.

When a software package has a high potential for reusability beyond the current project, it is a candidate for Ada, even though it might not be cost efficient on the current project. This becomes a cost saving factor realized on future projects due to DOD's commitment, the language's portability and maintainability.

Since Ada is the language of the future, it will be cost effective to train in Ada (Reference 4). The initial implementation of Ada will require a substantial investment in training. This is due to Ada's programming language strength, potential as a development tool, and its maintenance requirements. Personnel should be trained at levels compatible with their level of involvement. Five possible levels are for managers, support personnel, basic, intermediate and system software engineers (Reference 6).

The Ada vendors have developed object importers to import non Ada code into Ada. Very few have provided for Ada to be called by other languages, handle exceptions and share data with non-Ada code through parameter arguments calls, global variables or common blocks. One vendor, Digital

Equipment Company, using VMS operating system, has provided these capabilities. The DEC VAX Ada conforms to the VAX calling standards, which provides the ability to call and be called by code written in other languages. VAX Ada is also able to handle exceptions from non-Ada code, generate exceptions to be handled by non-Ada code, and share data with non-Ada code through global variables and common blocks (psects).

Telesoft, another Ada vendor, supplies a system interface program. The program allows an experienced programmer to interface between Telesoft Ada and another language based on the VAX operating system. Special modifications must be made to the interface routine. The routine must be recompiled and linked with the assembler runtime support package.

The decision to develop a module in Ada must be based on the host and target computer, the operating system and the Ada compiler vendor. In all probability a system level interface mechanism will have to be developed.

Two important issues concerning necessary or desired language features are programming methodology and software engineering (Reference 5). Programming methodology is concerned with the structured programming, program verification, information hiding and hardware representation. Software engineering is concerned with the issue of large system construction and maintenance. Ada was designed to support and incorporate both of these issues.

The decision criteria called other cost considerations includes; software design specifications, coding specifications, software testing, software design review, software configuration control and deliverable documentation.

The software design specification should address the use of Ada as a programming design language (PDL), the best way to package systems, subsystems, and guidelines for module composition.

Coding specifications and coding style guides will be required to insure that the delivered code is readable and maintainable.

Software testing becomes a factor since Ada encourages separate module development, compilation and independent testing. The module, package, subsystem and system level software test hierarchy becomes a time phase factor for integrated testing.

Software design reviews will require engineers to be knowledgeable in Ada so they can analyze code specifications for their area of expertise.

As compilers are upgraded the question of placing software under configuration control becomes important. A criteria must be established to determine when a compiler and an Ada tool set are sufficient to begin full-scale development.

| DECISION PARAMETERS | ADA DECISION CRITERIA | | | | | | DECISION WEIGHTS |
|---|---|---|---|---|---|---|---|
| | VERY STRONG | STRONG | GOOD | FAIR | WEAK | UNACCEPTABLE | |
| | 10 9 | 8 7 | 6 5 | 4 3 | 2 1 | 0 | |
| 1. The existance of a validated Ada compiler to produce Target Code | Compiler exists for Host and Target computer. Both operating systems and software are compatible. | | Compiler exists for Target computer and Host however both are not compatible. | | Compiler not validated for either target/host computer. | | 9 |
| 2. The coupling and cohesion of Existing Software Environments | Module easily decoupled from global common or data bases. Module cohesive and components identifiable. | | Coupling and cohesion of modules can be resolved with modifications. | | Existing software highly coupled or modules are not cohesive. | | 5 |
| 3. In Real Time environment, the efficiency of the code the compiler produces | Compiled code is efficient, runs in real time, on target computer and allows for expansion. | | Compiled code efficient on host is compatible with target but is not efficient doesn't allow for expansion. | | Compiled code runs on host but is not target efficient. | | 5 |
| 4. For life critical Software Environments the proven reliability of the compiler | Past performance has proven the compiled code to be fault free. (5 years) | | The compiled code is still in a test and checkout phase. Two years valid for non critical. | | The compiled code has not been fully tested or was tested and is not reliable. Use less than 1 year. | | 4 |
| 5. The adaptability of the existing Tool Set | The following tool sets exist: Data Base Control, Application Development, Target Development | | Some of the tool sets are in place or are under development. | | The tools sets have been identified and some exist or are planned. | | 4 |
| 6. The remaining Software Life Cycle | Additional cost to implement Ada is less than Life cycle cost saving over remainder of existing project. | | Ada implementation and LCC saving are equivalent. | | Cost to implement in Ada greater than LCC saving for remainder of existing project. | | 5 |
| 7. Independence of the Software Modification | Modification divide into program units for coding, checkout, integration and documentation. | | Modifications require improvisions to be devisible into program units. | | Modificaitons are very difficult to integrate part of into existing program units. | | 7 |
| 8. Potential Reusability of the software | Reusable beyond current project. | | Software might be reusable but has not been identified. | | Software will not be reused. | | 9 |
| 9. Cost effectiveness of Retraining | Adequate resources available projects life cycle justifies training effort. | | Limited resources available to train a number of key individuals for project. | | Project life cycle is short and resources not available. | | 7 |
| 10. Interface Mechanisms | Selected computer, compiler and operating system have interface mechanisms. | | Interface mechanisms can be developed or work arounds established. | | No provisions available for interfacing. | | 8 |
| 11. Necessary or Desired Language Features | Supports both Programming Methodology and Software Engineering. | | Need either programming methodology or software engineering. | | Languages features not supported. | | 8 |
| 12. Other Cost Considerations - Design, Coding, Testing, Design Review, Config. Control, Deliverable Documents | Cost consideration in these areas are minor impact. | | These conditions are marginal impacts. | | Major impact and could restrict implementation. | | 8 |
| | | | | WEIGHT SUMMATION | | | 79 |

**Figure 3. Ada Decision Matrix**

Well written Ada code will not in all likelyhood meet deliverable documentation standards. Therefore the questions left unanswered are what additional documentation will be required and will module packaging provide a system overview of systems and subsystems.

Using the twelve decision parameters a subjective set of action statements was developed to produce a prototype Ada Decision Matrix shown in Figure 3.

An example set of decision weights was applied to the matrix to show how a decision criteria would evolve. Applying the decision rule, Figure 2, the value .66 indicates Ada is the acceptable choice. A future study will be to validate the prototype matrix using the original software experts. Their independent responses will be combined to justify a decision.

## CONCLUSION

This paper investigated the feasibility of using Ada in non-Ada environments. A systems analysis approach was taken to develop a decision criteria. An Ada decision matrix and decision rule was developed. The Ada decision matrix relates conditions, which are the decision parameters, and actions to establish a decision rule. The decision rule incorporates all the conditions that must be satisfied for a related set of actions. The paper also exposes the need for Ada interface or linkage mechanisms as one of the decision parameters.

## REFERENCES


1. "Ada Progamming Language," Department of Defense, Washington, D.C., ANSI/MIL-STD 1815A-1983.
2. Baker, D. B., "Ada Decision Matrix,"The Aerospace Corporation, El Segundo, CA, March 23, 1984.
3. Barnes, J. G. P., "Programming in Ada," Addison-Wesley, Massachusetts, 1981.
4. Blake, G. A., "Ada Implementation Plan for Deputy FOR SIMULATORS (ASD/YW),"Wright-Patterson AFB, OH, January 16, 1984.
5. Bouch, G., "Software Engineering with Ada," The Benjamin/Cummings Publishing Co., 1983.
6. Habermann, A. N., Perry, D. E., "Ada for Experienced Programmers," Addison- Wesley, Massachusetts, 1983.
7. Senn, J. A., "Analysis and Design of Information Systems," McGraw-Hill Book Co., New York, 1984.
8. Yourdon, Edward and Larry L. Constantine, "Structured Design," Prentice-Hall, Englewood Cliffs, NJ, 1979.



James C. Helm received the B. S. in Mathematics and Physics from Missouri Valley College, the M. S. in mathematics from the University of Missouri at Rolla, and the Ph. D. degree in Industrial Engineering, Operations Research from Texas A&M University in 1962, 1964, and 1972 respectively. He is presently a Senior Software Engineering Specialist in the Flight Control Element of Software Systems with Ford Aerospace & Communications Corporation, Houston, Texas. He has had twenty years of industrial experience as: a member of TRW's Technical Staff, Systems Engineering and Analysis Department supporting NASA in navigation and trajectory analysis; a manager of software applications with M&S Computing, in charge of the Marshall Mated Engine Simulation Software; an integration manager with McDonnell Douglas Technical Services Company supporting the NASA Space Shuttle; a real-time programmer with IBM FSD supporting NASA on GEMINI and APOLLO missions. He was an instructor in Mathematics at the University of Missouri, Rolla, and has taught computer science courses at Texas A&M and presently at the University of Houston Clear Lake. His areas of interest are operations research, computer science, systems analysis and simulation techniques.



Theda E. Cook received a B. S. degree in Mathematics and Physics from East Texas State University, Commerce, Texas, in 1965. Since 1971 she has been employed at Ford Aerospace & Communications Corporation where she is currently a lead software engineer working on the design of a Shuttle Control Center for the Air Force. At Ford Aerospace & Communications Corporation she has worked on various projects for the Shuttle and served as task leader on telemetry data processing projects for the Shuttle. Prior to Ford Aerospace, she was employed as a member of the technical staff at TRW supporting NASA for Apollo and Skylab.

# General Dynamics Ada-based Design Language

Thomas S. Radi, Ph.D

General Dynamics/Pomona Division

**Abstract**

This paper describes an Program Design Language based on Ada language, the General Dynamics Ada Design Language (GDADL) and how it is intended to be used in support of the disciplined Software Development Approach methodology at the Pomona Division of General Dynamics. Ada language constructs are used to define the structure of the solution. A "quasi-freeform" Program Design Language (PDL) is used to describe the design intent. The algorithm and control flow at a higher level than the ultimate code. The **Ada** part of an Ada/PDL is used to define the program structure and to declare types and objects. The **PDL** part of an Ada/PDL is used to define the procedure unit's control structure and the procedure unit invocations.

By using "comment compatible" syntax for the PDL descriptions, the design intent is embedded in the final source code, thereby making both the maintenance enhancement task and the documentation update task a little easier.

A software tool, the GDADL processor, has been developed to assist the design team in software development. The GDADL processor capabilities are briefly described.

## Introduction

There has been a good deal of interest, recently, in the Ada language, especially in the area of using Ada as a design language. Several organizations, notably the IEEE sponsored, Ada as a PDL working group, and the SIGAda (formerly AdaTEC) supported, Design Language working groups have been formed in order to address that very issue. General Dynamics has been intimately involved in both the IEEE and SIGAda efforts.

In order to force contractors to get a headstart on using Ada, the government has begun to require the use of an Ada PDL in their latest Requests for Proposal. These requirements do not specify which Ada/PDL to use, how to use it, or what form the design should take after using it. This lack of definition is understandable since no *de facto* standard yet exists.

Several questions arise naturally when we look at the situation:

What causes all this enthusiasm to use Ada as a Design Language?

What is the real advantage of using an Ada PDL?

*Are all Ada/PDLs just as good, or is any one better?*

Let me try to answer some of these questions. But before I do, let me insert one caveat (that means let me hedge my answer). I am expressing only my opinion, and not the position or opinion of the General Dynamics Corporation. Oh yes, one other point: my opinion is very sound.

### What causes all this enthusiasm to use Ada as a Design Language?

The answer is very simply - Ada.

If there were no Ada, there would be no great desire to use Ada as a design language. This answer is not as flippant as it may at first appear. The goal for those advocating the use of Ada as a *design language* is to produce "good Ada designers" and good Ada designs.

The most effective means of producing those good designers is to start using the concepts found in the Ada language as early as possible in the current software development cycle. Program Design Languages, when properly used, provide an excellent vehicle for conveying software designs in a consistent manner. An Ada-based Design Language has the additional benefit of forcing designers to use the Ada constructs and features, such as strong typing, packages, and tasks early in the design phase.

**What is the real advantage of using an Ada/PDL?**

The advantage of using an Ada/PDL is learning how to use Ada to design good software. The idea, for the present, is to force our designers to use what some have labelled "the Ada mindset" when designing our software systems. In the future, once we all are familiar with Ada, Ada-based Design languages will offer the additional benefit of providing an excellent means of communicating both the structure of the program and the algorithms that describe the program control flow.

**Are all Ada/PDLs just as good, or is any one better?**

The lack of a standard for the Ada/PDL is regarded in some quarters as a cause for concern. My own opinion is that until a standard set of guidelines are developed and approved, an Ada PDL should be considered to be acceptable if it clearly conveys the designer's intent. Remember the purpose of a PDL is to formulate a solution and to communicate the solution at a higher level of abstraction than lines of code.

Of course, in PDLs as in anything else there is acceptable and there is **acceptable**! Let me paraphrase George Orwell's Animal Farm, "All PDLs are equal, but some are **more equal** than others." The General Dynamics Ada-based Design Language (GDADL) fits into the latter category.

**GDADL incorporates features from existing PDLs**

The General Dynamics Pomona Division and Data Systems Division are finalizing the development of an "Ada/PDL", which we call General Dynamics Ada based Design Language (GDADL) [2],[9]. GDADL will conform to the IEEE Ada PDL recommended practice for the development of Ada-based Program Design Languages [6]. The language itself, GDADL, would be only of minor interest if a processor did not exist in order to provide the design aids we have come to expect from Program Design Language processors such as Caine, Farber, and Gordon's PDL/91 [1] and Henry Fleine's SDDL [4]. While it is possible to design in Ada using only GDADL, i.e. without processing the design using an automated tool, such a tool provides many benefits to the designer.

A PDL processor allows the design team to produce reports which aid them in checking the validity and consistency of the design. These design reports provide the reader with an up to date view of the level of design refinement and detail. In other words, the PDL processor provides a design disclosure.

Table I is a list of some of the features of the GDADL processor, as well as a list of the reports which the user may request to be generated by the GDADL processor.

**Table I**
**GDADL processor Design Reports**

Program unit and task entry invocation trees
(shows the structure of the program)

Program unit and task entry cross reference table
(shows where the actual invocations are made)

Object declaration and usage cross reference table
(all object declarations are legal Ada, references to objects made either in design statements or parameter lists are listed)

Undefined design items (TBD) cross reference table
(highlights all TBD items)

User defined cross reference table
(unlimited number of items can be cross referenced)

Flow of control arrows (right side) and page reference numbers on all subprogram invocations and task entry calls

Exit arrows (left side) on all loop exits and returns

Summary of errors detected in the design description (incorrectly nested if..end if, loop..end loop, etc)

Pretty-printed design document
(level of indentation selected by user)

Data Dictionary
(automatically generated dictionary arranged alphabetically)

Subtype, derived type, base type reference table
(shows all subtypes and derived types)

Generic instantiation report
(shows all instantiations and the applicable generic)

Keyword enhancement and high-lighting
(highlights Ada keywords by underscoring or boldface printing)

Ada Identifier high-lighting
(all identifiers are automatically high lighted in the PDL and in the Ada declarations)

GDADL was developed after a thorough evaluation of commercially available products such as [1] and [2] and the Byron product [3] developed by Intermetrics. GDADL combines what we consider to be the best features of each of those languages and their processors, with some original additions, into a format that provides the designer with an easy to use, **and easy to re-use**, means of expressing multiple levels of design abstraction.

**Using GDADL means using Ada as it was intended to be used**

GDADL was designed with two primary goals in mind:

1. Avoid unnecessary repetition of Ada items when transitioning from the design phase to the coding phase.

2. Make the design intent visible as an integral part of the source code. This is accomplished by defining the PDL, in "comment-compatible" format in the prolog section of the final source code. Keeping the PDL control flow information in the same physical file as the source code, allows for easy modification of both the code and the design description during the maintenance phase of the software lifecycle.

Figure 1. shows an example of an Ada program unit after the design of the unit has been completed, and before the coding stage is complete. The design intent is described in the prolog of the program unit, thereby enabling the reader to readily understand the code which follows.

One of the biggest dangers in using a Program Design Language is to drive the design too far towards code. One of the main reasons for using a PDL is to provide a higher level of design disclosure (abstraction). When the design begins to look like code, it is time to review the level of definition of the design being used, and probably backup to a higher level of abstraction.

A good rule to remember is that PDL should not contain assignment statements.

I should note here that procedure STOP_LIGHT of Figure 1. will compile without error, but will not execute correctly until the enumerated (TBD) types are defined.

Figure 1. An Example of an Ada program unit fragment

```
procedure STOP_LIGHT is
   Requirements satisfied : 5.2.2.1
   Author : E.M. DeGuy

   STOP_LIGHT determines what to do when the car
   is at an intersection.

   the type and object declarations are shown below

type SPEED_TYPE is (STOP,SLOW,FAST);

type DIRECTION is (NORTH,EAST,SOUTH,WEST);

type DISTANCE is (TBD);

type LOCATION is (TBD);

type ROAD_OBSTACLES_REPORT is (Clear, Blocked);


DIST_TO_INTERSECTION : DISTANCE ;   -- distance &
     -- to the center of the intersection

INTERSECTION_ERROR : DISTANCE ;   -- minimum distance &
     -- which the car may be from the center of the intersection --
     -- and still be considered at the intersection.

INTERSECTION_LOCATION , CURRENT_LOCATION : LOCATION;

STOP_LIGHT_WORKING : BOOLEAN;

CHECK_OBSTACLES_AROUND : ROAD_OBSTACLE_REPORT := CLEAR;

function DIFF_BETWEEN (LEFT,RIGHT : LOCATION) return DISTANCE
          is separate;

procedure PROCEED (
     SPEED     : in  SPEED_TYPE;
     WAY_TO_GO_out  DIRECTION); is separate

   -- check to make sure the car is at the intersection
   if not at_intersection then
      return
   end if

   if the car has reached the intersection but the
      stop light is not working then

      The car must look for any obstacles in any
      direction using CHECK_OBSTACLES before
      using PROCEED to pass through the intersection
      at a safe speed

   else

      The car uses PROCEED to proceed according to
      instructions supplied by DIRECTIONS_AT_THE_INTERSECTION
   end if

begin

 DIST_TO_INTERSECTION := DIFF_BETWEEN ( INTERSECTION_LOCATION
          CURRENT_LOCATION );
   if DIST_TO_INTERSECTION > INTERSECTION_ERROR then
      return;
   else
      if not STOP_LIGHT_WORKING then
         while CHECK_OBSTACLES_AROUND = not CLEAR loop
               null;
         end loop;
         PROCEED (SPEED => LOW);
      else
         NEW_DIRECTIONS:=DIRECTIONS_AT_THE_INTERSECTION;
         PROCEED (LOW,NEW_DIRECTIONS);
      end if;
   end if;
end STOP_LIGHT


separate (STOP_LIGHT)
procedure PROCEED( ....
   ...

end PROCEED;

separate (STOP_LIGHT)
function DIFF_BETWEEN( Left, Right) return DISTANCE is
   ...

end DIFF_BETWEEN;
```

**How and when should we use GDADL?**

GDADL is intended for use in both the design and maintenance phases of the software lifecycle. During the design phase the designer's intent is documented using GDADL by embedding the description of the design in prolog portion of the source code. If any software changes are subsequently found to be necessary, first the embedded design description is updated to reflect the new design intent and then the source code changes are made. By having the design description and the actual source code in the same place, we try to insure that the code matches the design.

GDADL was originally developed to support the Disciplined Software Development Approach (DSDA), the software design methodology for the development of software which is currently being implemented at the Fomona Division. Figure 2 is a pictorial representation of the DSDA methodology. DSDA requires that the software requirements are defined and documented using Data Flow Diagrams, State Transition Diagrams, data dictionaries, and mini-specs [5], before the design phase is begun. In other words that the requirements are well understood.

The design phase of the DSDA results in the development of three stages of design refinement as shown in Figure 3. Each stage builds upon the previous stage, and is documented using GDADL.

During the first stage of the refinement the designer defines the structure of the solution. In Ada terminology, the top level program units are defined so that, initially, a set of packages is designed, where each package satisfies a set of requirements that could logically be grouped together. In order to insure that all of the requirements are satisfied a requirements allocation chart which cross references all requirements to Ada program units should be developed at the same time. This initial packaging of the top level architecture is examined to determine the best implementation for each package, e.g. a task, subprogram, generic or package. The top level set of program units is subsequently expanded and further defined. All "visible" parts of these program units, i.e. types, objects and program units declared in the specification part are defined in as much detail as possible.

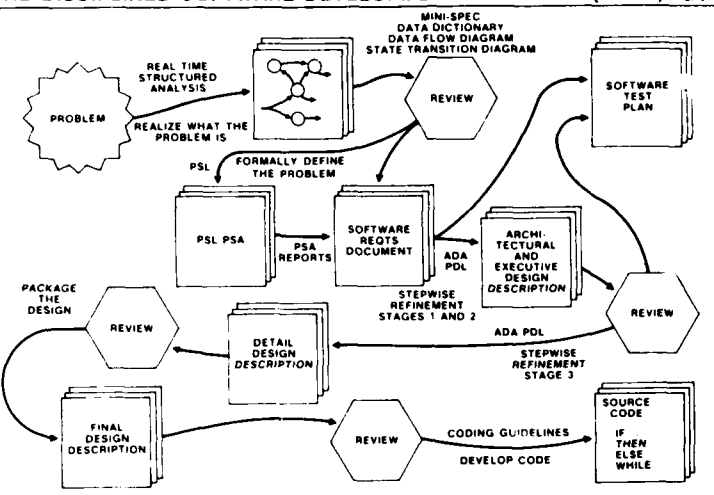
In stage two an Executive Module is designed

FIGURE 2

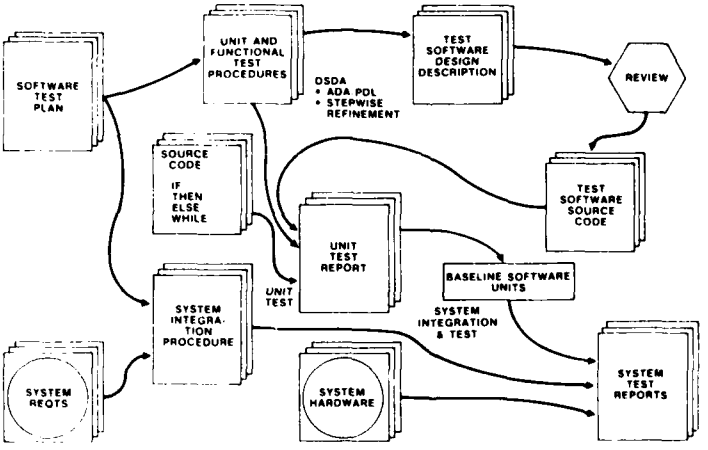
GENERAL DYNAMICS

UNCLASSIFIED

## THE DISCIPLINED SOFTWARE DEVELOPMENT APPROACH (DSDA) 1 OF 2




## THE DISCIPLINED SOFTWARE DEVELOPMENT APPROACH (DSDA) 2 OF 2




UNCLASSIFIED

and coded. The executive module includes the visible parts of the packages defined during stage one of the design process.

At the end of stage two both the structure of the program units identified in stage one, and the executive module are well defined. The interfaces for all visible program units have been defined and are tested by coding and compiling of an executive module. The precedence of modules invoked by the executive is known, but the actual conditions that would cause modules to invoke other modules within the packages has not been identified, that is, we do not know the control structure yet.

Stage three of the refinement defines the interfaces between modules, and the control structure of the modules. Ada/PDL is used heavily in this stage. The definition of who calls who and when is made using PDL statements at this time. Decisions on type representations based on design details may be *deferred at the start of phase three*, by declaring the items "TBD", (To Be Defined), but these TBD items must be identified by the end of stage three.

### Why not use ONLY Ada as the PDL?

There are three (at least) schools of thought on the use of an Ada-based Program Design Language.

The first school won't even consider the use of a "PDL". "We have used flowcharts for years and always delivered our product!" Despite several studies [6], [7], showing the superiority of PDL over flowcharts for both creating the code originally, and for enhancing the design later, this faction sticks to the "*tried and true*" method. (They also wear suspenders and drive Hudsons).

The other extreme is the "Ada is the PDL" faction. The thinking here is that Ada is rich enough to be used as a PDL without any enhancements. Proponents of the ADA_ONLY approach suggest that if one_uses_underscores_to_connect_everything_in_an algorithm_ then by_declaring_those_underscored_entities_either_as_procedures_or_functions_or_as_Boolean_objects, the design is the code. There may be some merit in this "underscore everything in sight" approach if the problem being solved is extremely simple, but I believe that in

Figure 5
Three stage process of Stepwise Refinement
Using Ada as a Program Design Language

```
Design Phase Stage 1 - Architectural Design

        Step 1.1: Define the initial top level structure

          Package the requirements.
            "Implement an initial solution using  packages."

          A: Group (up) all cohesive requirements together

          B: Allocate specific requirements to a package

          "Each requirement is satisfied by some package."

        Step 1.2: Refine initial packages into program units

          Identify the subprograms or "sub packages" that will
          satisfy the requirements.

          (Determine whether a task, procedure,
            function or package should be used to
            actually implement the solution.)

          Identify available generics where possible.


        Step 1.3: Identify the resources available in each
          program unit.

          Define and write specifications for the subprograms
          which are visible to each package.

          Define the interfaces of all subprograms.
            "the parameters that will be used"

          Define the interfaces for all other program units.


        Step 1.4: Define all visible types and objects
          (or make them private to the package)

          Use TBD constructs if necessary

  The Ada part of GDADL documents the design structure

Design Phase Stage 2 - Executive Design

        Step 2.1: Design the EXECUTIVE module
          Define all control flow in the executive.
          Define all module invocations to the program units
            identified in Stage 1, that the executive will make.


        Step 2.2: Code the EXECUTIVE Module
          Code and compile the executive and the
            program units defined in Stage 1, to check the
            interfaces to all top level packages.


    The PDL part is used to define any EXECUTIVE module
      control flow.

Design Phase Stage 3 - Detailed Design

        Step 3.1: Define the controls and interfaces
          of subprograms inside the package bodies.

            (these subprograms are identified while
              designing the body of the package)

            Define under what conditions calls
              are made (the control flow)

The PDL part of GDADL is used to define the control
flow.


        Step 3.2: Define the details and fill in
          omissions (define all types and objects)

            Define all TBD algorithms
            Define all TBD data types and objects

The Ada part of GDADL is used to define
the types and objects.

The PDL part of GDADL is used to
define the TBD algorithms.
```

most cases this approach adds an unnecessary burden to the design process. Rather than being able to describe the algorithm using a higher structured English approach, the designer is required to get bogged down in the details of the interfaces to these procedures and functions too early in the design stage. Another drawback is that this approach may create artificial "unnecessary" subprogram solutions rather than in-line code.

The design approach advocated using GDADL and the DSDM methodology is to gradually define the design in stages, conveying accurate representations of the design at each stage, with each stage being a refinement of the previous stage.

**An interesting speculation with regard to using only Ada as the Program Design Language**

While dramatic gains in productivity were reported by users of PDL to describe their FORTRAN and assembly language designs, the same may not prove to be true for Ada. The rationale for this speculation is as follows.

It may be that the difference between the levels of abstraction of assembly language or FORTRAN or JOVIAL and a properly used PDL are sufficiently different that by using a PDL to describe these designs, the comprehensibility of the design is much greater than the comprehensibility of the code.

On the Ada side of the coin however, it may turn out that the difference between the levels of abstraction of Ada and a PDL are so small that the PDL does not add anything to the comprehensibility of the code.

I do not believe that this will be the case. In our limited use of the GDADL to define and express Ada designs, I have found that the PDL is superior to the Ada code, with respect to having reviewers understand the design intent, but only if the PDL is used to describe the design at a high level of abstraction.

**REFERENCES**


[1] Caine, Farber, and Gordon Inc., PDL 81 Design Language Reference Guide [5500-PD8, v1.4], June 1982.

[2] Kleine, Henry Software Design and Documentation Language, JPL Publication 77-24 Revision 1, (August 1, 1979).

[3] Intermetrics Inc. Byron User's Manual Version 1.1, 1984.

[4] Radi, T.S., The Disciplined Software Development Approach. TM-6-370-058, General Dynamics Pomona Division.

[5] Yourdon Inc., Structured Analysis for Real-Time Systems, Course Notes. Yourdon Inc., Edition 3.0, January 1984.

[6] Ramsey, H.R, Atwood, M.E., and Van Doren, J.R. "Flowcharts Versus Program Design Languages: An Experimental Comparison". Communications of the ACM, June 1983 pp 445-449.

[7] Caine, S.H., and Gordon, E.K., PDL - A tool for software design, Proc. National Computer Conference, 1975, pp 271-276.

[8] Morales, H.D. and Radi, T.S. "General Dynamics Ada-based Design Language (GDADL) Requirements Document", M-6-370-HM-001, General Dynamics Pomona Division.

[9] Radi, T.S. and Morales, H.D. "General Dynamics Ada-Based Design Language (GDADL) User's Guide", M-6-370-TR-058.

# IBM-PC BASED ADA SYMBOLIC DEBUGGER

Daniel W. Lyttle
Rockwell International
Collins Government Avionics Division
Cedar Rapids, Iowa 52498

## Abstract

A symbolic debugger has been developed at Rockwell International Corporation, Collins Government Avionics Division, for use in the development and testing of avionics systems. The Debugger, an Ada(*)program running on an IBM Personal Computer, monitors the real-time execution of Ada software in an embedded computer system. The Debugger is part of a programming support environment built around the VAX-hosted ICSC (Irvine Computer Sciences Corp) Ada compiler.

This paper concentrates on the Debugger implementation, describing the connection to the system under test, generation of symbol tables by the compiler, development of the Debugger software, and the user command interface.

## Background

Since 1971, Collins Avionics has successfully used High-Order Languages in the development of avionics computer systems. The use of HOL for the design and coding of a complex program offers clear advantages over assembly language, in productivity and in the reliability and maintainability of the resulting system.

Unfortunately, systems programmed in HOL are often debugged at the assembly level. Embedded computers generally do not include a keyboard, printer, or disk drive for a convenient software debug interface. Instead, a hardware adapter is connected to the CPU or its bus to allow the programmer to monitor the execution of the software.

The traditional test interface provides commands to examine and modify memory locations and to set breakpoints at absolute addresses. The programmer needs assembly listings and a link map, a hex calculator, and an in-depth understanding of the target machine instruction set. Debugging at this primitive level is time-consuming and expensive, especially with the growing complexity of software systems and microprocessor architectures.

A symbolic debugger extends the more productive high-order language environment to the laboratory in the software debugging and testing phases. The debugger insulates the user from such details as the addresses of variables and peculiarities of the machine architecture. It filters and formats the information presented to the user.

Collins Avionics recognized the need for a symbolic debugger, and in 1980 began to develop a symbolic debugger for CAPS-6 (a proprietary stack-architecture avionics computer) and source languages AED (an ALGOL-60 derivative) and PL/I. It was successfully employed in the development of an experimental active controls system.

Although the advent of Ada has shifted the emphasis of debugging to the new language, the experience and feedback from users of the original tool have significantly influenced the design of a new Ada-oriented symbolic debugger.

## Implementation

Other Ada symbolic debuggers emphasize debugging Ada code on the host machine. Cross-debugging for an embedded computer may be supported by simulation of the target machine architecture, if at all.

In contrast, the Rockwell approach emphasizes debugging in the embedded computer environment, where real-time Ada software in the target computer interacts directly with hardware I/O devices.

This section describes the hardware interface between the target computer and the Debugger, and the symbol table interface with the compiler. The Ada-to-PC cross compiler is elaborated, along with several sticky design problems and their solutions.

## Target Interface

An eventual design goal is the capability to debug various target computers, including several commercially available microprocessors and the CAPS (Collins Adaptive Processing System) family. An additional requirement is the ability to test factory-assembled units, where all chips are permanently soldered in place. The cost of a test station should be minimized, so that a project can buy an adequate number of work stations.

This combination of requirements led to the decision to develop a flexible multitarget test interface, as opposed to purchasing off-the-shelf microprocessor emulators.

The target interface consists of custom-designed cards inserted in the expansion slots of an IBM Personal Computer. (An XT or AT model is preferred.) The first card provides control functions, including signals to reset, halt, run, and step the target, facilities to examine and modify target memory locations, an execution history buffer, and address-matching logic for breakpoints.

The second card is optional and contains 160K bytes of address-mappable RAM to supplement or replace target memory. These first two cards are common for all target systems.

A third card is connected directly to the target, isolating target dependencies from the rest of the system. For the Intel 8086 processor family (including the 8088, 80186, and 80188, with or without a numeric coprocessor chip), the interface is in the form of a "personality module" pod, attached via a short cable to a connector provided on the edge of the CPU board. Address/data lines and control signals are brought out to this connector when the CPU board is designed. This approach eliminates the need to remove the microprocessor chip from the board.

The CAPS processor family uses an asynchronous "Transfer Bus" with generous timing constraints, so the unit under test can be connected via a longer cable to a CAPS personality module inside the PC.

In order to display subroutine parameters and local variables, the Debugger needs to have access to certain registers inside the CPU, namely the program counter and stack frame pointer. Some of the CAPS processors can dump their registers to memory at a signal on

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

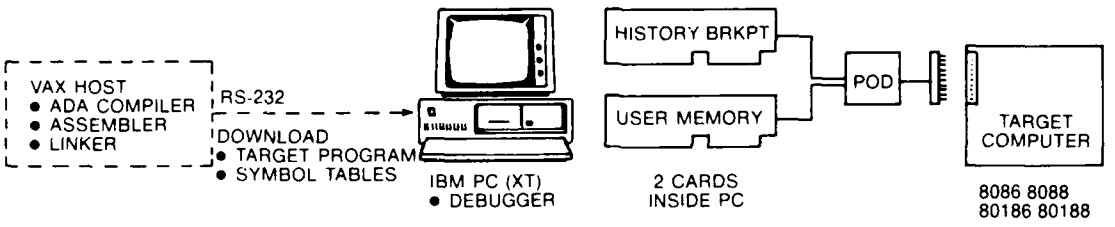



Figure 1. Hardware Requirements for Symbolic Debugging.

an input pin. Other systems must have a small "debug" interrupt handler embedded in the executive software.

## Compiler Interface

The Ada Symbolic Debugger relies on symbol table information generated by the compiler. The debugger database is entirely disjointed from the user program, as no special code is inserted in the user program by the compiler, and database files are separate from the user program files.

The VAX-hosted ICSC Ada compiler is comprised of two separate programs: a Front End (Pass 1) common to all targets, and a target-specific Code Generator (Pass 2). Collins Avionics has developed code generators for CAPS and 8086. ICSC has code generators for VAX-VMS, VAX-Unix, and several 68000- and Z8000-based systems. TLD Systems Ltd. working with ICSC, has a MIL-STD-1750A code generator.

The front end creates an intermediate form (.INT) file which is used by the code generator. When compiling a package specification, the front end also creates a symbol table attribute (.ATR) file describing the visible part of the package, for use when the package is "with"ed in a later compilation. The code generator creates an assembly source file, which is then assembled and linked. For the 8086, a VAX-hosted cross assmbler linker package was purchased from Microtec Research. A CAPS cross assembler and linker have been developed by Rockwell.

Under contract to Rockwell, ICSC has added enough information to the .ATR and .INT files so that symbol tables for all package specification and body data could be reconstructed. Line marks were added to the .INT file to identify the source line corresponding to each block of code.

The various parts of the compiler are executed in sequence by a command file. Extra steps are performed automatically when the symbolic debug option is added to the compiler command line. A Debugger Symbol Table (.DST) file is generated by a program reading the two files created by the front end. The Symbol Table file describes all procedures, types, and variables defined in the package. A Debugger Line Table (.DLT) file describes the relationship between Ada source line numbers and byte offsets in the assembled object module. The Line Table file is created by a program which scans the assembly listing file from the Microtec 8086 assembler. The CAPS assembler creates the Line Table file directly.

When the object modules are linked together to form an executable load module, a Debugger Module Table (.DMT) file is also created. The Module Table file gives the name of each object module and the starting and ending addresses of its code and data. The Module Table file is created directly by the CAPS linker, or by a program reading the Microtec link map.

The Symbol, Line, and Module Table files are transferred from the VAX host to the IBM-PC prior to debugging, along with the load module. Columbia University's public domain Kermit file transfer program is used for downloading via a 9600 baud RS-232 line.

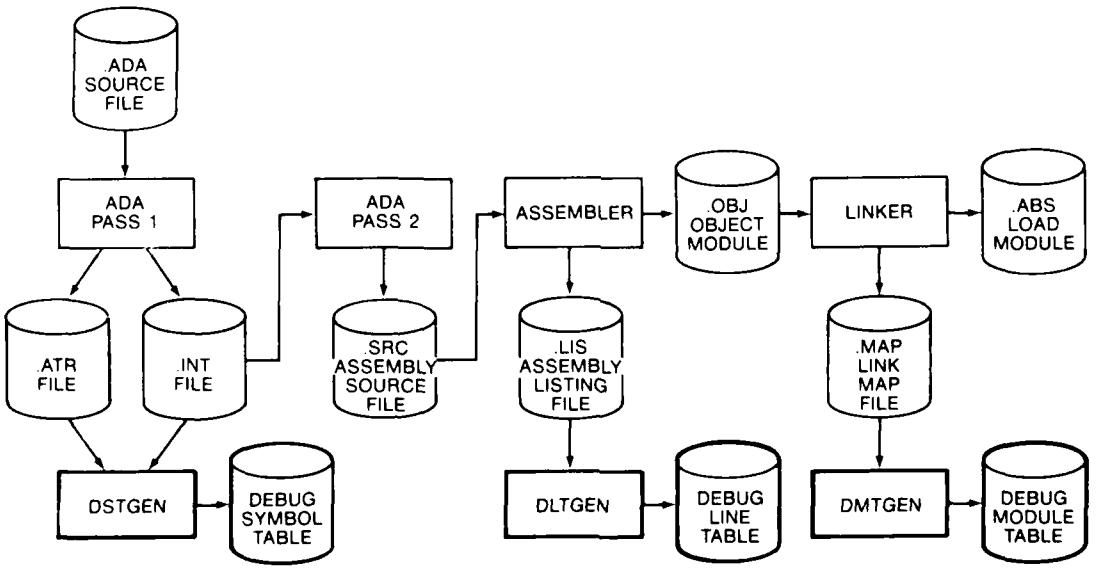



Figure 2. Code and Database Generation, 8086 Target. Steps Added for Symbolic Debugging are Shown in Heavy Outline.

## Software Implementation

Early in the project, it was decided to write the Symbolic Debugger in Ada and to host it on the IBM Personal Computer. No suitable Ada compiler for the PC was available at that time, so the Debugger was implemented using the ICSC Ada-VAX compiler. The Debugger, running on the VAX, communicated with the PC and the target interface via an RS-232 line. Avionics software was developed over a period of several months with the Debugger in this configuration.

The ICSC Ada-8086 cross compiler can generate code for the IBM-PC, since the PC contains an 8088 processor. Assembled object modules are converted from an ASCII Hex format to a binary format on the VAX, downloaded to the PC, and linked using the standard MS-DOS linker to produce an executable (.EXE) file. The "Medium" programming model is used, permitting multiple code segments and one 64K byte data segment.

Program initialization and exception support routines were implemented in assembly language. Ada-callable assembly language utility packages were developed to interface MS-DOS interrupts for I/O, to perform 32-bit integer arithmetic and logical functions, and to access absolute memory locations in the PC. Ada packages in the runtime library include STANDARD, TEXT_IO, CALENDAR, sequential and random-access disk I/O, heap management, and string manipulation.

Several projects besides the Debugger have taken advantage of this VAX-hosted Ada cross compiler and runtime library. PC-based tools and application programs are being developed, and avionics algorithms are being checked out on the PC.

The Debugger program is written entirely in Ada. Due to careful design, few changes were required when moving the program to the PC.

The predefined type INTEGER is implemented as 32 bits on the VAX, but as 16-bit words on the PC. To avoid relying on INTEGER, types BYTE, WORD, and LONG are used extensively. The efficient 16-bit WORD type is used where possible, while the 32-bit LONG is used when the extra precision is needed. Type LONG is defined as a record on the PC, with a complete set of conversions and overloaded operators.

The allocation of dynamic storage is carefully controlled, to avoid exhausting the 64K bytes of available data memory in the PC medium model. Functions returning arrays and strings, including the built-in "&" operator, are avoided. Packages where storage is allocated maintain a list of discarded objects for re-use.

The symbol, line, and module table files are received from the VAX and stored on disk in a compact binary file format. The first time a module is referenced, its files are read and converted to an easily scanned internal format. For example, symbol tables are stored as linked lists of variant records.

Access types are not used to manipulate symbol tables since the 64K data model would limit the size of user programs. Instead, a "database" package defines a "database pointer" record type (a block number and an offset within the block), and procedures to allocate, store, and retrieve objects. Blocks of the database are stored on the PC in a random-access disk file, with several recently used blocks cached in memory. On the VAX, database blocks are simply allocated from virtual memory.

## User Interface

The user enters commands and the Debugger prints a response to each command. The commands can be classified as File Manipulation, Execution Control, Breakpoint, Data Manipulation, and Miscellaneous commands.

To reduce the number of keystrokes needed to enter a command, any command name can be abbreviated. Many frequently used commands have one-letter abbreviations. In addition, the names of symbols from the Ada source program can be abbreviated, including module, procedure, and variable names.

Most commands can be aborted by pressing the Escape key.

### File Manipulation Commands

The LOAD command reads the linker-created user program load module into target memory for execution. The module table is also loaded into debugger memory.

The DBLOAD (DataBase Load) command loads the module table for debugging a program already in read-only memory.

The VERIFY command compares target memory against the load module on disk.

### Execution Control Commands

The RESET command asserts a hardware reset signal to the target. RUN releases mastership of the bus, or the HOLD signal to the processor, allowing the program to execute. GO combines these two commands, to execute the program from the beginning.

The HALT command requests bus mastership or asserts HOLD to stop the target processor. The current program counter is printed in symbolic form, as a module name, procedure name, and line number.

The target interface hardware maintains an execution trace buffer of 2047 bus transactions. The HISTORY command displays selected parts of the trace buffer (by default, the latest 16 entries). History is interpreted in a symbolic format, along with the traditional hexadecimal address and data trace. Data accesses are printed as a variable name and its value in the appropriate data type. Fetched instructions are disassembled and the address is interpreted as a module name, procedure name, and line number.

The EXECUTE command allows the target to execute a few instructions at a time and prints the line number of each line executed. STEP gives a more detailed trace in the symbolic history format. Both commands will print a given number of lines and/or stop at a specified point in the program.

### Breakpoint and Watchpoint Commands

A breakpoint causes the target to halt at a certain point in the program. (Actually, the tester hardware requests control of the bus when a specified address is detected.) Breakpoints can be set on execution of a given Ada source line number or procedure, or when accessing a variable defined in the Ada program. Data breakpoints can be further qualified by Read, Write, and value comparison. Breakpoints can also be set at absolute addresses and address ranges.

When a breakpoint is reached, the target halts and the Debugger prints a message giving the breakpoint address and symbolic information. A watchpoint is like a breakpoint, except that the target is restarted immediately after the message is printed.

### Data Manipulation Commands

The DISPLAY command can be given the name of a variable declared in the Ada source program. The Debugger looks up the address of the variable, reads target memory, and prints the value in a format appropriate for the variable's type. For example, integers are displayed in base 10, access variables in hex, and floating-point variables in decimal notation with the proper number of digits. Booleans are displayed as FALSE or TRUE, and a value of enumeration type COLOR might be displayed as BLUE.

Scalars, strings, entire arrays, entire records, and record components can be displayed. Complex Ada expressions can be evaluated, along with Ada attributes such as 'ADDRESS, 'SIZE 'RANGE, 'POS, and 'IMAGE. For 8086-family targets, the processor registers can be displayed.

After a variable or expression is displayed, entering DISPLAY alone will evaluate the same variable or expression again.

The SET command sets a variable to the given value. The value can be a literal, a named constant, another variable, or an expression.

The MONITOR command adds a variable or expression to a list of items to be continuously displayed while the target is running. Each

item is re-evaluated and printed at the top of the screen, at a rate of about 10 items per second. When the target halts or the user enters a command, the items will scroll off the screen.

The DUMP command displays a range of memory locations in hex and ASCII formats.

The DASM command disassembles a block of memory or an Ada procedure. The source line number of each new line of code is printed along with the disassembled instructions.

### Miscellaneous Commands

The SHOW command has several options. SHOW BREAKS prints a numbered list of the current breakpoints. SHOW CALLS interprets the stack to print a chain of subroutine calls. SHOW MODULE prints the starting and ending addresses and other info for a given module. SHOW SCOPE displays the module and procedure where the Debugger will look first when searching for a variable to display.

A disk file containing a list of Debugger commands can be executed by preceding its name with "@".

A JOURNAL file feature causes all commands entered to be written to a journal file on disk. The journal file can later be executed as a command file.

A LOG file feature causes all commands and responses to be written to a log file on disk, to be examined or printed later. The journal, log, and command file features can be used in combination to build powerful automated testing sequences.

The HELP command accesses an on-line help file, essentially printing sections of the Debugger user's guide on the screen. HELP can be displayed for a given command or topic.

The TYPE and PRINT commands send a text file to the screen or to the PC printer respectively.

### Future Plans

Although the PC-hosted Ada Symbolic Debugger has been released to users and is being used to develop actual real-time embedded computer applications, improvements will continue to be added.

The first Ada Debugger targets implemented were the Intel 8086 family. The next step, now underway, is to implement targets in the CAPS family, including the CAPS-7 and CAPS-10 processors used in Rockwell GPS equipment, AAMP (Rockwell's single-chip CAPS processor), and Advanced CAPS, an Ada-oriented stack architecture machine.

Other targets, for example 68000 and MIL-STD-1750, may be implemented, depending on project and contractual needs.

Several users have asked about an Ada debugger for PC-based applications; this would also facilitate changes to the Debugger itself. Such a debugger could be implemented either like the standard PC DEBUG program, or with a modified target interface pod and a second PC.

As part of an avionics project, a PC-based tester has been developed which simulates aircraft sensors and instruments via high-speed ARINC I/O. The Debugger will be integrated with this system to provide a powerful tool for running "canned" test sequences.

Users would like more sophisticated breakpoint functions and more hardware support for real-time program analysis and optimization. Features may be added if/when the tester hardware is enhanced.

Possible new features include a target memory diagnostic routine and a command to save the Debugger state and exit to MS-DOS temporarily. The command file feature may be expanded to include parameters, loops, and conditional execution.

The RS-232 link used for downloading files from the VAX is adequate, but not especially convenient. Several alternatives are being investigated, including a smarter downloader which sends only the updated files. A PC-to-DECNET link, or a similar high-speed hardware connection to the host, would be ideal.

It would be convenient if the Debugger were able to display the Ada source code in a history or disassembly listing. This would be easy to do, except for the increased time needed to download the source files, or expanded Line Table files, to the PC. With a faster downloader this would be feasible.

The Debugger has limited support for variant records, discriminants, and fixed-point. These areas will be enhanced as the compiler is upgraded to full Ada. ICSC plans to validate the compiler in 1985.

### Summary

This Ada Symbolic Debugger is a tool designed to improve productivity in software development and testing, in a real-time embedded computer environment.

The Debugger is currently being used to develop several major avionics systems based on 8086-family microprocessors. Interfaces to other target machines are planned or underway.

The Debugger is written in Ada, and is based on the inexpensive and readily available IBM Personal Computer. Application software is developed on the VAX host using the ICSC Ada compiler.

A secondary result of this project is a system for cross compiling Ada programs on the VAX for execution on the IBM-PC.

This system is being used to develop other PC-based tools and application programs.

### Biographical Data

Daniel W. Lyttle is a project engineer for support software in the Processor Technology Department, Collins Government Avionics Division of Rockwell International. He was responsible for the design and implementation of the Ada Symbolic Debugger.

Earlier projects include the CAPS symbolic debugger, the CAPS linker, a PC-based microcode assembler, and diagnostic software for the Fault Tolerant Multi-Processor.

Dan received the BS degree in Computer Science from Rose-Hulman Institute of Technology in 1979, and has done postgraduate work at the University of Iowa and Iowa State University.

# PARALLEL PARSING USING ADA

W. H. Carlisle and D. K. Friesen

Texas A & M University

Abstract

A parallel parsing algorithm for a grammar is developed and implemented in Ada. The parse of the string of terminals produced by the grammar achieves parallelism by making two assumptions of the grammar; first, that if the leftmost son of a production has been parsed, then the production generating that string is uniquely determined, and second, a scanner is able to determine the end of the string produced by this production. At this stage the parsing of the string can be split into two tasks, parsing terminals derived from the production and (assume this task will successfully complete) continue the parse of the string using the production as the leftmost son of the next production.
The algorithm is implemented in Ada for a simple expression grammar.

The Laboratory for Software Research at Texas A&M University is involved in the study of concurrent programming methodology. The Ada programming language has been an integral part of this study, for Ada has raised the language level at which parallel processing algorithms can be coded and evaluated. One area under investigation is that of parallel compiling. Considerable research and knowledge exists about the subject in a single processor environment, but there is a lack of research concerning compilers capable of using parallel architectures. The following is a report of the preliminary work done in this investigation.

A natural approach to the problem of parallel compilation is to break into tasking units the distinct phases of classical compilers - lexical analysis, parsing, etc. so that working in parallel the code can be parsed sequentially by these phases. This investigation, however has proceeded in a different direction, focusing on the parsing phase of compilation, and looking at the design of an algorithm to perform this one task in a multiprocessor environment. A design goal was to seek an algorithm that remains powerful enough to concurrently parse grammars whose description is reasonably close to that of current computer languages. It was deemed desirable not to try to force a redesign of the traditional syntactic constructs in order to make concurrent parsing easier. The realities of parallel parsing, however, suggests redesign of certain syntactic constructs is a direction worthy of study.

## Parallel Parsing Algorithms

Lexical analysis itself breaks into two natural subprocesses, scanning the text (comparison) and analysis (for example table look-up). These two processes are so closely interrelated that this does not appear to be a natural place to divide the work of parsing. Instead an alternative approach is to divide input and perform the same parsing task on the divided input. A similar approach was announced by Mickunas and Shell[1] but our work differs from their approach in that the breakup of the string to be parsed was not based on a uniform length breakup of input. Uniform string division requires communication between adjacent parsers in those situations where the breakup does not correspond to terminals derived from a common ancestor. Our approach was to seek to divide the string to be parsed along the family lines of the parse tree, that is divide the string according to subtrees whose root is a nonterminal of a production. For example, consider the following BNF description of a conditional statement:

```
conditional_statement::= if condition then
                           true_statements
                         else
                          false_statements
                         endif;
```

Upon recognizing this construction a parser can begin a parallel processing of the true_statements and of the false_statements and the condition. Raghavendra[2] has proposed a completely different approach to parallel parsing.

## Parallel Parsing by Productions

Both top down and bottom up parsing was considered. From the top down, starting at the root, the approach would be to determine the production, and break the string apart according to the terminals and nonterminals of the production, and produce associated subtasks to parse the parts of the string so determined. For the above example, in a top down approach a "statement_handling task" would parse the "if" and recognize the conditional statement construct. This task would then break apart the string being processed according to the "then" the "else" and the "endif" occurring at the same nesting level. The task could now spawn the "condition task" and two "sequence of statements tasks" to process the divided input.

Bottom up parsing would be conceptually the same except the direction of motion would be up along the leftmost sons of the parse tree, spawning tasks to complete the parse of strings produced from the non-terminals (in the same bottom up manner). Continuing the example, but now bottom-up, the parent task encounters the "if" once again signaling an "if production". A division of input is again possible using the "then" the "else" and the "endif". Four bottom up parses can be initiated, one for the string representing the condition, two for the strings representing the sequences of statements and one for the remainder of the string to be processed. Both top down and bottom up approaches are made possible with Ada since all tasks would be of the same type, and Ada supports dynamic task type instantiation.

## Top Down vs Bottom Up Parsing

Top down parsing is conceptually simpler, and this approach needs further study and consideration. However, the bottom up approach was chosen because the process of dividing input (scanning) appears to be more difficult when one is near the top of the parse tree. To come up with a reasonable algorithm seemed to be forcing restrictions on the description of the language to be parsed that was stronger than that desired by the goals of the study. Proceeding from the bottom up, one needs to scan only enough of the string to determine a production. Without analysis the scanner could then determine the limits of the string terminals generated by the production, and the task of parsing could be broken into several tasks to handle this string. The task itself could then reinitialize to continue parsing the remainder of the string or spawn yet another task to perform the remainder of the parsing. To make the approach work, the following restrictions would have to be met by the grammar being parsed. For determinism, a unique production must be established by the initial scan, and in order to maintain the string breaks along the ancestor lines of productions, it was decided that a production of the grammar would be determined by the next symbol after the scanner had processed the leftmost son of the production. Having chosen a production the scanner must be able to find quickly the end of the string determined by the non-terminals of the production. This forces restrictions on the grammar directly related to the complexity of the scanning process. Most programming language syntax productions have a terminating indication that, combined with level counting to handle nesting, provides the necessary information for a scan based only on compare and count. A unique symbol found at level 0 determines the end of strings generated by the production.

## An Example

A grammar that satisfies the restrictions required for a bottom up parse of the type we have been discussing is the familiar expression grammar for LR parsing[3]:

1. E -> E + T
2. E -> T
3. T -> T*F
4. T -> F
5. F -> (E)
6. F -> id

The parser is table driven, and has as input an initial state representing the state initial to the parse, a state representing the state after the parse of part of the string (the leftmost son of the production to be determined), the next symbol under the scanner, and the end of the string to be parsed. The task accepts this information and by table lookup determines the production. The scanner, based on knowledge of the production scans to find the ends in the string of terminals derived from the non-terminals of the production. At this stage parsers may spawn subtasks to parse these strings. For the remainder of the string to be processed, two possibilities arise: the task itself can continue parsing the string or the task can create yet another parser to perform this task. If parsing is the only goal of a task, then the parser should take on the work of continuing the parse. In anticipation that attributes should be considered along with the parse, delay to wait for subtasks to establish these attributes might be undesirable. Spawning another task to go on with the parse and waiting to handle attributes of the production is in this situation desirable. Both approaches were coded in Ada to anticipate experimentation with attribute handling.

A look at the parse of the string a + a*a + ..... will illustrate the action of the parser for the example grammar. The needed tables to trace this part of the parse are the GOTO table and the production table.

```
                                  non-terminals
                                    E   T   F
GOTO:              state: 0         1   2   3

                                  terminals
                                      +
production      states:(0,1)      E-> E+T
```

Entering the parse after the productions F->id; T->F; and E->T have been applied, the parser has initial state 0, final state GOTO(0,E) which is 1 and symbol under the scanner is '+' .

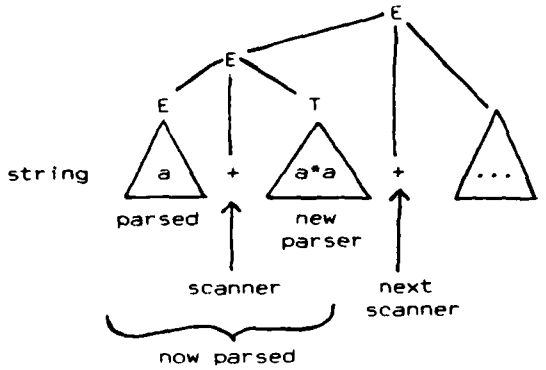



Table lookup determines that the next production is E -> E + T. The scanner now knows that the + symbol (in the follow of T at level 0) will end the terminals derived from T. A new parser can be initialized to parse the string derived from T, here (a*a), and either spawn a new task to go on (assuming T parses correctly) or restart itself with begin state 0, second state 1 = GOTO(0,E) (the left hand side of the production) and next symbol '+' (the next '+' in the string) and continue the parse itself. Note that each task has a goal, that of the main task being the start symbol, and that of the subtasks being the non-terminal that spawns the tasks. The implementation should check for this goal, and one coding of the algorithm incorporated this idea.

## Ada Code

The Ada code implementing the parse was run on the validated NYU Ada/Ed compiler/interpreter version 1.41. The version initiating the fewest number of tasks has as parser type:

```
task type PARSER is
  entry
    INIT(STATE1,STATE2,NEXT_SYMBOL,
         END_OF_STRING:in INTEGER);
end PARSER;
```

and as a driver:

```
begin
  MAINTASK := new PARSER;
  MAINTASK.INIT(0,PARSETABLE(0,
          INPUT(1)),2,STRING_LENGTH);
  end;
```

The body of the parser task type is to accept initialization and to parse until the next symbol being scanned is the end of the string being parsed.

```
  while NEXT_CHR < END_OF_TEXT loop
    PARSE (BEGIN_STATE,END_STATE,
           NEXT_CHR,END_OF_TEXT);
  end loop;
end PARSER;
```

The call to procedure PARSE is essentially table lookup to determine the production and new task instantiation if necessary. The case for the first production and the associated scan and task instantiation is included here.

```
begin
  case PRODUCTION_TABLE(BEGIN_STATE,
       END_STATE,INPUT(NEXT_CHR)) is
   when 1 =>  --production is E -> E+T
              -- so  find end of T and
              --parse
     COUNT := 0; -- level of nesting
     NEXT_CHR := NEXT_CHR + 1;
     SCAN := NEXT_CHR;
     while  SCAN <= END_OF_TEXT and not
                          DONE loop

       case INPUT(SCAN) is
         when '$' | '+' =>
           if COUNT = 0 then
             DONE := TRUE;
           else
             SCAN := SCAN + 1;
           end if;
         when '(' =>

...

       end case;
     end loop;
     if not DONE then
       ERROR_HANDLER.SCANERROR(SCAN,1);
       NEXT_CHR := END_OF_TEXT; --to
                --terminate gracefully
     else  -- initiate a parse of T
      STATEA  :=  PARSETABLE(END_STATE,
                      TERMINALS'('+'));
      STATEB    :=   PARSETABLE(STATEA,
                      INPUT(NEXT_CHR));
      SUBTASKS := new CHILD;
      SUBTASKS.INIT(STATEA,STATEB,
                      NEXT_CHR+1,SCAN);
      END_STATE  :=  GO_TO(BEGIN_STATE,
                                  'E');
      NEXT_CHR := SCAN;
     end if;

...
```

## Error Handling

A minimum of error handling was incorporated into the code, in that a table lookup error merely reported this to an error handler task. Appropriate errorhandling for the algorithm was not addressed.

## Future Directions

Further work is to be done with attributes (such as code generation or value for this example). The primary problem in attribute handling is that of communication. In bottom up parsing, if a task creates subtasks to perform the parsing of subtrees, then the attributes are established by much later generations. Completed tasks must then spend their time gathering information from their children and returning this information to their parents. A more distributed attribute gathering mechanism would be desirable, but this would mean linearizing the returned information. This could be accomplished by communicating the limits of the string along with the associated information, but would be computationally expensive. Care would have to be taken so that the information gatherer did not serialize the performance of the parsers. Additional work could also be done in evaluating more carefully a top down approach and what restrictions would need to be made on the grammar to effectively parse a string in this manner. Attribute handling in this situation should be simpler.

## REFERENCES


1. Mickunas, M.D., and Schell, R.M., Parallel Compilation in a Multiprocessor Environment, Proc. ACM Annual Conference, 1978, pp. 241-246.

2. Raghavendra, R.L. A Note on Parallel Parsing, SIGPLAN Notices, v. 19, #1, Jan. 1984, pp. 57-59.

3. Aho, A.V., and Ullman, J.D., Principles of Compiler Design, Addison-Wesley Publishing Co., 1979.



W.H. CARLISLE is an Assistant Professor of Computer Science at Texas A&M University where he has taught for four years. He holds a Ph.D in mathematics from Emory University. His research interests include abstract data structures, algorithm analysis, concurrent programming, and languages.

Department of Computer Science
Texas A&M University
College Station, Texas 77843
(409) 845-5481



D.K. FRIESEN is an Associate Professor of Computer Science at Texas A&M University where he has taught for x years. He holds a Ph.D in mathematics from Dartmouth College and a Ph.D in computer science from the University of Illinois. His research interests include algorithm analysis, artificial intelligence, and compiling.

Department of Computer Science
Texas A&M University
College Station, Texas 77843
(409) 845-5401

# A DYNAMIC PROGRAM PROFILER FOR TESTING ADA PROGRAMS

ANIL KUMAR SAHAI

DEPARTMENT OF COMPUTER SCIENCE, PLYMOUTH STATE COLLEGE

**ABSTRACT**

It is a well known fact that testing and debugging account for a major part of the total cost of design and development of a reliable software. Seldom is that software adequately tested before it is placed in the production. The major reason being the absence of implementable and efficient techniques for the above purpose. In this paper, I discuss some of the existing techniques in the context of the Ada language. It is proposed how a dynamic program profiler can be used to produce the necessary run time statistics of the control flow of program to help in program testing, debugging and possible code optimization.

## 1. INTRODUCTION

The question of determining whether or not a given program will do exactly what it was designed to do is not only intellectually challenging, but is also of primary importance in practice. Obviously, an ideal solution would be to develop techniques that can be used to produce a formal proof of the correctness (or incorrectness) of the program. There have been considerable efforts to develop these techniques, and several techniques have been reported. But all of them have been of theoretical interest, or they have not been developed to the point so that they can be readily applied in practice. The main reason being: in developing these techniques, the basic approach has been to translate the problem of proving program correctness into that of proving a certain statement is a theorem in a formal system. In section 2, I discuss some these existing techniques in the context of Ada. Symbolic Execution [1] is not practical in that it is difficult to implement efficiently. This technique suggests running the program using symbols in place of the input values and storing intermediate values of the variables in an algebraic expression form. The problem of efficiency is the amount of time spent in these manipulations. It also suggests forking the execution in various paths simultaneously after a decision statement, which is not practically possible, especially if there are many possible paths after a decision expression (eg: case statements). Finite Domain Testing [2], is another technique for program testing which is mainly of theoretical interest. It is claimed that all but a small class of errors can be detected by this technique. Functional Program Testing [3] is another approach to provide a tool for software testing. It is close to being practical but it assumes the knowledge of the language structure on the user's part, which is not practical in the real world. The idea is to break the program into a series of small functions computing certain variables and the sections of the program and then the whole program can be thought as the aggregate of these functions. It is suggested to is to compute these functions and then test for the correctness of the whole program. But the user has the responsibility of locating these small functions in the program. Finally, there is Software Probes [4] technique. This technique can be readily applied in practice, but the technique was developed in the context of FORTRAN and thus has its shortcomings. I have attempted to extend this technique and the idea of dynamic program profiler [5] to produce the run time control flow graph of Ada programs. In section 3, I discuss the design of dynamic program profiler, the statistics produced by the profiler and how this tool can be used for testing and debugging of Ada programs. I have also suggested that with slight modifications, how this technique can be used for software testing on multiprocessing systems.

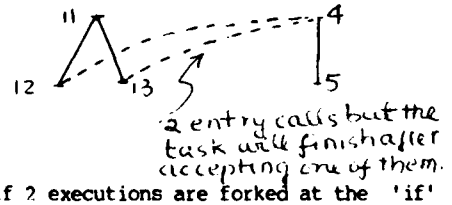
## 2. EXISTING TECHNIQUES

In this section I describe some of the existing program testing techniques in the context of Ada. While discussing these techniques, I give examples from Ada and propose some changes in the techniques to incorporate testing for Ada programs. It will be seen that in spite of these suggested changes, the techniques can not be developed for practical implementation.

### 2.1 Symbolic Execution [1]

The notion of symbolically executing a program follows quite naturally from the normal execution. The basic assumptions are:

```
MAIN: 11    IF K = M THEN
      12       T1.A(M);
      13    ELSE K := 3;
      14         T1.A(K);
      15    END IF;
      16    ...
      17    ...

TASK:  4    ACCEPT A(T:INTEGER) DO
       5                K := 7;
       6    END A;
       7    END T1;
```




Even if 2 executions are forked at the 'if' statement, only one of them will be able to perform the rendezvous because after that the task finishes and the other forked execution in the program (line 14) will wait forever for the 'accept' in the 'already finished' task. Symbolic execution was proposed using the notion of concurrent activities so that different parts of program can be forked at the same time. But if the language itself provides a tasking facility, the problem of testing with the help of symbolic execution becomes even more severe. More work will be needed to be done in this regard to take care of the tasking.

## 2.2 Finite Domain Strategy [2]

It is clearly obvious that the only completely effective strategy is an exhaustive testing (to test the program for all possible set of input values), which is totally impractical. Even a simple program to add two numbers will take a very long time for exhaustive testing. An idea could be to view a program as a function from N-dimensional input variables to M-dimensional output variables. Because of the nonfeasibility of the exhaustive, testing, it is suggested to select a small subset of the domain of the program function and then test for the program correctness. The finite domain strategy is used to determine the necessary set of input values for testing and is shown to be successful in detecting all class of errors except for a small subclass called 'missing path errors of reduced dimensionality'. The program is executed with the small set of 'chosen' test data. If the program produces an incorrect result, the the program is incorrect; otherwise that the program is correct. Obviously, the confidence in the above inferences will depend depend upon how well the stated as follows : 'Given no information other than the program to be tested, generate a small set of sample input data which if processed by the help of program will insure with a high degree of confidence that the program is correct'.

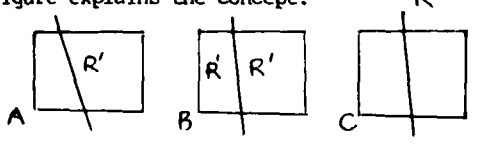
Broadly speaking, there are 4 classes of errors:

1. Coincidental Errors: The basis of the program testing is that from the observed correctness of the program over a small set, we infer that the program is correct. For example: J := I^2 will produce the same result for I=K or -K or for any K. The idea is that if the program produces the correct output using the wrong computations at the test points and it leads us to believe that the program is correct, the basis of inference is destroyed. This phenomenon is a basic obstacle in producing a completely reliable programs. However, in a practical sense, the choice of wide range of the input test data minimizes these kinds of errors.
2. Transformation Errors: Sometimes the value of a variable is computed incorrectly and consequently that variable used in a decision expression leads the program into an incorrect flow of execution.
3. On the other hand, if a variable is used to evaluate any other output variable and that variable is never used, it is not possible to check it by following the execution flow.
4. Missing Path Errors: Sometimes, there should be one more test along the path, typically when one condition has been omitted. These types of errors are called the 'missing path errors'. These types of errors can not be checked but the function domain strategy takes care of all but a small subclass of these kinds of errors.

The following table summarizes the kinds of errors:

| Class | Domain | Transformation |
|---|---|---|
| 1 | Correct | Incorrect |
| 2 | Incorrect | Correct |
| 3 | Incorrect | Correct |
| 4 | Incorrect | Correct |

Assuming that the missing predicates are simple, we can form 3 subclasses of the missing path errors. (1) Inequality predicates, (2) Equality and (3) Not equality predicates. If we have our test data in a plane region, these predicates are the hyperplanes, cutting the region under the test. Also note that inequality predicates will be on one of the sides of the test data plane. The Not equality will account for all minus the test data hyperplane. In any event, we can discover these kinds of errors but equality predicates will be some where we do not know. Also note that this hyperplane will form subregion of measure zero in comparison with the entire region, hence these types of errors can not be detected. Obviously, some these errors can be detected by expanding our set of test data. The following figure explains the concept:



1. There is a programming language and a normal definition of program execution of that language.
2. The program are not to be modified for the symbolic execution.

The idea is to use algebraic symbols in place of the input values (other constants of the program can assume their values). Once the program has been initiated with these 'input' values, the execution of the program can proceed in the normal fashion except when the symbolic inputs are encountered. But this could happen on each step, directly or indirectly. For example consider :

```
1   READ(X) ;
2   Y := X + 2 ;
3   Z := Y - 2 ;
```

Here the symbolic input X does not appear on line 3 directly, but the variable Y is computed using X on line 2. The evaluation in the terms of the input variables can occur in two cases :

1. Computation of expressions: The idea here is to delay the evaluation of the expression and the results are stored in an algebraic form. For eg: Y = V(X) + 2, then Z = Y - 2 ==> Z = V(X), where V is the value function. But this process could take a lot of time because it has to do some polynomial arithmetic in the intermediate steps.

2. Conditional Branching: Let us take an example of a decision making program statement.

```
IF E1 THEN S1;
ELSIF E2 THEN S2;
ELSIF E3 THEN S3;
ELSIF E4 THEN S4;
....
ELSIF En-1 then Sn-1;
else Sn;
```

The Ei's are boolean expressions of the language and Si's are the statements. Obviously, the value of E1 will depend upon the actual values of the input symbols (if any, in E1). Hence, V(E1) could be either 'TRUE' or

```
V(E1) = F1( V(A1), V(A2), .... V(An) )
```

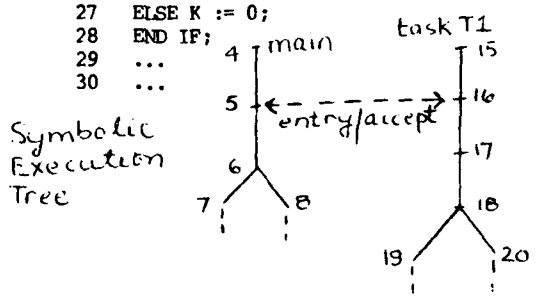
where F1 is some algebraic function and A1, A2, A3, ... An are input variables. Therefore the control flow of the program will depend upon the values of the expressions Ei's. The idea here is to fork the execution in 'n' different statements, and store the values of each V(Ei) and complement of the expression V(E1) AND V(E2) AND

In Ada, other decision statements such as the 'case', 'loop', 'while' etc can be similarly implemented because any of these statements can be implemented using the 'if' and a 'goto' statement (this is a fundamental result in the theory of computability).

Tasks in Ada provide a parallel computation facility. Since in the symbolic execution we use the idea of forking execution in several directions simultaneously, tasks can be easily incorporated. In case of an 'accept' or 'entry' call, the execution will be suspended if the related execution has not reached the required step for rendezvous. But there still exists the problem of deadlocks and livelocks. These problems have been discussed with an example in this section under comments.

The control flow of an Ada program can be represented by an execution tree using the sequence of execution. Each executable statement is represented by a node of the tree. The node also carries the algebraic expressions in the terms of symbolic variables and the value functions. If the statement is a decision making statement, several branches coming out of it will represent the different executions forked at that step. Each node also carries the expression for PC, so that the user can check those values for the program correctness. For the Ada programs with parallel executable tasks, a symbolic execution tree could be like a distributed network of trees such that there are communication lines between two trees for passing information in the event of a rendezvous. This will help the representation of the 'accept' and 'entry' calls. The following figures shows an Ada program segment, with the symbolic execution tree.

```
MAIN : 4    X := 10;
       5    T1.A(X);
       6    IF X = 10 THEN
       7    Y := 7;
       8    ELSE Y := 8;
       9    END IF;
       10   ...
       11   ...

TASK : 21   I := 7;
       22   ACCEPT A(K:INTEGER) DO
       23   K := 10;
       24   END A;
       25   IF K > 10 THEN
       26      K := 100;
       27   ELSE K := 0;
       28   END IF;
       29   ...
       30   ...
```




**2.1.1 Comments** King [1] accepts that the a loop in the program controlled by an input variable. The technique is far from being applied in practice. The problem of saving and manipulating the algebraic expressions at each step will normally increase the cost of testing a program tremendously. Also, in case of Ada, this technique may lead to a deadlock in the system as evident from the following section of code :

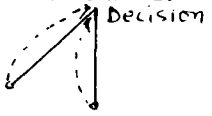
Rectangles represent the test data region.
In A, R' will account for data > or < test data, therefore one of the sides of 'Hyperplane' H.
In B, R' will account for <> test data, there it can be on the either sides, ie; > or <.
But in C, data could be anywhere in the region because missing test is for '=' comparison.

2.2.1 Comments The domain testing strategy offers an alternative to the exhaustive testing, but on the other hand it assumes that each path will be taken at least once. However the number of possible paths could be extremely large even in a program of small size. The problem of coincidental correctness has been identified as an inherent theoretical limitation to any testing procedure. A 'path selection strategy' should have been suggested because an incorrect computation in a path predicate can effect the subsequent path predicates. The technique is of more theoretical interest than to be applied in practice.

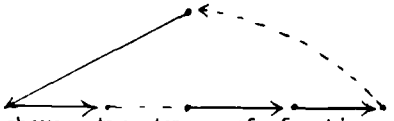
## 2.3 Functional Program Testing [3]

Each program can be viewed as a combination of smaller modules such that each module computes a function. The idea is to represent the program as a concatenation of these functions and then to test its correctness. it is required that complete set of functional tests are produced for each of the functions which are the parts of the program design. The design functions can be classified in the following three categories:

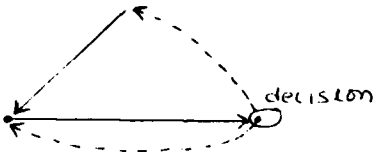
1. Some functions are parts of the program design and will correspond directly to the sections of the program and can be easily located. For example functions and procedures in Ada. The following figure shows such a function and the dotted line denotes the return of the control.




2. Program could be broken into series of codes, where each part corresponds to a small function. The following figure shows such a function.



3. The above two types of functions are computational types and can be easily located. But some programs exist in a program in a subtle ways. Recursive functions come under this category. Control of the flow of execution is not only in the main program but it is also there in the procedures themselves. These are called control functions. The following figure shows one of such functions.




2.3.1 Testing Functions in Context Suppose that a function f is part of design of a program P and one of the input variables X is a part of the function f. Suppose f is invoked when X < 2 and X can assume any value in the main program. If the program is tested for the arbitrary values of X, we shall be testing the function for the values of X > 2 also, which would be illegal. This problem of determining the context of function being tested imposes another problem to all the techniques discussed so far, as in selecting the test data we do not take care of this. Also note that if the context of a function is defined in terms of a complicated expression of other functions, it will become extremely difficult to evaluate the context of the function. Another problem here is to represent tasks as functions because of the fact that tasks execute simultaneously with the other procedures. Also the presence of entry calls will contribute additional problems to determine the context of this function.

2.3.2 Comments There are three key steps in this approach to program testing as suggested in the paper:

1. Identification of input and output data.
2. Functional decomposition of the data structures into design structure.
3. functional decomposition of program into design functions.

So far as the practical applications are concerned, this technique provides little help to a common user. The technique assumes that the user has a thorough understanding of the program structure (which is almost impossible in case of large programs) and also the user has the responsibility of decomposition of program into smaller functions.

## 2.4 Software Probes [4]

This technique is not intended for the program validation, but rather it is meant to gather the necessary run time control flow statistics to facilitate the user in program testing. The user is responsible for looking at the program code, the control flow and inferring the correctness of the program. The technique is primarily designed for inserting probes in the FORTRAN programs and thus has its limitations in the context of Ada. The restrictions are :

1. each program has single entry/exit point.
2. no functions in the computed 'goto' statements.
3. no provision for recursion or parallel executable tasks.

2.4.1 Program Graph Representation The idea is to represent each executable part of each statement of the program as a node. The program graph is constructed as follows:

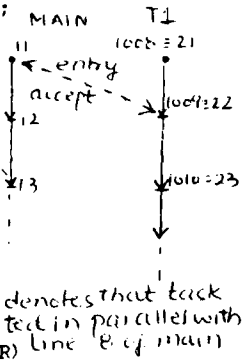
" First statement is assigned a node number. Then every new executable part of the next statement is

assigned the next number. Arcs are drawn from a node to other nodes to represent the control flow"

This technique can be applied to produce the program graph of an Ada program, with the following additional steps:

1. a recursive procedure or a functional call can be represented by a cycle.
2. tasks can be represented as separate trees/graphs with communication links to other tasks to perform the rendezvous. The tasks are numbered so that the parallel execution with other tasks can be inferred. For example, the following figure shows an Ada program with a recursive function call and an entry call, and its program flow graph.

```
    FUNCTION FACT(I:IN OUT INTEGER) IS
    BEGIN
1       IF I=1 THEN
2           FACT := 1;
3       ELSE I := I-1;
4           FACT := FACT(I)*I;
        END IF;
    END;

    MAIN:
        ...
        ...
10      J := 5;
11      PUT(FACT(J));
12      T1.A(J);
13      PUT(J);
        ...
        ...
    TASK T1:
21      K :=7;
22      DO ACCEPT A(I:INTEGER)
23          I := 9;
        END A;
        ...
        ...
```




2.4.2 Probes Insertion To produce the program graph, probes have to be inserted at certain strategic locations in the program. A *software probe is of the form*

CALL <probe name> (n1,n2)

where <probe name> refers to the special auditing program and n1, n2 are the number of the executable parts of the statement before and after the probe call. This auditing program prints out the pair (n1,n2) when invoked.

The probes are inserted at every branch of the program graph. An arc between two nodes is called as Decision to Decision (DD) path. When the program is executed after inserting these probes, the results include a sequence of pairs, denoting the DD paths of the program. When the program is run for a sample of test data, these pairs can be used to test the correctness of the program. There might be a case when a DD path is never taken for a sample test data. It can then be checked if this was due to the test data or inherent in the logic structure of the program.

2.4.3 Comments This technique does not provide automatic probe insertions and hence the user has the responsibility of inserting probes. The user is expected to have the knowledge of the language structure to understand the graph tree. At the same time he has to locate the points to put the probes and assign the node numbers to various executable parts of a statement. Because of the above mentioned problems, this technique is far from being applied in practice for a common use.

## 3 DYNAMIC PROFILER

It is evident from the previous section that none of the discussed techniques provides a practical solution to be applied in practice for testing and debugging of Ada programs. I intend to improve upon some of the techniques and implement a feasible solution. This profiler combines the technique of gprof[5] and the idea of software probes[4]. The profiler produces the statistics about the run time calls and transfer control in a tabular form for the user. The technique of using software probes required that the user inserts the probes at the right places. My profiler produces these statistics automatically. I had to improve the compiler of Ada at various places so that it becomes independent of the source program and produces the results for any program of the language.

3.1 Design The dynamic profiler was designed by adding code to the various phases of an Ada interpreter. The interpreter is implemented in the following phases:

1. Lexical Analyzer: to pick up the tokens of the language.
2. Parser: It is a 'Bottom Up' table driven parser. It produces a file called 'triple', specifying the reductions to be carried out.
3. Code Generator: This phase produces an intermediate code (quads) by carrying out the semantic actions corresponding to the reductions performed by the parser.
4. Interpreter: To interpret the quads generated by the code generator.

3.2 Run Time Statistics Each node of the execution tree is denoted by a statement line number, so that any common user can understand the output from the profiler. When a program is executed a file called 'Runtime' is produced with the following information in the chronological order (Real time, CPU time and System Time).

1. Transfer of control is represented by a pair (m,n), denoting the transfer of control from line number 'm' to line number 'n'.

2. If a line contains an 'if' or 'while' or 'exit' statement, it prints out the value of the conditional expression.

3. If there is a 'case' statement, it prints out the value of the 'case' expression and prints out the line number where the 'case' expression was matched.

4. If there is a 'for' loop, it prints out the value of the control variable, each time loop is executed.

5. In case of a procedure call, it prints out the name of the procedure called, the types and the values of the parameters passed. When the control is returned from the procedure, it prints out the values and the types of the returning parameters. In case of a recursive procedure call, it also prints out the level of recursion.

6. If the program contains any tasks, it prints out the currently executing task on each executable line.

7. In case of an 'entry' or 'accept' call, it prints out the names of all the tasks in the ready queue and suspended queue

The accompanying figure shows an Ada program and the runtime statistics produced by the profiler. Each transfer of control to other line is denoted by a pair (m,n) as mentioned before. By looking at the runtime statistics and the program listing, runtime control flow can be easily traced.

3.3 Program Testing Howden [3] has defined two broad categories of program errors under the names 'Domain Errors' and 'Computation Errors'. A domain error occurs when a specific input follows the wrong path due to an error in the control statement. A computation error occurs when an input follows the correct path, but because of error in computation statements, the wrong function is computed for one or more of the output variables. The most important data required for program testing and validation is the frequency of occurrences of different types of errors and their severity. It is a reasonable to assume that the number of decision statements increases as the size of the program. The other types of errors like incomplete specification, erroneous data accessing, erroneous automatic computations etc., can be detected using compilers and debuggers. The following table (reproduced from [6]) shows the frequency of sequencing errors found in programs.

| Error Category | Total Number | Percent |
|---|---|---|
| Branch Test Incorrect | 28 | 20 |
| Computation in the Wrong Sequence | 9 | 6 |
| Logic Sequence Incorrect | 98 | 71 |
| Branch Test Set Up | 2 | 2 |

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
 listing of the source program
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

procedure test4;
  1  i,j,k : integer;
  2  task t1 is
  3     entry a(a1: in out integer);
  4  end;
  5  task body t1 is
  6  l,k : integer;
  7       procedure p(i : in integer) is
  8       kk : integer;
  9       begin
 10            i:=i+2;
 11            put(i);
 12       end;
 13  begin
 14       k:=15; l:=6;
 15       p(k); put(k);
 16       accept a(a1 : in out integer) do
 17       l:=l+a1;
 18       end a;
 19       p(l); put(l);
 20  end t1;
 21  begin
 22       i:=10; j:=3; k:=4;
 23       t1.a(i); put(i,j,k);
 24  end ;
 25

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$                                               $
$ Run Time Statistics of the source Program     $
$                                               $
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$


 Real Time CPU Time System Time          Flow
 --------- -------- -----------          ----

 00:43:23    883        133     (21,22)
                                The task executing is  main
 00:43:23    883        150     line 23'entry' call

                                other tasks status
                                ------------------
                                The task executing is  main

                                The followins tasks are suspended:
                                ----------------------------------


                                The following tasks are in ready queue:
                                ---------------------------------------
                                        1.  t1
 00:43:24    916        150     (22,13)
                                The task executing is  t1
 00:43:24    916        150     (13,14)
                                The task executing is  t1
 00:43:24    933        150     line  15 call procedure 'p'
 00:43:24    950        150     parameter 1 is 'in' type, value is    15
 00:43:24    966        166     (14,10)
                                The task executing is  t1
 00:43:24    966        183     (10,11)
                                The task executing is  t1
 00:43:24    966        183     returning parameter 1 is 'in' type, value was    21
 00:43:24   1000        183     (11,15)
                                The task executing is  t1
 00:43:24   1000        183     line 16'accept' call

                                other tasks status
                                ------------------
                                The task executing is  t1

                                The followins tasks are suspended:
                                ----------------------------------
                                        1.  main
 00:43:24   1033        200     (15,17)
                                The task executing is  t1
 00:43:24   1033        216     line  19 call procedure 'p'
 00:43:24   1066        216     parameter 1 is 'in' type, value is    16
                                recursion level is  2
 00:43:24   1083        233     (17,10)
                                The task executing is  t1
 00:43:24   1100        233     (10,11)
                                The task executing is  t1
 00:43:24   1100        233     returning parameter 1 is 'in' type, value was    32
 00:43:24   1133        233     (11,19)
                                The task executing is  t1
 00:43:24   1150        233     (19,23)
                                The task executing is  main
The following lines, if they had any executable code were never executed
line  24
```

It can be seen from the table that 91% of the sequencing errors are caused because of either incorrect branch test or wrong sequence of computation. These types of errors can be easily tested using the results of the profiler. For example the following section of code tests whether a year is leap year or not. Notice that if the year is divided by 400 it is unnecessary to check for the division by 100. Hence the logical expression formed by the conjunction of two conditions is equivalent to just the second term alone.

```
16 REM4   := YEAR - 4 * (YEAR/4) ;
17 REM100 := YEAR - 100 * (YEAR/100);
18 REM400 := YEAR - 400 * (YEAR/400);
19 IF (REM4 = 1) OR ((REM100 = 0) AND (REM400 = 0)
20   THEN DAYSIN(2) := 28;
21   ELSE DAYSIN(2) := 29;

  ......

  ......
```

The profiler prints out the values of the input in parameters and the output parameters after the procedure call. These values can be tested to locate the errors due to parameter mismatch. Sometimes the parameter passing errors are because of the wrong types of parameters. For example, if an 'in' type of parameter is used to retrieve a value from the procedure it can be tested and checked using runtime statistics. A user can see the types and the values of the 'going in' parameters. The presence of 'infinite loops' is particularly easy to detect using the control flow information produced by the profiler.

The profiler prints out the status of all the tasks at each 'entry' and 'accept' call. If one finds that, at all the 'entry'/'accept' calls, no other task was suspended, he can infer that his program may be in a deadlock situation. Since an entry call has to be matched to an accept call, there should be at least one task suspended at some entry or accept call. Also if he finds that at any time no two tasks were executing, it can be inferred that there was no parallel execution. Obviously, this will always be true on a single processor system, because at any instant only one cal execute.

The profiler produces all the required statistics of Software Probes[4] technique. It also produces some additional information about the conditional expressions, loop variable values, case expressions. Hence the statistics completely determine the runtime flow graph of a program. Since all of the data is generated in chronological order, the user can determine other timing statistics of the program execution. For example, he can determine the execution time in the terms of the various units, he can find out the inter leave time of tasks, he can also infer the scheduling policy and finally if there is any time slicing he can determine its effect on (1) program execution time, (2) executions of the tasks.

**3.4 Program Debugging** The program profiler can be very well used for program debugging. Most of the common mistakes are; 'if' expressions testing, wrong values of passing and returning parameters, wrong values of 'for' loop variables (being a common technique for specifying the indexes of an array), infinite loops, wrong calculations and matching of the 'case' expressions etc. The runtime statistics produced by the profiler provides all the information needed to detect the above errors. The most important thing is that the user does not have to put any probes to produce the control flow. Since the output is in terms of the line numbers the output can be used by any common user for program debugging. Deadlock problems in case of the tasks can be detected while looking at the tasks status at the time of an entry or accept call.

**3.5 Code Optimization** The number of decision statements increases as the size of the program. It has been seen that in large programs many statements are never executed because the path following to those statements are never taken for most of the input data. The profiler produces the list of statement numbers which were never executed. This information may be used for code optimization. The code for these statements may be omitted from optimizing in order to reduce the overall code optimization cost and hence the incurred execution cost will decrease. For example, in the large database programs some of the special routines are rarely invoked. Then there is no need of optimizing the codes of these routines.

If the user finds that for most his input data only a few 'case' alternatives are executed, he may want to change the code by replacing the 'case' statements by an 'if' statement. This is particularly significant when the cost of implementing a 'case' statement is much larger than that of a group of 'if' statements.

The runtime statistics can be used to detect if there was any true parallel computation. This can be inferred if it is found that it was never the case that two or more tasks were executing at the same time. For instance, we can determine the percentage of I/O time and the CPU time and let the operating system use these figures to select an appropriate job mix.

**3.6 Limitations** The profiler developed could generate the runtime statistics for any program in the language accepted by the parser. I have added additional variables in the different phases so that the actual code of the interpreter is not affected. In order to do that the size of profiler increased a lot. I also had to pass line number with each token to the parser and with each semantic action to the code generator. This has increased the execution time of the interpreter. I added the fifth field to each 'quad' so that the code for profiler can be added to interpreter. Probably, quads could be generated for this purpose. I avoided it so that numbering of the quads is not effected while doing 'backpatching' or inserting 'jumps'. The profiler was running on

a single processor system, hence the tasks status information will not be of much use, except that the status can be checked at the multiple CPU's, the chronological order will not make much of a sense since each CPU will have its own time. As the profiler itself will be running under some CPU and hence it will not be able to print out the other tasks' information running under different CPUs. I suggest the method of polling other CPU's for status and time stamping each executable statement. The profiler can be implemented as a monitor, while the tasks execute and transmit information to it. Much work needs to be done in this area.

## 4. CONCLUSIONS

In this paper I have discussed some of the existing techniques for program testing in the context of Ada and pointed out that none of these technique is implementable and efficient for program testing. Either a technique is of theoretical interest or it is not suited for the various control structures of the language. The use of software probes [4] was a good attempt towards the solution of the problem of software testing, but its applications are restricted to the Fortran language with some more restrictions. I have explained how a dynamic profiler [5] can be used in testing and debugging an Ada program. The suggested solution is supported by the results I have obtained by designing a profiler and running it. I have also explained how the statistics obtained from the profiler can be used for program testing, debugging and possible code optimization. These statistics can be used for many different purposes by the variety of users for their interests. The main idea of the paper is to purpose an efficient and implementable technique for program testing using a dynamic program profiler. It is shown that the run time information produced by the profiler completely explains the control flow of an Ada program. The purposed technique is implementable and can be readily applied in practice for producing reliable software.

## 5. REFERENCES


1. James C. King, Symbolic Execution and Program Testing : Research Report, R 5082, Oct' 1974
2. Edward Cohen and Lee White, A Finite Domain Strategy for Computer Program Testing : IEEE Transactions on Software Engineering, Vol SE-6, May' 1978, 247-257
3. William E. Howden, Functional Program Testing : IEEE Transactions on Software Engineering, April' 1980, 162-169
4. M. R. Paige and Marshal K. Mckusick, The use of Software Probes in Testing Fortran Programs : Computer, July' 1980
5. Susan L. Graham, Peter B. Kesslerand Marshal K. Mckusick, gProf: A call Graph Execution Profiler: ACM 1982
6. Raymond J. Rubey, Joseph A. Dana and Peter W. Biche, Quantitative Aspects of Software Validation: IEEE Transactions of Software Engineering, Vol SE-1 No 2. June' 1975



Mr. Anil Kumar Sahai was born on April 28, 1960 in Mirzapur, India. He received a B.A. (Honors) degree in Mathematics from St. Stephen's College, Delhi, India in 1979, a M.S. degree in Math/Computer Science from Ohio University, Athens, OH, in 1982 and a M.S. degree in Computer Science from University of Pittsburgh, PA in 1983. Currently, he is an assistant professor in the Department of Computer Science at Plymouth State College, Plymouth, NH. His areas of interest include programming languages, software design and testing, database management systems and compiler construction.

He is a member of Ada Methodman Review Working Group, IEEE, NHACES and the honor society of Phi Kappa Phi.

# AN ADA* DISTRIBUTED SYSTEM

*Yul J. Inn and Mark Rosenberg*

*ESL, A Subsidiary of TRW*
*Sunnyvale, CA*

**Abstract** -- This paper describes an approach to using Ada to program applications which execute in a distributed computing environment. Package specifications from the distributed application program are scanned by a preprocessor to generate additional Ada code which facilitates interprocessor communication with subprograms and tasks on remote processors. Advantages of this approach include transparency to the programmer, use of existing Ada compilers, and hardware independence. A prototype demonstration of this Ada distributed system on an Ethernet based network of 68000 microcomputers is planned for the second quarter of 1985. The software for the system will be a collection of APSE level tools which can be used to develop distributed application programs. Candidate tools include the aforementioned preprocessor, a reusable distributed software library, a distributed interactive debugger, etc.

## Introduction

The traditional approach to building systems has been to have a centralized computer which is responsible for the control and processing of all subsystems. However, the concept of centralized control has a number of inherent weaknesses:

(1) *Poor fault tolerance:* Should the central computer become inoperative, little system capability remains available.

(2) *Resource contention:* Since all subsystems share a single computer, each competes with the others for cpu cycles, disk accesses, and other resources. As a result, response time and system throughput are hard to predict and generally degrade as system activity increases.

(3) *Limited modular reusability:* Incompatible hardware interfaces may constrain the reuse of an existing subsystem in a new system which employs a different type of central computer. Similarly, applications software containing operating system calls requires extensive reimplementation when transported to a computer using a different operating system.

The distributed system approach partitions the system control and processing over a number of computers. A typical distributed system might consist of a hierarchical local area network of heterogeneous computers each of which has limited responsibilities.

This type of system architecture has a number of advantages over a system with centralized control:

(1) *Graceful degradation:* Since the system responsibility of each computer is limited, a single computer failure has limited system impact.

(2) *Localized contention for resources:* Distributed systems can be designed such that the resources required for some subsystems are isolated from other subsystems not requiring their use thereby leaving the performance of one subsystem relatively unaffected by use of another.

(3) *Ease of system expansion:* Additional subsystems can be readily added to the system. The ease with which such additions can be made is a consequence of the modularity and interface standardization which is required to support communication between networked computers. A further consequence of the standard subsystem interfaces imposed by networking is that new subsystems providing greater performance or new functions can be added to the system with little or no redesign.

(4) *Modular reusability:* Subsystems developed for one system can be used in another without change if both systems employ compatible network interfaces and control philosophies.

The decreasing cost of microprocessor based computers coupled with increasing performance provided by advances in hardware technology make the distributed system approach practical as well as attractive. The software development environments for such systems have, however, not advanced as rapidly as the hardware technology. There are only a limited number of high order languages suitable when programming distributed systems. The choice to use Ada (together with its associated programming support environment) as the language to program distributed applications was made for several reasons: (1) Ada supports certain necessary features required of a distributed programming language, (2) Ada compilers will be available for a wide range of target computers, and (3) the DoD is strongly backing the use of the Ada language.

The Ada language offers the necessary features for a language which is to be used in a distributed environment. Multitasking provides the language level support for concurrent programming. Exception handling provides the means to deal consistently with errors arising in the distributed environment. Finally, separate compilation allows the division of complex processes into tasks and supports the

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

mapping of these tasks onto distinct physical processors for parallel execution.

In directive 5000.31, the DoD prescribes Ada as the programming language for all of its mission critical software. Since much of ESL's work is DoD related, Ada is a language which is firmly embedded in our future plans. In addition, this has led to a large effort by compiler writing companies to produce Ada compilers for many of the minicomputers and microprocessors available today. Thus there will not be a shortage of Ada compilers for the programming of the heterogeneous distributed systems of the future. The DoD directive is also the basis for the selection of Ada over other modern languages which support concurrent programming (such as Modula-2 [7], for example).

The decision to use Ada as a distributed programming language is not without its problems however. Although the Ada language does support multitasking, the language reference manual[1] does not require that the compiler support parallel execution on distinct physical processors even when the target system has multiple processors. At the present time, there are no announced plans for compilers which will support execution in a distributed multiprocessor environment and even when such compilers do become available, there may still be compatibility problems when the target system is a network of heterogeneous processors.

The strengths of distributed systems and the reasons for using Ada as a distributed programming language which have been outlined above motivate our investigation into this problem. This investigation has been the object of a research project over the past two years. In the next section the goals and requirements of this project are discussed. In the following two sections we present an approach to the use of Ada in a distributed environment. Finally, the current status of this project along with our future plans are described and the results of our investigation into the use of Ada as a distributed programming language are presented.

## Goals

The use of Ada as a programming language for distributed systems is our primary goal. To reach this goal, it is necessary to design and implement interprocessor synchronization and communication mechanisms which provide run-time support for distributed Ada programs. The design must be practical and the implementation efficient; practicality will lead to reduced software development costs while efficiency will guarantee the suitability of the implementation for real-time applications. In order to achieve these goals, we have identified, during the design phase of the project, three further objectives: (1) transparency to the programmer, (2) use of an existing Ada compiler as a code generator, and (3) hardware independence.

Complete transparency to the programmer, i.e. having the distributed nature of the target system be invisible to the programmer, is a very desirable property. It would be extremely difficult, if not impossible, to provide complete transparency within the scope of our project. Therefore, we have strived to provide the maximum transparency possible within the limits of our other objectives.

There are numerous advantages of transparency to the programmer. For one, the programmer need not be concerned whether his application program will run on a single computer or will be distributed across a network of computers. In the context of the life-cycle of a single system, this results in minimal modification to source code upon reconfiguration and in little or no software modification when upgrade to more powerful processing elements is made possible through technological advances in hardware. The reusability of software between systems is also increased. When building newer, similar systems, debugged application software from previous systems can be reused with very little rework since it does not reflect the system hardware configuration.

Transparency similarly reduces other software life cycle costs. There is less training required for application programmers since they will have to learn neither new language constructs for distributed programming nor details of operating system and network communication primitives. Minimized program debug time, also a result of transparency, is another factor which leads to further cost reduction. Program debug can be performed without source code modifications on a uniprocessor before the final test on the distributed system. Since debug techniques on distributed target systems are less well understood and not as refined as those on single processor systems, overall code debug time is minimized.

It would be impractical to *write* an entire Ada compiler for a distributed targets; that would be an extremely complex task and would extend far beyond the scope of this project. Ideally, the use of Ada to program a distributed system would require no modification to the compiler itself. This would derive the maximum benefit from existing Ada compilers and their associated run-time packages. Since many compiler companies are currently working on compilers for a variety of target processors, obtaining compilers in the future for any of the processors in the distributed system should not be a problem.

In order to make the Ada distributed system programming environment portable to different hardware configurations, the Ada level programming of such a system should be independent of the underlying hardware. Rewriting hardware drivers is typically required when porting software from one hardware system to another. If the Ada distributed system approach is structured to isolate hardware dependencies, then porting application software onto new hardware systems would involve modification to low level drivers but no change to the application software and would therefore require the minimum possible effort. Thus the reusability of the application software is increased and the cost of building similar systems in the future is decreased.

## Approach

Previously proposed approaches to the use of Ada as a distributed systems implementation language provide varying levels of transparency to the implementer. It is to be hoped that in the future run-time support for distributed

Ada programs will be standardized so that large applications can employ a wide range of different computers, network technologies, and compilers without explicitly considering the problems imposed by network unreliability and the possibility of remote failures. The difficulty of these problems suggests that such a standard is not likely to emerge in the near term.

Unlike languages designed to support distributed systems such as Argus[8] and NIL[10], Ada contains features which are difficult to support efficiently in a distributed system environment. In addition, the semantics of other features are difficult to extend consistently from a uniprocessor or tightly-coupled multiprocessor to a network of loosely-coupled processors. Before describing our approach in detail, we will outline some of these problems and compare various other methods which have been considered.

Perhaps the greatest obstacle to providing complete network transparency that is imposed by Ada is the result of the ability of Ada subprograms and tasks to be nested and share variables declared in an outer scope. If such software modules are to be arbitrarily distributable, consistency between multiple copies of shared variables must be maintained or some means of mapping remote addresses into the local address space must be provided. Several distributed operating systems [12] [16] provide such service but require a network of homogeneous processors, special hardware and/or microcode to do so. Such a reliance on homogeneity makes incorporation of new subsystems based on new computers more difficult if not impossible. Honeywell's approach[4] requires this level of support to meet their transparency goals and may be satisfactory if their goal is a networkable Honeywell processor and distributed system architecture which provides transparent support for distributed Ada programs.

An alternative approach[15] [6] [5] [13] requires considerably less run-time support and provides correspondingly less network transparency. This technique has in the past been referred to as transparent remote procedure call (RPC)[9] despite the fact that other types of remote transfers of control are supported. For this reason, in this paper, we will refer to this method as remote transfer of control (RTC).

The essence of this method is to provide *surrogates*[15] for procedures and tasks that are remote to a caller. These surrogates provide interfaces that are identical to their remote counterparts. On the calling machine, a surrogate for the callee processes a call by marshalling the in parameters into a packet for transmission to the remote callee. When this packet is received by the callee machine, a surrogate for the caller unmarshals the parameters from the packet and calls the actual callee. After the call returns to this surrogate, the out parameters are returned to the remote caller via a similar marshalling, network transmission, unmarshalling process.

The transparency supported by this method and the layering of its run-time support is apparent from Figures 1a and 1b. Figure 1a depicts a normal Ada rendezvous between tasks executing on a single processor. Figure 1b shows that when these tasks execute on different computers, they interface with the surrogate level in the same way that they interface with each other within a single computer. The underlying run-time support is layered such that interprocessor communication (IPC) kernel software which implements a special high performance RTC transport protocol is separated from the more hardware-dependent network driver software.

This alternative to supporting network shared memory provides highly transparent support for a wide range of Ada control transfer primitives including subprogram calls, task entry calls, and exceptions. However, the absence of network shared memory imposes restrictions on program structure and the type of parameters which may appear in calls to remote subprograms and tasks. Designers are required to structure their application into disjoint collections of tasks which share an address space. Access and task type parameters must not appear in remote calls since these types require the callee to access the caller's address space.

If it was necessary for implementers to write surrogates for all modules which make or receive remote calls, the additional effort required, to say nothing of the increased opportunity to introduce bugs into the system, would make RTC impractical and ineffective. Fortunately, surrogate generation can be automated to a very high degree. A preprocessor can generate appropriate surrogates for remote clients and servers from interface information present in Ada package, task, and subprogram specifications with minimal input from an implementer.

One of the attractions of this approach is that modules adhering to this discipline may be collocated without the need to involve surrogates or other non-standard Ada run-time support. Run-time efficiency is not sacrificed in the interest of the configuration flexibility provided by the ability to collocate distributed programs. This capability is useful during development when software destined for embedded targets with limited debugging support can be debugged using APSE tools resident on a development host. In addition, it provides an Ada-oriented framework for software reuse in a distributed computing environment. Since the semantics of remote communications models that of local communications, designers need not learn and relearn methods of utilizing operating system monitor calls or ad hoc message-passing packages.

We believe that Ada RTC can support a variety of distributed systems architectures. In particular, it is especially well suited to a *free market*[9] model where clients negotiate and receive service from remote servers. In such a system, servers register their availability with a network-wide service directory. Clients can then acquire the address of a required service by importing the service from the directory. Such run-time binding of servers and clients provides the foundation for dynamic system reconfiguration in response to partial system failure.

Since RTC is intended to be used as a low-level primitive for constructing distributed systems, run-time performance is a critical concern. Previous studies[9] [14] suggest that the end-to-end protocol selected to support RTC at the transport level has a profound effect on performance. Moreover, general purpose transport protocols, especially those requiring explicit connection set-up and shut-down or those oriented toward bulk data transmission can be utilized only at considerable expense at run-time.

While not in the best interests of high performance, a layered RTC run-time increases the practicality of providing language-level homogeneity in support of large systems

employing a variety of processor types and network technologies. In addition to isolating hardware dependencies within the lowest layer, this architecture enables the machine generation of the application-dependent surrogate layer. The possibility of implementing high-performance RTC run-time support coupled with the potential to improve both software reusability and system fault tolerance has resulted in ESL's commitment to prototype and evaluate the Ada RTC approach to distributed systems implementation.

## Ada in a Distributed Environment

We are currently working toward a demonstration of Ada RTC in a distributed system composed of a small number of Sun workstations which communicate via Ethernet. This section describes the particulars of our methods.

As noted, prohibiting shared variables between modules resident on computers with physically disjoint address spaces avoids the difficulties of implementing some form of network virtual memory but requires that applications be decomposed into disjoint collections of tasks which share an address space. When such collections of tasks are not collocated they must communicate with each other solely via Ada RTC. These collections of tasks we call *activity packages*. According to our methods, each processor in a distributed system executes an integral number of activities.

The surrogate generation software is the primary tool which supports the activity abstraction. We have avoided the use of an auxiliary configuration language in favor of a simpler mechanism which utilizes several pragmas of our own definition to specify activity constituents and remotely accessible entry points. Since the Ada language standard requires that the compiler ignore any unrecognized pragmas, activities can be compiled together without preprocessing and run in a single processor environment.

In essence, the implementer inserts a pragma into the specification of an activity package which instructs the preprocessor to generate surrogates for the specified task entry, subprogram or exception. A block diagram of the surrogate generation process is shown in Figure 2. The Ada surrogate preprocessor (ASP) consists of three passes. The first pass takes as input an activity package specification and generates the surrogate code and some intermediate source code which contains local activity information. The second pass links together the local intermediate code with those from other activity packages. Finally, the third pass produces global tables containing global identifiers, packet contents, etc.

ASP output is comprised of modules which are destined to reside on both client and server processors. The context clause of the client activity package contains the name of the server activity package. When both packages are to be collocated this name corresponds to the actual application package written by the implementer. When the client and server are distributed, this name corresponds to the surrogate server generated by the preprocessor. Since the server has no static knowledge of who its clients are, surrogate software which calls the actual server on behalf of a remote client is not referred to in the server's context clause.

Many features of the Ada language support this approach. Ada's separate compilation facilities permit the generation of individual surrogate activity packages. Ada's multitasking support provides the ability for distributed programs to call each other without locking competing tasks from using cpu resources while the call is in process. Ada's exception support enables remote errors to be reported to the client in a manner which is consistent with local error reporting.

For example, suppose that a server task raises TASKING_ERROR. If the server is collocated with the client, the standard Ada rules for exception propagation apply. When the server is remote, the exception is propagated to a surrogate which prepares and forwards an exception packet to the kernel for transmission to the client. When the packet is received by the client computer, the surrogate can re-raise the exception which will then be propagated to the client. Contrast this transparent exception mechanism with application level checking of status codes embedded in messages transmitted by a message package.

Consistent handling of local and remote error conditions is but one of the reasons that we prefer Ada RTC over message-passing systems. A library package which would provide tasks with the ability to send and receive messages between processors could certainly be implemented. Such facilities are frequently provided by real-time executives and operating systems for uniprocessors. However, the semantics of message-passing are inconsistent with respect to Ada's procedure call and task rendezvous semantics. This inconsistency drastically reduces the flexibility with which existing application software modules can be partitioned and recombined to fit the topology of different distributed systems.

Run-time support for RTC is provided by the IPC kernel, the binder, and low-level network support software. Among the most important of our design objectives is performance and support for run-time binding of clients to servers. Although an exhaustive description of our design is beyond the scope of this paper, some of our design decisions are worthy of discussion here.

Tasks are required within the kernel on both client and server machines so that the kernel can initiate processing of one call without locking out others. In order to optimize performance, the kernel needs to create and destroy tasks sparingly due to the high overhead associated with initializing a task's state, creating control blocks, etc.. In our design, tasks are managed by the kernel and dynamically assigned to incoming and outgoing calls.

Since the overhead associated with transmitting and receiving packets coupled with the latency of the network medium is considerable, it is desirable to minimize the number of packets needed to support RTC. Reliable, connection-oriented protocols such as TCP[11] require explicit connection establishment prior to data transmission as well as explicit disconnection when the connection is no longer needed. In addition, acknowledgement packets are sent for data packets. Such protocols have their place in support of remote file transfers and other bulk data transfers but are inappropriate for RTC implementations where performance is a major concern.

We are implementing a transport protocol for RTC which is modeled after that used by Xerox's CEDAR project[2]. This protocol relies upon an underlying unreliable datagram service and is designed to minimize the number of packets necessary to support the most prevalent types of RTC. If packets can never be lost, damaged or duplicated, the minimum number of packets required to support a remote procedure call is two: one to transmit the call and one to return the results. When remote communication is unreliable, both call and return packets must be acknowledged. However, since call and return packets are paired by the semantics of procedure call, receipt of a return packet indicates to the caller that the callee received the call packet last sent by the caller. Since the calling procedure cannot have two calls outstanding at once, the callee may interpret the receipt of a new call packet as an indication that the previous return packet was received by the caller.

Timers are used to trigger packet retransmission when these implicit acknowledgements are not received. When retransmitted packets are received, the kernel immediately returns an acknowledgement. Timeout intervals are variable. At Xerox, the first retransmission interval is set to a value slightly greater than the round-trip time between machines. Subsequent retransmissions are issued after increasingly long delays until after approximately ten minutes, retransmissions are issued every five minutes. Communication failures and remote crashes will be detected but no indication of server problems such as deadlock, infinite loops, etc. will ever be provided. This is as it should be: if the caller and callee are collocated, the caller will receive no indication that the callee will never return. This protocol optimizes what is expected to be the most frequent case: a one packet call is processed quickly and a result packet returned before retransmission is necessary.

During initialization of an activity package, entries in a network-wide directory of available servers are inserted by each server which is available for remote access. Clients import the network addresses corresponding to the server they require from this directory. Fault tolerant capabilities can be supported by this mechanism by providing exception handlers in clients which attempt to rebind to another server in response to a TASKING_ERROR exception.

## Status

At the present time, we have completed the preprocessor responsible for generating surrogates. Preliminary designs for both the binder and the IPC kernel have been specified.

A demonstration of an Ada distributed system prototype is planned for the second quarter 1985. The demonstration will consist of a distributed application which has been programmed in Ada and is hosted on a collection off Sun workstations interconnected by an Ethernet. Each Sun workstation is a 68000 based microcomputer running Berkeley 4.2 BSD UNIX*. The application program will make use of remote transfer of control as the basis for interprocess communication and its surrogate level will be automatically generated by the preprocessor.

In order to create a productive software development environment for distributed programming, it is desirable to develop an APSE specialized for use in programming distributed target systems. The adoption of Ada RTC as the primary means for communication between distributed programs strongly influences the nature of this toolset. We have already described the Ada Surrogate Preprocessor which is essential to relieve the implementer from the tedious and error prone task of writing surrogates. We now describe other tools which belong in our set of APSE extensions but are yet to be realized.

The initial binding of activity packages to computer is a task that currently must be done manually. For each computer in the distributed target system, this involves assembling its activity packages together with the surrogates for those activities which are located on remote computers. The necessary tools to automate this process would not be difficult to construct and would enhance the development process.

Debugging distributed programs is notoriously difficult. One source of these difficulties is that even the best uniprocessor debuggers fail to provide the appropriate level of granularity for program control and data inspection. Debuggers for distributed targets need to provide a set of capabilities which are analogous to those provided by uniprocessor debuggers. The means of monitoring and controlling the interaction of software executing concurrently in a distributed system should be as convenient as that used to inspect local data areas and monitor subprogram calls.

Since all Ada-level interprocessor communication utilizes surrogates, the Ada Surrogate Preprocessor could generate additional code which would permit the inspection of remote call parameters and allow processors to be halted upon remote call or return. The demonstration environment we have selected provides an attractive setting for such a distributed interactive debugger. In particular, the Sun's windowing capability could be utilized to provide separate windows for each processor.

There are limitations imposed by our approach to the use of Ada in programming distributed systems. Programming within these limitations is currently the responsibility of the programmer and only limited protection from the use of illegal constructs is supplied during the development process. It would be desirable to provide automated support for the detection of shared variables between activity packages, access and task type parameters in calls to remote programs, and other constructs which are anathemas to our Ada RTC implementation.

A further consequence of these limitations is the activity package. If a library of reusable, distributable Ada programs designed according to our approach is maintained, it will consist largely of activity packages and associated documentation and test routines. The inclusion of the reusable software library and manager in the APSE will allow the APSE to grow and provide greater support to applications as experience is gained in the use of the Ada distributed system.

* UNIX is a registered trademark of Bell Laboratories.

## Conclusions

Our experience has indicated that despite shortcomings with respect to standardized run-time support for distributed systems, Ada is nonetheless suitable for programming distributed applications. The implementation of a distributed run-time system which supports the entire Ada language is difficult and, in our opinion, impractical when the objective is support of heterogeneous distributed systems. On the other hand, confining interprocessor communication support to an Ada library package restricts software reusability by requiring intertask communication to be achieved by two independent mechanisms. Our intermediate position is practical for heterogeneous distributed systems and provides a strong framework for software reuse by masking the differences between intertask communication between and within processors. The remote transfer of control (RTC) mechanisms we have described utilize Ada's multitasking, implementation-defined pragmas, and separate compilation facilities to provide distributed runtime support for many types of rendezvous, subprogram calls and exceptions.

Since not all possible forms of communication between distributed Ada programs are supported, we require that distributed applications be decomposed into a set of *activity packages*. These activity packages are the fundamental unit of software reusability. Our experience has indicated that the precompilation processing of activity packages to transform uniprocessor programs into distributed programs can be efficiently performed by software. The use of remote transfer of control as the basis for interprocessor communication endows our approach with three desirable characteristics: compiler independence, hardware independence, and transparency to the programmer.

Hardware independence contributes to the portability of applications from one system to another while transparency enhances programmer productivity and reusability. Since our approach is not tailored to any one Ada compiler and since the software-generated surrogate layer is Ada source code, a variety of compilers and target computer environments can be accommodated. Such compiler and target system independence is required if distributed systems composed of a variety of processor types are to be supported. Both hardware independence and transparency to the programmer are results of the layered RTC protocol. Although our implementation does not provide complete transparency, when used within our distributed system framework, it allows for a high degree of software reusability and portability.

Since the interfaces to the lower level network functions are well defined and the IPC level packages communicate via this uniform interface, support for new target systems only requires new driver level software. The IPC and application level software requires no changes. This permits easy expansion to future systems with new hardware and allows for a high degree of software reusability.

Continuous execution in the presence of hardware failures can be supported by appropriately designed RTC protocols and binders. In our design, the dynamic binding of activities to processors enables functions resident on failed processors to be relocated to other processors. The remote exception propagation facilities we support provide support for reporting remote failures.

The software development environment for the Ada distributed system consists of the Ada compiler and support environment together with a collection of APSE-level tools for distributed programming. In addition to ASP, a distributed interactive debugger, a distributed design checker, and a reusable distributed software library and manager are required to maximize programmer productivity.

Our first Ada distributed system prototype will execute on a network of Sun workstations connected via an Ethernet. We expect to complete this prototype by mid-1985 and are confident that empirical data collected at that time will confirm that distributed systems can be cost-effectively developed using Ada RTC.

## References


[1] *ANSI MIL-STD-1815A Ada Programming Language*, 1983.

[2] A. Birrell and B. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, February 1984.

[3] D. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, vol. 1, no. 2, 1984.

[4] D. Cornhill, "A Survivable Distributed Computing System for Embedded Application Programs Written in Ada," *Ada letters*, vol. III, no. 3, November/December 1983.

[5] A. Dapra, S. Gatti, S. Crespi-Reghezzi, F. Maderna, D. Delcredi, A. Natali, R. Stammers, and M. Tedd, "Using Ada and APSE to Support Distributed Multimicroprocessor Targets," *Ada Letters*, vol. III, no. 6, May/June 1984.

[6] D. Lane, G. Huling, and B. Bardin, "An Ada Network - A Real-time Distributed Computer System," *Proc. 2nd Annual Conference on Ada Technology*, March 1984.

[7] P. Leach, P. Levine, B. Douros, J. Hamilton, D. Nelson, and B. Stumpf, "The Architecture of an Integrated Local Network," *IEEE Journal on Selected Areas in Communication*, vol. SAC-1, no. 5, November 1983.

[8] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic support for robust distributed programs," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 3, July 1983.

[9] B. Nelson, "Remote Procedure Call," Technical Report CSL-81-9, Xerox Palo Alto Research Center, 1981.

[10] F. Parr and R. Strom, "NIL: A high-level language for distributed systems programming," *IBM Systems Journal*, vol. 22, no. 1,2, 1983.

[11] J. Postel, "DoD Standard Transmission Control Protocol," Defense Advanced Research Projects Agency, 1980.

[12] R. Rashid and G. Robertson, "Accent: A communication oriented network operating system kernel," *Proc. 8th Symposium on Operating Systems Principles, ACM*, 1981.

[13] M. Rosenberg, "An Ada-based Approach to Distributed Systems," *1984 Government Microcircuit Applications Conference Digest of Papers*, November 1984.

[14] J. Saltzer, D. Reed, and D. Clark, "End-to-end Arguments in System Design," *Proc. 2nd Int. Conf. Distrib. Syst., Paris, France*, April 1981.

15 S. Schuman, E. Clarke, and N. Nikolaou. "Programming Distributed Applications in Ada: A first approach." Technical Report CADD-8103-3102, Massachusetts Computer Associates, March 1981.

16 B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System." *Proc. 9th Symposium on Operating Systems Principles, ACM*, October 1983.

17 N. Wirth, *Programming in Modula-2*. New York: Springer-Verlag, 1982.

**Yul J. Inn,** Senior Engineer, has been at ESL since February, 1981. His major area of interest is in programming language design and implementation. Currently, he is working on the Ada distributed system. His previous work at ESL has included the signal processing algorithm implementation and the design and implementation of a compiler for a data flow language. Mr. Inn received a B.S. degree in mathematics from University of California, Riverside in 1974 and M.S. degrees in mathematics and computer and information science from Ohio State University in 1980.

**Mark Rosenberg,** Senior Engineer, has been at ESL since June, 1981. His professional interests include concurrent programming and software reusability. At the present time, his major responsibility is for the development of Ada software technology to support distributed systems implementation. In 1976, he received a B.A. degree in music from Bard College. Two years later, he received a M.F.A. in music from Princeton University where his work was primarily focused upon computer generated sound.

Authors' address:
ESL
A Subsidiary of TRW
495 Java Drive
Post Office Box 3510
Sunnyvale, CA 94088-3510

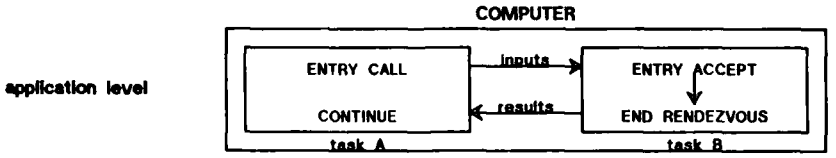



figure 1a
Intertask communication within single computer

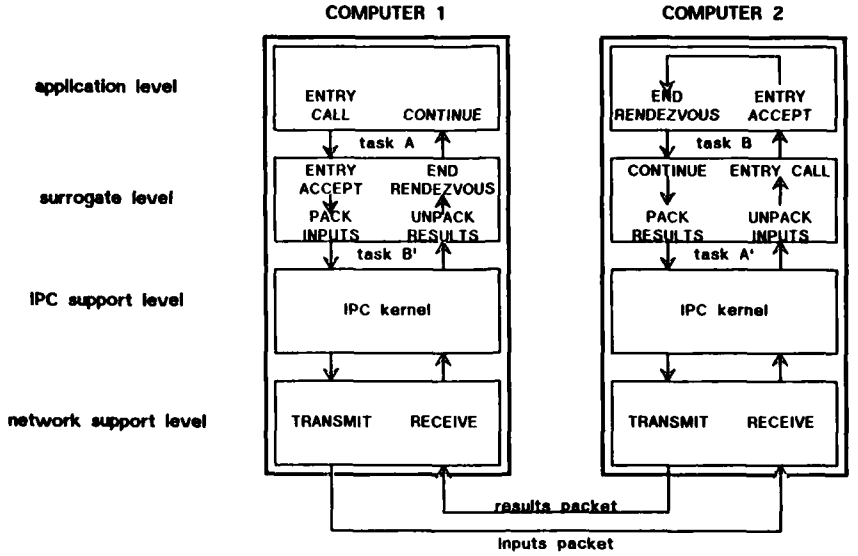



figure 1b
Intertask communication between computers

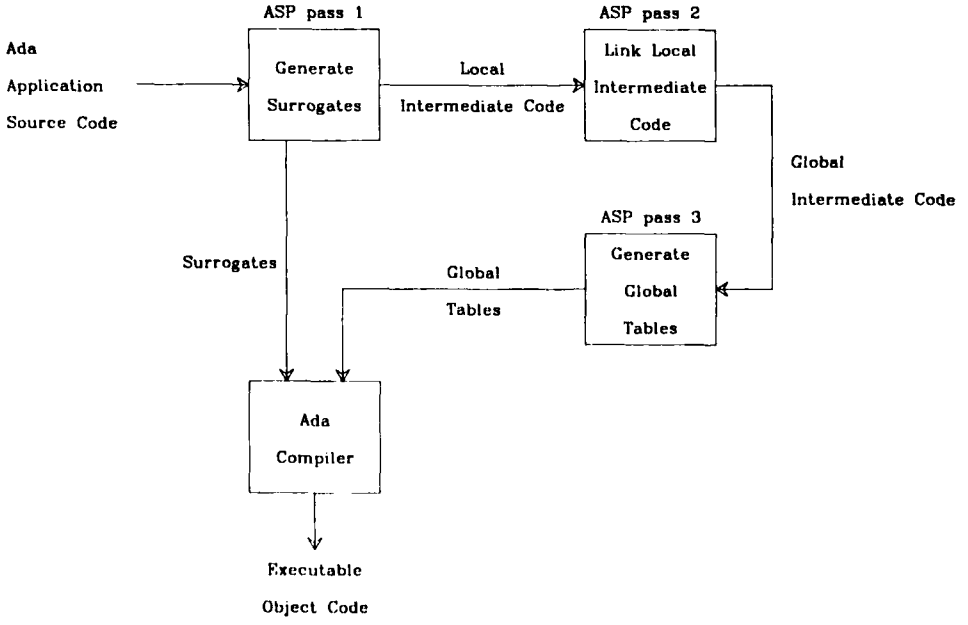



Figure 2. Surrogate Generation Process

# AN ABSTRACT MACHINE SPECIFICATION FOR THE PROCESS NODE SECTION OF THE COMMON APSE INTERFACE SET (CAIS)

Chandra S. Srivastava
and
Timothy E. Lindquist

Department of Computer Science,
Virginia Tech, Blacksburg, Virginia 24061

## ABSTRACT

This paper describes the use of an Abstract Machine approach to specifying kernel software interfaces. To provide a specification that is appropriate for developing a validation mechanism, we describe the functionality of the operations in the process node model of the Common Ada* Programming Support Environment Interface Set (CAIS). Each operation is described by an Abstract program in an Ada-like package body. In this paper we present the technique, describe CAIS Process Nodes, and show the Abstract programs for two of the operations.

## INTRODUCTION

The Ada language has been developed by the Department of Defense to reduce the software costs of embedded computer systems. To achieve this objective, the Ada project has extended beyond the development of a programming language to address the Ada Programming Support Environment (APSE) and the kernel facilities needed to support an APSE. The underlying concept of this development is: If a common interface to the underlying kernel is used then programs, tools, and data will be more transportable over different APSE's. To address this problem the Ada Joint Program Office (AJPO) has formed the KIT/KITIA (Kernel APSE Interface Team / Industry and Academia) and the APSE Evaluation and Validation Team (E&V). The KIT/KITIA have designed a preliminary version of a kernel interface set to support APSE tools. This interface set is called the Common APSE Interface Set (CAIS)[1].

The APSE Evaluation and Validation Team (E&V) is developing a CAIS Validation Capability (CVC). This suite of test tools will be used to assess the implementations of the CAIS in a manner similar to the Compiler Validation Suite used for Ada. To develop the CAIS Validation Capability it is essential to have a clear and complete specification. A preliminary study of APSE validation needs [2] has indicated that CAIS specification should not be limited to syntax and functionality. Subsequently, Lindquist [3] presented a specification technique in which the interactions that exist at the interface and pragmatic limits must also be specified.

This paper uses the Abstract Machine approach to specify the Process Node Model of CAIS. This approach aids in constructing a CAIS validation mechanism. Example Abstract Machine descriptions of Process Management are given to present CAIS Process Nodes.

## ABSTRACT MACHINE SPECIFICATIONS

Among the methods that could be used to describe the functionality of CAIS facilities are natural language commentary, formal semantics, and abstract machines. Although natural language commentary is easy to construct and comprehend, the major drawback is that the intended audience is often left to interpret key semantic issues. The validator or the implementor may either make arbitrary decisions based on interpretations or ignore key semantics.

There are several formal methods for specifying semantics. Examples are axiomatic specifications, denotational semantics, or validation assertions. The methods use mathematical formalism to define semantic information and are subject to mathematical analysis. In the axiomatic approach, axioms are designed in the form of logical statements to describe the functionality of the operation. This approach, while quite expensive and time consuming to construct, is precise and rigorous.

---

*Ada is a registered Trademark of the US Department of Defense Ada Joint Program Office.

With Abstract Machines, the functionality of an operation is defined in terms of a program, which describes what the operation does. The program is written for an Abstract Machine, which if executed would exhibit the function of the operation. This approach is more formal and concise than natural language commentary and easier to understand than other formal methods.

There are two drawbacks of the Abstract Machine approach. One drawback is that it is bottom-up, which means that the instructions and the data recognizable by the Abstract Machine must be designed before the operations can be understood. We describe CAIS functionality in an Ada-like Abstract Machine which is familiar to the intended audience. The second drawback is that Abstract Machines tend to bind the implementor to a specific implementation technique. While an Ada-like Abstract Machine program may suggest an implementation, functionality is defined by the effect of executing the instructions not by the instructions themselves.

## CAIS ABSTRACT MACHINES

We have defined the operations of CAIS Process Management by an Ada-like Abstract program. In constructing the Ada program some operations and data are needed which are neither convenient to detail nor part of the Ada language. These operations and objects are treated as primitives of the Abstract Machine.

We us Ada because it is rich in control constructs and typing, for example, separate compilation units (packages), tasking for concurrency, and exception handling. Further, users of a CAIS specification will be familiar with the semantics of the Ada language.

Our Abstract Machine programs for Process Management make extensive use of other CAIS operations. For example, the abstract program for SEND uses OPEN, CLOSE, and KIND, which belong to CAIS_NODE_MANAGEMENT. We assume that these operations are fully specified. The specification includes an implementation package for Process Management. This package defines the data structures and procedures used in the Abstract Programs. The implementation package can be viewed as containing elements private to detailing the Abstract code for processes.

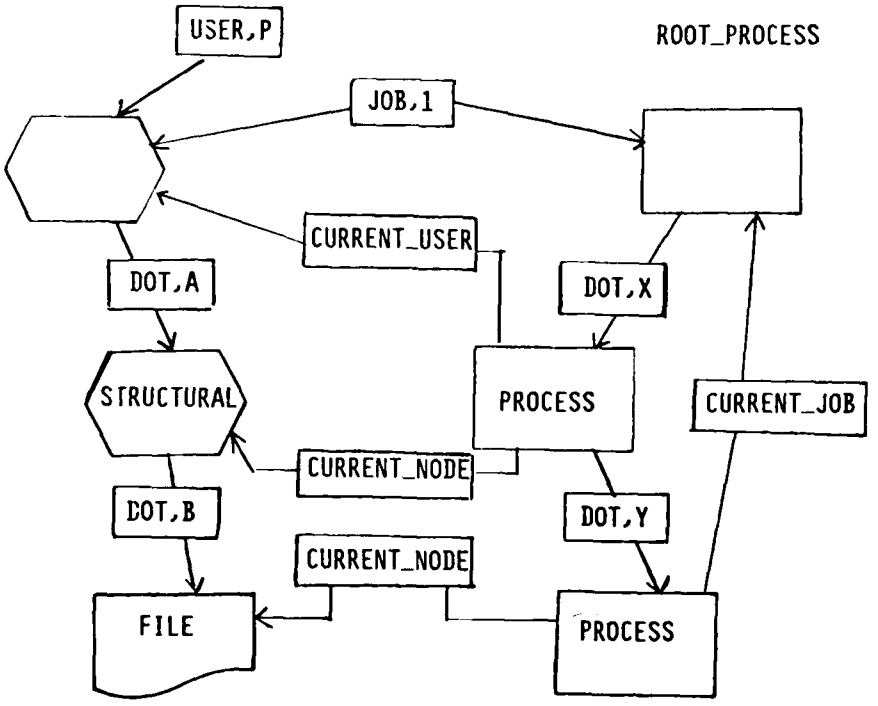



Figure 1. CAIS Nodes.

## CAIS PROCESS NODES

The CAIS can be viewed as a manager for a set of entities such as files, processes, and devices. To manage these entities four types of nodes have been defined: structure, file, process, and device. Nodes are distinguished by their contents, and structural nodes have no contents. Contents act as a carrier of information about the entity being described. Figure 1 shows the various nodes in the CAIS environment.

Process nodes represent the execution of an Ada program by means of its code and context. A single process node represents all the tasks associated with a program. Whenever a user enters the APSE, a root process node is created dynamically as the top level node for the user. The root process node acts as the root of the tree for all dependent processes created by the user. Whenever an Ada program is invoked, a process node is created and attached to the parent process. Thus, the process structure grows and shrinks as programs begin and complete execution.

When compared to other kinds of nodes, process nodes are active rather than passive. They have parameter passing and suspension/resumption capabilities. Resources are dynamically bound to the process node during execution. For example, memory, cpu capacity, files, devices, and other processes are bound as a process requires.

A process is identified by providing a pathname to its node. A pathname consists of a sequence of relation/relationship-key pairs that traverse a path to the node. There are at least three predefined relationships associated with each process node. The CURRENT_JOB relationship refers to the root node for a process node's job. The CURRENT_USER relationship refers to the user's top level node. The CURRENT_JOB and the CURRENT_USER relationships are maintained by CAIS. The CURRENT_NODE relationship refers to a node which represents the process node's current focus or context and can be manipulated by the process. These relations provide a convenient means for accessing other CAIS nodes. Figure 2 shows a process node and its relations together with the program and tasks associated with it.

The current state of a process which is known as its PROCESS_STATUS, can be determined from another process. CAIS defines READY, SUSPENDED, ABORTING, and TERMINATING states for a process. CAIS also defines facilities to perform the following process management functions:

- management of Ada program execution
- interprocess communication
- process monitoring and control

CAIS Process Management packages have been defined to achieve these three objectives.

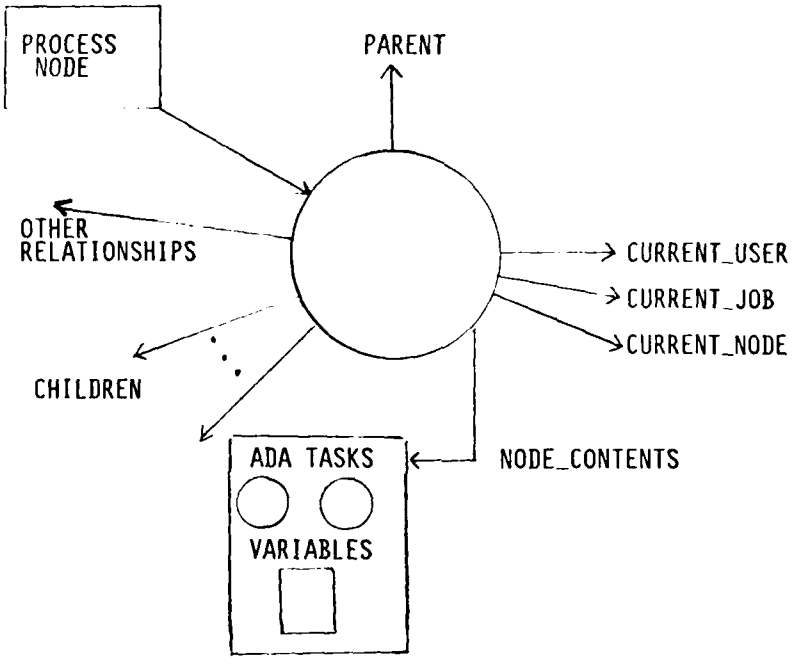



Figure 2. Process Node.

The CAIS_PROCESS_DEFS package defines types and definitions associated with the process nodes as needed by the other packages. This paper presents sample details of the specification of routines from PROCESS_CONTROL.

## CAIS PROCESS CONTROL

CAIS_PROCESS_CONTROL provides for invocation, suspension, resumption, and termination of processes. Each time an Ada program is invoked a new process and a process node associated with it are created. The new process may execute synchronously or asynchronously by using INVOKE_PROCESS or SPAWN_PROCESS respectively.

Processes may complete their execution by one of two methods. The first one is normal termination, which is accomplished by RETURN_TERMINATED. A process calls RETURN_TERMINATED to cause normal completion of itself after completion of its dependents. The second method is abnormal completion which is achieved by RETURN_ABORTED. Here, the process and its descendent processes are forced to abort.

ABORT_PROCESS is used to abort a process along with its descendant processes. Control returns immediately to the calling process without waiting for the processes to abort. The process node is not deleted when ABORT_PROCESS is used. Instead, it has to be explicitly deleted.

Processes can be suspended from the ready state or resumed from the suspended state by using SUSPEND_PROCESS and RESUME_PROCESS. The state of a process can be viewed from another process, using the function STATE_OF_PROCESS.

CAIS_PROCESS_CONTROL includes functions JOB_INPUT and JOB_OUTPUT which return the standard input and output files as defined at the initiation of the root process tree.

Our Abstract Machine programs use Ada tasks and objects to represent CAIS processes. The NODE data structure, as defined in the CAIS_IMPLEMENTATION package, defines the contents of a process node. The node contains the data necessary to manage the process and the tasks needed to support program execution, process synchronization, interprocess communication, and process interrupts. Figure 3 shows the node data structure in which circles represent tasks and boxes represent data.

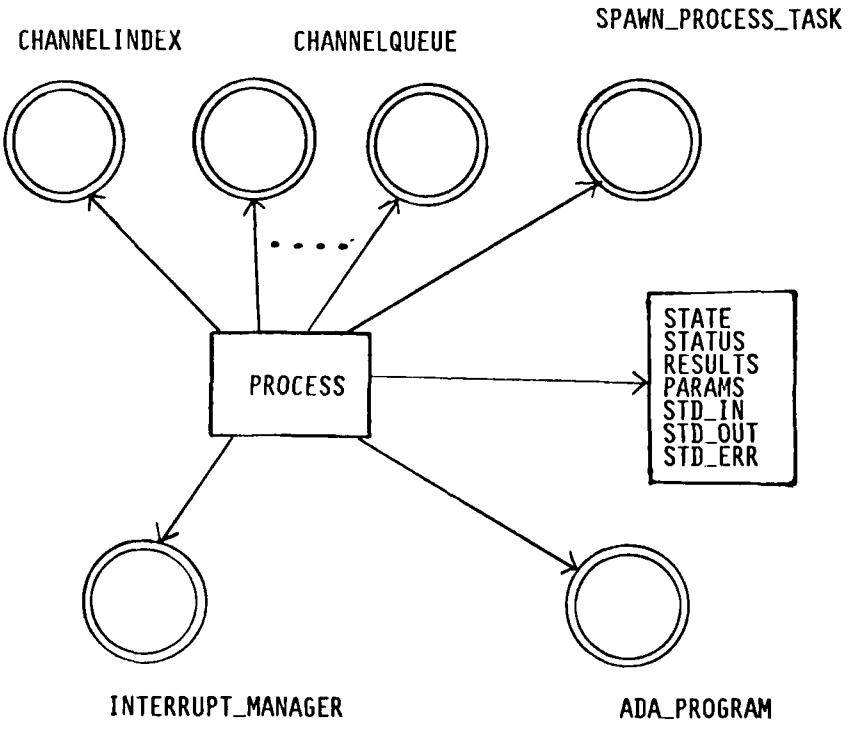



Figure 3. Process Tasks.

The task ADA_PROGRAM is used to represent the execution of a PROGRAM associated with the process. The parent makes an entry call to this task in the procedure SPAWN_PROCESS and the calls return immediately. The execution of the program is started in this task. The task SPAWN_PROCESS_TASK is used to achieve synchronization between the parent and the child process. To wait for a child process to finish, the parent makes an entry call to an entry within this task. When the child process completes, it also makes an entry call to this task providing the RESULTS and the COMPLETION_STATUS. The parent is then awakened and returned the parameters of the child process.

## SPAWN_PROCESS

As examples, the Abstract Machine specifications of SPAWN_PROCESS and AWAIT_PROCESS are presented. A call to SPAWN_PROCESS results in a new node and a new process being created to represent the execution of the specified program. Control returns to the invoking process, and no technique is provided for coordination of the new process with its parent, except for communication and termination. Termination of the parent will not be completed until all children are terminated or aborted. Similarly, no technique is provided for returning a result string to the invoking process.

```
procedure SPAWN_PROCESS (PROGRAM : in      PROGRAM_STRING;
                         PARAMS  : in      PARAMS_STRING;
                         NODE    : in out  NODE_TYPE;
                         KEY     : in out RELATIONSHIP_KEY :=
                                           UNIQUE_CHILD_KEY;
                         STD_IN  : in FILE_TYPE :=
                                   CAIS_TEXT_IO.CURRENT_INPUT;
                         STD_OUT : in FILE_TYPE :=
                                   CAIS_TEXT_IO.CURRENT_OUTPUT;
                         STD_ERR : in FILE_TYPE :=
                                   CAIS_TEXT_IO.CURRENT_ERROR;
                         CURR_NODE : in NAME_STRING :=
                                   "'CURRENT_NODE") is

     IS_UNIQUE : BOOLEAN := TRUE;
     NODE_C,NEXT_NODE : NODE_TYPE;
     FILE_TYPE      : CAIS_LIST_UTIL.LIST;
     ITERATOR       : NODE_ITERATOR;

begin
     OPEN(NODE,PROGRAM);
     if KIND(NODE) /= FILE
        then CLOSE(NODE);
             raise NAME_ERROR;
     end if;

     GET_NODE_ATTRIBUTE(NODE,"file_type",FILE_TYPE);
     if CAIS_LIST_UTIL.IDENTIFIER(FILE_TYPE) /= 'executable_image'
        then CLOSE(NODE);
             raise NAME_ERROR;
     end if;

     CLOSE(NODE);
     OPEN(NODE_C, CURRENT_PROCESS);

     ITERATE(ITERATOR, NODE_C, KIND =>PROCESS);
     while (IS_UNIQUE and MORE(ITERATOR)) loop
         GET_NEXT(ITERATOR, NEXT_NODE);
         if PRIMARY_KEY(NEXT_NODE) = KEY
            then KEY := UNIQUE_CHILD_KEY;
                 IS_UNIQUE := false;
         end if;
     end loop;

     CREATE_PROCESS(NODE_C,KEY,NODE,PROGRAM,PARAMS,
                    'ready',STD_IN,STD_OUT,STD_ERR,CURR_NODE);
     CLOSE(NODE_C);

end SPAWN_PROCESS;
```

Figure 4. Spawn Process

## ABSTRACT PROGRAM FOR SPAWN_PROCESS

Figure 4 is the Abstract Machine code for SPAWN_PROCESS. The pathname to the program to be executed (the PROGRAM parameter) is first tested to assure that it is syntactically correct, that the file exists, and that it is in executable form. A NAME_ERROR is raised if it is a syntactically incorrect pathname or if it is not executable. Different systems have different conventions for the executable image depending on the protocol with the linker.

ITERATE, MORE, and GET_NEXT procedures of the CAIS_NODE_MANAGEMENT package have been used to traverse all the existing child processes to determine whether the key supplied by the user is unique. If the user supplied key is not unique (or if the user does not specify a key) then the system generates a unique key for the spawned process. A procedure CREATE_PROCESS (which is not shown) is used to create the process node and necessary tasks for the spawned process. The parameters, standard input, standard output and standard error as specified by the parent process are stored in the spawned process. The state of the spawned process is made READY. The CURRENT_NODE of the spawned process is initialized to the CURR_NODE of the parent. The execution of the PROGRAM, which is passes as a parameter by the parent process, is initiated by the task ADA_PROGRAM. An entry call GET_PROGRAM is made from the CREATE_PROCESS procedure to this task to indicate what program to execute and to indicate that the execution may begin. The execution of the PROGRAM starts concurrently with the SPAWN_PROCESS call.

## SUMMARY

In this paper we have presented a specification technique for kernel facilities that can be used to aid in developing a validation technique. The technique, which uses an Ada-based Abstract Machine approach, has been used to specify the CAIS Process Management section. We have presented the Process section of CAIS and in particular described the Abstract Machine specification of SPAWN_PROCESS, which is the facility for concurrent execution of programs. We have shown how the functionality of operations that manage a multi-process program development environment can be detailed in terms of Ada tasking. An executing Ada program together with all of its tasks are represented by a single CAIS process. In our description, each process is represented by a group of Ada tasks, needed to manage the program's interactions with the CAIS environment. A single task within the group is used to represent the parent of all tasks in the executing Ada program. Thus, a CAIS environment of several active processes is defined by our specifications with a cluster of Ada tasks for each process. Ada has proved useful in describing the Process Management facilities, and the rendezvous mechanism of tasking has been used to define synchronization and communication among CAIS processes.

## REFERENCES


1. Common APSE Interface Set (CAIS) Draft Military Standard, Prepared by the KIT/KITIA CAIS Working Group, Version 1.4, Ada Joint Program Office, October 1984.

2. Kafura, D.G.; JAN Lee; T.E.Lindquist; T.Probert, Validation in Ada Programming Support Environments, in KAPSE Interface Team Public Report Vol. II, NOSE, pp. 30-1 to 10-59, October 1982

3. Lindquist, T.E.; J.L.Facemire, and D.G.Kafura, A Specification Technique for the Common APSE Interface Set, Journal of Pascal, Ada and Modula-2, Vol. 3, No. 5, pp. 25-32, Sept/Oct. 1984


## Acknowledgement


This research was supported by the Ada Joint Program Office through the Office of Naval Research Information Sciences Division under ONR contract number n00014-83-K-0643. The effort was under the technical direction of V.L. Castor, Wright-Patterson AFB, Ohio. The views expressed herein are solely those of the authors.

# ADA AS OUTPUT FROM SOFTWARE CAD SYSTEMS

T. A. Mizell
H. S. Osborne

Teledyne Brown Engineering
Huntsville, Alabama 35807

## Abstract

This paper describes the theoretical concept of a project by Teledyne Brown Engineering to develop a unique set of tools for systems engineering that both improves productivity and lowers the error rates. While there are many efforts underway to build tool sets for productivity improvement, none are based on using the Ada language as an intermediary in system construction. The sum total of this attempt has been to impose a yet higher order graphics tool between the user and the Ada language, thereby producing labor savings and error reduction. It is important to emphasize that this paper deals only with using Ada as an intermediary language and the human interface and not with the general characteristics of the toolset as a whole.

## The Human Interface

There are a number of ways in which the interface between people and machines could be improved. Instead of having a program in ASCII streams, for example, it would be more productive to have a graphics "front end" to improve readability, maintainability, and provide living specifications. The main reason for graphics should be obvious -- a quick synopsis of the activity is provided. Second, the system development process could be facilitated by using a natural math notation, via graphics, rather than the typical kinds of mathematical expressions found in programming languages. Algorithm development and refinement are two of the more significant tasks to be accomplished in scientific programming. Graphical displays of the algorithmic procedures using natural math notation facilitate the process, saving tedious checking in a programming language. Third, the human interface could be greatly improved by installation of more elaborate diagnostic aids, such as the highlighting of errors found using such tools as MACPASCAL. Fourth, the graphics front end should show a system at a time instead of a program at a time, which typically happens at the present. If the system is very large and complex, as a typical DOD system, it is impossible to see the "big picture" without some sort of visual representation. If the graphics can show not only the main components but also the algorithms, timing constraints, data characteristics, and flow of control, then the system designer can significantly improve productivity and efficiency by seeing the component fit into the whole.

It is recognized that defending the thesis that systems designers should use graphical notations for system design that use Ada as an intermediary will be difficult in the eyes of many traditional programmers who see Ada as a super programming tool. Teledyne Brown has already developed this process, which is now being refined. This is based on the recognition that Ada can be much more than another coding tool; it is indeed of such power that it can be used in new and revolutionary ways that were not foreseen when it was first conceived.

Ada has been simultaneously hailed and cursed as a high-level programming tool since its conceptual beginnings in the mid-70's. Many felt that it was too inclusive in that it attempted to incorporate the strongest features of all common coding languages, including COBOL, FORTRAN, CMS-2, JOVIAL, PASCAL, and TACPOL. They argued, among other things, that any compiler that incorporated all the strong features of these languages would be, by necessity, so complex that it would become burdensome in terms of the required overhead that it would not be possible to meet requirements of reasonable compilation and execution speed. While some of the concerns about Ada have been borne out, it should not be thought that Ada is just another coding tool. Ada is complex, but necessarily so given the requirements laid down by those who conceived it. To this end, it is argued by the current authors that Ada is the best higher order software tool yet devised. It should also be noted that the tool is just one component of an entire programming environment, the Ada Programming Support Environment (APSE), which, at this writing, is still under development and beyond the scope of this paper.

An interesting aspect in the development of Ada is the fact that it was a Department of Defense effort to bring the cost of system software development back into acceptable limits. However, the same range of studies pointed out the exorbitant costs were not due to the available coding tools, but rather to problems of requirements and design. Some coding tools were thought to be obsolete, specifically COBOL and FORTRAN, but they are still the most common coding tools in the DOD environment. However, if one must carry out requirements and design with a coding tool, the problems associated with software

cost overruns will continue to grow. Experience has shown that any project exceeding 20K lines of code becomes difficult to manage, given traditional paper and pencil methods. In addition, designing with a coding tool invariably leads to failures in expectations, confusion, and low morale.

### Invariants in Commercial Compiler Development

Compilers are a necessary part of any digital system, since they greatly facilitate the development process of any large computerized activity. Those familiar with the MIS world today know that some huge data processing shops in the insurance and banking industries still code in assembly languages, since they will not shut their operation down long enough to develop and test the code in a higher order language. The inertia found in some of these operations is a source of bewilderment to those familiar with the advantages afforded by compilers, high-level design toolboxes, and sophisticated editors. However, in their behalf, it should be noted that the cost constraint precludes many things that the scientific world assumes that it could not live without, e.g., testing.

A commonly observed, if not readily explainable, phenomenon in the development of compilers in commercial environments is that the level of effort seems to be a constant. Across a wide range of languages that vary in complexity, it generally takes about 3 man-years to develop a compiler from base zero to minimum functionality. No attempt will be made in this writing to provide a statistical analysis of development time for other software products than commercially available compilers, so it should be borne in mind that we are not directly addressing other software difficulties. In addition, experience has shown that to move from minimum functionality to production requires 2 additional man-years. These figures are for an operational compiler; any enhancements and "bells and whistles" will invariably add but little to that minimum. (The Ada compiler development effort has taken longer -- so much longer, in fact, that it is difficult to estimate the delta. A plausible reason is the enormous amount of testing and validation required by the DOD.) One of the authors of this paper has been part of six major language projects, ranging from a BASIC interpreter to a COBOL at ANSI level II compiler with some implementations using standard tools such as LEX, YACC, and some ad hoc parsers. By all rational expectations, such a wide range of deliverables calls for a corresponding wide range of schedules, yet the observable final totals were all within 25% of 5 man-years. Others may have had different experiences, but we have found that those efforts requiring more than 5 man-years may have had some hidden costs rolled up into the final figures.

The explanation of the 5 man-years from conception to delivery may have more to do with corporate requirements than language requirements. New language products are implemented for a variety of reasons. A COBOL compiler enables ingress by a vendor into a wide array of business applications, with respect to both the ability to build new packages and to convert existing ones. Not all COBOL compilers are optimized, since speed of performance is rarely a critical ingredient in that environment. "Scientific" compilers, on the other hand, may have to have some additional refinements, in terms of optimizing compilation and execution speed, setting them slightly apart from the rest of vendor-supplied software deliverables. It is argued, therefore, that cost and not schedule becomes one of the significant drivers in commercial compiler development. The value associated with a commercial compiler is constant, usually some percentage of the value of the hardware. A vendor, therefore, needs a particular kind of compiler in a cost range that enables him to market the hardware successfully, and it is this value that becomes the significant driver in what he is willing to spend for the product to be developed. This reflects what the customer is willing to spend.

### Better Ways to Spend the Compiler Effort

Despite the obvious advantages of Ada as a programming or coding tool, questions are still raised about its effect on the current software crisis. The problems that produced Ada -- need for embedded systems tools, proliferation of coding languages, transportability, cost of development, and maintenance of computerized systems -- are going to continue. They have nothing to do with Ada, or any other coding tool for that matter. The problems just mentioned are significant, indeed, but they will not be solved by widespread implementation of Ada alone. In the large programming environment (here arbitrarily defined as any system larger than 20K lines of source), coding is around 10 to 15% of the total effort. Much larger problems portend in the requirements and design of the system, from which the code is supposed to flow logically. Therefore, no additional coding tool, regardless of the power, will be able to do much about the problems that really plague this industry.

Like other systems engineering firms, Teledyne Brown Engineering is concerned with two simultaneous and highly related problems: overcoming unclear specification documents and incorporating the Ada language into its programming environment. Ada is indeed the most powerful coding tool yet developed, but it does require a little more "front-end" design work than other languages before the coding can begin. It has been the experience of many in the Ada community that it takes an experienced programmer from 3 to 6 months of involvement in Ada before the tool is used effectively. The learning curve in this sense is slightly slow, if one is to do "Ada in Ada and not FORTRAN in Ada." Ada is, after all, a methodology as well as a language.

But becoming an efficient Ada programmer does nothing for overcoming problems of undecipherable specification documents. It is this logical intersection between Ada as a coding tool and systems engineering design tools that has intrigued Teledyne Brown Engineering.

## Ada as Intermediate Code

As noted earlier, it generally takes about 5 man-years to move a commercial compiler from the conceptual phase to production quality. From the designer's point of view, it is irrelevant what the compiler produces as intermediate code, if efficiency does not matter. Following are three areas in which the use of Ada as the intermediate language in compiler design can subtract from the 5 man-years expectations given above:

1. Parser

While this is not the major area of savings, it is interesting to note that the grammar rules of a new language can be significantly shortened in the area of operator precedence handling. This can be accomplished by the expedient of riding "piggyback" on Ada's precedence handling capability. Depending on the character of the expression, decomposition required, and the style of the BNF grammar with which the compiler writer chooses to construct the parse engine, as many as two dozen productions may be saved, with corresponding reduction in the semantic support associated with these productions.

2. Built-ins

The amount of run-time code required to support a new language obviously varies widely with the nature of the language. It should be noted, if only in passing, that several specific benefits could ensue by emitting Ada as intermediate code rather than another kind of code. First, the kinds of languages that could profitably use the Ada emission techniques suggested here are typically very high-level languages with correspondingly heavy run-time support requirements, well suited to Ada implementation. Second, even in the context of "conventional" sets of run-time support, Ada could be a fine medium for implementation, given that efficiency is not a dominant requirement. One GENERIC routine, for instance, may serve a given, fixed, floating, and even complex number function, where three routines would be required in a more traditional solution of the problem.

3. Code Generation

The major, and potentially overwhelming, benefit of Ada as a "pseudocode" is the near elimination of the largest single piece of work in a commercial compiler: namely, the code generation phase, which is about half of the total effort of compiler development. Direct emission of functional Ada in the semantic intermediary phase has been found by us to be no more complex or laborious than the more normal emission of triplets or directed graph updates, which usually characterizes this phase. The true code generation phase then becomes someone else's compiler, thus deferring the traditional and very painful problems of register allocation, code efficiency, and so forth. Essentially, the new language's implementation has the character of a translator, rather than a compiler, which would seem to be a realistic way of breaking the invariance expressed earlier. The expected development cycles should drop to about 3 man-years, and perhaps less.

4. The Sore Spot

The technique of emitting Ada is remarkably pointless for the compiler of a conventional programming language. At this writing, Ada should be given that market segment for its own, being the "programmers' friend," until a different and superior idea comes along. However, the observed convenience of the technique is an open invitation to the language designer (more than the compiler writer) to attempt to develop ways of representing systems at higher levels. Roughly speaking, an extra 2 man-years are freed from machine considerations, which may be reinvested in the system design.

# REUSABLE GENERIC PACKAGES
# DESIGN GUIDELINES BASED ON STRUCTURAL ISOMORPHISM

Mícheál Mac an Airchinnigh

Generics (Software) Limited,
Unit 7, Leopardstown Office Park,
Foxrock, Dublin 18, Ireland.

## Abstract

A fundamental criterion for reusability of an Ada(R) package is that it be generic. The generic package encodes abstract structure. All instantiated structures are then said to be isomorphic. Different classes of generic package give rise to different kinds of abstract structure. A taxonomy of such generic packages based on formal parameter classes is presented. Special attention is given to agent abstractions which are dynamic objects encoded as tasks encapsulated within generic packages.



## Introduction

"Finally there are generics and the whole question of parameterisation. Should we write specific packages or very general ones? This is a familiar problem with subprograms and generics merely add a new dimension. As stated at the beginning of this section there is as yet little experience of the use of Ada and so it seems somewhat premature to give advice in this area."[1]

It is approximately four years since the above remarks of J.G.P. Barnes were first penned. We have now had at least four years exposure to Ada, two of which with ANSI MIL/STD 1815A. Examples of generic packages have mushroomed in the Ada literature and tentative guidelines as to their design and construction have already appeared[2]. In his textbook "Software Engineering with Ada", Booch[3] explicitly recommends four different applications for Ada packages - (i) named collections of declarations, (ii) groups of related program units, (iii) abstract data types, and (iv) abstract state machines. He also points out elsewhere that in order to include tasks as library units, they must be encapsulated in packages. We call such a packaged task the incarnation of an agent abstraction. Naturally, Booch's taxonomy is immediately applicable to generic packages.

The construction of a taxonomy of generic packages from the perspective of application domain is a valid and worthwhile approach but not the only one. We have complemented this work by focusing on the kinds of abstract structure and functionality which is rendered feasible by generic packages. Such packages are templates from which actual instances may be "stamped" out. In this sense there is an analogy between the latter and the construction of a mould and the kinds of objects cast from the mould. In particular, the more complex the structure of the object to be cast, the greater the difficulty in the design and construction of the mould. Furthermore, the mould incorporates the required structure in an non-obvious manner. That is, by looking at the mould it is not possible to perceive immediately the structure of the resulting casting. Although the analogy is useful in considering the relationship between generic package and instantiation, it can not be carried too far. In the case of the mould the structure of the casting to be is the space to be filled - an inverted structure, whereas the structure of the instantiation to be is directly given by the structure of the generic package.

From an educational view-point, most classes of generic package arise as a result of applying the piagetian principle of "concrete-to-abstract" evolution[4]. Thus, STACK_OF(REAL_NUMBER), STACK_OF(COMPLEX_NUMBER), STACK_OF(MAILBOX), etc., all suggest the classical abstraction STACK_OF(...). Each instantiation has the structure of STACK and thus it may be said that each is

structurally isomorphic to the other. The mathematical principle of isomorphism is central to modern algebra and has its counterpart in the design of generic packages. Isomorphism literally means "same structure". One can readily develop such packages for hardware colour models, coordinate systems, vector spaces and classical computer science structures. However, just as in mathematics, the elegance and simplicity of the abstraction provided by the generic package is counterbalanced by the greater semantic burden that is placed on the package designer, implementor, and end-user.

In constructing guidelines for the design of reusable generic packages we have found it useful to distinguish between the Ada programmer roles just mentioned, even if the same person should at present play all three. In the future we would consider it desirable that distinct programmers are assigned distinct roles with respect to a given package. Such role separation forces one to look carefully at the relationships that (ought to) exist between the generic formal parameter list, the package specification, visible part, the package implementation (private part and body), and the package instantiation and end-use.

The remainder of the paper is divided into three sections. First, we consider the issue of reusability of Ada software components. Second, we provide a taxonomy of generic packages based on the form of parameterisation. Third, we consider a taxonomy of agent abstractions which are incarnated as single tasks embedded within generic packages.

A note of caution is in order. There is a considerable amount of Ada code presented in the paper. We believe that it is correct with respect to the Ada Language Reference Manual[5]. In designing the generic packages presented here, we had much difficulty in interpreting the "semantics" of the English language text of the Reference Manual. There are many fine details that must always be kept in mind. It is not easy to present these in a written paper of this nature and we have decided to treat them in the oral presentation. To illustrate the kind of difficulty we had, let us consider a typical problem. In the case of the KEYBOARD_DRIVER_MODEL example where addresses are passed as generic formal objects we believe that it ought to be valid Ada even though generic formal parameters are never static. Should it happen that the Ada Language Committee rules against it, then Ada must be deemed to be seriously wanting in the application domain of reusable components for real-time systems.

## Reusability

"The very concept of reusability must be defined more rigorously, in terms of the dependence of the component on enclosing or higher level environments (probably defined in terms of number and complexity of algebraic structure parameters). It should be possible to develop metrics for the measurement of such component dependence, enabling quantification of potential reusability."[6]

The concept of generic package is inextricably bound up with that of reusability. The success/failure of Ada as a programming language in competition with existing and yet-to-be-defined languages depends to a large extent on the degree of reusability of Ada software components that can be achieved. With respect to Ada, the desideratum of reusability has been mooted in the context of the DoD STARS Program[7,8]. There are many aspects to the concept of reusability. First, there is the issue of granularity of the reusable Ada software component (RASC). At the macroscopic level one can forsee RASCs as complex as, say, lexical analyser generators such as Lex[9] or parser generators such as YACC[10]. On the other hand, the concept of reusability is applicable at the microscopic level - that of individual generic packages. I hypothesise that it is at this latter level that the issue of reusability for Ada is most crucial. Therefore, we have proposed and adopted the following fundamental principle:

P1. To be reusable, a package must be generic.

A second principle immediately follows as a corollary to the first:

P2. All generic packages are reusable to a certain degree.

Second, there is the issue of domain applicability of the RASC. Reusable software components have proved successful in the past, chiefly in the area of mathematical software[8]. It is not difficult to prophesy that classical computer science structures such as STACK_OF(...), QUEUE_OF(...), etc., will give rise to RASCs. However, let us consider the SET_OF(...) example which appears frequently in the literature to illustrate

Ada's generic package capability[1,3,11]. The degree of reusability of this example is restricted by the domain of discrete types. It is not reusable if one wishes to have sets of structured objects. For example, it cannot be instantiated to give SET_OF(RGB_COLOUR) where RGB_COLOUR is a limited private type exported from a package that incarnates RGB_COLOUR as an abstract data type. If we wish to have a generic package which handles sets of structured objects, then we must rely on some implementation such as hashing and the simplicity of the implementation for the set of discrete type objects is lost. This causes a real dilemma with respect to reusability. Either we use the more general set package to cover all applications or we have two separate set packages - one for discrete types and one for structured types. Of course, the domain of applicability for such generic packages will be determined by the generic formal parameter list. It is clear that there are degrees of reusability and that perhaps one can find a measure thereof. Such a measure may be determined from the structure and applicability of a generic package. Hence, we propose:

P3. A measure of the "genericness" of a package will provide the basis for a measure of reusability.

Now we come to the problem of determining the genericness of a package. To achieve this goal it is essential to have a taxonomy of generic packages, to identify those significant features that contribute to the genericness and a mechanism to map features on to values in some numerical measure space. Similar proposals towards the determination of a measure of reusability in the context of Ada appeared in[6]. In this paper we focus on a taxonomy of generic packages with particular emphasis on the abstract structure which it encodes and the parameterisation of that structure via the generic formal parameter list. The structure encoded by the generic package is essentially an abstraction of the structure of all possible instantiations. There are two distinct possibilities that arise. The generic package may abstract on the maximal functionality required by the instantiations. Those which require less than the full functionality may be obtained by using a generic package in the role of filter[12], which is an illustration of the principle of derivation. On the other hand, the generic package may abstract on the minimal functionality required by the instantiations. Added functionality may be obtained by using a generic package in the role of kernel which is an illustration of the principle of enrichment[13]. A good introduction to the issue of reusability of software in a general setting may be found in the special issue of the IEEE Transactions on Software Engineering, vol.SE-10, no.5, September 1984. Finally, it is important to note that the design of reusable generic packages will require extensive education/training in Ada, computer science, and mathematics, and also extensive experience of and familiarity with many kinds of existing generic packages.

## Taxonomy of generic packages

We restrict our taxonomy to generic packages only, thus excluding generic subprograms whose usefulness is limited and possibly redundant. Furthermore we assume that a generic package will have the form:

```
-- some with/use clauses
generic
  -- zero or more formal parameters
package Z_MODEL is
  type Z -- possibly parameterised
    is limited private;
  -- functions/procedures
private
  -- FOR IMPLEMENTOR'S EYES ONLY!
end Z_MODEL;
```

At first glance generic packages having this form may seem to be excessively restricted. However we have found it to be sufficient to cover a wide variety of data abstraction incarnations - the abstract data types, agent abstractions, and state abstractions. Moreover we hypothesise that only generic packages of this form will ever really be reusable. Whereas in this paper we present our analysis in the context of limited private types, a requirement of our Ada Software Methodology, similar results may be obtained in the weaker context of private types, except for the treatment of agent abstractions, of course, which involve tasking. Note moreover that according to our design philosophy, the package is treated as an envelope for the abstract object being encoded. The type Z defines the object in question and the corresponding package name is this type name suffixed with "_MODEL" to make this clear. There is an alternative design approach wherein the package itself is the encoded object and there is no type of focus in the package specification.

An example contrasting these two distinct approaches is that of the STACK which is presented in the Ada Language Reference Manual, pp. 12-15, 12-16.

There is an obvious partition of the set of all (possible) generic packages into two disjoint subsets - those which do not have a generic formal parameter list, denoted GPØ, and those which do, denoted GP1. Let us first consider GPØ.

GPØ - Parameterless Generic Packages

There are three distinct kinds of parameterless generic package. First, there are those which have state and may be used in a multi-tasking environment. The classic example which has been cited is that of the package which provides the incarnation of a pseudo-random number generator[14,15]. All instantiations of such generic packages are identical in structure which is that of the generic package. This gives us our first design guideline:

DG1. To be reusable, packages with state must be generic.

Second, there are those packages which are incarnations of abstractions of fixed dimension and fixed element type. Many Ada packages which are incarnations of abstract data types - the ADT packages, belong to this class. We call such packages potential instantiations of a generic package. For example, a Bézier bicubic surface patch may be incarnated as the ADT package BEZIER_BICUBIC_MODEL[16]. To render it reusable, we prefix it by the keyword generic, install it in the universal standard Ada software library (USASL), provide a default standard instantiation BEZIER_BICUBIC_MODEL_INST and use that in place of the original in the software assemblage. All instantiations of such packages are identical in structure which is that of the generic package. This gives us our second design guideline:

DG2. Fixed ADT packages are by definition deemed to be reusable. They are entered into the USASL as generic packages and a default standard instantiation is provided for the end-user.

A note on the status of the USASL is in order. From the end-user's point of view, the USASL contains only reusable generic packages which may be instantiated for a particular application or it contains standard instantiations (not necessarily default ones) for direct use. There are other objects in the USASL neither available for nor visible to the end-user. However, the USASL is not an Ada library in the sense of the Reference Manual though it does share many of its properties. It is convenient to think of it coexisting with the usual Ada library for a given software assemblage. Before a candidate Ada module may be entered into the USASL it must be validated by some standard body, whether within a given company or organisation. The geographical extent of reuse of such a module depends entirely on the geographical extent of the body imposing the norm.

Third, there are those generic packages which encode structure that is truly an abstraction of existing or potential instantiations. Such instantiations are distinct and structurally isomorphic subject to either derivation or enrichment. The first example of this form of package that we actually used was the incarnation of hardware colour, an abstract data type which had actual instantiations RGB colour, CMY colour, and YIQ colour[13]. Another obvious candidate is an ordered pair of reals which has potential instantiations complex number and real coordinates. Such generic packages usually arise as a result of applying the piagetian principle mentioned above to existing (non-generic) packages. The following is the generic package specification, visible part only, for HARDWARE_COLOUR:

```
-- some with/use clauses
generic
package HARDWARE_COLOUR_MODEL is
  type HARDWARE_COLOUR is limited private;
  function MAKE(X,Y,Z: REAL_NUMBER)
    return HARDWARE_COLOUR;
  function SELECT_X(C: HARDWARE_COLOUR)
    return REAL_NUMBER;
  -- etc.,
  function IS_EQUAL(C1,C2: HARDWARE_COLOUR)
    return BOOLEAN;
  procedure ASSIGN -- CL := CR
    (CL: in out HARDWARE_COLOUR;
     CR: in HARDWARE_COLOUR);
  -- ...
private
  -- FOR IMPLEMENTOR'S EYES ONLY!
  -- ...
end HARDWARE_COLOUR_MODEL;
```

To obtain, for example, the RGB_COLOUR_MODEL we proceed in two stages. First, we instantiate HARDWARE_COLOUR_MODEL to give a kernel package RGB_COLOUR_MODEL_KERNEL. This resides in the USASL but is protected/hidden from the end-user. Second, we build the RGB_COLOUR_MODEL from it by (i) renaming the type (using a

subtype declaration) and functions of the generic package, and (ii) enrichment[13]. The resulting package is generic and a default standard instantiation provided as before. This leads to our third design guideline:

DG3. Packages that are perceived to be structurally isomorphic may be replaced by standard default instantiations of a generic package subject to derivation or enrichment.

There is one observation to be made that has an important impact on the end-user. Let OLD_MODEL_INST and NEW_MODEL_INST be produced according to DG2 and DG3, respectively, where both are incarnations of the same abstract data type. Although they have exactly the same type and function names, the syntax differs considerably. In the case of OLD_MODEL_INST, the structure and functionality is explicit and independent of any other package, except for context clauses. On the other hand, in NEW_MODEL_INST, there is a kernel that is clearly dependent on a generic package instantiation. Furthermore, there is no private part! The syntax of NEW_MODEL_INST may be constrained to be identical to that of OLD_MODEL_INST provided the end-user is willing to pay the price for an extra level of function call for each function in the kernel. From the point of view of the end-user's conceptual model this approach is more appropriate. It results in replacability of code even at syntactic level! Let us now turn to GP1 - those generic packages that have a non-empty formal parameter list.

GP1 - Generic Packages with Parameters

We have identified four distinct classes of attribute with respect to which the abstract structure encoded by a generic package may be parameterised - dimension, element type, functionality, and structure. In general, the more complex the formal parameter list, the more abstract is the encoded structure of the generic package. Parameters serve a dual purpose - they are generally used in the implementation of the package, and they ought to have significant semantic connotation for the end-user. We will discuss each class of parameter in turn.

DIMENSION: Stacks, queues, buffers, vectors, etc., all have the attribute of dimension. The Ada language mechanism we use for dimension is a generic formal object of discrete type which is a subrange of INTEGER. The general form is:

```
generic
  -- DIMENSION attribute
  N: POSITIVE_INTEGER; -- N >= 1
  -- ...
package Z_MODEL is
  type Z is limited private;
  -- ...
```

The majority of encoded structures are linear, i.e., possess only one dimension. The number of dimensions d contributes directly to the "genericness" of the package. Let us consider a particular instantiation of Z_MODEL for some N = NØ. Then objects of type Z produced from this instantiation are all of the same fixed dimension NØ. In other words, object dimensionality is established at instantiation time. This approach is particularly appropriate for the encoding of vector spaces, general linear groups (i.e. group of invertible nxn matrices), etc.

An alternative approach to the incorporation of the dimension attribute into a structure is to use Z as a parameterised type. The general form is:

```
generic
  -- ...
package Z_MODEL is
  type Z(N: POSITIVE_INTEGER) -- N >= 1
    is limited private;
  -- ...
```

In this case, dimensionality of objects is not fixed at instantiation time but at declaration time. This approach is particularly appropriate for conceptually infinite dimensional vector spaces - the space of polynomials is a typical example. Considerations such as these lead us to formulate our fourth design guideline:

DG4. In the encoding of a structure where it is required that all objects of that type have the same dimension, then one must use a dimension parameter N of subrange INTEGER, N >= 1, in the formal parameter list.

One word of caution is in order with respect to the use of the parameterised type approach. Suppose that one wants to encode a structure with dimension N where the implementation is required to use a task type - the buffer is the classical example. We hypothesise that it is impossible to use the parameterised type approach in such a case and we leave it as a challenge to the reader to try to prove us wrong.

ELEMENT TYPE: All of the examples

cited above for dimensionality are also usually parameterised with respect to element type. The Ada language mechanism that we normally use for element type is the limited private type and consequently, in such cases, we insist that one use a limited private type for instantiation. As a result, the introduction of such an element type raises the question as to whether or not we will require operations to test for equality and for assignment. The most general form is:

```
generic
  -- ...
  -- ELEMENT TYPE attribute
  type ELEMENT is limited private;
  with function IS_EQUAL(E1,E2: ELEMENT)
    return BOOLEAN;
  with procedure ASSIGN -- EL := ER
    (EL: in out ELEMENT;
     ER: in ELEMENT);
  -- ...
```

The number of element types e contributes directly to the "genericness" of the package. It is frequently the case that e = d, i.e. the number of element types is conumerous with the number of dimensions. Again, many structures are linear with respect to element type.

Let us look once more at the HARDWARE_COLUR_MODEL given above. It is clear that we can abstract *further from* it to give a TRIPLE_MODEL where the dimensionality is fixed at 3 and the element type is general:

```
generic
  type ELEMENT is limited private;
  with function IS_EQUAL ...
  with procedure ASSIGN ...
package TRIPLE_MODEL is
  type TRIPLE is limited private;
  function MAKE(A,B,C: ELEMENT)
    return TRIPLE;
  -- ...
```

From this package it is possible to produce the various colour models. Finally, one further generalisation with respect to element type is possible, keeping dimensionality fixed. Instead of a homogeneous triple, we could develop a hetereogeneous triple:

```
generic
  type ELEMENT1 is limited private;
  with ...
  -- ...
  type ELEMENT2 is limited private;
  with ...
  -- ...
  type ELEMENT3 is limited private;
  with ...
  -- ...
package TREE_MODEL is
  -- named after the tree concept of VDM
  type TREE is limited private;
  function MAKE(E1: ELEMENT1;
                E2: ELEMENT2;
                E3: ELEMENT3)
    return TREE;
  -- ...
```

Note that whereas the TRIPLE could have been derived from an encoded vector space because it is homogeneous with respect to element type, such is not the case for the TREE. Here we have reached the upper limit of the expressiveness of the generic package construct - it cannot handle arbitrary dimensional structure which is hetereogeneous with respect to element type. A similar limitation will arise in the case of the next class of parameter - that of functionality.

Now we must address the issue of providing guidelines to cope with the wide range of levels of abstraction such as that above which are possible. Because of the replacability concept developed above as a result of DG3 we propose the following:

DG5. The level of abstraction to be *encoded should* match the "computing maturity" of the package designer at that time.

A direct consequence of this is that designers of generic packages must be highly skilled and have a strong mathematical and computing science background. By DG3 evolutionary development in levels of abstraction are possible if desired or required. In order to inforce this kind of evolution, we propose another design guideline to counterbalance DG5:

DG6. The level of abstraction to be encoded should be just adequate to fulfill the requirements of the current software assemblage development, taking previous assemblages into consideration.

In other words, one should avoid being "too abstract" in the context of a particular project. We do not think that abstraction for its own sake is the proper criterion for generic package design. Hence, when we used Ada for interactive computer graphics software, we began with the concrete packages RGB_-

COLOUR_MODEL, CMY_COLOUR_MODEL, and YIQ_COLOUR_MODEL. At the next stage, the software evolved to the level of the HARDWARE_COLOUR_MODEL. Both TRIPLE_MODEL and TREE_MODEL are too abstract for our needs at present.

FUNCTIONALITY: This attribute is most often associated with those generic packages which encode agent abstractions, examples of which are to be found in the section on tasking below. The usual Ada language mechanism is the function or procedure. It is necessary to state this explicitly since we have shown[16] that there is a large class of mathematical functions which can be abstracted as objects, encoded as ADT packages and thus passed to other packages as element types! Functionality parameterisation requires that one also have element type parameterisation. Furthermore, the number of arguments is fixed at the generic package design time. One typical form for a function of one argument is:

```
generic
  type ELEMENT1 is limited private;
  -- ...
  type ELEMENT2 is limited private;
  -- ...
  with function TRANSFORM(E1: ELEMENT1)
    return ELEMENT2;
  -- ...
```

The number of functions f, times the number of their respective arguments a, contributes fxa to the "genericness" of the package.

STRUCTURE: This attribute is commonly employed in the use of a generic package as a filter for further derived structure and as a kernel for further enriched structure. An example whereby an equivalence class structure is imposed on a group structure is given in[6]. Mathematical structures lend themselves to this form of parameterisation in a natural manner. We do not know the range of applicability of this approach beyond the mathematical domain at the present time. We will identify generic packages by the parameter configuration (d,e,f,s) where d,e,f,s denote the number of dimensions, element types, functions or procedures, and structures, respectively, appearing in the generic formal parameter list.

## Embedding Tasks in Packages

"Since the language rules do not permit the declaration of tasks as library units, we often encapsulate a task inside a package"[3]

Agent abstractions are encoded in Ada as generic packages which encapsulate tasks. Considering the package specification, visible part only (PSV), it is not possible for the end-user to determine from the syntax whether the implementation is a static one, using non-task Ada constructs, or a dynamic one, using tasking. Furthermore, if the designer and implementor roles are played by distinct programmers then there is nothing in the PSV to tell the implementor that tasks are to be used in the case of an agent abstraction. Clearly, there is a need for the development of the principle of "information revealing" which is the direct opposite to the often cited principle of "information hiding". Naturally, such revelation cannot be done by allowing the end-user to see the imlementation. This would be contrary to the spirit of Ada packages. What is required is a formalism for expressing the dynamism of the encoded object in question. We are currently experimenting with conceptual graph notation[17] (CG notation), which can be used as a knowledge representation language, for that purpose. Among those things to be revealed by the CG notation at PSV level are (i) the object is dynamic, (ii) the calls made by the agent to other agents, (iii) the selection strategy by the agent of calls made to it by others, (iv) timing and guard conditions. Of course, all of this information is directly available as the task body. However, since it is forbidden to reveal it directly, then we must present the essential information necessary for the proper use of the package by the end-user.

The encoded agent abstractions presented below were designed by adhering to the piagetian principle of concrete-to-abstract. We took existing tasks from the literature[1,3,18], identified those elements which were a fixed part of the agent structure - calls on the agent (which were incorporated as procedures in the PSV), and internal data structure (which was incorporated in the implementation part). Note, that in this respect, the agent bears a strong resemblance to an abstract data type. The variable elements in the tasks were identified to be calls to other agents and their identies, plus the type of data to be passed at rendezvous time. These were candidates for the generic formal parameter list. Finally, dimension and functionality

parameters were considered to complete the encoding. We have adopted a single paradigm for the encoding of such agent abstractions. The PSV is almost identical to that of an ADT package and similar to the Z_MODEL outline shown above. Procedures NAME1, NAME2, etc., are used in the PSV and correspond to the entries NAME1, NAME2, etc., of the task being encoded. In the private part we use a CONTROLLER task type:

```
private
  task type CONTROLLER is
    entry NAME1 ...
    entry NAME2 ...
    -- ...
  end CONTROLLER;
  type Z is new CONTROLLER;
```

This paradigm is taken directly from the BUFFER example of Barnes[1].

Taxonomy of Tasks

Ada tasks and therefore agent abstractions can be partitioned into three distinct classes:

1. Those tasks which are called but do not call (CN tasks) - purely passive tasks:
   - MAILBOX
   - BUFFER
   - SCHEDULER
   - KEYBOARD_HANDLER
2. Those tasks which are not called but which call (NC tasks) - purely active tasks:
   - TRANSPORTER
   - USER
3. Those tasks which are called and which call (CC tasks) - mixed active and passive tasks:
   - SERVER

Let us consider each of these classes in turn.

CN TASKS: Such tasks are among the easiest to encapsulate as generic packages. This is simply due to the fact that they need never know the identies of their callers. I present two simple examples - the MAILBOX which is a generalisation of that to be found in[1] and which has already appeared in[19], and the KEYBOARD_HANDLER which is a generalisation of that to be found in[18]. First, let us look at the MAILBOX. The generic package is:

```
generic
  type ITEM is limited private;
  with procedure ASSIGN(IL: in out ITEM;
                        IR: in ITEM);
package MAILBOX_MODEL is
  type MAILBOX is limited private;
  procedure DEPOSIT(MB: in out MAILBOX;
                    I: in ITEM);
  procedure COLLECT(MB: in out MAILBOX;
                    I: in out ITEM);
private
  -- FOR IMPLEMENTOR'S EYES ONLY!
  -- ...
end MAILBOX_MODEL;
```

The MAILBOX has the parameter configuration (0,1,0,0), i.e., is parameterised with respect to one element type - ITEM. Note in particular the parameter ASSIGN which provides assignment. Since this procedure is only used in the package body then the package designer must provide a rationale for its occurence that is meaningful to an implementor on the one hand, and an end-user on the other. This may be accomplished by specifying that the semantics of depositing and collecting imply that of assignment. Let us now look at the actual implementation:

```
private
  task type CONTROLLER is
    entry DEPOSIT(I: in ITEM);
    entry COLLECT(I: in out ITEM);
  end CONTROLLER;
  type MAILBOX is new CONTROLLER;

package body MAILBOX_MODEL is
  task body CONTROLLER is
    STORE: ITEM;
  begin
    accept DEPOSIT(I: in ITEM) do
      ASSIGN(STORE,I);
    end accept;
    accept COLLECT(I: in out ITEM) do
      ASSIGN(I,STORE);
    end accept;
  end CONTROLLER;
  --
  procedure DEPOSIT(MB: in out MAILBOX;
                    I: in ITEM)
  is
```

```
  begin
    MB.DEPOSIT(I);
  end DEPOSIT;
  --
  procedure COLLECT(MB: in out MAILBOX;
                    I: in out ITEM)
  is
  begin
    MB.COLLECT(I);
  end COLLECT;
end MAILBOX_MODEL;
```

Our second example is liable to be somewhat controversial. It is the KEYBOARD_HANDLER_MODEL:

```
with SYSTEM;
generic
  DONE_INTERRUPT: SYSTEM.ADDRESS;
  CHAR_ADDRESS: SYSTEM.ADDRESS;
package KEYBOARD_HANDLER_MODEL is
  type KEYBOARD_HANDLER is
    limited private;
  --
  procedure TAKE
    (KBH: in out KEYBOARD_HANDLER;
     CH: out CHAR);
  --
  procedure KB_DONE
    (KBH: in out KEYBOARD_HANDLER);
private
  -- FOR IMPLEMENTOR'S EYES ONLY!
  -- ...
end KEYBOARD_HANDLER_MODEL;
```

Note that this package is parameterised with respect to addresses. However, since dimension, element type, functionality, and structure is fixed or non-existent, then this generic package is assigned the parameter configuration (0,0,0,0) which is in agreement with our intuition about the genericness of such an object as a keyboard handler. Typical usage might look like:

```
with KEYBOARD_HANDLER_MODEL;
package NADIR_KEYBOARD_HANDLER_MODEL
  is new
  KEYBOARD_HANDLER_MODEL
    (DONE_INTERRUPT => 8#100#,
     CHAR_ADDRESS => 8#177462#);
```

It is this form of instantiation where the actual parameters are constants that leads us to suppose that the approach ought to be valid. But as mentioned earlier, since generic formal parameters are never static, then there appears to be some doubt as to the legality of the Ada package presented. For completion, I give the details of the implementation:

```
private
  task type CONTROLLER is
    entry TAKE(CH: out CHAR);
    entry KB_DONE;
    for KB_DONE use at DONE_INTERRUPT;
  end CONTROLLER;
  type KEYBOARD_HANDLER is
    new CONTROLLER;

package body KEYBOARD_HANDLER_MODEL is
  --
  task body CONTROLLER is
    BUF: CHAR;
    DBR: CHAR;
    for DBR use at CHAR_ADDRESS;
  begin
    loop
      accept KB_DONE do
        BUF := DBR;
      end accept;
      accept TAKE(CH: out CHAR) do
        CH := BUF;
      end accept;
    end loop;
  end CONTROLLER;
  --
  procedure TAKE
    -- much the same as for MAILBOX
  procedure KB_DONE
    -- much the same as for MAILBOX
  --
end KEYBOARD_HANDLER_MODEL;
```

NC TASKS: These are among the most difficult to encapsulate as generic packages simply because they must know the identity of those agents which they call. We will consider the TRANSPORTER which is taken from[18]. A transporter task calls another task to collect some item. It then calls a second task to receive delivery of the same item. We encapsulate it as the MESSENGER which has the form:

```
generic
  --
  type ITEM is limited private;
  --
  type SENDER is limited private;
  S: in out SENDER;
  with procedure COLLECT
    (S1: in out SENDER;
     I: in out ITEM);
  --
  type RECEIVER is limited private;
  R: in out RECEIVER;
  with procedure DELIVER
    (R1: in out RECEIVER;
```

```
    I: in ITEM);
  --
package MESSENGER_MODEL is
  type MESSENGER is limited private;
private
  -- FOR IMPLEMENTOR'S EYES ONLY!
  -- ...
end MESSENGER_MODEL;
```

The MESSENGER has the parameter configuration (0,3,2,0), i.e., is parameterised with respect to three element types - ITEM, SENDER, RECEIVER, and of functionality of order 2 - COLLECT, DELIVER. These two operations in turn imply the types of agents to be called and their identities which in this case are to be supplied as the actual parameters for the formal objects S and R. Note the simplicity of the PSV. It consists solely of a type! Let us consider the details of the implementation where we have kept the name TRANSPORTER in preference to CONTROLLER:

```
private
  task type TRANSPORTER;
  type MESSENGER is new TRANSPORTER;


package body MESSENGER_MODEL is
  task body TRANSPORTER is
    POUCH: ITEM;
  begin
    loop
      select
        delay 0.0;
        COLLECT(S,POUCH);
        DELIVER(R,POUCH);
      or
        terminate;
      end select;
    end loop;
  end TRANSPORTER;
end MESSENGER_MODEL;
```

CC TASKS: To illustrate this class of tasks we will look at the generalisation of the server task presented in[1].

```
generic
  --
  type IN_ITEM is limited private;
  with procedure ASSIGN
    (IIL: in out IN_ITEM;
     IIR: in IN_ITEM);
  --
  type OUT_ITEM is limited private;
  with procedure ASSIGN
    (OIL: in out OUT_ITEM;
     OIR: in OUT_ITEM);
  --
  with function TRANSFORM(II: IN_ITEM)
    return OUT_ITEM;
  --
  type MAILBOX is limited private;
  with procedure DEPOSIT
    (MB: in out MAILBOX;
     OI: in OUT_ITEM);
  --
  type MAILBOX_ADDRESS is
    limited private;
  with function OWNER
    (MBA: MAILBOX_ADDRESS)
    return MAILBOX;
  with procedure ASSIGN
    (MBAL: in out MAILBOX_ADDRESS;
     MBAR: in MAILBOX_ADDRESS);
  --
package SERVER_1_MODEL is
  type SERVER_1 is limited private;
  procedure REQUEST
    (S: in out SERVER_1;
     MBA: in MAILBOX_ADDRESS;
     II: in IN_ITEM);
private
  -- FOR IMPLEMENTOR'S EYES ONLY!
  -- ...
end SERVER_1_MODEL;
```

The SERVER_1 has the parameter configuration (0,4,2,0). One important observation must be made. The actual parameter to be used for MAILBOX_ADDRESS is supplied by an object from an instantiated generic ADT package that essentially provides handles (or identities) for any kind of object encoded as a limited private type. This package provides one level of abstraction above the access type. There is a function which generates "addresses" much like the gensym function of Lisp, and is implemented using an allocator. The function is invoked by a USER agent requiring service. Anotner function is used to dereference the object from the address. It has the comparable effect of ".all" on a value of an access type. This function is used to instantiate the OWNER function given above. The name SERVER_1 is used to signify that the job to be performed is done by the TRANSFORM function which takes 1 argument. Note that this generic package can be used as a paradigm to replace many of the generic subprograms often cited. Details of the implementation follow:

```
private
  task type AGENT is
    entry REQUEST(MBA: MAILBOX_ADDRESS;
                  II: IN_ITEM);
  end AGENT;
  type SERVER_1 is new AGENT;


package body SERVER_1_MODEL is
```

```
  task body AGENT is
    REPLY_ADDRESS: MAILBOX_ADDRESS;
    JOB: IN_ITEM;
  begin
    loop
      accept REQUEST
        (MBA: MAILBOX_ADDRESS;
         II: IN_ITEM)
      do
        ASSIGN(REPLY_ADDRESS,MBA);
        ASSIGN(JOB,II);
      end accept;
      DEPOSIT(OWNER(REPLY_ADDRESS),
              TRANSFORM(JOB));
    end loop;
  end AGENT;
  --
  procedure REQUEST
    (S: in out SERVER_1;
     MBA: in MAILBOX_ADDRESS;
     II: in IN_ITEM)
  is
  begin
    S.REQUEST(MBA,II);
  end REQUEST;
end SERVER_1_MODEL;
```

This work that we have carried out in relation to encapsulating tasks in generic packages leads to our final and tentative design guideline:

DG7. In providing tasking for a software assemblage, one classifies the task to be used and encapsulates it into a generic package.

This is a tentative guideline in so far that we are unsure as to the level of granularity to be employed in encapsulating tasks. As stated above, we use one task per package at present. It may be the case that task clusters, rather than single tasks, might be more appropriate for encapsulation! This implies that we need to find a higher-level of agent abstraction with respect to interaction. This is one of our current research directions. Finally, note that in the task taxonomy we have ommitted any mention of tasks with entry families. Again this is meat for further thought.

## Conclusion

Formerly, we held the view that the package was the principle building block of the Ada programming language. Now, in view of the need for and desirability of reusability, we qualify the word package with the adjective generic. In order to design good generic packages one must start with many concrete examples of good non-generic packages. Just as in any learning situation, the piagetian principle of concrete-to-abstract is applicable. In the training of Ada programmers who will be responsible for the design and implementation of generic packages, we consider it essential that they be exposed to a wide variety of the same. In the first instance, we have indicated that any non-generic package may be immediately converted to a generic one of type GP0 and a default instantiation provided in its place. This process ought not to give rise to any significant compilation or run-time overhead, and it has the advantage of introducing the concept and use of generic packages as painlessly as possible to the new Ada programmer. A second source of generic package of type GP1 is also immediately available - these include the classical computer science structures.

We have also shown that it is possible to provide an evolutionary development in levels of abstraction whereby instantiations are replacable without affecting the end-user. We have asserted that reusability of Ada software components is inextricably bound to the degree of genericness of a given package. To measure such genericness we have proposed a taxonomy based on the form of the generic formal parameter list and identified four classes of parameter - dimension, element type, functionality, and structure. The parameter configuration (d,e,f,s) is presented as a first attempt at such a measure.

To illustrate our approach and provide comprehensive examples, we decided to concentrate on the area of agent abstractions. Much work has already been done for the encoding of static objects. As a first step, we have identified agent abstractions with single tasks and provided a simple taxonomy thereof. The PSV of encapsulated tasks requires a greater degree of supplemental information for the end-user and the implementor of such packages than the comparable PSV for static objects. This was the principle of "information revealing". Work in this area is essential and urgent if such agent abstractions are to be (re)usable at all! Finally, we must look at the possibility of identifying higher levels of abstraction of interaction units and the consequent encoding of task clusters rather than single tasks in generic packages.

Acknowledgement

I wish to thank my colleague Hans-Juergen Kugler for the many discussions on the encoding of agent abstractions and, in particular, for his time in contributing to the interpretation of the "semantics" of the English language text used in the Ada Language Reference Manual which resulted in the coded examples contained herein. Much of this material was presented in rough draft form as a position paper to the members of the Ada-Europe Education WG whose comments and criticisms are gratefully acknowledged.

Author

The author, Mícheál Mac an Airchinnigh, may be contacted at Generics (Software) Limited, Unit 7, Leopardstown Office Park, Foxrock, Dublin 18, or at Department of Computer Science, University of Dublin, Trinity College, Dublin 2, Ireland. He graduated from the University of London (1978) with an external B.Sc degree in Mathematics. The following year he completed his M.Sc degree in Computer Science at the University of Dublin. After doing one year postgraduate work in SUNY, Stony Brook, New York, he joined the faculty of the Department of Computer Science, University of Dublin, Trinity College, (1980).

The author was a cofounder of and is currently a Technical Director of Generics (Software) Limited, a new Research and Development Company specialising in Ada. He is a member of ACM, SIGAda, SIGCHI, SIGGRAPH, SIGPLAN, an affiliate member of the IEEE Computer Society and a member of EUROGRAPHICS and Ada-Europe.

References

1. J.G.P. Barnes, Programming in Ada, 2nd edn., Addison-Wesley Publishing Company, London (1984).

2. G. Booch, "Dear Ada Column," ACM SIGPLAN AdaTEC Ada Letters Vol. III(3) pp. III-3.25--III-3.28 (November, December 1983).

3. G. Booch, Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California (1983).

4. M. Mac an Airchinnigh, "Some Educational Principles Relating to the Teaching and Use of Ada," pp. 201-210 in Proc. of the Third Joint Ada Europe/AdaTEC Conf. on Industrial Applications of Ada and Ada Programming Support Environment, ed. J. Teller,Cambridge University Press (1984).

5. DoD, The Ada Language Reference Manual, ANSI/MIL-STD 1815A, Alsys, La Celle-Saint-Cloud, France (1983).

6. S.D. Litvintchouk and A.S. Matsumoto, "Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification," IEEE Trans. on Software Engineering Vol. SE-10(5) pp. 544-551 (September, 1984).

7. Department of Defense, Application-Oriented Technologies and Reuse. October, 1982.

8. J.C. Batz, P.M. Cohen, S.T. Redwine Jr., and J.R. Rice, "The Application-Specific Task Area," IEEE Computer Vol. 16(11) pp. 78-85 (November, 1983).

9. M.E. Lesk, "Lex - A Lexical Analyzer Generator," Computer Science Technical Report Vol. #39,Bell Laboratories, (October, 1975).

10. S.C. Johnston, "YACC - Yet Another Compiler-Compiler," Computer Science Technical Report Vol. #32,Bell Laboratories, (July, 1975).

11. N. Gehani, Ada - An Advanced Introduction, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1983).

12. G. Booch, "Dear Ada Column," ACM SIGPLAN AdaTEC Ada Letters Vol. III(4) pp. III-4.34--III-4.36 (January, February 1984).

13. M. Mac an Airchinnigh, "Mathematical Lattices of Abstract Data Types in Ada," Proc. of Colloquium on Object-based Design, pp. 5/1-5/7 Institute of Electrical Engineers, Savoy Place, London, (November, 1984).

14. B.A. Wichmann and J.G.J. Meijerink, "Converting to Ada Packages," pp. 131-139 in Proc. of the Third Joint Ada Europe/AdaTEC Conf. on Industrial Applications of Ada and Ada Programming Support Environment, ed. J. Teller,Cambridge University Press (1984).

15. G. Booch, "Dear Ada," ACM SIGAda Ada Letters Vol. IV(2) pp. IV-2.15--IV-2.18 (September, October 1984).

16. M. Mac an Airchinnigh, "CAD, GKS, and Ada," in Workshop on Interfacing Old and New Graphical Software to GKS, ed. K. Brodlie, G. Pfaff,Springer-Verlag, Berlin Heidelberg New York Tokyo (to appear 1984).

17. J.F. Sowa, Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley Publishing Company, London (1983).

18. R.J.A. Buhr, System Design with Ada, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1984).

19. M. Mac an Airchinnigh, "The CONTEXT, a high-level structuring concept for GKS input," Computers and Graphics, (to appear 1985).

# THE USE OF ADA[R] IN IMPLEMENTING A RAPID PROTOTYPING SYSTEM

Guylaine M. Pollock
Sallie Sheppard

Laboratory for Software Research
Department of Computer Science
Texas A&M University

## Abstract

The use of Ada in implementing a rapid prototyping system for software metrics is evaluated by focusing on the project issues dealing with Ada and discussing advantages and disadvantages incurred by the utilization of Ada. The procedure used in conducting the work consisted of defining a set of kernel primitives from which the prototyped software is constructed and defining procedures to facilitate implementations from the kernel primitives. Generic Ada packages have been designed to incorporate the kernel primitives and define the structures and operations necessary for their manipulation. The implementation utilized in developing software illustrates the feasibility of rapid prototyping via reusable software parts. Based on experiences with the prototyped system, an evaluation is presented on the use of Ada in developing rapid prototyping systems.

## Introduction

A rapid prototyping system for the development of software metrics is currently being designed and implemented in Ada within the Laboratory for Software Research at Texas A&M University. This system supports the development of software metric prototypes from the reuse of software components called "kernel primitives" predefined within the system which can be optionally extended by new components provided by the user. The resulting prototype is used for initial analysis of the metrics being studied and may be altered repeatedly until a promising model has been devised.

Ada was selected as the implementation language because the packaging and generic constructs available in Ada provide an attractive environment for the development of reusable software. This approach follows the recommendation of current literature which advocates the utilization of reusable software parts and heavy parameterization as possible techniques to use in the development of rapid prototyping systems. Implementing the system in Ada has also provided an opportunity to evaluate the use of the language in constructing software development tools. Preliminary experiences indicate the attractiveness of Ada for this and similar projects. The advantages and disadvantages experienced because of the utilization of Ada are discussed after a brief description of the system being implemented.

## System Description

The basic premise of the system being developed to support the rapid prototyping of software metrics is that metrics within common classes tend to measure the same objects. Traditional implementation strategies have often led to a duplication of effort in constructing software tools to collect data for analysis of the various metrics. This duplication of effort is evident in current literature in both function and language as described below. Duplication of effort due to function is intuitively obvious as one would expect metrics that measure similar qualities to be somewhat similar in form and function. This duplication occurs frequently in software complexity metrics. Twenty-two software complexity metrics were examined to test this hypothesis. Certain measurements were found to be common to nearly all the metrics, while each individual metric had some measurement common to at least one of the other metrics. For example, an incidence matrix is common to nearly all of the metrics although the individual metric definitions do not indicate this fact; that is, the construction of an incidence matrix is useful in deriving many of the metrics even though the metric definitions

---

[R] Ada is a registered trademark of the Department of Defense, Ada Joint Program Office.

This material is based upon work funded in part by a research grant from IBM.

seem to indicate other methods of evaluation. Utilization of the incidence matrix may, however, require a slightly different algorithm in evaluating the resulting data.

The other area of duplication in the current development of software tools for metric research is language. Many research efforts heretofore have been concerned with developing previously defined and implemented software tools to collect data on programs written in a language different from the coded language of programs the original software tools were developed to measure or examine. A review of metric literature illustrates this duplicity: researcher A has developed a system to collect and analyze data from FORTRAN programs according to Halstead's metric; researcher B has developed a system to collect and analyze data from COBOL programs according to Halstead's metric; etc. Although duplication of effort is clearly evident, it has been largely unavoidable due to limitations and restrictions imposed by available computing environments.

The rapid prototyping system developed in this project attempts to reduce or eliminate duplication of effort in both function and language. The system accomplishes this goal through the definition and subsequent utilization of reusable software modules or "kernel metric primitives". The kernel primitives perform certain predefined operations common within chosen metric classes. Typical classes of metrics include software complexity, software reliability, and software quality. To avoid duplication of effort due to function, the researcher utilizes as many of the predefined kernel primitives as possible in developing software to collect data for a new metric. Many metrics tend to extend ideas found in previous metrics, thus the same code may be useful in many different instances. The objective is to define primitives to serve as basic building blocks in constructing new software. Important parameters are also defined and abstracted out of the primitives allowing redefinition with a minimum of effort. Thus the researcher has the capability to redefine the primitives in a "primitive" fashion. This system characteristic also facilitates the fine-tuning of metrics during validation without full scale software development.

Duplication of effort due to differences in the language being measured are avoided by abstracting out the concept of language from the kernel primitives. The metric researcher instantiates an appropriate version of the desired kernel primitives for any of the predefined languages or provides language tokens and semantic rules for new languages. Thus similar measurements can be made on programs with a minimum of regard to the code language.

Although some researchers have advocated the use of Ada in developing rapid prototyping systems whereby the prototype eventually becomes the final product after successive iterations[3], the system being developed at Texas A&M University[7] utilizes a different approach. Once a prototype that warrants further study has been developed using the predefined kernel primitives and user extensions, the prototype is typically discarded and research continues by incorporating the metric model into a more efficient software tool for further experimentation and analysis. Rapid development and fine-tuning of the prototype in this manner frees the metric researcher from an initial large development time in implementing a software tool to model new metrics until preliminary analysis identifies a promising model for more intensive study. Thus the approach used in the design of this system consists of the development of kernel primitives or reusable software parts to be utilized as building blocks in developing a desired prototype. The technique is somewhat similar to approaches described in reusable software literature.[5,6]

The procedure used in this project consists of defining a set of kernel primitives from which the prototyped software metrics are constructed and defining procedures to facilitate implementation of metrics from the kernel primitives. Generic Ada packages incorporate the kernel primitives and encapsulate definitions of the structures and operations necessary for their manipulation. Approximately 3000 lines of code have been implemented with an estimate of 5000 lines of code as the size of the completed system. The system includes ten packages.

A prototype of the rapid prototyping system is being developed on a Digital Equipment Corporation VAX 11/750 under VMS version 3.5. The system development was started on version 1.1 of NYU's Ada/ED System and was transferred to version 1.41, a more recent validated version. In addition, attempts have been made to run the prototyped system on version 2.1 of the TeleSoft Ada Compiler under VAX/VMS but have been unsuccessful up to this point as generics and tasking are not yet fully implemented in version 2.1.[12]

Definite advantages and some disadvantages have been encountered during system development due to the use of Ada

as the implementation language. Overall the decision to use Ada has been considered to be advantageous. Implementation of the system design has been greatly facilitated by the use of Ada, and the disadvantages experienced are rather minor as the causes are transitory in nature. Consequently, they should not impact future research to the same degree.

## Advantages of Ada

Many of the constructs and capabilities available in Ada have enhanced the development of the system design and implementation. These include features such as tasking, concurrent processing, package specifications, separate compilation, generics, naming conventions, and typing facilities that allow data abstractions and strong typing restrictions. Although several other languages do provide some of the capabilities listed here, none of those languages provide as many of the attractive capabilities and constructs as are offered in Ada. The impact of several of these features on the rapid prototyping system that is under development will be examined to illustrate the benefit of implementing the system in Ada.

### Generics.

The generic facilities in Ada provide one of the greatest advantages of the language for the construction of reusable software parts. This is evident by considering the applications of the generic parameters to reusability. Three types of generic formal parameters are possible: types, objects, and subprograms. Generic types present the most obvious reusability aspect as too frequently in the past numerous routines have been rewritten simply because the data to be manipulated by the algorithm was in a form different from that utilized by an already coded and debugged routine. Figure 1 gives an example which demonstrates the value of generic types in promoting reusability in the package specification of a generic queueing routine and a procedure which employs multiple queue elaborations based on the given specification. This package is available for use in the metric prototyping system and a modified version has been used in a simulation package implemented in Ada.[4,9,10]

The generic package QUE provides a template for the algorithm to create and manipulate a queue of objects. An instance of the package created by instantiation defines a queue for specific objects, the form of the objects being

```
generic
 type Q_ELEMENT is private;
package QUE is
 type PNT_Q_ELEMENT is access Q_ELEMENT;
 type QUEUE          is private;
 procedure INIT(Q      : out QUEUE);
 procedure PUT(ENTITY : in  PNT_Q_ELEMENT;
               Q      : in  out QUEUE);
 procedure GET(ENTITY : out PNT_Q_ELEMENT;
               Q      : in  out QUEUE);
private
 type LINK_RECORD;
 type LINK is access LINK_RECORD;
 type LINK_RECORD is
   record
     ELEMENT_LOC  :  PNT_Q_ELEMENT;
     NEXT         :  LINK;
     PREV         :  LINK;
   end record;
 type QUEUE is
   record
     SIZE  :  NATURAL;
     EMPTY :  BOOLEAN;
     HEAD  :  LINK;
   end record;
end QUE;

with QUE;
procedure EXAMPLE is

 package INTEGER_QUEUE is
   new QUE(Q_ELEMENT => INTEGER);
 use INTEGER_QUEUE;
     -- creates new queue for integers

 type JOB_DESCRIPTION is
   record
     -- a suitable description
   end record;

 package JOB_QUEUE is
   new QUE(Q_ELEMENT => JOB_DESCRIPTION);
 use JOB_QUEUE;
     -- creates new job description queue

begin
 -- desired processing
end EXAMPLE;
```

Figure 1: Generic Queue Facility

identified at the point of instantiation. Generic types thus allow the abstraction of implementational details such as the internal representation of data from the algorithmic representation, providing reusable code. This code may be reused multiple times within a single application or may be utilized in many different applications that require the functional abstraction provided by the generic unit.

Generic objects and subprograms extend the power of this type of reusability. Generic objects specify global values to be utilized within the

generic unit while localizing the access of the value to the generic subunits. Specification of the types of these objects may be postponed until elaboration of the unit. Generic subprograms permit the definition of operations that are not provided within the language for user defined data types. This allows full freedom of data abstraction in generic units. However the generic definition may place restrictions upon the types possible for specification by the user. For example, the underlying form of a generic type may be restricted to a type such as character, integer, or real.

In developing new systems, implementation time is reduced by incorporating previously written and debugged code that is functionally reusable, thereby improving software development productivity. However, time is still wasted in converting old code into different representational forms even if the code is functionally abstract. Generics eliminate this need if the abstraction is properly defined when initially implemented. The kernel primitives defined within the rapid prototyping system are generic constructs, allowing a user to instantiate new versions of the primitives based upon the format of the data that is being measured. Furthermore, the generic facilities combined with other aspects of the language definition permit the definition of primitives that demonstrate semantical and syntactical reusability. The primitives support changes in the definitions or meanings of the data that is being processed in addition to changes in the syntactical form, thus permitting the abstraction of language from the primitives so that software tools constructed from the prototype may be processed on programs coded in different languages.

Tasking.

Tasking has been used within the system as a means of encapsulating related functions and operations. This allows the logical separation of functions which can be performed in parallel. For example, several of the kernel primitives encapsulate data structures which contain multiple linkages to desired information. Many of these structures require substantial housekeeping functions to maintain the required links. Often the requested information is determined before the housekeeping functions are performed. Such housekeeping functions are performed by tasks in order to allow these duties to be logically performed in parallel with the operations performed by the calling routine. Instead of completing all necessary operations before returning the information to the caller, a task is invoked to process remaining details while the retrieved information is passed back to the calling routine. Following this procedure mandates the necessity of a guard or proctor task to prevent access to the structure before all housekeeping has been completed. An example of these task structures is given in Figure 2.

```
task GUARD is
  entry SEIZE_STRUCTURE;
  entry FREE_STRUCTURE;
end GUARD;
task body GUARD is
begin
  loop
    accept SEIZE_STRUCTURE;
    accept FREE_STRUCTURE;
  end loop;
end GUARD;

task HOUSEKEEPER is
  entry CLEAN_UP (DUTIES : in JOB_LIST);
end HOUSEKEEPER;
task body HOUSEKEEPER is
  DIRECTIONS : JOB_LIST;
begin
  loop
   accept CLEAN_UP(DUTIES : in JOB_LIST)do
     DIRECTIONS := DUTIES;
   end;
   GUARD.SEIZE_STRUCTURE;
   -- execute duties from given directions
   GUARD.FREE_STRUCTURE;
  end loop;
end HOUSEKEEPER;
```

Figure 2: Example of Tasking Usage

Perusal of this example illustrates a typical use of tasking that is quite straightforward. Note that the rendezvous consists only of obtaining a copy of the entry parameter. This releases the initiating routine to continue in parallel with the task execution.

Although utilizing tasking program structure degrades performance if performed on a serial architecture, it offers the advantage of actual parallel execution where multiple processors are available. In such cases the tasking approach potentially improves the operational efficiency of the metric prototypes.

As many of the kernel primitives contained within the rapid prototyping system are designed to be low level functions, certain primitives when utilized to construct a metric prototype may be performed concurrently. The tasking facilities within the language definition provide the ability to incorporate concurrency aspects within the system in a relatively simple and straightforward manner. Thus concurrent

processing of primitives may occur in addition to the concurrent processing of housekeeping functions such as link maintenance in various encapsulated structures.

Packages.

The packaging features facilitate the application of standard software engineering principles such as abstraction, information hiding, and principles of modularity. Most kernel primitives are encapsulated by packages which allow multiple procedures and structures to be utilized in defining the primitives and at the same time offer a simplistic method of elaboration to the user. By including an elaboration of the primitive package within the declarative section of a metric definition, one is automatically provided with all declarations and procedures crucial to obtaining either a particular metric or a particular primitive. Thus not only is the user freed from having to familiarize himself with unnecessary detail concerning the operation of the elaborated primitive, as essential information is clearly identified within the package specification, but he is unable to alter any aspects of the data collected by the primitive except through the defined operations.

Typing.

Typing facilities provided in the Ada language definition aided the simplification of data specifications for the prototype system and promoted clarity of use by prototype developers utilizing the system. The strong typing constraints assisted in restricting usage of the primitives in the manner for which they were defined. These language defined constraints help maintain the integrity of any objects exported by the primitives by regulating the operations instigated by the parent unit upon the object. Furthermore, if there is a dependency between procedures based on generic parameter data types, it is strictly enforced by the language.

Separate Compilation.

The separate compilation feature as provided by Ada advances development of the prototype in that many of the components are already compiled. Thus the construction of a prototype with the system is quicker and allows faster modifications or fine-tuning in the resultant prototype as changes do not necessarily mandate recompilation of all routines. Obviously the desired modification greatly impacts the magnitude of the required recompilations. Furthermore, this language trait is very useful as it allows the metric definition facility to create new primitives through modifications to existing components without having to recompile other primitives contained within the same encapsulation as long as the modified components are nongeneric.

In addition to aiding utilization of the system, separate compilation supported development and improved morale by permitting faster implementation of various components and forcing earlier specifications of exact programming interfaces. Furthermore, having parts of the system up and running rather quickly promoted a sense of accomplishment.

Other Features.

Other features of Ada were useful in the system development such as the naming conventions which helped to promote clarity during implementation, and statements that helped to encourage good structured programming techniques. These features were definitely supportive; the first 800 lines of code contained only three logical bugs. Obviously, the language does indeed promote the incorporation of many software engineering principles such as modifiability, efficiency, reliability, and understandability as advocated by visible Ada language proponents.[2]

## Disadvantages of Ada

The disadvantages encountered as a result of selecting Ada as the project implementation language for the rapid prototyping system were mainly caused by the status of the state of the art of Ada implementation. Specifically, the dearth of production quality compilers and the impact of the validation procedures proved to be the largest inconveniences of the language selection--inconveniences which are due to the fledgling status of the language implementation and are hoped to be transitory in nature. These problems are briefly discussed in order to identify problems similar research projects might expect to encounter if developed in a similar environment.

Compiler Quality.

The NYU Ada/Ed System is an excellent tool for educational training but as stated in the user's reference guide is not intended to be used as a production quality compiler. Designed for accuracy more than efficiency, the Ada/Ed System is slow in operation. Pedagogical problems are not as adversely affected by this slowness as are larger, application oriented software. Future versions of the

compiler will provide improved efficiency; improvements are already evident between earlier versions.

Ada/Ed's size restrictions constrain the volume of implementable code in terms of lines of code and number of procedures contained within a package; certain memory constraints depend upon the available SETL workspace. The size restrictions are disheartening to encounter if unknown, as they may require redesign of various aspects of the system and may prohibit project completion. Other projects have also experienced this problem.[1]

Error detection by Ada/Ed is quite good with excellent error messages that direct the user back to the appropriate location within the language reference manual that indicates the aspect of the language definition being violated. However, the user must be prepared to differentiate between error messages generated as a result of a language compilation error, a language execution error, or a SETL execution error. Furthermore, it is possible to encounter discrepancies between the compiler actions and the language definitions, or "bugs" within the compiler. This possibility is decreasing with time but was particularly troublesome in early work with Ada/Ed.

Version 2.1 of the TeleSoft Ada compiler does not fully implement generics and tasking; among other problems, recursive data types within generic units are not accepted by the compiler. Consequently, the prototype metric system development was unable to utilize this compiler.

Compiler Validation.

Annual compiler validation, a characteristic of the Ada language environment, also created some disadvantages for the implementation of the rapid prototyping system under investigation. The major problem was the need to modify previously completed code in order to transfer the system onto the newer compiler. Although only minor changes were required in this instance, modifications were necessary throughout every program unit. As the changes were made, the units were also executed to make sure that additional errors were not introduced inadvertantly while upgrading the code. For an application of this size a couple of days were required to perform the transfer. If the necessary changes had more significant impact upon the design of the system, more time would have been required.

The concept of validation also affected morale in that the project was initiated on an older version of the compiler which meant that there were known errors in the compiler being utilized but the exact errors were basically unknown. This had the effect of promoting occasional interpretation of obscure errors as defaults of the compiler rather than possible misinterpretations or misapplications of the language definition.

Language Definition.

The language definition states "an implementation may also require that subunits of a generic unit be part of the same compilation" which indicates that separate compilation of units contained within generic units is not necessarily supported by all implementations of the language. Consequently, this feature must be avoided by generic units in order to satisfy the requirement of portability for the metric prototyping system. Separate compilation would be an extremely useful feature in this particular application as generic packages constitute the primitives and generally contain multiple subunits.

One technique available for the creation of new primitives is the modification of existing components. Because separate compilation of subunits within generic units is not allowed within the system design, any such modifications require recompilation of the entire generic package, even if the modifications are supported by the metric definition editor. This disadvantage also occurs during development of completely new packages. While debugging new primitives under development, the entire package must be recompiled after each subunit alteration even though coexistent subunits maintain consistent states.

A final disadvantage incurred by the language definition was the inability to interface with programs written in other languages, forcing the recoding of standard applications that had already been implemented so that they could be utilized within the system. This is also a transitory problem as other languages will define interfaces with Ada as more applications begin to appear which require them.

## Conclusions

The main consensus on the evaluation of Ada as the project source language is that Ada has enhanced the project through the advantages provided by various features contained within the language definition. The inherent disadvantages encountered due to the language selection are largely transitory in nature and

should not greatly influence future work. However, in the future, attention must be given to the careful distribution of knowledge concerning differences between the latest validated compilers and their previous versions in order to lessen the impact of the revalidation process.

## References


1. Basili, Victor R., Nora Monina Panlilio-Yap, Connie Loggia Ramsey, Chang Shih, and Elizabeth E. Katz. "A Quantitative Analysis of a Software Development in Ada", Dept. of Computer Science, University of Maryland, TR-1403, May 1984.

2. Booch, Grady. Software Engineering with Ada. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1983.

3. Duncan, Arthur G. "Prototyping in ADA: A Case Study", ACM SIGSOFT Software Engineering Notes, Vol. 7, No. 5, (December 1982) pp. 54-57.

4. Friel, Patricia and Sallie Sheppard. "Implications of the Ada Environment for Simulation Studies", Proceedings of the 1984 Winter Simulation Conference, (December 1984) pp. 477-489.

5. Lanergan, Robert G. and Charles A. Grasso. "Software Engineering with Reusable Designs and Code", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, (September 1984) pp. 498-501.

6. Matsumoto, Yoshihiro. "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, (September 1984) pp. 502-513.

7. Pollock, Guylaine M. "Evaluating the Feasibility of Software Metric Prototypes Via Reusable Software Parts", Ph.D. dissertation, Texas A&M University, 1985.

8. Reference Manual for the ADA Programming Language, U.S. Department of Defense, Ada Joint Program Office (MIL-STD 1815A), January 1983, p. 10-10 (10.3.9).

9. Sheppard, Sallie, Patricia Friel and Donna Reese. "Simulation in Ada: An Implementation of Two World Views", Simulation in Strongly Typed Languages: Ada, Pascal, Simula, Vol. 13, No. 2 (February 1984) pp. 3-9.

10. Sheppard, Sallie, Usha Chandrasekaran and Karen Murray. "Distributed Simulation Using Ada", Proceedings of the Conference on Distributed Simulation, Vol. 14, No. 3 (January 1985).

11. Taylor, Tamara and Thomas A. Standish. "Initial Thoughts on Rapid Prototyping Techniques", ACM SIGSOFT Software Engineering Notes, Vol. 7, No. 5 (December 1982) pp. 160-166.

12. TeleSoft ADA Compiler, Version 2.1 User's Manual for VAX/VMS (Preliminary draft), ADA-USER-06 (October 30, 1984) Version 1.1.

GUYLAINE M. POLLOCK is a Ph.D. candidate at exas A&M University. In 1979 she joined the Computer Science faculty at Texas A&M University where she is an instructor. Her dissertation research is in rapid prototyping of software metrics. Other projects she has worked on include analysis of the space shuttle software, design of database query languages, compiler design, simulation modeling, and heuristic gaming techniques. Current research interests are high level language design, data abstraction, distributed databases, and software engineering. She was named Outstanding Student of the Year in 1978-79, received the Gulf Oil Foundation Fellowship in 1979-80 and was a recipient of the National IEEE/Computer Society Scholarship for scholarship and organizational activities in 1982-83.

Department of Computer Science
Texas A&M University
College Station, Texas 77843
(409) 845-4306

SALLIE SHEPPARD is an Associate Professor of Computer Science at Texas A&M University where she is director of the Laboratory for Software Research. Her research interests are concurrent high level languages, software engineering and simulation. She is currently principal investigator on a National Science Foundation project which will utilize multiple microprocessors working in parallel. A research grant from IBM has been recently received to investigate the use of Ada in rapid prototyping of software metric systems. She was the Halliburton Professor of Computer Science in 1983-84.

Department of Computer Science
Texas A&M University
College Station, Texas 77843
(409) 845-5466

# Analytical Approach to Software Reusability

Deanna G. Whinery
Ford Aerospace & Communications Corporation

Gary H. Barber
Intermetrics, Incorporated

## Abstract

A recoverability analysis procedure is defined which may be used to systematically evaluate whether existing software is cost effective within a new design. The procedure assumes that there is an inventory of candidate software elements and that the system requirements are well defined and have been allocated to computer program configuration items. The procedure consists of: mapping the requirements onto the candidate software; determining the quality of the existing software through documentation evaluation, maturity analysis, and complexity analysis; determining the modifications required to meet design requirements; evaluating the language to be used; the difficulties of interfacing with Ada; and costing the implementation via the use of an industry standard development cost model and a unique documentation cost model. Finally, the cost is compared with the budget for that configuration item. If the cost is less than budget, the design proceeds, otherwise redesign is required.

## Introduction

There is significant interest today in the cost of software development. Anyone who makes use of software products (that is, nearly the entire population) would like to have them at a lower price. Functionality has increased significantly in the hardware community with a coincident decrease in cost resulting in large productivity increases. But the improvement in software productivity has been very much slower. The software community has made advances, via language development, generating languages with abstractions more capable of representing the problem domain. The advent of Ada promises to improve productivity through the software engineering techniques of abstraction, information hiding, and packaging coupled with a rigidly maintained language definition. But Ada will not overtake the software world overnight. There will be a transition period in which it will be cost effective to reuse existing software. The hardware community has been making use of reusability in chip and board design for some time. Reuseability has equal potential, although greater problems, in the software world. This paper describes an approach to analyze existing software to determine its applicability in a new but similar system. The procedure provides a consistent and documented selection procedure, quality review, and cost comparison to determine whether the potential recovered software would be less costly than new development. The paper also describes a procedure to select an appropriate language for new development and describes how Ada may be interfaced with existing languages.

## Technical Overview

This Recoverability Analysis is composed of six major activities, which are illustrated in the data flow diagram in figure 1. The activities are supported by automatic tools and structured methods (indicated by the larger arrows in figure 1.)

- Activity 1 - REQUIREMENTS MAPPING.

  This activity determines which system level software requirements are at least partially satisfied by existing software processors from the software inventory.

- Activity 2 - REQUIREMENTS ANALYSIS

  This activity applies computer program configuration item (CPCI) level software requirements to the software candidate(s) and determines what modifications are required for recovery of the candidate(s).

- Activity 3 - QUALITY ANALYSIS.

  An evaluation of the complexity, maturity, and existing documentation of recoverable candidate(s) is performed by the quality analysis activity.

- Activity 4 - COTS COST ANALYSIS.

  This activity determines the acquisition cost of a Commercial Off-the-Shelf (COTS) software candidate.

- Activity 5 - MODIFICATION EVALUATION.

  This activity determines what modifications are required for recoverable candidates based on a specific host candidate (target machine). These modifications are consolidated with modifications due to requirements for a total modification evaluation of the recoverable candidate(s) for the target environment. The development cost is determined based on modifications, source code complexity, and documentation upgrades.

- Activity 6 - NEW DEVELOPMENT ANALYSIS.

  This activity determines a new development cost for those CPCI level requirements, which are not met adequately by recoverable software candidate(s).

The recoverability analysis is to be executed for each CPCI and its associated host candidate (target machine) within the current design for the target environment. Before the analysis is initiated, a software inventory of software processors that may become candidates for recovery to the target environment is generated.

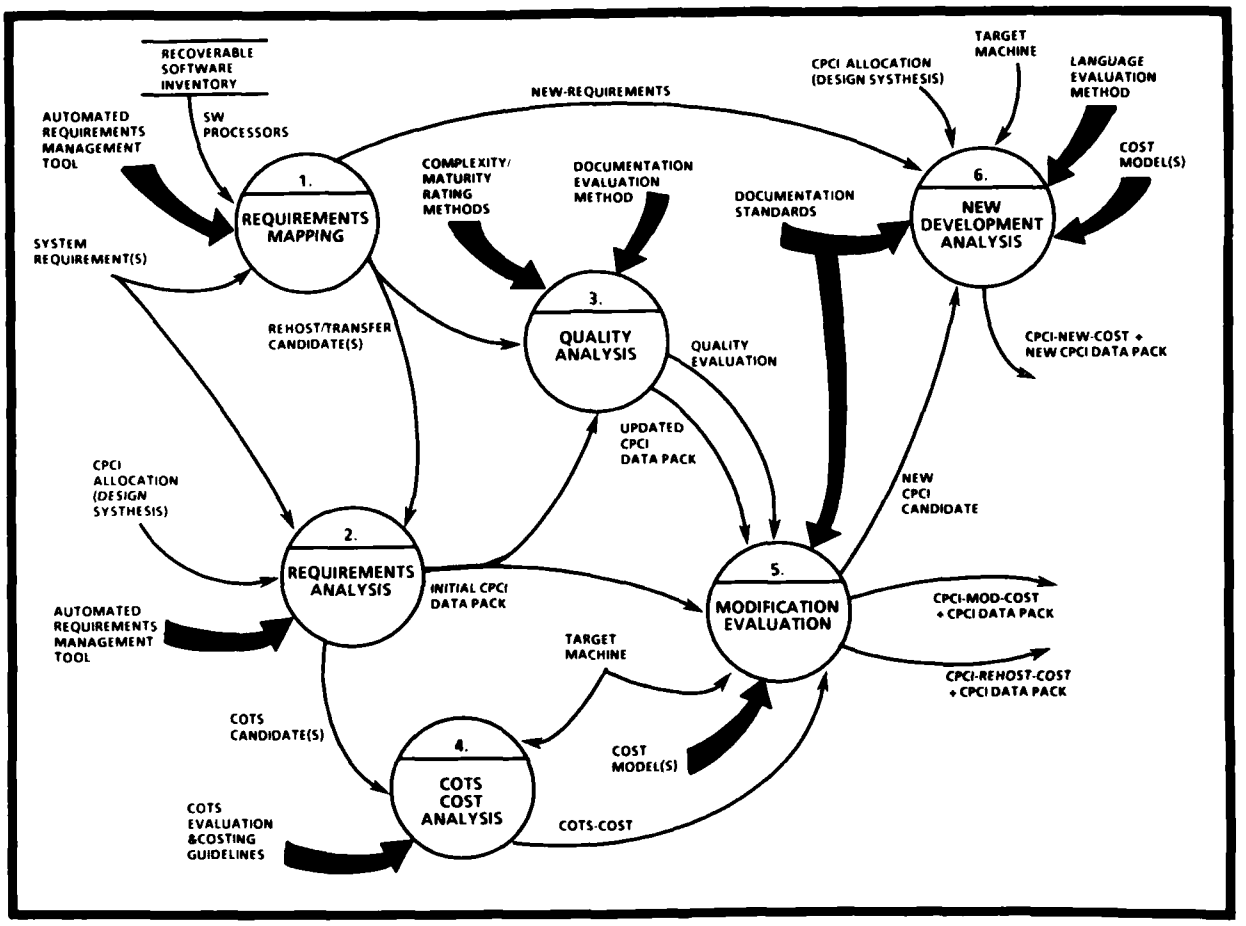



**Figure 1. SOFTWARE RECOVERABILITY ANALYSIS**

This inventory should include application software from similar systems and support software which could be used to assist in the development or maintenance of the application software. This software inventory is the software baseline for the recoverability analysis.

A CPCI Summary Form should be initiated for each CPCI as it starts through the analysis and will be complete at the end of the cycle. This form is used to document information about characteristics of a CPCI that affect its development. The form should provide a field to document the adaptation of recoverable software to a CPCI. Specifically, the percentage of design and code modification required to adapt the software to the target CPCI must be specified. Most of the data required on the data form should be pertinent and necessary information to be determined in order to derive the inputs to the software cost model. Other fields should contain descriptive information about the software candidate and its source and target environments. Several of the data fields will have to be estimated at this point in the CPCI life cycle. However, these estimates will represent the first iteration of the CPCI Summary Form. As the CPCI moves through its life cycle the form should be updated at milestones such as Preliminary Design Review (PDR), Critical Design Review (CDR), Preliminary Qualification Test (PQT), and Final Qualification Test (FQT).

Upon completion of the recoverability analysis, each CPCI will be recommended for development in one of three categories as described in the following paragraphs.

a) Rehosted.

A CPCI for which recoverable software processor(s) exist, and the software processor(s) do not have to be modified to meet functional requirements for the target environment. There may or may not be minor modifications to the code required because of the rehosting of the software processor(s) to an upward compatible target machine. There may or may not be upgrades required of existing documentation. This category also includes the case of recovered software processors that will simply be moved to the same equipment in the target environment with no modifications at all.

b) Modified

A CPCI for which a recoverable software processor(s) exists, and the software processor(s) must be modified to meet functional requirements for the target environment. Thus, some percentage of existing design, coding, and documentation is recoverable to the target environment. The target machine will be selected

c) New Development.

A CPCI for which no recoverable software exists that is acceptable to the target environment. New design, coding, and documentation are required. The language and target machine will be selected.

The associated development cost base through FQT (Final Qualification Test) is also determined for each CPCI.

Commercial-off-the-Shelf Software

Commercial-off-the-shelf (COTS) software is evaluated before it is entered into the software inventory. This evaluation screens out software that is not applicable in the target environment or which comes from unsuitable vendors. After placement in the inventory, a COTS software processor is analyzed just as any other recovery candidate by the Requirements Mapping activity and the Requirements Analysis activity. If a COTS candidate meets requirements, it is sent to the COTS Costing activity where it is evaluated for a specific target machine and where its acquisition cost is determined. Finally, the COTS acquisition cost is sent to the Modification Evaluation Activity for integration into the appropriate CPCI development cost base.

Requirements Mapping

The first activity in the analysis is the mapping of system level requirements of the target environment into software processors from the software inventory. The requirements mapping process is based on the technical judgment of the software engineer and his familiarity with the software in the inventory. The software engineer's mapping decisions must be documented. It is recommended that this documentation process be automated. For example, if an automated requirements management system is being used on the project, a technique for documenting the mapping of existing software processors into the target requirements in the data base could be devised. This technique should provide a method of documenting the percentage of a software requirement that is satisfied by a candidate processor In addition, the technique should provide a method of showing traceability between the target requirements and the software inventory.

The software engineer's mapping decisions are the basis for selection of software candidates from the inventory which will be further analyzed for recovery to the target environment. Any system level requirements not mappable to recovery candidates are sent directly to the New Development Analysis activity.

Requirements Analysis

This activity applies CPCI level software requirements to the rehost/transfer candidates identified in Activity 1 and determines what modifications are required for recovery of these candidates to the target environment. In order to perform this activity, a target architectural concept and its associated requirements allocation must have been determined and entered into a requirements data base. Additionally, the allocated system requirements for the architecture must be decomposed at least to the CPCI level so that requirements are assigned to CPCI's. When the requirements decomposition has been entered into a data base, then the same procedure as recommended in Activity 1 should be used to document the mapping of CPCI requirements to rehost/transfer candidates.

The CPCI level requirements are mapped into the rehost/transfer candidates identified in Activity 1. The mapping process is based on the technical judgment of the software engineer, his familiarity with the software candidates, and his analysis of the CPCI requirements. It is recommended that the same technique discussed in Activity 1 for automated documentation of the software engineer's mapping decisions also be used in this activity to document CPCI level requirements mapping to recovery candidates. When the mapping process is completed, recoverable software candidates will be associated with a target CPCI. The engineer must determine delta requirements between the recovery candidate(s) and the target CPCI(s). The determination of what modifications need to be made to the recovery candidate(s) in order to meet target CPCI requirements is based on the software engineer's technical judgment and analysis of target CPCI requirements which have not been adequately satisfied by recovery candidate(s).

Interface Control Documents (ICD's) are considered requirements and should be assessed in this requirements analysis activity. The engineer must determine whether any modifications to the rehost/transfer candidate(s) are necessary to meet interface requirements in the target environment as defined by the Design Synthesis. Interfaces should at least be defined to the CPCI level in order to make an adequate assessment of a software candidate for a particular target CPCI.

A CPCI summary form is required for each CPCI to be delivered. However, an extremely large CPCI should be decomposed into Computer Program Components (CPC's) of a manageable size (small enough to be developed by a single programming team). Each CPC will be costed separately using a development cost model and the documentation costing model. In this case, a CPCI Summary Form would be generated for each CPC included in a CPCI. Then, at completion of the entire recovery analysis, the CPC forms are integrated into one form for the associated CPCI. The CPC forms are then attached to the CPCI forms for supporting information to the customer. In addition, an input form for the cost model should be initiated for each block of software specified on a CPCI/CPC summary form.

Quality Analysis

The purpose of this activity is to perform an evaluation of the existing documentation, complexity, and maturity of the rehost/transfer candidates, which will be assigned to a target CPCI. This quality analysis activity is composed of three types of evaluations to be performed on recovery candidates (documentation evaluation, complexity evaluation, and maturity evaluation).

Documentation Evaluation

An analysis of the existing documentation of rehost/transfer candidates is performed to determine the quality of the documentation. This evaluation is intended to determine the completeness, consistency, and accuracy (agreement with current source listings) of the documentation. It is recommended that this evaluation be done with a set of questionnaires which are oriented to the following three major types of documentation: requirements; program description; and test. When the evaluation has been completed, the existing documentation quality will be quantified with a number between 0 and 1 representing a "goodness factor". There will be three "goodness factors": one for program description documentation, one for test documentation, and one for requirements documentation. These factors will be used as input to the documentation costing model.

Complexity Evaluation

An analysis of the existing source code of rehost/transfer candidates is performed to determine the complexity of the code.

Software complexity refers to the difficulty to comprehend and thereby maintain/develop a given piece of software. Complexity has several different aspects. It can refer to 1) the complexity of the problem to be solved, 2) the complexity of the algorithms used to solve the problem, 3) the complexity of the program's response to its environment (i.e., real time, multitasking), 4) or the complexity of the logic structure of the code. Since it is clear that complexity has a bearing on the cost to develop or maintain a piece of software, it would be beneficial to find a way to measure this factor in all of its different facets. In the case of software to be newly developed, only the first three aspects apply. None of these are amenable to automated assessment or evaluation by a measurement algorithm. Only guidelines to form boundaries on the subjective judgment of independent evaluators can be provided. These guidelines are typically a part of the input to the software cost model. The fourth aspect of complexity has hope of being automated. This aspect applies only to existing code and would therefore benefit mostly the determination of life/cycle support cost. Control constructs can be measured to produce a metric (e.g., McCabe's metric), or the number of operators and operands in a program can be measured to indicate the level of effort to comprehend it (e.g., Halstead's metric). Both of these measures have been automated, and evaluators are encouraged to use such automated tools for measures of source code complexity.

Maturity Evaluation

The maturity of a software processor is defined as an indicator of the potential cost to maintain a processor's functional performance capabilities. Given that software functional performance errors or failures are a result of unaccommodated environmental stress (input conditions under which the software fails to perform as expected), and given that operation will produce such unaccommodated stress; then software maturity is a measure of the amount of operational stress a software processor has successfully endured.

It is important to obtain a maturity measure for the rehost/transfer candidates during this recoverabiltiy analysis for the following reasons:

- identification of high risk software areas
- life cycle cost implications
- effects on the projected cost to rehost a software processor from the current to the target environment
- test criticality matrix implications

The procedure for obtaining a maturity measure on an existing software processor is to examine the rate history of Discrepancy Report (DR) traffic associated with that processor. The time correlated trend of DR's is compared with the current number of outstanding DR's to arrive at an estimate of the number of DR's per month which can be expected. If the trend is increasing the software is less mature than if the trend was decreasing. In the event that the DR history is not available, the procedure provides an algorithm for determining a stability rating. This stability rating is based on the software engineer's best technical judgment of user satisfaction and assessment of requirements changes.

COTS Cost Analysis

The purpose of this activity is to determine the acquisition cost of a COTS software processor. The preliminary evaluation process is intended to provide a first order filter on the multitudinous COTS software items that exist. The following questions must all be answered in the affirmative to enter the item into the inventory data base.

1. Does the software perform a function which has reasonable likelihood of being required in the target environment?

2. Is it marketed by a stable vendor? That is, has the vendor been in business for at least 5 years? Is there reasonable probability that the vendor will still be in business 5 years from now?

3. Is it hosted on a computer and operating system which is likely to appear in the target environment?

4. Is the user documentation adequate?

5. Does it provide for open-ended growth to include external interfaces, new functions, and increased capacity?

If the software item receives an affirmative answer to all the above questions, it should be entered into the software inventory data base.

The costing is broken into two parts: acquisition and support. In the acquisition area, the costs of licensing, installation, modifications, and documentation should be determined. In the support area, the costs of warranty and subscription should be determined.

Modification Evaluation

The purpose of this activity is to determine the total development cost base for a CPCI based on modifications required of recovery candidate(s) which compose the CPCI.

Target Machine Evaluation

The first step in this activity is to evaluate the recovery candidate(s) which compose a CPCI in terms of a specific host candidate (target machine). This evaluation is based on the technical judgment of the software engineer. For example, the software engineer may need to perform a CPU loading study to validate that performance requirements of the software candidate(s) comprising a CPCI can be met on the target machine. The percent modification information on the CPCI Summary Form will change for some software and must be updated appropriately. The more machine dependent a block of software is, the more likely it will be affected by this target machine evaluation.

Software Development Costing

When the software engineer has determined all modifications required for recovery candidate(s) composing a CPCI, a software cost model is be used to determine the development cost of each block of software identified on a CPCI/CPC Summary Form.

The specific cost model in use should be examined carefully to determine exactly what portions of the development cycle are covered by the model and which are not. For example, if the model does not cover the development of the requirements specification, this effort must be estimated and added to the total cost for each CPCI. A convenient method for doing this is as a percentage of the remainder of the development effort.

If a CPCI was decomposed into CPC's for applying the development cost model, the cost of integrating those CPC's to form the CPCI must be calculated. This CPCI integration cost is then added to the cost of each CPC to arrive at the total software

development cost of the CPCI. It should be noted that the sum of the CPCI development costs is still not the total cost of the software. The final integration and test costs will still have to be added.

Documentation Costing

It is frequently the case that software, which is otherwise a good candidate for reuse, will be poorly documented. Typically the documentation will have neither the minimum information content nor be in the format desired by the end user. The cost of upgrading to a minimum standard must be determined in order to arrive at the total cost of reusing the candidate software. This is complicated by the fact that the software development cost model frequently includes documentation to a standard, which is likely not the same as the minimum standard. It may also be desirable to define alternate costs for multiple documentation standards. The developing organization might have a minimum standard and a proposed standard with stricter format requirements or more information content. A method for arriving at the documentation costs is described in the following paragraphs.

The standards to be costed should be defined by the organization analyzing the project. It is assumed that the developing organization will also have a data base of previous projects done with the various standards used to cost against. A linear regression technique is then used to relate the alternate documentation standards against that used in the development cost model. This will result in an average number of pages per thousand lines of source code for each of the documentation standards. The effort required to produce existing documentation to one of the given standards will consist of a fix portion, (to get minimum information content), a reformat portion, and an upgrade portion (see figure 2). The fix portion is determined using the numbers derived from the documentation quality rating, which used questionnaires oriented toward the minimum standard. The expected size of the minimum documentation is found by multiplying the average page count per thousand lines of code by the candidate source lines. The amount of pages that have to be produced is found by multiplying the quality rating (as a percentage of perfect) by the expected page count. An engineering estimate is then made of the effort required to reformat the documentation. The pages requiring rewrite are found by multiplying the estimate (as a percentage of the effort to write the original documentation) by the expected page count. The upgrade page count is determined by the difference between the expected page count of the upgrade standard and the minimum standard. The page count for each standard can be converted to a cost by using the data base to determine a typical number of man-hours per page.

New Development vs. Recovery

The software engineer must determine whether a CPCI should be considered for new development rather than being recovered. Acquisition cost as well as life cycle cost must be considered. Maintainability of recovered software, its maturity rating, and its complexity are critical factors in life cycle cost considerations. For example, if a recovery candidate is extremely unstable (has a low maturity rating), then it should be considered a high risk piece of software for the target environment. The cause of the instability should be located and the modifications required to "fix" the software should be costed for implementation. This cost may be so large that new development should be considered. Therefore, at this point in the recoverability analysis the software engineer must evaluate recovery versus new development based on all the previous software attributes obtained during the analysis:

- complexity rating
- existing documentation quality rating
- maturity rating
- acquisition cost of the modifications necessary to meet target requirements and to accommodate a specific target machine
- cost to generate maintainable, accurate documentation

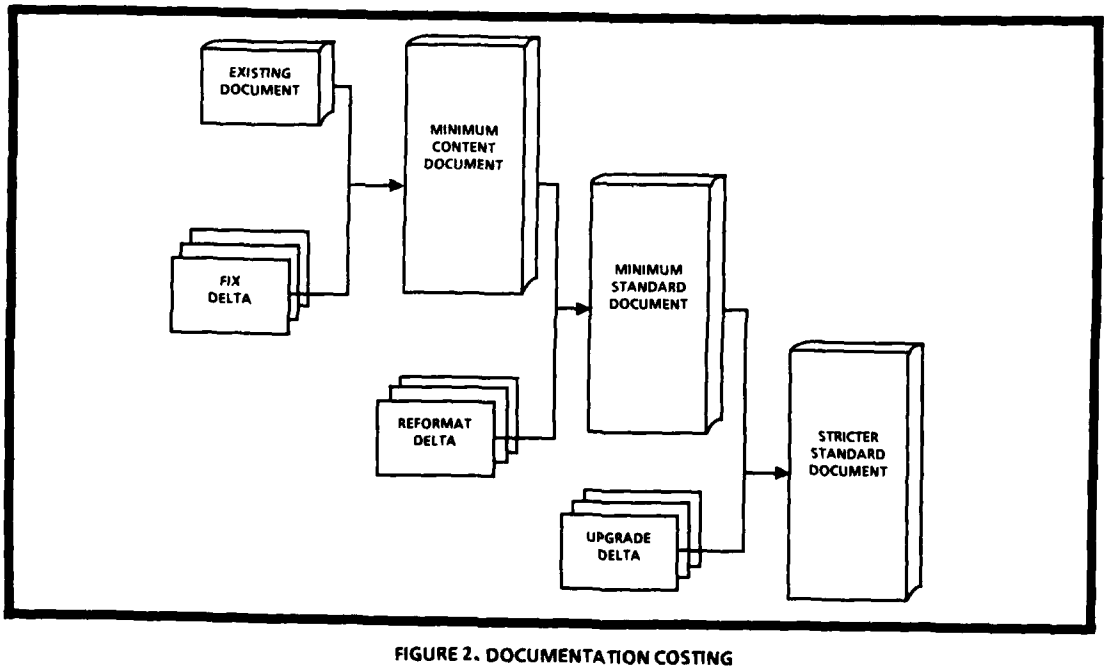



**FIGURE 2. DOCUMENTATION COSTING**

## New Development Analysis

The purpose of this activity is to determine the development cost of performing new software development based on:

1. Target CPCI requirements for which there is no acceptable recovery candidate(s)

2. Target CPCI requirements for which a cost comparison is needed between modification cost of a CPCI composed of recovery candidate(s) and new development cost of the CPCI.

## Language Selection

The software engineer must select a programming language for the CPCI to be developed. The potential languages should be picked with an attempt to minimize the proliferation of languages in the final system. The languages should be evaluated for their technical capability to support the problem environment requirements and for the development tools which are available with the language. Ada is a viable candidate for most new development. However, interfacing Ada with other languages can be difficult. The following paragraphs discuss some of those difficulties.

Ada provides a mechanism for interfacing other language subroutines to Ada, called "pragma interface". This capabaility is intended to allow a programmer to define an Ada subprogram specification in the usual manner, and then specify that the body is not in Ada, but is to be provided in some other (foreign) language. Support of this pragma is entirely at the option of the compiler implementor, who may make any restrictions he wishes.

Passing of data to/from foreign language routines will be easiest via parameters and function results. This conflicts with the common practice of using global data and will likely require major rework of routines which do use global data. Routines that make use of system level calls will need to be reworked also since the services they are requesting will now need to be provided through the underlying KAPSE or run-time kernel. The scope of this rework may make it more attractive to consider converting the foreign language routines to Ada where it is determined that a waiver is not appropriate.

There are several key issues/problems in interfacing Ada to other languages.[1]

- Calling conventions and register usage. The issue with calling conventions is how to pass data back and forth between routines written in other languages, which are likely to have different assumptions about data structure layout and register usage.

- Multiple run-time system maintenance. The multiple run-time system problem involves the fact that each language provides its own mechanisms for exception/error handling, may require preinitialization, and may conflict in basic requests made of the operating system for run-time support

- Input/Output (I/O) Interactions. The I/O interaction problem is actually a subset of the multiple run-time system issue, relating to what happens when both languages want to communicate with the same I/O device

- Linking. The linkage problem involves getting one linker to recognize the linker formats, libraries and external symbolic names for routines in more than one language

The area of interfacing Ada to another language other than assembler is a difficult one. It is important to find the most severe restrictions that can be lived with, to reduce the complexity of the problem, and insure a reliable solution. The first suggested restriction is that Ada be the master program because it makes the most demands on the operating system, and of its own run-time system. The second suggested restriction, is that Ada calls other language routines, but not vice versa. The third suggested restriction, is that only scalars be passed as parameters and return values, if any are passed at all.

## Summary

The final result of this procedure will be a cost for each CPCI within the system making maximum utilization of existing software which may be adapted to fit the new system. This cost may be compared with a budget for each CPCI that could trigger some redesign of the system or a reallocation of the budget. In addition, the procedure has generated information that may be used to estimate the cost to maintain the software after development. This evaluation will also lead to tradeoffs in new development versus recovered software.

## References


1) B. Wilcox, "Pragma Interface Complexities", Internal Intermetrics Memo, September 1984.

2) W. Harrison et.al , "Applying Software Complexity Metrics to Program mainter ance", IEEE Computer, Vol. 15-9, September 1982, pages 65-79.

3) B. Boehm, "Software Engineering Economics", Prentice-Hall, New Jersey, 1981.

Biographies

Deanna G. Whinery

Ms. Whinery, Ford Aerospace & Communications Corporation, Space Information Systems Division, is currently the project engineer for software methodology and analysis technique development in the Advanced Systems Engineering Department. Ms. Whinery received a B.S. degree in Education/Mathematics in 1964 from Oklahoma State University, where she was the holder of a tuition scholarship. Upon graduation, she was elected to the Phi Kappa Phi Honorary Society. While at Ford Aerospace & Communications Corporation, she has worked in the areas of requirements analysis, software design, and software implementation with special emphasis in real-time telemetry processing software. Prior to joining Ford Aerospace & Communications Corporation in 1978, she held a variety of analysis and software development assignments related to the Apollo/Space Shuttle programs at the Johnson Space Center while she was employed by TRW Systems, Lockheed Electronics, and the University of Texas.

Gary H. Barber

Mr. Barber, Intermetrics, Incorporated, has been a program manager in the Simulation and Technology Division since 1984. Mr. Barber received a B.S. in Electrical Engineering from the University of Wisconsin in 1970 and a M.S. in Electrical Engineering from the Air Force Institute of Technology in 1972. Prior to his current position, Mr. Barber was assigned to Control Systems Division of Intermetrics where he managed the development of several products for the oil and gas industry. Before joining Intermetrics in 1978, Mr. Barber served in the U.S. Air Force where he managed the development of software support resources for such programs as the Minuteman, F-4, and F-16. Mr. Barber is a member of the IEEE and ACM. His interests include software engineering and software project management.

# The Marriage of Ada* and an Adaptable Multiprocessor Architecture

Dan Malek and Gary McIntire

Ford Aerospace and Communications Corporation
Space Information Systems Division
Houston, TX

## Abstract

The data processing needs of the government often require custom systems that incur the high cost of design and development. Examination of the custom equipment expected to be needed for NASA's Space Shuttle and Space Station programs leads us to the conclusion that there are indeed similarities in hardware and software functions, and that these functions can be supported by modular, reusable hardware and software. Consequently, we have devised a "building block" approach to creating custom data processors that is flexible enough to meet new and changing requirements, and will provide a means to combat the high costs of the technology race.

## Introduction

General-purpose computers are not well suited for those data processing applications that combine high performance, real-time computing, and sophisticated near-real-time and batch processing. Ground-based spacecraft data processing requirements results in systems that are typical of this situation. Computers that are large and fast enough have simple interfaces and are one-task oriented; those that have sophisticated user interfaces and are adaptable to many tasks don't have enough speed. Custom data processing equipment is then designed to provide the required data processing functions.

If the Government could use commercial off-the-shelf hardware and software in these spacecraft data processing systems, they would not have to bear all the cost of design and development. But computers available in the general marketplace are ill-suited to many custom data processing needs. Therefore, NASA and the DoD often find it is less costly to design and develop a new specialized system to meet performance needs than to implement a system that uses misapplied commercially available equipment.

Rapid technological growth also increases the cost of data processing to the Government, because capabilities of existing ground support systems must change frequently to meet the technological advances in the spacecraft they are supporting. Design and development of these system upgrades are expensive.

Custom data processing systems are, by definition, systems that can't be bought commercially. How are they different? How do the numerous "custom solutions" differ from each other? How are they similar? How can components of one system be "reused" in another system to reduce costs? Can concepts of "reusability by virtue of modularity" be applied to a diverse range of custom systems? These are some of the basic questions we contemplated before designing our system.

Our company has designed and developed most of the custom processing systems in NASA's Mission Control Center. Examination of the custom equipment already in use and expected to be needed for the Space Shuttle and Space Station programs leads us to the conclusion that there are indeed similarities in hardware and software functions. We also believe that these functions can be supported by modular hardware and software which can be reused. Consequently, we have devised a "building block" approach to creating custom data processors that is flexible enough to meet new and changing requirements, and will provide a means to combat the high costs of the technology race.

We have incorporated many state-of-the-art methodologies, along with some

*Ada is a registered trademark of the United States Government (Ada Joint Program Office)

novel techniques, to reduce the costs of custom data processing systems. Although our particular target is ground-based spacecraft data processors, we feel the products of this project apply to many problems which have a combination of sophisticated, flexible needs coupled with high-speed, real-time needs.

## System Design

### General Approach.

Our approach uses multiple VonNeumann microprocessors in tightly coupled and distributed multiprocessing arrangements. A key aim of our concept is to avoid any rigid architecture by using combinations of these two processing arrangements. The software is designed to work in these different arrangements without modification.

VonNeumann vs. Non-VonNeumann. Our research turned up many interesting research and development projects that are not based on the VonNeumann concept of a central processor unit (CPU) executing instructions from its main memory. Although those projects will ultimately lead to new and better computers, the state-of-the-art of such projects currently limits their utility in the immediate future. Consequently, the multiprocessor approach chosen for this project is based on current and near-future "state-of-the-art" hardware and software put together with some interesting twists.

Avoiding A Rigid Architecture. One of the primary reasons that one processing system is more suitable to a given problem than another is that the arrangement of CPU's, memories, I/O, and their interconnecting bus structures more closely maps to the problem than the less suitable machine. Processing systems can't be measured only in terms of million instructions per second (MIPS), memory size, and I/O capacities; the crucial factor of system performance is how closely the architecture fits the problem. Every commercial system known to us has expansion capabilities, but is fixed with respect to the architecture initially designed.

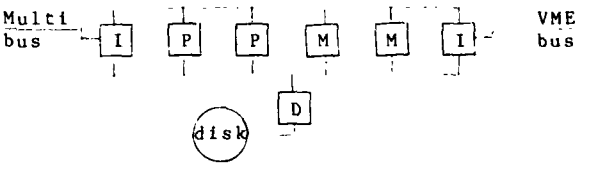
One of our principle design rules has been not to presume a fixed architecture but to support all architectures we can imagine; a few examples are shown in Figure 1. This rule extends to the capability of changing the architecture after a system is installed because its requirements have changed. This rule is supported by hardware with dual bus boards connected by ribbon cable buses, which can literally be changed overnight. This rule is also supported in the operating system by the concepts of virtual memory and virtual I/O. These capabilities provide a significant degree of protection from the costs associated with inadequate, late, or changing requirements.

### Definition of Terms.

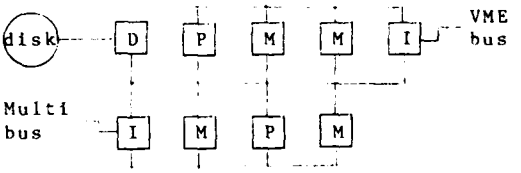
The definition of multiprocessing related terms are not unanimously agreed upon. Although we perceived the following definitions to be generally accepted, other authors may disagree, so accept these definitions only in the context of this project.

Distributed Multiprocessing. A distributed multiprocessor system is an arrangement where each processor has its

Dual bus architecture




Three bus architecture




Two distributed processing centers

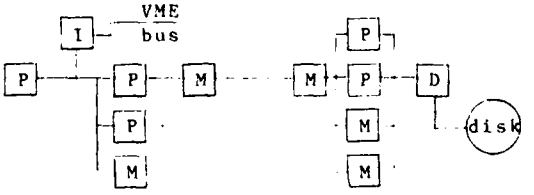



Figure 1. Example Hardware Configurations

own memory with I/O channels to communicate with other processors. Processors need not be alike and do not necessarily run the same type of operating system. There is an a priori assignment of tasks to processors. A typical example is a host processor, with another processor handling disk I/O and another handling terminal I/O. Advantages of this are the functional separation (physical as well as logical) and expandability (since processors have relatively low bandwidth communication requirements). Disadvantages are that the a priori assignment of tasks to processors is time consuming and subject to large oversights, that inefficiences result from processor workload imbalance, and that there is inflexibility with regard to changes (it is difficult or often impossible to shift a task from one processor to another).

Tightly Coupled Multiprocessing. A tightly coupled multiprocessor system is an arrangement in which several processors share a common memory and common I/O channels. Although this can be logically configured like a distributed system with a priori task/processor assignments, the advantages of this arrangement are best realized with a symmetric operating system (described below). The advantages are dynamic task/processor allocation, automatic load balancing, and ease of upgrade. Adding MIPS is simply a matter of plugging in another processor board. The disadvantages are that the common memory and common bus become bottlenecks, limiting the expansion capability, and that processor boards, buses, and memory must be designed with this arrangement in mind. The everyday, off-the-shelf boards will not work in this arrangement.

Network Multiprocessing. Architecturally, a network multiprocessor system is similar to distributed multiprocessing but does not imply fixed task/processor assignment. It also does not imply that a Local Area Network (LAN) is used. Hardware realization may be accomplished with a LAN, but higher performance can be achieved with what is often referred to as "network in a box", where processors communicate over multiple high-speed parallel buses. A Network Operating System (NOS) governs all processors and resources. Advantages are that this system can be expanded almost as easily as a distributed system, that more processors can be added than in the tightly coupled system, and that higher processor utilization efficiencies can be achieved than in a strictly distributed system. Disadvantages are that processor utilization efficiencies are much lower than the tightly coupled system, good load balancing is difficult, and tuning the NOS can be a long and arduous task.

Our system concept currently addresses distributed multiprocessing and tightly coupled multiprocessing. We refer to the tightly coupled arrangement as a "processing center" and allow multiple processing centers within a system. These processing centers interact with one another in a distributed manner.

We have not concentrated our efforts on the NOS because we view the need for distributed and tightly coupled multiprocessing as more immediate. We also view NOS as requiring more effort than our budget allows, and there is another local effort underway that directly addresses NOS and considers our system as part of its processing resources.

MIPS, "Buy em by the yard."

Most computer systems offer memory expansion capabilities. As more users are added to a system, memory boards are added to accommodate the extra memory requirement. This allows the customer to upgrade as needs increase. Eventually, however, the CPU becomes compute-bound. The customer is forced to throw away the old computer and buy the latest, greatest computer he can afford. Often this requires that software be rewritten and users retrained. Being able to add MIPS by plugging in another processor board greatly increases the system's lifetime and reduces costs.

Multiprocessor Operating System.

The functional capabilities of a multiprocessor operating system include:

- o Resource Allocation and Management
- o Table and Data Set Protection
- o Prevention of System Deadlock
- o Abnormal Terminations
- o I/O Load Balancing
- o Processor Load Balancing
- o Reconfiguration

The three basic operating systems for multiprocessors are master/slave, separate executive, and symmetric. In the master/slave system, only one processor executes the operating system and performs the I/O; the "slaves" are assigned work by the master and can spend a large amount of time waiting. With separate executives, each processor has its own operating system and functions like a single processor system with its own resources; a process assigned to run on a processor runs to completion on that processor. Symmetric operating systems are the most powerful and reliable, offer the best utilization of resources, and have inherent load balancing. The operating system "floats" from

one processor to another, and many processors may be executing the operating system at the same time. For this reason, the operating system must be reentrant and make use of mutual exclusion techniques. A process running in a symmetric system may be run at different times by any of the processors. All of the processors may cooperate in the execution of a particular process.

Symmetric Operating Systems. The symmetric operating system was chosen for this project because:

- o Identical microprocessors are used
- o Hardware configuration is flexible
- o Powerful, high performance operating system is possible
- o No old system constraints are imposed

One requirement for a symmetric operating system is to have similar processors. Since identical Motorola MC68010 or MC68000 microprocessor boards are used, the operating system many run on one or many of the processors without any changes. This approach also eliminates the need to design a processor communication protocol that all processors can work with, now or in the future. The responsibility for this design was placed on the NOS group.

There can be many different hardware configurations. The differences may be in the number of processors or location of the processors within the system. Since all of the processors are the same to the symmetric operating system, it doesn't matter if they move around in the system. The symmetric operating system works with any number of processors in the system, whether it is one or many.

Some of the uses of this system demand the performance that only the symmetric operating system can offer. The performance is provided by making best use of the processors. The processors are never waiting for something to do, they are always "out looking for work." Any process may execute on any processor, so the processor looking for something to do simply starts working on the first process that is ready. All of the processors are capable of doing input/output. This capability allows the balancing of the input/output and eliminates any bottlenecks in this area.

Since this is a new system, there is no requirement to provide upward compatibility of software. This is one problem that forces many designers into using other operating system organizations. Starting with a symmetric operating system organization allows future systems to benefit from the performance and flexibility that is in the design from the beginning.

Multiprocessor Ada.

There seem to be as many ways to support an Ada host or target system as there are host and target systems. Several approaches were considered early in the system design, but these quickly dwindled to one best approach. The final design decision was to purchase an Ada programming environment and adapt it to a small multiprocessor kernel operating system developed exclusively for this hardware. A major advantage with this approach is that the system is not dependent on one particular Ada tool set vendor. A new Ada tool set can be quickly adapted to the multiprocessor kernel with minor modifications to the Ada support kernel, without the loss of existing Ada support.

The Multiprocessor Kernel.

The operating system plays a major role in the performance of the overall system. It is always required to do a maximum amount of work with a minimal amount of overhead, while providing a standard user interface. Additional requirements for this system were to support multiprocessors and the chosen Ada kernel with minimal modifications to the Ada kernel software. The UNIX* System V operating system was chosen as the starting point for symmetric multiprocessor kernel for many reasons. This operating system is amiable to multiprocessor modification and is readily available for the target microprocessor. It also provides the very important kernel support to many of the Ada programming environments currently available.

Virtual Input/Output.

When the system hardware configuration changes, the software should not have to change unless I/O devices have been added or removed. For example, if a disk device is in the system and is located adjacent to a processor, moving the disk device so it needs to be accessed through a memory device or another processor should not impact the software. In order to provide this function, the operating system kernel contains system topology information that it collects on initialization or reads from the disk if the system is too complex. Data is transferred between a user and a device through the most efficient virtual channel, similar to a distributed system

*UNIX is a trademark of AT&T Bell Laboratories.

communication channel. The user will not notice any changes, except in the response times due to path lengths.

Design Summary.

The design of this system was based on evaluation of custom real-time systems in use today and expected to be needed in the future. The hardware flexibility is provided by a few hardware devices configured as required by multiple system buses. Ada provides the software flexibility since the packages that exist in libraries can be easily used and new packages can be added as necessary. The high performance of these systems is provided by the use of multiprocessors in both tightly coupled and distributed configurations. Adapting a commercially available Ada system to the hardware was viewed as a difficult task, so a multiprocessor kernel was developed to isolate the Ada programs from the hardware by providing the required software interface.

## Hardware Design

Minimizing the Bottlenecks.

In the processing center, the tightly coupled processors use a common shared memory for storage of both instructions and data. The processors are addressing the memory almost constantly; this shared memory and connecting buses form the limiting bottleneck. Several hardware techniques minimized the impact of these bottlenecks. Solutions included the following:

Cache. Each processor board has an on board 4Kbyte, associative cache. The cache has two major benefits:

1. It allows the processor to run at full speed most of the time.
2. It unloads the shared bus and memory by reducing the number of accesses to main memory.

Split Cycle Synchronous Bus. With common microprocessor buses, a processor places the address on the address lines, holds it while the addressed memory board looks up the data and then transmits that data word to the processor. The processor then relinquishes the bus allowing another processor to use it. The split cycle bus requires the processor to relinquish the bus as soon as it has transmitted the address to the memory board. When the memory board looks up the requested data, it momentarily acquires access to the bus in order to return the data word to the processor. While this is rather complicated for a single processor bus, its benefits come alive in the multiprocessor case. After processor A uses the bus to request data from memory 2, processor B can use the bus to memory 1, processor C to memory 3, etc. The memories return the data in much the same manner. This decouples bus bandwidth from memory access delays. It also allows effective exploitation of concurrency among memory boards thereby allowing two memory boards to appear to have twice the memory bandwidth of one board.

Interleaved Memories. Rather than assigning a processor board to a memory board, as in the above example, it is better to assign even addresses to one board and odd addresses to another board. In this manner, a processor executing a linear segment of code uniformly distributes the load across both memory boards. This is called 2-way memory interleaving. We also allow 4-way memory interleaving, if at least four memory boards exist in the system. A more subtle benefit occurs when a contention arises (two processors trying to access the bus and the same memory at the same time); one processor is forced to wait momentarily. This delay causes it to be momentarily delayed in requesting data from the second memory. If all accesses are to alternate memories, the multiple processors would fall in step behind each other avoiding any more contention delays. In practice this situation occurs for a while but is disturbed when one processor executes a branch and the others do not. Since most code segments are sequential the benefits of this effect occur the majority of the time.

Dual Buses. The principal function of the dual bus boards is to allow the flexible architectures as described earlier. An added benefit is having twice as much bus bandwidth, eliminating more of the bottleneck. The buses can also be interleaved, with even addresses on one bus and odd on the other, with benefits similar to interleaved memories. A second type of 4-way interleave can be done here if the processor board and two memory boards are connected by the same two system buses. Up to sixteen devices, which are processor, memory, disk controller and I/O adapter boards, can be connected on one system bus. There is no limit to the number of system buses.

Processor Board Local Memory. Each processor has up to 64K bytes of on board fast memory to store frequently used code segments redundantly. This offloads the buses and common memory even more. It also creates a deterministic environment necessary for accurate calculations of response times for interrupts. While this seems to

violate the tightly-coupled concept, it actually does not because the local memory mirrors the common memory as long as the corresponding area of common memory is unaltered.

The above techniques are used to enhance the performance of the processing center, allowing more processors to exist in the tightly-coupled environment. How many processors? Our calculations with 12.5MHz MC68000's show that 16 processors would perform at about 80% efficiency yielding an effective value of 12.8 processors, or about 12 MIPS.

MIPS ratings are a source of ambiguity in the literature. Reduced Instruction Set Computers (RISC) can execute more MIPS than Complex Instruction Set Computers but the RISC machine generally takes more instructions to do the same job. The MC68000/MC68010 microprocessor is rated between .8 and 2.5 MIPS by different OEM vendors. The 12.5MHz MC68000 can actually execute 3.1 million register to register adds per second. But for ballpark comparison to 32 bit minis and mainframes, this project assumes the MC68000/MC68010 to be a 1 MIPS machine.

The P-Board and the M-Board.

The hardware building blocks developed are the P-board and the M-board. We often refer to these as the processor board and the memory board, although each board has both processor and memory.

The M-board has .5 Mbyte of access protected, error corrected dynamic RAM that is usually used as common memory to the P-boards. In addition the M-board contains a very fast 29116 processor. This 16 bit processor is organized similar to cascaded bit slices. It is programmed in microcode (70 bits wide) and executes 8 million of these 70 bit instructions per second. It always acts as a parallel coprocessor to the P-boards. Its purpose is to provide the high speed, unsophisiticated processing that is so common with real-time data processing. The processing functions currently assigned to these processors are bit/byte pattern matching, direct memory access (DMA) data moving, semaphore primitives, and M-board access protection set up. As many of these 29116 processors can be operated in parallel as required.

The P-board is basically the processor board but contains memory as described earlier. Though we refer to the processor as the MC68010, the P-boards not requiring demand paging can use the pin compatible MC68000 to gain some performance. An address translation unit (ATU) contains page map information and provides the 24-bit virtual to 25-bit physical mapping. Some of this information in the ATU determines how page faults should be generated and what pages should be cached on this board. The P-board also contains interval timers and a serial communications port. Both the P-board and the M-board provide fault isolating diagnostics on command.

The D-board and the I-board.

The hardware building blocks for I/O are the disk controller and the I/O adapter boards, referred to as the D-board and the I-board. The disk controller board allows SMD-type disks to interface directly to the high speed system buses. The I/O adapter board provides an interface to industry standard Multibus and VMEbus.

The D-board features an intelligent microcontroller architecture based upon the same 29116 processor that is contained on the M-board. This gives it the flexibility to support disk drives with different densities and data rates and to implement custom programming for unique job requirements. This board can support up to sixteen disk drives and be can commanded from up to sixteen different sources (P-boards). Multiple commands to be completed in sequence can be linked together. This board also contains 128Kbytes of dynamic RAM that can be used for data buffering.

The I-board is used to interface to commercially available devices that utilize industry standard buses. The I-board basically provides a one-to-one mapping between the system bus and the standard bus. This board contains an ATU like the P-board, that can be loaded externally (by the operating system) and provides logical addressing of the system by the devices on the standard bus. This is especially useful for devices with limited 16- or 20-bit addressing capability. The devices on the standard bus appear in some physical address range on the system buses. The I-board also provides interrupt management from the devices on the standard bus that allows the balancing and directing of interrupts among the P-boards. One I-board can only support one type of standard bus, but multiple I-boards each supporting a different standard bus can be used in the system without difficulty. Standard bus bandwidth problems can be eliminated by using multiple I-boards. The I-board supports bus and memory interleaving like the P-board to eliminate system bus contention problems.

## Software Design

### Ada "Building Blocks".

Ada packages provide the software building blocks for the system. Since it is easy to use existing Ada packages, new systems can be more quickly developed because there is much less software to implement and test. The operating system kernel was designed with the Ada package concept, but the commercially available systems provided an absolute minimal kernel most of which was not written in Ada. Since research and development support for the kernel did not exist as a vendor option, it was decided to develop a kernel that could maximize the performance of the hardware and provide the user interface the Ada programs needed. The operating system kernel was then developed from the existing UNIX System V/68 operating system. With this approach, a commercially available system could be easily installed on this kernel, and any system could be supported that generated code for this target processor.

The software building blocks that exist today are a single processor kernel, a multiprocessor kernel, and Ada application packages. This still supports our building block concept very well, since the operating system kernels do not require modification and the Ada application packages change to meet the needs of the new processing system. Since the latest compiler purchased supports all features of the Ada language, the operating system kernel is scheduled to be implemented as Ada packages in the near future.

### The Multiprocessor Kernel.

The operating system kernel was designed with three major goals in mind. One goal was to support an Ada embedded application or host development environment. A second goal was to develop a symmetrical operating system to efficiently manage the tightly coupled hardware and to provide distributed capabilities between processing centers. The third goal was to develop an operating system that could be used on radically different hardware configurations by changing only the configuration information tables in the kernel.

Supporting the commercial Ada systems requires standard kernel interfaces and any utilities (editor, linker/loader, etc.) that the vendor does not supply. Some kernel data structures and processing policies were modified to provide better supoort for multiprocessor Ada.

The symmetric operating system is assisted by several hardware features that provide high performance and ease of implementation. The 29116 processor manages the Dijkstra-style semaphores and guarantees atomic operations by the processors. The I-boards were designed to manage external interrupts for the entire multiprocessor system, which greatly simplified interrupt handling. The disk cache buffers are mapped to the RAM on the D-board. This eliminates DMA data transfers across the system bus between the disk and system buffers.

The operating system kernel makes use of other hardware features to improve system performance. The virtual address mapping capability provides a means of implementing interprocess communication with full protection. The mapping capability also eliminates many data copy operations by being able to "just shuffle the pages" in a logical address space. This is very helpful in the network interface functions because data messages can be reassembled from their packets (that don't always arrive in the correct order) very easily without the overhead of copying or buffering. The virtual mapping makes the operating system immune to changing hardware configurations by making the physical devices appear in the same virtual address space regardless of the system configuration. Virtual mapping also provides the capability to implement demand paging, although this was not the primary design goal.

Just as the virtual address mapping provides the capability to handle different hardware configurations at the P-, M-, D-, and I-board level, the virtual I/O provides distributed processing centers the capability to handle different I/O configurations. The virtual I/O concept may be best explained by using the example of the two distributed processing centers in Figure 1. Suppose that hardware configuration exists because the processing center on the left is heavily involved in pre- and post-processing of data transferred across the several high-speed parallel interfaces on the industry standard bus. The system on the right is involved with the storage and retrieval of this high rate data as well as program storage. The virtual I/O subsystem exists in much the same way as a LAN exists between two workstations, except that the virtual I/O subsystem is simplified in its commands and data transfers. The system on the left knows that a disk is available through one M-board to M-board link and the system on the right knows that a serial terminal port exists through a same link. A user could log into the system on the left and use the virtual disk capability, or log into the system on the right using

the virtual serial communication capability. In either case, the system depends on the DMA capability of the 29116 processor on the M-boards and the distributed system configuration information (system topology and virtual device paths) in both systems.

Multiprocessor Ada.

Installation of an Ada run-time support kernel on the multiprocessor operating system kernel usually requires no modification of the Ada kernel. This environment does not make very efficient use of the multiprocessor capability since all of the Ada task and memory management is performed by the run-time kernel. If the data processing is an embedded application consisting of only one run-time kernel, an increase in performance can't be gained in the multiprocessor environment since the Ada run-time kernel is perceived as one process by the operating system.

To provide true parallel tasking and to take advantage of the multiprocessor performance, the Ada run-time kernel is modified. These modifications include the moving of the inter-task communication, task management, and memory management functions to the multiprocessor operating system kernel. Since the multiprocessor kernel already provides these functions, the modifications to the Ada run-time kernel are to map Ada system calls to the appropriate multiprocessor kernel system calls. Any information contained in the Ada run-time kernel required to support Ada tasking that is not contained in the multiprocessor kernel, was moved into the multiprocessor kernel for protection and ease of access. The multiprocessor kernel functions contain minor modifications to best support Ada tasking. Other functions of the Ada run-time kernel are not modified, unless some of these functions required the use of information moved into the multiprocessor kernel. This information is made available to these functions by standard multiprocessor kernel system calls.

Future Research.

We are currently studying the approach to implement Ada tasking with the use of networks, and a network operating system is beginning to emerge as a system goal. A multiprocessor kernel supporting a standard Ada interface is planned when the specifications become available. An interface to the system bus is being designed to handle high speed telemetry data streams. System performance is increased whenever possible by using the latest hardware and software technology feasible.

References.

1. Enslow, Philip H., Jr., "Multiprocessor Organization - A Survey", CACM, Vol. 9, No. 1, March 1977, pp. 103-129.

G. McIntire and D. Malek, "Advanced Microprocessor-Based System Design", Ford Aerospace and Communications Corp. R&D Report, 1984.

Dan Malek, Senior Software Engineer, has previously worked on software development for the Shuttle Downlink Telemetry Preprocessing Computer Complex. He holds BA degrees in Computer Science and also in Chemistry from the University of Northern Iowa. His mailing address is:

Ford Aerospace and Communications Corp.
P.O. Box 58487 M/S M3B
Houston, TX 77258

Gary McIntire, Senior R&D Engineer, has previously worked with several microprocessor projects in support of the Space Shuttle Telemetry Downlink. He holds a BS in Electrical Engineering from the Lamar University. His mailing address is:

Ford Aerospace and Communciations Corp.
P.O. Box 58487 MS4B
Houston, TX 77258

# VHSIC HARDWARE DESCRIPTION SYSTEM OVERVIEW

Alfred S. Gilman

Intermetrics, Inc., Bethesda, Marvland

**Summary:**

The Department of Defense's Very High Speed Integrated Circuits (VHSIC) Program is sponsoring the design of a hardware description language and the development of some tools to support its use. The purpose of this language, the VHSIC Hardware Description Language (VHDL), is to become the standard medium of exchange of design data for VLSI devices used in defense electronics and for systems/subsystems employing these devices. This paper provides a brief introduction to the language and tools with particular emphasis on the applications of Ada technology in the VHDL development and the potential for synergy between the support environments for Ada and VHDL.

**Background:**

Early in the VHSIC program it was decided that a standard medium of expression was needed to definitize and communicate the massive amounts of design data associated with designs of VHSIC complexity. Therefore the DoD defined language requirements for the VHSIC Hardware Description Language (VHDL) and initiated a two-phased procurement of VHDL and its support environment. One requirement stated that VHDL should make use of the Ada* language whenever the necessary constructs were present in Ada.

The team of Intermetrics, IBM, and Texas Instruments was awarded the contract to design VHDL and to implement the VHDL support environment software. Intermetrics is the prime contractor for this program, with particular responsibility for designing the language, and establishing the VHDL support environment architecture. Intermetrics is also implementing the VHDL analyzer which reduces the VHDL source text to the intermediate form from which the simulator runs. IBM is directing the language requirements analysis. They are also developing usage scenarios and coding and simulating five "benchmark" descriptions in VHDL. Texas Instruments has contributed to the language requirements and the system architecture. They are implementing the VHDL simulator. TI is also coding and simulating three benchmark designs. Team-wide review is used on critical decisions.

The VHDL design effort started on July 31, 1983. The design was completed on July 31, 1984 and was reviewed by a group of experts from DoD organizations, universities and private industry during August and September. Based on the favorable response from the reviewers, the DoD has authorized the team to proceed to implement the VHDL. An initial capability to use VHDL will be created by support software deliveries scheduled for November 30, 1985.

**The Language:**

The VHDL incorporates some important modern language concepts [1]. These include the ideas of independent semantics, hierarchical description, and configuration management. Independent semantics means that the language is fully specified as to its semantics, independently from the tools that support it. The hardware description is complete and definitive from the text written in VHDL and the definition of the VHDL as a language [2]. Hierarchical description is an integral part of the overall organization of VHDL. This is evident in the modularity of design afforded by the design units of the language (analogous to compilation units in Ada). It is also evident in the breadth of levels of abstraction supported by the language by the inclusion of strong, user-defined types and the exclusion from the language of any presumption as to primitive design units. Good support for hierarchical description is an absolute necessity when dealing with designs of VHSIC complexity. The modularity of the language, together with features in the language and support tools supporting the tracking of variants and versions of designs lay a firm foundation for the implementation of configuration management of designs described in VHDL.

---

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

**Design decomposition:** The primary language abstraction for representation of a hardware component is the design entity [3]. A design entity is composed of an interface and one or more alternative bodies. The interface of an entity defines all of that entity's external characteristics. Each body represents an alternative description of the entity. This degree of freedom -- the simultaneous existence of alternative bodies -- allows the language to capture related high-level and low-level descriptions of the same entity. It may also be used for competing design approaches to realizing the same entity. All bodies must conform, however, to the characteristics established in the interface.

The design entity interface contains not only its externally visible characteristics, such as ports and generic parameters, but also standard items applicable to all bodies of that entity. Ports define the channels of real-time communication between the entity and hardware outside itself. Design entities may be defined to be parametric or generic by the inclusion of generic parameters in their interface. The description of the entity adapts to the values supplied for these parameters.

A body of a design entity completes the description of that entity, working within the constraints laid down in the entity interface. Aside from declarations and specifications, a body is built up out of both concurrent and sequential statements. The execution of a VHDL description is performed by the execution of one or more concurrent statements. Sequential statements serve to define a single concurrent statement by a sequence of steps to be performed. The sequential statements within a single concurrent statement are executed in the order in which they appear. Concurrent statements, on the other hand, execute independently with respect to other concurrent statements.

A VHDL design entity is a template to be used in creating specific instances of a component. The component instantiation statement creates these instances, identifying the actual signals attached to its ports and the actual generic parameters for that particular instance.

**Time-Based Execution:** Time is one of the most important aspects of an HDL, because the timing characteristics of hardware are difficult to represent in a textual description. There is massive parallelism that can exist in a description of hardware. The language must express the dynamic behavior of one hardware component over time, and it must also capture the correct temporal relationships of all the interactions among the components involved in the design.

There are two time scales in the execution model for VHDL: a macro-time scale and a micro-time scale. The macro-time scale is a quantitative representation of physical time as experienced by the hardware being modeled. The micro-time scale is in effect a unit-delay mode and is not measurable in terms of macro-time; there may be an arbitrary number of micro-units of time within one macro-units of time. Thus the micro-time scale allows the user to define the time order of actions performed by concurrent statements without having to quantify the precise delays.

A VHDL description of a component specifies an instantaneous mapping from the past and present values of its inputs to the future values of its outputs. That is, given a set of inputs, a VHDL description predicts the expected outputs at future points in time. As time advances, these projected outputs become actual outputs at which point they are propagated along data paths to become inputs to other components.

Evaluation of a VHDL description is event-driven. That is, the description of a component is evaluated when an event occurs at one of its inputs. The result of evaluation is a new set of projected values for the outputs of the component. This stimulus/response approach to computation is a natural way to describe the behavior of hardware in the digital system to logic gate range.

**Features of the Language:** In order to support the wide range of descriptive capabilities required to model hardware in the logic gate to digital systems range, VHDL incorporates a number of useful features, including user-defined data types, signals as well as variables, attributes, assertions, regular structures and packages.

A type is a collection of values and a set of operations on those values. Like Ada, VHDL provides the capability for the designer to define the data types he needs. VHDL also allows the user to define types with units, called physical types. Because of the intended use of VHDL as a definitive description of hardware, the precision of computation with discrete types is fully defined in the language without dependency on host arithmetic types. Arbitrary precision is supported for discrete types, which includes all physical types. Thus the semantics of arithmetic in VHDL is more portable than the semantics of arithmetic in Ada.

Signals provide for information flow between concurrent processes in VHDL. A signal has a history and a future: historic values of the signal are visible but unalterable; future values are alterable but invisible. Assignment to a signal may change the projected (future) values of the signal, but not the current or past values of the signal. A signal may be assigned a single value, delayed in time; the time specification is the time relative to 'now' at which the new value is to take effect for that signal. A signal may also be assigned a waveform, which is expressed as a sequence of values and associated times.

Signals that are driven by multiple sources are called buses. In order to resolve the values supplied by multiple sources into a single value for the bus, VHDL allows the designer to specify a bus resolution mechanism to be associated with each bus in a description.

VHDL also provides variables for use in abstract computations. Variables have no relationship to time, so they have only a single value that is both visible and alterable. Assignment to a variable changes its value immediately, and therefore variables can be used in algorithmic descriptions. While variables may be either static or dynamic, they are not used for inter-process communication.

One of the goals of VHDL is that all of the information contained in a data sheet for a component be expressible in VHDL. VHDL provides a general-purpose mechanism that allows the designer to define attributes of objects. This allows the designer to "decorate" a description with extra information about a component or its parts.

Assertions allow a designer to specify conditions that are expected to be true in the course of executing a design. They allow the designer to specify information about the intent of a design so that errors can be detected close to their source. An assertion definition consists of a boolean expression that specifies the condition being asserted, optionally followed by a severity level and an error message.

A regular structure is a pattern that is repeated many times. VLSI designs often contain regular structures. In order to facilitate description of such structures, the language includes the generate statement. This statement functions like a macro expansion capability within a body. Often, the boundaries of a regular structure exhibit slightly different connection patterns than the rest of the structure. The conditional generate statement is used to described this variation in structure.

A package contains a group of declarations that are related in some way. These include type and subtype declarations, attribute declarations and specifications, and function declarations. Packages do not declare signals or variables. This exclusion enforces the requirement that communication channels between components be explicitly declared as ports in the design entity interface. Once a package is defined, it may be referenced by other descriptions to share its declarations.

Packages are a convenient way to encapsulate all the declarations relating to some abstraction. For example, the contractors developing submicrometer technology under the VHSIC program will be describing and simulating their device designs using VHDL. These three contractors must also collaborate on defining and following interoperability standards for their designs so that the resulting devices may be used in combination without regard to the originating company. The interoperability standards for these chips are an example of an abstraction which one would want to express in one or more VHDL packages.

**The Tools:**

Part of the definition effort in Phase A of the VHDL program has been to define a support environment for the language. This support environment is both an open-ended integration framework for what is hoped to be a growing list of tools organized around VHDL, and a specific set of software which is being implemented in Phase B of the program.

Recall that the mission of the VHSIC Hardware Description Language is to support insertion of VHSIC technology in military systems and to facilitate hardware design. To carry out this mission, the language user needs tool support so that he can record and communicate digital designs, and verify digital designs by simulation. In order for VHDL to serve as the standard interface to a growing spectrum of design automation tools, the kernel support software implemented initially will construct and make available for interfacing to other tools an intermediate form of hardware description derived from the VHDL description written by the user.

The VHDL support software being implemented in the current phase (B) of the VHDL program consists of five tools and a design library. Four of the tools manipulate design descriptions. These tools are an analyzer, a reverse analyzer, a profiler and a simulator. The design library is a collection of data representing hardware descriptions. The fifth tool is the Design Library Manager which the other tools use to manage and access the common data in the Design Library. The Analyzer checks hardware descriptions for static errors, i.e. those that are evident without simulating the passage of time. It also translates VHDL text to intermediate form and places this in the design library. The reverse analyzer produces VHDL text descriptions from intermediate form descriptions in the Design Library. The Profiler collects modules of hardware description to build a unified model of a hardware design. The Profiler reduces the layers of hierarchy in the model representation in the interest of simulation efficiency. The Simulator computes what would happen as the modeled hardware executes and detects dynamic violations of the language semantics and user assertions. It does this by transforming the model representation into an Ada program which computes the dynamic behavior of the whole model. The Design Library consists of the intermediate form of already-analyzed hardware descriptions and is supported by the library access and management routines of the Design Library Manager.

The Analyzer functions can be divided into syntactic and semantic analyses. The Intermediate Form is constructed in the course of these two analyses. The core structure of the Intermediate form is an Abstract Syntax Tree (AST). This is a tree-form representation of the syntactic structure of the VHDL description being analyzed, using an abstraction of the full language syntax. The final Intermediate Form consists of this tree structure with semantic attributes and relations added as decorations.

The VHDL language gives the user the capability to associate both high-level functional descriptions and more detailed, decomposed descriptions with the same entity being designed. Design entities can also be incorporated in other design descriptions as components. This gives a very flexible hierarchical description capability. Using the modular, hierarchical form of description with embedded variant descriptions at various levels, the user can construct from the same library of descriptive modules a wide variety of simulatable models -- at various levels and of mixed levels. This very high level of programming in terms of design entities is accomplished with a language capability called the configuration body and a tool called the profiler. The configuration body is analyzed into the design library by the analyzer as with any other VHDL design unit. Operating from this information, the profiler creates a unified description of the aggregate model.

The purpose of the Simulator is to compute the successive signal values which occur as a result of the time-dependent behavior of a particular hardware entity, as described in a VHDL model of that hardware.

The simulator draws the definition of this hardware Unit Under Test (UUT) from the Intermediate Form representation in the Design Library. This hardware model is exercised in the context of a simulated test bench environment of signal sources and receptors, and a controller. This test bench equipment is predefined as part of the Simulator. The user can also add pieces of user-defined test equipment by describing these in VHDL and invoking them as components in a VHDL description of the total test bench. The Simulator will be capable of simulating any hardware behavior within the scope of the dynamic semantics defined in the language specification.

The design library in essence defines the integration framework of the VHDL support environment. All the tools communicate through the data in the design library. The design library houses the intermediate form representation of VHDL design units analyzed by the analyzer. The Design Library will also contain the Intermediate Form of the flattened structure constructed for a model by the Profiler. The design library manager functions include detailed representation access functions organized around the abstract syntax tree representation internal representation of VHDL, library-organization functions organized around a network file-system model, and a few other utilities.

**Applications of Ada Technology:**

The support tools for VHDL being developed by the VHDL team are to be implemented in Ada. The chief reason for this is for the portability it affords. Moving the tools to a new host with a full Ada implementation is relatively easy. To demonstrate this portability, the VHDL support environment will be delivered on two hosts, both VAX/VMS and an IBM 370-class machine with MVS.

The Design Library Manager is the collection of library control and access services which all tools use in order to have a correct and high-level interface to the Design Library. The Ada package concept provides a natural and effective mechanism for collecting, isolating and sharing this common code.

The separate specification and body facility of Ada is being used to structure the top-down design of the support environment software. The principal medium of design is the source library itself. Essential facts for the program maintenance manuals are extracted from the source files by the Byron(tm) system.

The portable virtual file system employed in the Design Library is drawn from the CAIS, or Common APSE Interface Set [4]. This is a virtual operating system interface proposed as a standard tool interface for Ada programming environments. This model provides an organization which is effective for the needs of the current tools and flexible enough to allow the Design Library to grow as tools are added.

The detailed internal representation of VHDL design units employed in the Design Library is, as described above, a decorated abstract syntax tree. Thus the VHDL intermediate form bears the same relationship to VHDL as Diana bears to Ada. In fact, a common language, the Interface Description Language (IDL) is used to formally specify the data model [5] employed in building both intermediate forms

**Unified support:**

Use of the CAIS and IDL technologies developed initially for Ada programming support environments has been effective in the design of the support environment for VHDL. Thus the core of common details required in a tool-integration framework for Design Automation tools gathered around VHDL and for Ada programming tools can be the same. Both languages are targeted to the same application, intelligent Defense electronics. The systems of the future require close coordination between the design of the hardware and software for a system, including the capability to trade

hardware and software capabilities late in the system development cycle. Both the user requirements and the software interfacing resources thus point toward a common standard tool integration framework for tools addressing hardware design (expressed in VHDL) and software design (expressed in Ada).

It has not proven possible to have VHDL be just a collection of Ada packages and still meet the fundamental hardware description requirements. Still the application of Ada technology in the VHDL program has been fruitful, and the prospects for a common tool integration framework spanning hardware and software design automation appear good.

**Acknowledgement:** The information in this article is the result of work by many people throughout the team. Any inaccuracies are the author's but the contribution is theirs.

**References:**

1. Lt. Al Dewey, The VHSIC Hardware Description Language, **VLSI Design**, November 1984, pp. 33-39.

2. VHDL Language Reference Manual, Version 5.0, Intermetrics, Inc., July 30, 1984.

3. Moe Shahdad, et. al., VHSIC Hardware Description Language, **IEEE Computer**, March 1985

4. Common APSE Interface Set, Version 1.1, KIT/KITIA CAIS Working Group for the Ada Joint Program Office, Sept. 30, 1983.

5. Interface Description Language Formal Description, Draft revision 2.0, Carnegie-Mellon University, Computer Science Department, June 1982.

**The Author:** Alfred S. Gilman received his D. Sc. in Control Systems from Washington University, St. Louis MO. Over the last ten years at Intermetrics he has worked on a variety of advanced avionic systems and the computer aided engineering environments for developing these systems.

# SOFTWARE QUALITY ASSURANCE AND ADA

BRUCE BROCKA

U.S. Army Management Engineering Training Activity
Rock Island, IL. 61299-7040

Abstract

Ada was designed to be life cycle oriented; one outgrowth of life cycle driven software development should be the ready application of software quality methods and metrics. Current software quality measures may be applied to Ada, in concert with the program management environment. This paper discusses the application of software quality methodology to Ada and is intended to stimulate how quality improvement is intrinsic using Ada.

## I. INTRODUCTION

Ada was designed with software engineering principles and the software life cycle clearly in mind. Because of this, software quality assurance techniques are more readily applied to Ada than to other languages, particularly those languages available for real time systems. Ada is unique in its conception in that it is applicable to more than one life cycle phase, and the Ada Program Support Environment (APSE) was designed to assist the program development process.

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

The life cycle may be divided into the following five phases:

- o Requirements Definition
- o Design
- o Code
- o Test and Integration
- o Operation and Maintenance

Although hardware reliability concepts have been around since the 1950's, such ideas as mean time between failure or sample testing have no meaning when applied to a system of intangible thoughts or instructions. Further, reliability typically applies to the design phase, quality assurance to the production phase. Thus the distinction between reliability and quality assurance becomes blurred in practice when dealing with software. This paper explores ways in which Ada can be used throughout the life cycle, and how Ada can be used for quality assurance activities which last throughout the life cycle, such as configuration management and verification and validation.

The existing software environment is rapidly growing in terms of lines of code per program and embedded software applications; weapons systems are becoming software driven. This leads to a sharply increased need for better software management, particularly in the areas of:

- o configuration control
- o contractual specifications
- o integration into hardware
- o productivity

## II. EXISTING QUALITY AND RELIABILITY TECHNIQUES

This section examines how existing quality assurance techniques may interact when applied to Ada. We examine the techniques

first by each life cycle and then by activities that encompass the entire life cycle.

Part of the overall problem with software quality assurance is that each software project contains many components, each of which may be unique to that particular application. This leads to difficulty in measuring or assessing the software and thus determining product quality. Further reason why a large portion of traditional quality assurance techniques must be abandoned is the lack of a production phase; a phase in which traditional hardware quality assurance techniques play a major role in determining product performance.

Requirements Phase

The goal of the requirements phase is to produce a set of software specifications, which will provide sufficient detail to be fully testable, yet the requirements should not design the system.

Ada assists in this phase by providing a common language, and a way to document the software using the program unit specifications. Using this method should increase the traceability of the requirements throughout the rest of the development phase.

Traditional methods for development of requirements are not obviated by the introduction of the Ada discipline. User documentation quality must still be developed and evaluated using structured methods that interface well into the Ada context.

Writing compact, testable, and easy to design from specifications may be achieved by using Ada and structured methodologies, as Ada allows for a structured format.

Design Phase

This phase produces the detailed design from which the programmers will produce the code. Obviously, if this phase is done well, the coding effort becomes relatively simple. The design phase, however, typically undergoes several iterations before the programmer can begin work.

Ada may be directly used in this phase as a program design language (PDL). There are several advantages to this:

- Program documentation is being provided while designing the program, allowing for better configuration control.
- The PDL can virtually be the program code, thereby almost eliminating the coding phase.
- The integration testing phase can be provided for by means of designing stubs and/or drivers. (This aspect is not unique to Ada.)
- Traceability is increased by naming conventions appearing at this stage.

Ada as a PDL can improve the overall quality due to the early documentation generated, and a consistency of style, usage and concepts at an earlier stage. Using Ada (or any language) as a program design language is not a panacea; persistent problem areas still exist:

- Confusion between "structured english" and "pseudo-code". Since the design process requires several iterations, it seems logical to proceed from a more abstract, high level description (i.e., "structured english") to a more code like description (i.e., "pseudo-code").
- Design is constrained by syntax details.
- What portions of the language to emphasize (e.g., interface-definitions, modularity, tasking, etc.).

- o Requiring a code like description of everything. It may be advantageous to describe "traffic control" modules differently from modules that perform a very specific function, and are readily described by HIPO charts, or some other tool.

So in using Ada as a PDL the disadvantages are the same as those that would be encountered in any other language.

Other techniques that may be applied at this phase are such structured methods as structure charts which aid measures of cohesion and coupling, use of Nassi-Schniederman charts, Warnier-Orr charts, Jackson techniques and interface documentation.

Coding Phase

There are advantages to coding in Ada even if the target language is not Ada. Ada is highly structured, and it is fairly easy to proceed from a structured language to an unstructured one. Eventually, automated tools should assist (if not completely convert) the code from one language to another. On the other hand, it is very difficult to convert from an unstructured language to a structured one, particularly if one wants to take advantage of such features as parallel tasking. The advantages of Ada over traditional declarative languages are, in terms of quality assurance:

- o Modularity which is conducive to readily readable, testable and maintainable code. With program modules, however, interface documentation becomes more crucial.

- o Parallel tasks can provide for a fast run time environment increasing the real time capability of the program.

- o Type and variable declarations must appear at the beginning of the program and the program units. This forces the programmer to put all type and variable references in an easily locatable spot. With care in the program design, global variables can be reduced to minimum, further increasing each modules coherence and reduce coupling. Ada also does not limit identifiers to 8 characters, so identifiers may be given a comprehensible name.

- o Exception handling is a very important feature in real time systems which rely on data from input sensors, particularly sensors prone to failure or damage.

- o Separate compilation may be used to test the program in units, thus eliminating or reducing the need to create stubs or drivers.

Coding walkthroughs, reviews and inspections have proven to be a very valuable means of detecting errors that occur during the coding phase.

Other actions that are appropriate at this point are configuration control, and the establishment of a program library.

Testing Phase

Testing may be applied in three phases: unit testing, integration testing, and operational or acceptance testing. The first two phases would correspond to the traditional military concept of development test and evaluation, the latter to operational test and evaluation.

Testing planning is made easier by the prior documentation. Interfaces should be known, and glass box testing performed on critical modules should have readily definable inputs and outputs. Testing order can be determined from the structure charts or diagrams. Care should be taken to allow time to adequately test the software, and that each unit be tested in some measure before any integration testing is made. If the testing phase is not initiated until all units are coded and integrated, disaster is likely to result.

One of the major causes for cost overruns in software is the failure to detect requirements errors prior to release of the software. Since Ada was used as a backdrop throughout the development life cycle, requirements should be readily traceable from the requirements phaser through to the testing phase. Because of this, requirements errors should be detected at the unit testing or integration testing phase. Methods for unit testing include:

- o Path Analysis
- o Cause Effect Graphing
- o Boundary Value Analysis
- o Equivalence Partitioning

Integration of the units may be achieved in one of three basic wasy: top-down, bottom-up, and the "big-bang". In the first two methods, either stubs or drivers must be created, respectively, but the pay-off is high in that errors are easier to locate, and testing procedes in a logic and comprehensible fashion. In the big-bang method, the entire program is tested as a whole. The disadvantages of this are obvious.

Operation and Maintenance Phase

Ada is well suited to be modified because of the following:

- o Modularity as mentioned earlier, code that is written in a small module (somewhere between 60-200 lines of code) with few global variables is easy to maintain, and new modules may be readily added.
- o Consistency in types and variables since the location of all variable and type declarations are known, and with the help of the APSE, data item dictionaries are easily m ade and maintained.
- o Interfaces tracing interfaces is made easier by the interface documentation.
- o Standardization since Ada is rigidly controlled in its implementation, Ada source code programs should be highly transportable from one system to another.

When Ada is compared to other popular higher order languages (HOLs), it is clearly a language designed with maintenance and documentation tasks in mind.

Since the Ada Language System is very structured, it is very suitable to such total life cycle activities such as configuration management and verification and validation.

Reviews, Walkthroughs, Audits

Reviews, walkthroughs and inspections are useful tools in eliminating errors in the design and coding phase. Because Ada can be made readable, preparation time for the walkthrough should be minimized, as well as misunderstandings arising from poorly structured and named code.

Configuration Management

Configuration management is the consistent labeling, tracking and change control of the computer program configuration items (CPCIs). Configuration management involves the careful tracking of the documentation and version history. This is simplified in Ada by use of the APSE. Revisions are automatically logged, access can be controlled, and previous versions can be recreated.

Verification and Validation

Validation refers to the activity of ensuring each end item product functions as specified in the requirements. Verificaiton ensures that the current development phase proceeds correctly from the intentions of the previous phase. Again Ada is useful in Verification and Validation activities because of its structured approach.

## III. FUTURE RESEARCH INTO ADA

### QUALITY IMPLICATIONS

This section outlines some areas of potential future research and exploration.

Requirements Phase Whose function should it be to write the specifications; an analyst, programmer or user? How extensive should the system be specified? How can Ada be used in specifications; as structured english using Ada reserved words, or something more extensive? What development time is required to write specifications using an Ada backdrop?

Design Phase Ada may be used as a PDL, but what structured methods work best with Ada? Ada introduces timing or rendezvous considerations and real time constraints are not always considered in structured methods.

Coding Phase How does the productivity of a programmer coding in Ada compare to other languages? Is error detection and correction easier?

Testing Phase Due to the prior measures taken with Ada, is it easier to test and integrate an Ada program? Without prior measures, is Ada easier to test? Are any testing methodologies especially well suited to Ada?

Ada may be used throughout almost the entire life cycle of software. The advantage in doing so is that a common background provides for easier control of a project, and the highly structured aspects of the language allow it to be maintained and managed with a minimal amount of difficulty.

### Author

Bruce Brocka is currently a faculty member of the United States Army Management Engineering Training Activity, located on the Rock Island Arsenal. He received a B.S. in Physics from St. Ambrose College, Davenport, Iowa, in 1981, and a M.S. in Electrical and Computer Engineering from the University of Iowa in 1984. Mr. Brocka has taught various courses in computer science and engineering.

# Software Management Control System
# S M C S
# (An Ada Approach)

Raymond J. McGlynn

Center For Tactical Computer Systems (CENTACS)
Fort Monmouth, New Jersey

## ABSTRACT

The Software Management Control System (SMCS) is a set of automated tools designed to support Army Life Cycle Software Support Centers (LCSSCs). These tools have been written in Pascal and as they are reworked in Ada, SMCS will ultimately be available within the ALS. Currently the Library tool is being written in Ada and designed to be portable, flexible and maintainable. This paper describes the purpose of the SMCS set and the detailed functions of the Library tool. Two approaches to using an Ada Program Design Language (PDL) are discussed and details of the selected approach are presented.

## SMCS Description

The concept of a Software Management Control System evolved out of studies into the needs and problems of managers at the Army LCSSCs. There are seven automated tools within SMCS: Baseline Controller, Tracker, Project Status Reporter, Tracer, Scheduler and Library. Each tool performs the function implied by its name. Creation of a common data base provides displays and reports of system information. The tools are written in Pascal and operate on the VAX 11/730, 11/750 and 11/780 under VMS.

## Library Tool Description

The SMCS Library tool was chosen to be written in Ada first since it is representative of a medium level tool in terms of programming difficulty and the number of functions performed. The following sections describe the functions of the Library tool. This Library tool allows the user to maintain an inventory of items in a software collection. An item is composed of number, title, owner, media, volume, copy, subject and classification. Item location, expiration, author, vendor, system and language are also specified. The item descriptions are kept in an item file. The tool also maintains an account of all items loaned and identifies the borrower. When an item is borrowed the loan date and due date are recorded for every borrower.

## Library Functions

The user will give some command to activate the tool such as the word LIBRARY and will then have to enter a correct System code, user name and password. This type of security is required to avoid unauthorized people from gaining access to the listings of the library where sensitive information may be stored. The tool will check the user name and password against an authorized user list. If the name and passwords do not match, then an exception routine will handle the error and thus inform the user. On the other hand if the name and password match, then the user will gain entry to the tool and a main menu will appear. The Library tool operates through the use of multiple menu driven screen forms which contain highlighted fields for data entry. Three screen forms and three reports can be selected from the Main Menu. Any selection opted, except Exit, will return the user to this Main Menu. The Control_Y sequence can be activated for fast, emergency exits from the tool. The Exit can return the user to the operating system command level or can log the user off the system depending upon the preference of the particular installation.

The Item Maintenance screen form maintains a file with a record for each library item. The user options of Add, Delete, Update, View and Exit are supported. The Disposition Screen form maintains the variant portion of the Library Item record. Its function is to record and display those items on loan to users of the Library. When selected, item records are expanded to include information of the Disposition screen. Options of Add, Delete, Update, View and Exit(to main menu) are provided. The View option lists information of all items on loan. When the screen becomes filled with information during the view

option, the user is prompted with a message to "Press RETURN to Continue or SPACEBAR to Quit". The Owner/Holder screen form displays information on library items as related to the owner or holder of the items. For example, all the software and manuals borrowed (held) by John Smith can be displayed. All the items in the library owned by another installation can also be displayed.

Four printed reports can be generated which can be printed on 8 1/2 by 11 inch paper. The information contained in the Item Maintenance screen form is the same as that given in the Inventory report. This report prints all the items in the Library tool. The Disposition report produces all items currently on loan with the repective due dates. The Expiration report lists items contained in the Library which are on loan from other owners. The report corresponds to the Owner/Holder screen form and lists all items that the Library is holding and whose dates have expired. The Overdue Report lists the Holders who have held items past the item due date. The phone number of the holder and title of the item are given. Listed below is a summary of the Library tool functions.

1. A display is maintained to identify all items of the software collection. An item can be given an I.D. number, title, owner, volume, subject etc..

2. A display to record all existing Library items on loan to borrowers.

3. A display to show all Library items of a specified owner. For example, many people or groups of people could own items contained in the Library.

4. CRT screen forms will be implemented using a menu driven scheme to maintain the three displays described above. Addition, deletion and modification of the data will be supported.

5. Hard copy reports will be generated using the data from the three displays above.
   a. Inventory Report - Lists all items in the library.
   b. Disposition Report - Lists all items on Loan.
   c. Expiration Report - Lists all items held belonging to other Owners.
   d. Overdue Report - List all overdue items.

## Alternative Approaches

One approach to this design would be to place the emphasis of design on the programmer. This would require less detail and time during the program design phase. The Ada software design can consist of a package structure of the program. Package specifications and how they relate to subprograms and other library units can be depicted in the design. Exceptions that handle a system wide scope of error conditions must be designed in the system at an early stage. The mention of the program units and subunits that raise and handle these exceptions can be given as comments. Among the generic packages used in the high level design, only subprograms that provide an independent function as expressed through the package specification need be mentioned. Subprogram units used as subordinate modules to such independent functions should not be identified for high level design. Such an approach produces the following effects.

a. LESS INFORMATION IS CONVEYED to the programmer and reviewer. While this may be suitable for an overview of the system the user or designer would need more information closer to the actual product. This is comparable to a community developer who formulates the layout of the streets, houses, schools etc. in a planned community. The homebuyer or housing contractor needs to know much more specific information to base their respective decisions upon. Likewise, while it may be desirable at first to see the overall layout of a system, the software builder or buyer cannot afford the luxury of dealing only in highly abstract terms.

b. The design is at a HIGHER LEVEL OF ABSTRACTION. At such an abstract level of design, full advantage is not taken of features existent in a high order language. The PDL will not contain the expressive power of the higher order language which can be used to express the structure of the design.

c. FEWER CONSTRUCTS would be needed in the Program Design Language and might only consist of package and procedure specification parts. Consequently, the resultant design is likely to be less robust and flexible than would be desired. Too much design work is left to the programmer who is already overburdened with implementation details.

d. A LESS ROBUST AND THOROUGH program design language results than would be desired. Powerful tools cannot be used to check the design at the various stages of development. For example, an Ada

compiler would not be feasible to check such a design at the data flow level. Traceability and visibility (two important features at high level design) cannot be checked without a thorough program design which is close to the implementation language.

e. IMPLEMENTATION IS HARDER to achieve for the programmer when only a general outline of the system is given. The programmer is forced to analyze the situation himself to fill in the missing portions of the design. Thus the programmer becomes an analyst and systems designer. Even if the same person by himself performs the three functions mentioned above, the same concepts hold true. A more general high level program design language will make it harder for that person to implement. That person will be forced to deal with the more specific design decisions later in the life cycle and coding will be started earlier.

Deferring the program design is not a vice at high levels of abstraction. To defer detailed program design until implementation time by using a lax methodology or a liberal PDL does not take full advantage of the features existent in a higher order language nor the tools available to process design work. Disagreement exists yet as to the relationship of Ada to the life cycle development phases. However much research has been produced to show that Ada can be used as a specification language which applies to most phases of software development and support. For this reason an Ada Program Design Language would have to be used even if this first alternative approach were taken.

The second alternative approach, which has been chosen, is to place the emphasis of detail on the designer to bridge the gap between the design and the implementation. At a low level of design, stepwise decomposition of high level design is carried out until coding is ready to begin. Such decomposition can be performed entirely in Ada since the high level design can also be specified in Ada. Information hiding can be increased through the use of decomposition and stepwise refinement.

Bodies for the procedures and packages of a high level design are elaborated for the low level design. As this is done, other packages and procedures to be used as subordinate modules emerge. Any newly declared exceptions during this process must not be visible to other parts of the program structure.

During the low level design every effort should be made to take advantage of existing routines previously developed. Also, operations that can be used in many parts of the program should be identified as far in advance as possible to make use of the commonality of fan-in library modules.

The Ada based PDL used for this application strives for a rigorous syntactic and semantic definition of all the Ada constructs in the Ada Language. Such a method can be used to state the system requirements as well: "(It was interesting in this context, that through a rigorous specification of requirements in Ada, one contractor saved a significant amount of time in the detailed design and coding phases.)" (5). Using such a rigorous approach to formulate the Program Design Language produces the following characteristics:

a. MORE INFORMATION IS CONVEYED if needed. Simplicity and understanding of the design can be maintained if the reader decides to review the specification sections first before investigation of more detailed body parts. Since the syntax of the Ada language closely resembles that of the English language, its constructs are more readable and understandable. The syntax of the Ada design language constructs are compatible with the syntax of the Ada programming language using this approach.

b. MANY LEVELS OF ABSTRACTION can be obtained. By drawing upon the strengths of the software engineering concepts used by the designers of the Ada programming language, the Ada design language inherently possesses the same expressive power. The constructs supporting design characteristics such as modularity, abstraction, information hiding, generics, exceptions, strong typing and data description are provided in Ada. These characteristics can be used to express both high level requirements of a system and detailed system specifications. Such features are not only desired but required in a design language at all phases of the life cycle. When abstraction of all the details of the implementation language are expressed at a higher level the problem is the tendency to start actual coding of the system while laying out the design. However, if the various levels of abstraction are followed throughout the design then this problem can be avoided. No standard rules have been adopted that

specify when the design stops and the coding begins.

c. MORE CONSTRUCTS available in the PDL provide the designer with more expressive power. This expressive power can be used to specify the detailed design which is close to the implementation stage if so desired. An abundant amount of constructs of the implementation language should be used as the Ada based PDL at this level to accomplish this. Constructs from both the specification section and body section of an Ada program unit are used. The semantic meaning of the Ada PDL constructs must have the same exact meaning as the Ada programming language specified by the Ada Language Reference Manual (LRM).

d. A ROBUST DESIGN can be produced by subscribing to the conventions of the target language. A major issue to deal with between the two alternative approaches is the choice of using a relaxed syntax format versus the exact syntactical requirements of the Ada language. The latter was chosen for the reasons discussed below. First, the Program Design Language was envisioned as a subset of the Ada language and therefore the exact syntax of the language is desirable. Second, it may be desirable at times to compile the design segments at various stages to aid in the creation and correction of the evolving design product. Third, if and when a legal requirement is adopted that insists the Ada design be machine processable then this Design Language (DL) will satisfy such a requirement. Verification of the design throughout the development phases can be checked by a compiler and other design tools. A design program processed in this manner can be used to locate interface errors, scope and visibility errors, and procedure and function definitions. The effort to actually code the design is bridged with less difficulty when the syntax of the design is the same as that of the implementation language. The compatibility with the Ada language is obtained by the following mechanisms: Ada syntax, Ada semantics, Ada compilability, Ada extentions and Ada tools.

e. EASE OF IMPLEMENTATION from the design is produced especially if the target language is the same or similar to the design language. Every construct in valid Ada is acceptable to the design language. On the conservative end of the spectrum, this means that every construct of the Ada language can be processed by the same tool used to process the design language. This does not mean that every construct of the Ada design language need already exist in the Ada programming language.

It has been shown many times in the history of computer program development that the more time and effort spent in the design phases during system development results in reduced time and cost of implementation, maintenance and enhancement. This is the primary reason why the second approach was chosen.

## Ada Program Design Language Usage

The following sections describe some of the constructs which are useful and desired in an Ada Program Design Language for this SMCS Library application. The use of an Ada based program design language for this application can be seen as a tool for software design. The Ada based PDL is used to express the structure and design of the software to be programmed. Due to the advances made in software engineering in general and the Ada programming language in particuliar, PDL's can now express more than just local algorithmic information. Projects utilizing a PDL in this manner realize gains in productivity, debugging and maintenance efforts. The four possibilites of using the Ada language as an Ada PDL as portrayed graphically by Grau and Comer of the Harris Corporation (5) are: the Ada PDL can be exactly Ada, a subset of Ada, a subset of Ada plus extentions or a superset of Ada. There is no doubt that not every construct of the Ada language is needed during design; however,an Ada PDL that incorporates more Ada features will be of greater use to the Ada community. Extentions and additions to the Ada language can be used to supplement the Ada PDL. Although some of these extentions and additions are not exact duplications of features existing in the Ada language, they can be compatible nevertheless through the use of the Ada comment. It is intended that the Ada based PDL remain as the commentary for the source code once the system has been Implemented. It should be noted that the Ada design language can be used for systems which will be implemented in languages other than Ada. The Ada based PDL is used to express the structure and design of the software to be programmed. The PDL can be used with other documentation as a basis for the Preliminary Design Review (PDR) and Critical Design Review (CDR). For each construct of the PDL host language, Ada the following will be discussed: first, why the construct in the Ada design language is useful for this type of application; second, the operational

effect of using the construct and third, the cost effectiveness for the use of each construct.

**Predefined Types**

Predefined types as defined in package STANDARD of the Ada Language Reference Manual are boolean, integer, float, character, natural, positive, priority, string and duration. Literals can be of type character or numeric. Character literals use a single quote for identification as shown in the example below of type Option. Double quotes denote character strings such as:

```
Rec_type.Title := "TAGGED FOR DELETION";
```

Numeric literals can be integer or real. Boolean types have two possible values True or False. Natural types can assume only positive integer values.

1. The predefined types are easily employed due to the ease of use with package Standard.
2. Character type, which is a predefined type, is essential to have since the application deals with character input and output processing.
3. The cost savings can be realized by using the package STANDARD and thus avoiding the development cost of producing a generic package for the predefined types.

**Enumeration Types**

Enumeration types are discrete types that elaborate a complete list of all values for the data object. The operations of =, /=, <, <=, >, >= and := can be used with enumeration types. The following statement represents a user defined enumeration type based on the predefined type CHARACTER in the package STANDARD.

```
type Option is ('A','a','U','u','V','v',
                'D','d','E','e');
```

The procedures of Add, Update, View, Delete, Help and Exit for each screen form are selected by the user with a one character input which can be an upper or lower case letter.

1. This construct type enables a programmer to list the legal set of values of an ordered list for the defined type.
2. This type enables a complete list of data values to be specified alphanumerically with operations which can be performed on the objects of this type.
3. Enumeration types effectively reduce the cost of expressing data abstraction, data structures, parameter interfaces and data dependency.

**Scalar Types**

Scalar types cannot be decomposed and objects of these types can only assume one value at a time, such as the type Option shown above. 'First, 'Image, and 'Val are some attributes available for scalar types. Attributes are applied by preceding the attribute with the appropriate type; for example Option'First.

1. Ada scalar types are utilized to reference single value objects according to some attribute.
2. Readability and ease of maintenance are affected mostly by this construct.
3. The cost savings exist in the form of increased productivity, especially in the maintenance phase.

**Composite Types**

Composite types group logically related parts of data. Composite data objects consist of multiple values. Item_rec, shown below, is a record type which combines the components into an ordered set. The operators: =, /= and := are used with record types.

```
type Item_rec is
  record
     Item         : integer;
     Title        : string(1..30);
     Owner        : string(1..15);
     Version      : string(1..5);
     Media        : string(1..10);
     Subject      : string(1..10);
     Day_stored   : integer;
     Month_stored: string(1..3);
     Year_stored  : integer;
     .
     .
     .
  end record;
```

Each component of the record has an object identifier which is declared of type String or Integer.

1. This type is used due to the necessity of having record types and arrays for this particuliar application. Records need to be defined for database manipulation and arrays for the extensive string processing which is needed.
2. This allows operations on an ordered set of several parcels of data pertaining to a particular type. Also the operations of equal, not equal and assignment can be performed with array and record components.
3. Savings are realized since every component of a composite type need not be specifically referenced for the file operations of READ and WRITE.

A slice is a consecutive portion of a one

dimensional array. Slices and assignments of one dimensional arrays are useful as shown by the example below:

```
procedure Edit_Field
        (Astring : in out String) is
    Que : String(1..1);
    CR  : Character := ASCII.CR;
  Count : Integer := 1;  --Last char pos.
  Blank : String(1..30); --Longest field.
begin
 loop
  get_line(Que,1);
  if Que(1) = CR then
     Astring(Count + 1 ..Astring'last):=
     Blank(Count + 1 ..Astring'last);
     exit;
  end if;
 end loop;
end Edit_Field;
```

This Edit_Field procedure accepts a variable length string and detects when the user has entered a carriage return. If so, the remaining slice from the last character entered to the end of the Astring is padded with blanks.

**Derived Types**

1. The derived types and subtypes are used in the PDL to enable the designer to create new types from old types which have the same properties.
2. This will have the effect of restricting and constraining the newly derived types.
3. Such derivation of types will maintain consistency of data and provide to the compiler a means of checking such consistency. Since the data is strongly typed, checks by the compiler or other design tools can point out flaws before effort is expended building upon them.

**Private Types**

1. This type is useful in an existing PDL because the details of packages and procedures developed in the application need to be hidden from the users of these packages and procedures.
2. This feature could affect the operation of the Library tool by allowing certain parts to be untouched by users of the common utility packages.
3. Limited private types will not allow any operations in other program units to affect the data objects declared.

**Access Types**

The method shown below makes use of semaphores to protect data at the record level. This method utilizes access types.

```
package Read_Write_Record is
  type semaphore is access boolean;
  procedure P (Proceed:in out semaphore);
  procedure V (Proceed:in out semaphore);
end Read_Write_Record;

package body Read_Write_Record is
  procedure P(Proceed:in out semaphore)is
  begin
    while (Proceed.all = false) loop
     null; -- waiting.
    end loop;
    Proceed.all := false;
  end P;

  procedure V(Proceed:in out semaphore)is
  begin
    Proceed.all := true;
  end V;
end Read_Write_Record;
```

A danger exists if there are several processes operating concurrently. One process may check to see if Proceed is set to TRUE and if so continue to set Proceed to FALSE. If another process at the same time has checked Proceed and found it to be TRUE, then two processes would enter a critical area at the same time.

1. Access types are used by the Ada PDL since the application may require linked records.
2. This will obviously affect the application as described, and result in easier manipulation of linked records and pointers.
3. This feature is an efficient mechanism for linked data structures in the application where dynamic processing of pointers to records is desired. Access types should be added to this PDL in the future once the exact design for linked records in the application is defined.

**Task Types**

A second method which guarantees mutual exclusion, at the record level, is the use of tasking. If one process is writing, then another process is not allowed to be reading or writing.

```
package Read_Write_Record is
          Value : Data_type;
   procedure Read (Info : out Data_type);
   procedure Write(Info : in  Data_type);
end Read_Write_Record;

package body Read_Write_Record is
     Common_data : Data_type := Value;

  procedure Read ( Info : out Data_type) is
  begin
    Common_controller.Read(Info);
  end Read;

  procedure Write (Info : in Data_type) is
```

```
  begin
    Common_controller.Write(Info);
  end Write;

  Task Common_controller is
     entry Read (Info : out Data_type);
     entry Write(Info : in  Data_type);
  end Common_controller;

  Task body Common_Controller is
  begin
    select
      accept Read (Info : out Data_type) do
        Info := Common_data;
      end Read;
    or
      accept Write (Info : in Data_type) do
        Common_data := Info;
      end Write;
    end Common_Controller;

end Read_Write_Record;
```

Three observations can be made about an Ada design which includes tasking.

1. The major packages would operate independently.
2. The designer may need to know if certain data types have to be guarded against multiple access.
3. It is unclear if new tasks should be introduced during the refinement of the high level design. Many dynamic tasks spawning other tasks can disrupt timing constraints making it impossible to calculate the effects in a real time system.

## Specification Parts

The Ada specification region of a module or package contains information about the subprograms and parameters which are passed in and out of the package. In an Ada based PDL there may be times in the high level design when only the specification portion of a package will be used. For low level design the body parts for both packages and procedures should contain at least the basic data structural outlines to be used. Data types and objects can be declared in this specification part. The example below shows an outline of the specification and body parts of a package. Note the importation of the package text_io by the "with____"; and "use___"; clauses.

1. The specification region for program units serves as a commented header to each module of the application design. Information about the interface parameters can be clearly stated. A global view of the next layer of partitioned modules as well as horizontal use of other packages can be clearly seen using this construct.
2. This region improves the resultant design and implementation of the application. Each software unit can be identified as well as the internal and external interfaces.
3. The specification part supports the design characteristics of abstraction, dependency and traceability through its use.

```
with text_io; use text_io;
package Screen is
  Video : file_type;
 Bold,Rev,Off : string(1..8); -- escape
         Home : string(1..3); --sequences
         Que  : string(1..1);
                        -- Highlight
 Blank1 : string(1..9);  --1 char. field.
 Blank2 : string(1..10); --2  "      "
 Blank3 : string(1..11); --3  "      "
 Blank5 : string(1..13);
 Blank10: string(1..18);
 Blank15: string(1..23);
end Screen;
Package body Screen is
begin
   open(Video, In_file,"W.");
   get(Video,Home); get(Video,Bold);
   get(Video,Rev);    get(Video,Off);
   close(Video);
   Blank1 := Rev & " " & Off;
   Blank2 := Rev & "  " & Off;
   Blank3 := Rev & "    " & Off;
   Blank5 := Rev & "      " & Off;
   Blank10:= Rev & "           " & Off;
   Blank15:= Rev & "                "&Off;
End Screen;
```

Procedures can be encapsulated within a package. The subordinate relationship can be viewed from the specification part as shown below.

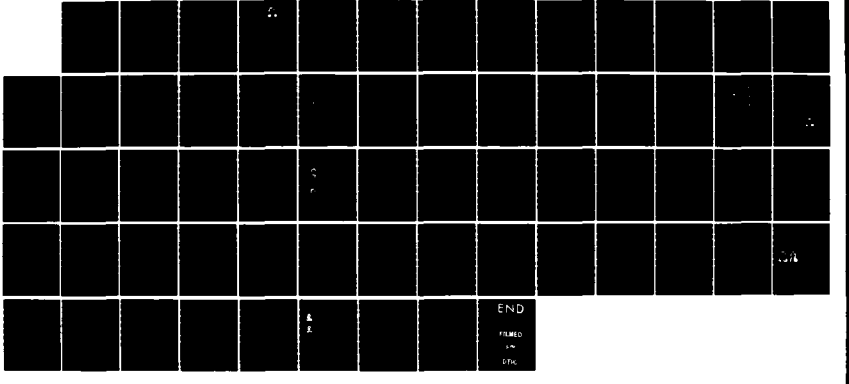
```
with direct_io;
with text_io, screen;
use  text_io, screen;
package Item_Screen is        --subordinate
 procedure Display_Item_Screen;  --module
end Item_Screen;                          |
package body Item_Screen is               |
  .                                       |
  procedure Display_Item_Screen is -------|
     .                                    |
  begin                                   |
     .                                    |
  end Display_Item_Screen;          -------|

begin   --Package Item_screen.
  .
end Item_Screen;
```

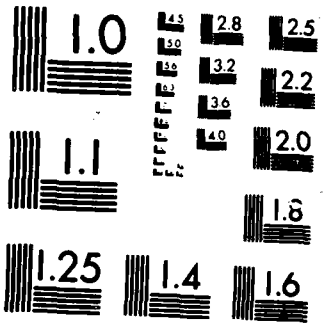
## Conditional Constructs

There are two kinds of conditional constructs, the If statement and the Case statement as shown below.

AD-A164 338 PROCEEDINGS OF THE ANNUAL NATIONAL CONFERENCE ON ADA (TRADEMARK) TECHNOLO..(U) ARMY COMMUNICATIONS-ELECTRONICS COMMAND FORT MONMOUTH NJ CENT .. 1985 3/3

UNCLASSIFIED F/G 9/2 NL





```
 if Que(1)='y' or Que(1)='Y' then
  put("ITEM TO BE DELETED AFTER EXIT");
 elsif Que(1)='n' or Que(1)='N' then
  put("DELETION ABORTED");
 end if;

case Menu_Choice is
     when 0 => Exit;
     when 1 => Display_First_Screen;
     when 2 => Display_Second_Screen;
     when 3 => Display_Third_Screen;
     when others => null;
end case;
```

1. The IF statement is utilized to prevent forward jumps used by the conditional GOTO construct of other languages. The CASE construct can reference previously defined procedures and thus prevent the use of the GOTO for backward jumps. The CASE structure provides a set of mutually exclusive choices for each of the predefined actions.
2. Maintenance is improved by these constructs. The programmer and designer do not have to deal with intertwining code. The flow of control is more easily followed and organized.
3. The cost of debugging the application for implementation or maintenance is reduced. Finally the cost of maintaining, modifying and enhancing the design is greatly minimized.

**Repetitive Constructs**

The repetitive constructs are:

```
loop ... end loop;

while < > loop ... end loop;

for < > loop ... end loop;
```

1. The loop, while loop, and for loop perform iteration of execution on a sequence of statements. This repetition can take place zero or more times as specified in the <conditional expression> by the designer. The loop construct continues execution repeatedly until some condition within the statements of the construct causes an exit to occur. The while <condition> loop, will iterate until the boolean expression is satisfied. The for <condition> loop, executes just once for each discrete value in the expression. Exit from any of these three constructs may occur by using the "exit;" statement.
2. The operational effects of these three constructs are: control flow, algorithm control, data flow and reusable data structures.
3. Cost for the application is lessened by the additional control of execution and algorithm flow. This control makes implementation and maintenance easier and more understandable.

**Block Construct**

A simple sequence of statements can be grouped into the basic building unit called the BEGIN block. This BEGIN block is used to formalize a series of statements in a functional structure. The format is:

```
begin
.   Ada statements;
.      "
end;
```

1. The BEGIN .... END block construct allows grouping of statements into functional building blocks.
2. Control flow and data structure are improved through its use. The executable statements for each program unit are clearly outlined. Such a block construct is also useful for loop control where errors occur and exception handlers are needed while still iterating in a loop.
3. The cost of implementing the Library tool and maintaining it is reduced through the ability to clearly define begin..end blocks without calling a subprogram.

**Stubs**

This construct is very useful for progressive stepwise decomposition of a software system into smaller units. For example:

```
package Library_Tool is
 procedure Display_Main_Menu is separate;
end Library_Tool;
```

can later be elaborated as:

```
procedure Display_Main_Menu is
  Top : string(1..80):=" MAIN MENU      ";
  One : string(1..80):=" 0 - Exit       ";
  Sec : string(1..80):=" 1 - First Menu";
begin
  put(Home); New_line(5);
  put_line(Top);
  put_line(First);
  put_line(Sec);
end Display_Main_Menu;
```

1. The "is separate" construct assists in high level design and stepwise decomposition where abstraction is critical.
2. Dependencies of packages and procedures can easily be traced in the specification part by using this feature. The construct supports abstraction in both the early design and can be kept through to the implementation phase.

3. Costs associated with vertical and horizontal tracing can be lowered with this feature. The time spent in determining the scope and visibility of a module is also lessened.

**Exceptions**

Exceptions can be added to the BEGIN block for error handling and other exceptional conditions. An exception can also be forced to occur by the use of the raise statement:

```
raise numeric_error;
```

An exception handler can be added at the end of a "BEGIN END;" construct to negotiate an occurence of an error as shown below:

```
procedure Display_Main_Menu is
  Top : string(1..80):=" MAIN MENU      ";
  One : string(1..80):=" 0 - Exit       ";
  Sec : string(1..80):=" 1 - First Menu";
  Junk: string(1..1);
  Que : integer;
begin
  loop                  --Bad input is
    begin               --accepted by Junk
     put_line(...);     --and loop execution
     get(Que);          --continues.
    exception
     when data_error => get(Junk);
             put("ERROR MESSAGE");
    end;
  end loop;
end Display_Main_Menu;
```

If the user enters an error at the get(Que); statement then an exception raises a data error. The exception handler can display an error message, accept the incorrect entry and resume loop execution.

1. Exception declarations and exception handlers provide localized as well as global handlers of error situations.
2. Effects of operation on the design are; control of execution and algorithm flow, and interfacing dependency requirements. For example, if a local procedure does not contain an exception handler for a particuliar error, then the error is propagated to the exception handler in the calling procedure.
3. Handling of error situations at the desired level of dependency avoids redundant code and allows the designer greater flexibility.

**Summary**

The matrix which follows, was taken from the IEEE Ada PDL guide(2). The matrix displays the relations between constructs of an Ada PDL, some of which were discussed above, and the supported design characteristics very well.

Language Features and Constructs

| | | | |
|---|---|---|---|
| A | Packages | I | Comments |
| B | Subprograms | J | Private |
| C | Tasks | K | Exceptions |
| D | Spec/Body | L | Rep Specs |
| E | Pragma | M | Allocations |
| F | Types | N | Generics |
| G | Objects | O | Predefined Pkg |
| H | Stubs/Is Separate | | |

| Design Traits | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Abstraction | * | * | * | * | | * | | * | * | * | | * | | * | * |
| Data Flow | | * | * | | | | | | * | | | | | | * |
| Control " | * | * | * | | | | | | * | | * | | | | |
| Algorithm | | * | * | | | | | | * | | * | | | | |
| Data Design | * | | | | | | * | * | | * | * | | | * | * |
| Data Ref. | * | * | * | | | | | | * | | | | | | |
| Interfaces | * | * | * | * | * | * | * | | * | * | * | * | * | | |
| Dependencies | * | * | * | * | * | * | * | * | * | * | * | * | * | | |
| Reusability | * | | | | | | | | | | | | | * | * |

As shown above the Ada approach to an application, such as the SMCS Library tool, greatly minimizes the cost and complexity through the use of a rigorous Ada design language. A subset of useful constructs in the Ada language were presented to illustrate this point. As Ada is put to greater use in other area applications more insight into the use of Ada as a design language will be revealed.

**Acknowledgements**

The author wishes to acknowledge the reviewers of this paper: Mary Bender, Andrea Cappellini, Mike Crawley, Frank Laslo, Anthony Serfarty and Thomas Wheeler. The opinions expressed herein are those of the author and should not be construed as policy of the U.S. Army or its Center for Tactical Computer Systems (CENTACS).

**BIBLIOGRAPHY**

1. Ada Case Studies II. United States Army Center Tactical Computer Systems (CENTACS), January 1984.

2. Ada as a DL. IEEE Working Group on Ada as a PDL, July 1984.

3. Ada PDL Is The Answer. (But What Was The Question?). N. Lomuto, Softech, Inc., May 1983.

4. Ada Primer. United States Army Center For Tactical Computer Systems (CENTACS), January 1984.

5. Ada Process Description Language Guide. J.K. Grau, E.R. Comer, H. Krasner and P.B. Dyson, Harris Corporation, Melbourne, FL., March 1982.

6. Ada Programming Design Language Survey, Final Report. Naval Avionics Center, October 1982.

7. Ada Software Design Methods Formulation Final Report. United States Army Center For Tactical Computer Systems (CENTACS), October 1982.

8. Advanced Ada. United States Army Center For Tactical Computer Systems (CENTACS), July 1984.

9. Case Study II Final Report Developed For Large Scale Software System Design of the Missile Minder AN/TSQ-73 Using The Ada Programming Language. United States Army Center For Tactical Computer Systems (CENTACS).

10. Military Standard Ada Programming Language. ANSI/MIL-STD-1815A, United States Department of Defense, January 1983.

11. Real-Time Ada. United States Army Center For Tactical Computer Systems (CENTACS), July 1984.

12. Software Technology Impact on System Development. J.E. Kernan, Software Technology Division, U.S. Army Communications - Electronics Command and Fort Monmouth.

13. The Development of Ada Program Design Languages. United States Army Center For Tactical Computer Systems (CENTACS), January 1984.

Raymond J. McGlynn is a computer scientist in the Center for Tactical Computer Systems (CENTACS) at Fort Monmouth, New Jersey where he has been employed since 1982. He is responsible for coordinating Software Management Control System (SMCS) activities and has participated in the IEEE Working Group on Ada as a DL. Mr. McGlynn received his B.S. degree in Management Science / Computer Science from the State University of New York (SUNY) at Geneseo and is pursuing a Masters degree in Computer Science at Fairleigh Dickinson University.

# AN Ada® MEASUREMENT AND ANALYSIS TOOL

S. E. Keller and J. A. Perkins

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

ABSTRACT

ADAMAT™, an Ada Measurement and Analysis Tool, provides immediate assistance for 1) improving the quality of Ada software, and 2) training Ada programmers. The underlying metrics framework is hierarchical, based on the McCall metrics framework, tailored to the Ada language, and formally defined using Prolog. The automated data collection component is automatically generated using compiler generation techniques, which include a description technique for describing pattern matching in a well-defined language. The quality analysis component, based on the formal definition of the metrics, provides users with interactive analysis of the metric data, and allows users to step through the Ada metrics hierarchy to pinpoint problem areas.

Keywords

software metrics, software quality, software measurement tools, software management, Ada, Prolog

## 1. INTRODUCTION

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

Unfortunately, most of these kinds of benefits of software quality metrics are contingent upon validation. That is, one must have a high degree of confidence that the metrics actually measure the software characteristics that affect software users and developers. Validation requires gathering sufficent data to demonstrate solid relationships between measured quality and actual quality. Metrics researchers are in the early stages of gathering this needed data. Software quality metrics have been applied on only a handful of projects; metrics specifically for Ada have been introduced just recently.

Our approach to developing Ada metrics is to develop metrics and metric tools that provide substantial benefits even in initial applications, before validation. Program managers have been reluctant to introduce software quality metrics into their projects because of concerns about the cost-to-benefit ratio of applying metrics not yet validated. Metrics and metric tools with immediate benefits can get us over the initial cost/benefit hump, leading to the widespread application of metrics, and ultimately to a set of validated metrics.

Our basic design goal is to develop a software quality metrics framework that provides the user with sufficient details to identify the cause of indicated quality problems. We believe that the assessment of low quality must lead to the identification of the cause of the problem, thereby enhancing the ability of managers and engineers to make decisions affecting software quality.

Our approach to achieving this goal revolves around two key metrics design rules. First, each metric should measure the expression of a good software quality principle. For example, "When using fixed point types with a DELTA that is not a power of 2, always use a length representation clause for SMALL that is the same as the DELTA ... " [Booch84]. Second, each metric should be defined in terms of specific features of the Ada language. For example, based on the above quality principle concerning fixed point types, define a metric to be the percentage of fixed point types with DELTA not a power of 2 and without a length representation clause for SMALL = DELTA.

Since Ada was designed to support software quality practices [Ada83], metrics can be devised that measure the expression of good quality principles in Ada source code and that relate quality directly to features of the Ada language. Moreover, such metrics are less dependent on validation, and hence, immediately beneficial. These metrics measure the degree to which an Ada development effort is adhering to the software quality principles Ada was designed to support. Thus, the metrics address this question: Is the programming in the spirit of Ada ?

Section 2 describes our approach to the design of an Ada metrics framework. Section 3 describes the Quality Analysis Assistant, a tool based on the Ada metrics framework that provides analysis capabilities for pinpointing software quality problems. The Data Collection Tool, which analyzes the Ada source and collects basic data values is described in section 4. Finally, the status of Ada metrics and tool development is described in section 5.

## 2. Approach to Ada Metrics

We applied a formal top-down approach, illustrated in figure 1, to develop the Ada software metrics framework. Initially, each element of the McCall metrics framework was reviewed with respect to users' needs, adherence to metric design rules, applicability to Ada, automation potential, cost-effectiveness of implementation, and impact on the data base size. Then the resultant Ada metrics were defined in terms of features of the Ada langauge, associated with software quality principles, and formally specified in first-order logic, using the logic programming language Prolog. The formal definition ensured a complete, consistent definition of the metrics before automation, permitting a balanced treatment of metrics design rules against automation concerns. The executable specification allowed us to experiment with the behavior of the metrics before automation of the data collection.

Section 2.1 provides an overview of the metrics framework that forms the basis for the Ada metrics framework. Section 2.2 describes investigations that established the foundation for Ada metrics development. Section 2.3 illustrates some of the metrics that resulted from our development approach. The formal definition and a tool supporting understanding and modification of the formal definition are described in section 2.4.

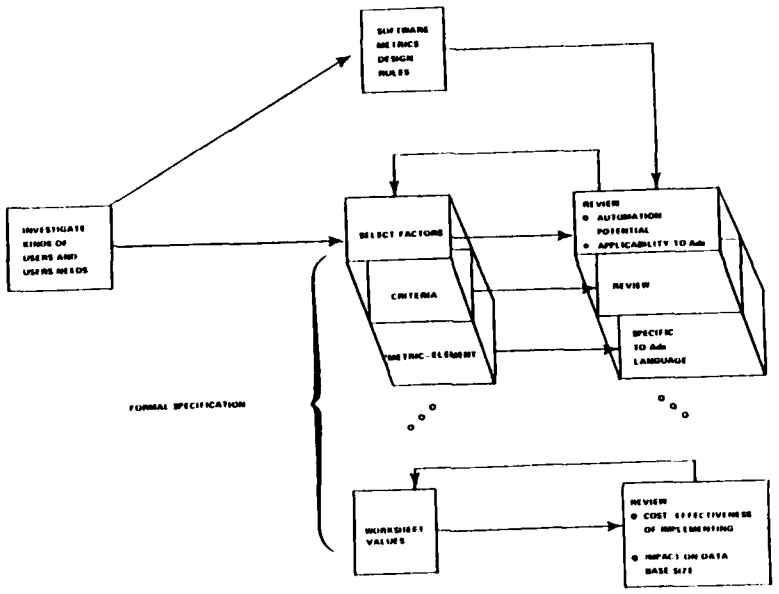



**Figure 1: Approach to Ada Metrics**

## 2.1 Baseline Metrics Framework

The software quality framework developed by McCall and extended by RADC [Bowen83] and DRC [San Antonio83] forms the basis for our metrics framework. The McCall framework results from a thorough analysis of software quality concepts and terminology and incorporates software characteristics important to users, developers, testers, and maintainers of Ada software. Quantitative measurement of software quality has been conducted based on this framework [San Antonio83].

The McCall framework is hierarchically structured. The various levels of the hierarchy are described below, and the interconection of different levels is illustrated in figure 2.

Factors are management-oriented terms, such as maintainability, testability, reliability, and reusability, that represent important software qualities. Each factor is composed of one or more criteria.

Criteria are software-oriented terms, such as simplicity, generality, self-descriptiveness, and modularity, that relate software characteristics to factors. Each criterion may support more than one factor, and is composed of one or more metrics.

Metrics are objective measures of software characteristics. Each metric may support more than one criterion, and may be subdivided into metric-elements or be calculated directly from one or more data items.

Data items are primitive software measures, directly derived from the software source. Each data item may support more than one metric or metric-element.

## 2.2 Needs of Metrics Users

To establish a solid foundation for designing the metrics framework, we conducted an investigation of the needs of metrics users. The investigation allowed us to establish a set of metrics design guidelines, described in section 2.2.1, that help to ensure that users' needs are addressed. The investigation also allowed us to firm up our understanding of the relationship between standards and metrics, described in section 2.2.2, and to distinguish several classes of metrics, described in section 2.2.3.

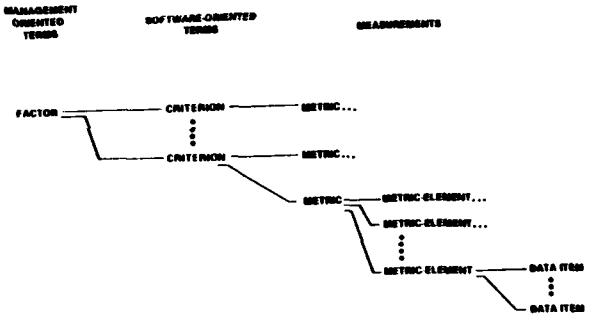



**Figure 2: Metrics Framework Hierarchy**

Our investigation of metric users focused on the needs of managers in the software areas of acquisition, development, and training.

Acquisition managers are interested in developing objective software quality requirements and in measuring software for compliance with these requirements. Thus, the acquisition manager needs metrics that assess the overall quality of the software product and that indicate the difficulty of testing, porting, and maintaining the software product. He requires metrics that provide the basis for an acceptance criterion of the software product, and that form the foundation for the software science of predicting the cost of installing the software, the cost of maintaining the software, and the expected error rate when using the product.

Development managers are interested in tracking quality during software development. Thus, the development manager requires metrics that provide constant assessment of the quality, that promote early detection of quality problems and bad quality trends, that pinpoint areas within the software product where quality improvement is possible, and that isolate groups or individuals whose performance is having a negative impact on quality.

Training managers are interested in determining which individuals and what language features require training, and in evaluating the effectiveness of training in both areas. Thus, the training manager needs metrics that provide information concerning the use of features of the language, that pinpoint constructs of the language being abused, misused, or ignored, and that isolate groups or individuals that improperly use aspects of the language.

### 2.2.1. Metric Design Guidelines

Although the metric design rules of relating metrics to quality principles and to features of the Ada language provided the greatest impact on the definition of our Ada metrics framework, the following additional metric design guidelines were also important.

When possible, define a metric to be collectible on a program, module, or construct basis. For example, the quality of commenting should be measureable for all modules and all constructs, for a module over all constructs, and for a construct over all modules. Metrics that are defined for individual modules or constructs support pinpointing a quality problem to the area of the source code where the problem occurs.

When defining a metric, consider the consistency of the units of the data items. For example, when formulating a metric for the use of boolean expressions, the ratio of the number of complicated boolean expressions to the number of boolean expressions has better unit consistency than the ratio of the number of complicated boolean expressions to the number of executable statements.

All metrics in the framework should be represented by a value between 0 and 1, where 1 represents the highest quality. Moreover, any metric directly calculated from a data item should be monotonic with respect to that data item. For example, the metric representing the number of branches should be normalized, such that an increase in the number of branches results in lowering the metric value closer to zero. Metrics which are strictly monotonic with respect to a data item are more desirable than binary metrics, since they provide better resolution, allowing users to isolate areas for improving quality.

### 2.2.2 Relation of Standards to Metrics

Our desire to associate metrics to software quality principles caused us to examine the relationship of software standards and software metrics.

Most software standards, such as 1) never use GOTOs, 2) declarations must be commented, 3) two statements must not appear on the same physical line, and 4) statements within a decision block must be indented, correspond to metrics that have fixed thresholds. Standards tools can be viewed as calculators of binary metrics, where the values 0 or 1 indicate noncompliance or compliance, respectively.

Other software standards are formed by combining a quality principle with empirical evidence. For example, the software standard, "a maximum of 10 branches are allowed in a subprogram," combines the quality principle, "avoid the excessive usage of branching constructs in a subprogram," with the empirical evidence, "subprograms containing more than 10 branch constructs show extensive lose in quality" [Walsh79]. Again, compliance to these standards can be measured; however, such validation will not differentiate between a module containing 10 branches and a module containing 3. These standards measure the number of instances where extensive quality is lost due to branching. A metric which measures the amount of branching better reflects the quality of the software, with respect to software factors such as testability and maintainability.

Presently, there is insufficient empirical evidence to associate fixed thresholds with most quality principles, thereby precluding the establishment of corresponding software standards. However, metrics provide a means of measuring compliance to a quality principle in the absence of such evidence. Furthermore, metrics based on quality principles can be used to gather the data necessary to determine meaningful thresholds.

### 2.2.3 Classes of Metrics

Our analysis of the metrics in the baseline framework resulted in distinguishing three classes of metrics.

Absolute metrics are measurements of the absolute amount of a software characteristic. Many of the traditional metrics fall in this category. Examples are: 1) the number of lines of code, 2) number of operators, and 3) the number of branches. Absolute metrics support comparisons between problem spaces, comparisons of interest to acquisition managers. The value of most absolute metrics is calculated using the rule, 1 / (N + 1), where N is the number of occurrences of the language feature.

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

Language-use metrics measure how and how many language features are used. Examples are: 1) the number of generic packages, 2) the number of fixed point types, and 3) the percentage of types used that are not predefined. Language-use metrics are primarily of interest to training managers and may be either relative or absolute metrics.

## 2.3 Definition of Ada Metrics

The definition of our Ada metrics framework involved a review of each metric outlined in RADC's Guidebook for Software Quality Measurement [Bowen83]. To illustrate how our metric design guidelines and the unique features of the Ada language affected the definition of our resultant Ada metrics, our definitions of several metrics, associated with the criterion simplicity, are discussed below.

Some Guidebook metrics apply to Ada without any need for modification. One such example is the maximum level of nesting of branch constructs within a module. The value of our branch nesting level metric is calculated by the rule, 1 / (N + 1), where N is the maximum nesting level of branch constructs in the module.

An example of modifying a Guidebook metric to satisfy units consistency is our metric for measuring the number of complex boolean expressions. The value of our complex boolean expression metric is calulated using the rule, N / (C + N), where N is the number of non-complex boolean expressions in the module, and C is the number of complex boolean expressions in the module. The Guidebook proposes the rule, C / S, where C is the number of complex boolean expressions in the module, and S is the number of executable statements in the module.

Our metric for measuring module flow top to bottom differs from the Guidebook's proposed metric, because of our desire to define strictly monotonic metrics when possible. The value of our top to bottom flow metric is calculated using the rule, 1 / (N + 1), where N is the number of branch constructs which cause backwards branching in the module. The Guidebook proposes a binary metric indicating the existence or absence of backwards branching in the module.

Some Guidebook metrics always have the value 1 for Ada. For example, the metric that measures the number of modified loop indicies within a module, is guaranteed to always have a value of 1 for Ada, since the loop indices of a for loop cannot be modified in Ada. This metric, which is based on the quality principle, "loop indices that are modified decrease simplicity", contributes to an overall measure of absolute quality, but it does not aid in measuring the proper use of Ada. However, a new quality principle, based on the previous one, that supports measuring the proper use of Ada is, "loops that are guaranteed not to have modified indices are simpler than those that are not."

A metric based on this principle is, (T - L)/T, where L is the number of loops that should be written as for loops, and T is the total number of loops. The precise meaning of "should be written as for loops" is beyond the scope of this paper. Nevertheless, the example illustrates how the tailoring of metrics to Ada can increase their usefulness for measuring Ada software.

Another example pertains to the mix of variables in a module. The Guidebook states, "From a simplicity point of view, local variables are far better than global variables." We defined two metrics based on variations of this quality principle. The first metric is defined by the rule (L / V), where L is the number of unique local variables referenced by the module, and V is the number of unique variables referenced by the module. This metric is identical to the metric proposed in the Guidebook. The second metric is defined by the rule (GL / G), where GL is the number of unique global variables referenced by the module that are declared in the body of the package containing the module, and G is the number of unique global variables referenced by the module. This second metric shows how special features of the Ada language allow quality distinctions not possible in other languages.

The Guidebook proposes measuring if the structure of the code reflects a top-down design, based on the quality principle, "each level of the tree structure should reflect a lower level of detail." We made this principle specific to Ada as follows: "If package P WITHs packages B1, B2, ..., Bn, then no package Bi should WITH a package Bj." Our desire for monotonic metrics caused us to count the number of packages WITHed by package P that violate the principle. The top-down metric for a package P which WITHs B1, B2, ..., Bn is defined by the rule (1 - (N / W)), where N is the number of packages WITHed by package P that satisfy the property that Bi is WITHed by Bj, and W is the number of packages WITHed by the package P. Since this metric depends only on the package structure of the Ada source code, the metric can be applied early in the coding cycle.

## 2.4 Formal Definition

The development of a formal definition for the metrics was approached by building a metrics-definition language that supports a set of predetermined goals.

The metrics-definition language was molded by the three goals described below.

1) The formal definition must explicitly describe a sufficiently wide variety of operational views of the metrics to allow the user to identify specific quality-related problems with the software. Kinds of views include decomposing metric values into the constituent parts, views of metric values by module or for a set of modules, and views of metric values by construct or for a set of constructs.

2) The formal definition must provide an unambiguous specification of the metrics that is easy to understand and modify. The formal definition is the specification for implementing the automated metrics data collection.

3) The formal definition must be executable to allow experimentation with the behaviour of the framework before implementation of the automated data collection.

The first goal deserves further clarification, since it is central to the ADAMAT design. The issues pertinent to this goal are best illustrated by example. Consider the metric, "Amount of fixed point types with a length clause for small", which we chose to measure, because fixed point types without a length clause for small are subject to variations in accuracy between compilers and machines. The metric is obviously a function of the number of fixed point types in a module and the number of length clauses for small in a module. However, the choice of how to aggregate these values into a metric for a set of modules is not obvious. Possible candidates are: average(i) (L(i) / F(i)), or (sum(i) (L(i)) / sum(i) (F(i))) for the ith module, where L is the number of length clauses for small and F the number of fixed point types. The formal definition must specify one of these two choices.

In most cases, for metrics of this kind, we selected the first definition, because the value of the metric for a set of modules is comprised of values of the same metric for the individual modules in the set (a property not shared by the second definition). This property supports our goal of decomposing metrics into their constituents in an understandable manner.

In spite of the selection of the first definition as the definition of this metric, it may be advantageous for the user to view components of the second definition. For example, the value, sum(i) (L(i)), might be useful, even though this quantity is never used in another metric. Consequently, we define L for a set of modules as a sum.

Aggregation of basic counts, such as the number of length clauses for small, is usually a sum, and aggregation of higher metrics is usually an average. Although these are the normal form of aggregation, our goal was a metrics-definition language that allows sufficient expressive power to express metrics that do not follow this pattern, as well as allowing expression of basic counts that are sums of other basic counts, and expression of weighted averages.

### 2.4.1 Metrics-Definition Language

We chose the logic programming language Prolog as the basis for the metrics-definition language. The metrics-definition language is expressed in Prolog notation, using a set of predicates designed to define metrics. The use of Prolog notation allows us to capitalize on the Prolog interpreter for the easy development of a set of automated support functions, collectively called the Framework Assistant. This section describes how the specialized predicates for defining metrics satisfy goals one and two, and the subsequent section describes how the Framework Assistant supports goals two and three.

Each metric is defined as a function using two predicates, "vdef" for defining the domains of the function, and "value" for defining the calculation. "vdef" is an n-ary predicate, where the first argument is the name of the metric, and the second through nth arguments are sets. For example, consider the definition of the metric, "Amount of fixed types with a length clause for small." The name of this metric is "aftwlcs", and the domain definition is:

```
vdef(aftwlcs,modules).
```

indicating that "aftwlcs" is defined for the set of modules. Each set is defined using the predicate:

```
set_definition(<set_name>,<list_of_elements>).
```

There are three predefined sets currently in the framework:

"modules", the set of modules in the analyzed system, is defined by the Data Collection Tool;

"constructs", the set of language constructs counted by ADAMAT, is defined as part of the formal definition;

"constructs_needing_comments", the set of language constructs for which comments are counted, is defined as part of the formal definition.

The "set_definition" predicate is the basis for providing the capability to define sets to the user.

The domain definitions are used to check that a query on the value of a metric is a legal query.

The predicate "value" is an n-ary predicate, where the first argument is the metric name, the second through (n-1)th are formal arguments, and the nth is the result of the calculation. The calculation is specified in two parts, an expression for the calculation, and a part for describing the values in the expression in terms of other metrics. The "value" predicate for "aftwics" is specified by the following rule:

```
value(aftwics,modules,Val) :-
1)  (for_all(module,modules),
2)      value(noc,module,length_clause_for_small,N),
3)      value(noc,module,fixed_point_type,T)),
4)  average((N/T)).
```

Line 4 contains the expression, and lines 1-3 describe the constituent values. The "for_all" operator on line 1 may be regarded intuitively as a subscripting operator, i.e.,

```
for_all(e,S) g(e)
    = {g(S(1)),g(S(2)),...,g(S(i)),...,g(S(n))}.
```

The scope of "for_all" is either the "value" predicates or the "for_all's" that follow.

The "value" predicate on line 2 states that N is the value of the metric "noc" ("Number of occurrences of a construct") for a given "module" and the construct, "length_clause_for_small". Finally, the "average" predicate is a combining operator that computes the average of the set of ratios (N/T) defined by lines 1-3. Two other combining operators are available, "summation" and "maximum".

The definition of "aftwics" is an aggregate of the form average(i) (N(i)/T(i)). The alternative aggregation discussed in section 2.4, i.e., (sum(i) (N(i))/(sum(i) (T(i))), may be expressed as follows:

```
value(aftwics,modules,Val) :-
   (value(noc,modules,length_clause_for_small,N),
    value(noc,modules,fixed_point_type,T)),
   N/T.
```

Aggregation of the values of "noc" over modules as a sum for "length_clause_for_small" and for "fixed_point_type" is deferred to the definition of "noc".

Note that the definitions for "aftwics" are not robust, since they are undefined for T = 0. Two predicates are provided for handling singularities. The predicate "if_then_else" is a 4-ary predicate; the first argument is a condition, the second is the value of the result if the condition is true, the third is the value of the result if the condition is false, and the fourth is the value of the result. A more robust definition of "aftwics" is:

```
value(aftwics,modules,Val) :-
   ((for_all(module,modules),
      value(noc,module,length_clause_for_small,N),
      value(noc,module,fixed_point_type,T)),
   if_then_else(T = 0, N/T, 1, V),
   average(V)).
```

The "such_that" predicate provides for refining the set of values to which the expression part applies. "such_that" is a unary predicate with a condition for its argument.

In addition to the function definition for each metric, the metrics-definition language provides for associating several other kinds of information with each metric. Figure 3 shows a complete definition of the metric "literals". Each of the predicates available for defining a metric are described below.

my_name The "my_name" predicate introduces the metric identifier and a longer, more descriptive name. The descriptive name is used for interfacing to the user.

description The "description" predicate provides a detailed English description of the metric.

rationale The "rationale" predicate allows the definer to document what aspect of software quality is measured by the metric, how the metric measures this aspect, and why the particular form of normalization was chosen. The rationale is made available to the user.

method_of_improvement The "method_of_improvement" predicate associates a description of changes that can be made to the software to improve the score on this metric.

Ada_related_improvement The "Ada_related_improvement" predicate specifies Ada language constructs that can be used to improve the metric score.

Relation_to_other_metrics The "related_to_other_metrics" predicate documents closely related metrics, e.g., metrics that may be influenced by software changes made to improve this metric.

Default The "default" predicate allows the specification of a default value for each data item. Default values for data items help control the volume of data produced by the Data Collection Tool. If a collected data item has the corresponding default value, the Data Collection Tool does not emit that data item. The Quality Analysis Assistant assumes the corresponding default value for any data item not emitted. Note that default values for data items are common. For example, consider the number of floating point types in a text processing program.

### 2.4.2 Framework Assistant

The Framework Assistant is a collection of utilities written in Prolog that support the understanding and modification of the formal definition. The utilities may be divided into three categories: interactive queries, automatic documentation, and automatic validation.

#### Interactive Queries

The Framework Assistant combines the interactive querying capabilities of Prolog with specialized predicates for manipulating structures in the metrics-definition language. This combination provides capabilities to access and manipulate the formal definition and results in a very flexible assistant. The user is not restricted to the querying operators developed specifically for the Framework Assistant. Essentially, any information that can be derived from the framework definition is available to the user.

The user of the Framework Assistant is assumed to have a basic understanding of Prolog. This is not a problem, since the Framework Assistant is used internally by a small number of metrics developers to modify the framework.

A full description of the queries available in the Framework Assistant is beyond the scope of this paper, but we can illustrate by example some of the capabilities. A user can ask directly about any of the predicates in the metrics-definition language. For example, the query

```
rationale(literals,X).
```

evokes the rationale for the metric "literals", i.e.,

```
X =
  ['The fewer number of literals used in a module',
   'the more breath of function for that module.',
   'Expanding the capabilities of a module that',
   'contains literals (eg., increasing array sizes',
   'or increasing precision) requires modification',
   'of all relevant literal usages.']
```

By asking

```
vdef(Metrics_id,Argument,constructs).,
```

the user can obtain one-by-one the metrics with constructs as the second argument, e.g.,

```
Metric_id = n_comments,
Argument = modules.
```

or, by asking

```
vdef(Metric_id,Argument,constructs),
description(Metric_id,Description).,
```

the user can obtain the descriptions,e.g.,

```
Metric_id = n_comments,
Description =
   ['The number of comment lines in a module',
    'for a given construct where blank lines',
    'and blank comment lines are not counted.'].
```

In addition, the "calc_desc" operator is available. "calc_desc" accepts one argument, a metric identifier, and returns a pretty-printed description of the metric calculation.

Automatic Documentation

The Framework Assistant automatically provides a complete set of documentation for the framework. Automatic generation of the framework documentation supports maintenance of up-to-date documentation. The "document_metric" command is the most important command for documenting the framework. Invoking "document_metric", with a metric identifier, produces a complete description of the metric, eg., the command "document_metric('literals')", produces the complete description of literals, shown in figure 3.

The metrics can be documented in various orders, depending on the particular documenting command. The command, "document_all_metrics", outputs the metrics in the order they appear in the framework definition. The command, "document_framework", outputs the metrics in the order of a depth-first traversal of the framework. The command, "document_sorted", outputs the metrics in lexicographical order.

Automatic Validation

The "validation" command checks that each metric has a domain definition ("vdef" predicate), a calculation ("value" predicate), and a description ("description" predicate), and that every used metric is defined.

## 3. Quality Analysis Assistant

The Quality Analysis Assistant is a collection of utilities, written in Prolog, that provides the user with analysis capabilites to pinpoint software quality problems. The analysis capabilities can be divided into two groups, the computation of factor, criteria, and metric values; and a user interface providing access to the factor, criteria, metric, and data item values that users need to isolate problems.

```
metric: literals = The Amount of literals in the source code.

The literals metric is based on the number of literals used in
the module. Literals include integer literals (eg., "32"), real
literals (eg., "2.."), enumeration literals (eg., "red"),
character literals (eg., "'c'"), and strings (eg., ""abc"").

literals(modules) =
  average( integerlits(i) + reallits(i) + enumlits(i) +
           charlits(i) + stringlits(i) + 1)
    for all i in modules
    where integerlits(i) is the number of integer literals
    where reallits(i) is the number of real literals
    where enumlits(i) is the number of enumeration literals
    where charlits(i) is the number of character literals
    where stringlits(i) is the number of string literals
```

The fewer number of literals used in a module the more breath of function for that module. Expanding the capabilities of a module that contains literals (eg., increasing array sizes, or increasing precision) requires modification of all relevant literal usages.

The software can be improved by inspecting each use of a literal within the module to determine if replacement of the literal by a constant, attribute, or parameter is appropriate.

Ada provides several features that allow a minimum usage of literals. Often, the FIRST and LAST attributes are appropriate substitutions for literals associated with the "in" and "for" constructs.

In some cases, a literal is appropriately replaced by a parameter with default; thereby, increasing the generality of the module without increasing the complexity of the interface for the simple cases.

In many cases, literals should be replaced by Ada constants.

Figure 3: Description of Literals

Values for the calculations in the framework are computed by directly executing the framework definition with the Prolog interpreter. Direct execution means that metrics developers can determine the behavior of new metrics immediately after the new metrics are defined. Several predicates are needed to support direct execution of the framework definition: rules that check if a requested value is a legal request, and check if a requested value is the result of an average, summation, or maximum operator, a basic data item, or a default value. The "collected" predicate indicates whether a data item is collected by the Data Collection Tool. If a data item is collected, but no data value for a given data item and domain is found, then the data item defaults to the default value. If a data item is not collected by the Data Collection Tool, then values may be supplied by the user. This approach supports incremental development of the framework definition and Data Collection Tool.

Our goal for the user interface was to provide convenient access to all of the values that can be computed from the data items in accordance with the formal definition. We felt this was essential for pinpointing software quality problems.

The value for all elements of the domain of each metric can be obtained by using the "ask" command, which is of the form,

```
ask(metric_id(arg1,...,argn)),
```

where arg1 through argn are predefined or user-defined sets. "ask" returns the value of the metric for the given argument list. The sets may be denoted explicitly by listing the elements or referred to by name. The "set_definition" command allows users to define new sets. Other commands allow users to obtain a list of values of a metric for the elements of a domain, and a list of values less than a threshold.

Repeated typing of metric_ids and argument lists, as required by the "ask" command, is inconvenient. Moreover, the user usually wants to know the constituent values that comprise a requested metric value, so quality problems can be isolated. The concepts of a stored "view" and a "breakdown" of a view increase the ease with which quality problems can be isolated.

A view is a metric_id associated with a set of arguments, i.e., "metric_id(arg1,...,argn)". A view may be stored by using the "set_view" command, "set_view (metric_id(arg1,...,argn))". The current view is shown by asking "current_view".

Once the current view is established, the user can obtain a break-down of the view by using the "breakdown" command. The response to a breakdown command is a metric value for the current view and metric values for the metrics at the next level down in the metrics hierarchy. Figure 4 shows the response to a breakdown command for the view "reusability(modules)". The breakdown command creates new views of the metrics that enter into the calculation of the value for the current view; in this case, generality(modules), independence(modules), etc. The argument lists of the new views are derived automatically from the argument list of the current view based on the metrics calculation.

The breakdown command allows users to decompose a quality deficiency into more specific deficiencies. In figure 4, poor simplicity and poor self-descriptiveness are shown as major detractors from reusability.

A user can further decompose a deficiency by using the "down(n)" command to select the nth view and go further down into the metrics hierarchy. For example, "down(5)" would break-down the values contributing to the value for simplicity. In addition to producing a break-down, the "down" command sets the current view to the selected view; in this case, "simplicity(modules)".

By using the "down" command to decompose metrics values and the "set_view" command to narrow or widen the domains under exploration, a user can conveniently isolate quality deficiencies to specific software characteristics (as expressed by metrics or data items) of a particular module or set of modules. Complementary commands facilitate exploring other quality deficiencies in the same run of the Quality Analysis Assistant, including the "up(n)" command, for returning to the nth previous view, and the "history" command, for listing a history of the views.

### 3.1 Use of Quality Analysis Assistant

The following scenario illustrates how the interactive analysis capabilities of the Quality Analysis Assistant can help a user pinpoint quality problems in a software product. This scenario also shows the benefit of a hierarchical framework structure and the importance of defining metrics in terms of both individual modules and sets of modules.

Suppose that a development manager, responsible for the implementation of a software library, is interested in the reusability of the software being developed. Initially, the manager can view reusability with respect to all modules in the library. Such a view, illustrated in figure 4, indicates that reusability is negatively effected by the generality, simplicity, and self-descriptiveness of the modules. Clearly, a deeper analysis of each of these three criteria is warranted. Assume that the generality of the modules is the first criteria selected for further examination. The view of generality, figure 5, indicates that the structural generality of the modules is low. A breakdown of the structural generality of the modules, figure 6, indicates that the use of literals is contributing negatively. A graph of the literals metric for each module, figure 7, indicates that module G has the worst use of literals. Further investigation, figure 8, reveals that module G contains several integer literals. Next the development manager might wish to determine if the literals usage is a recent problem. A graph of the number of integer literals in module G with respect to time, figure 9, indicates that the number of integer literals in G has recently risen sharply. Moreover, a graph of the number of literals in all modules with respect to time, figure 10, indicates that the recent rise in the number of literals is a system wide problem.

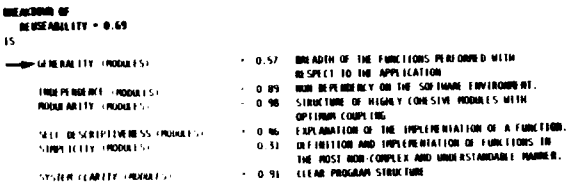



Figure 4: Reuseability

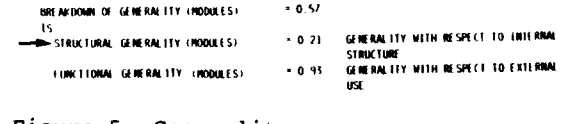



Figure 5: Generality

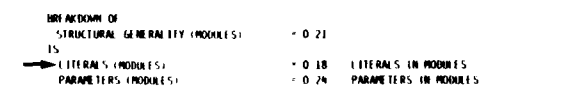



Figure 6: Structural-Generality

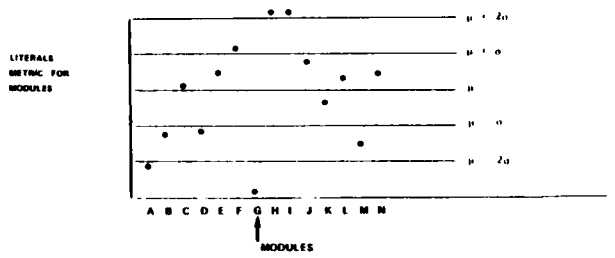



Figure 7: Literals Metrics for the Modules

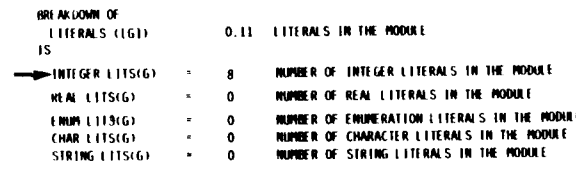



Figure 8: Literals

Now, this information can be combined with facts not in the framework, such as the project is in a new phase of software life-cycle or the personnel is different, so that a decision can be reached about the actual cause of the problem. In this case, the development manager concluded that the cause of the rapid rise in literal usage was related to an influx of new hires. Actions were then taken to locate and train the new hires responsible for the literals problem.

Movement back up the framework hierarchy would allow the examination of simplicity or self-descriptiveness to be conducted in a similiar fashion.

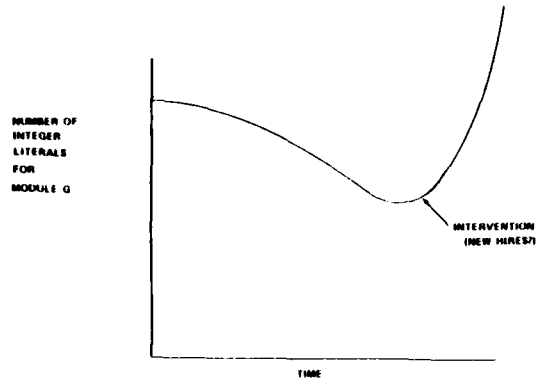



Figure 9: Integer Literals Usage in G

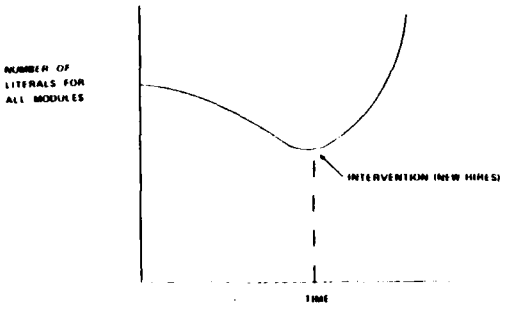



Figure 10: Literals Usage in the Modules

## 4. Data Collection Tool

The Data Collection Tool analyzes Ada source code and produces a set of values for data items. The data item values can be input into the Quality Analysis Assistant to support quality analysis. The Data Collection Tool was designed for high flexibility and high portability. High flexibility is a goal because of the need to add or modify data items collected by the tool. Metrics technology is still young, and as we gain insight into the metrics from experience, the metrics will be modified. Furthermore, our experience in applying metrics at DRC indicates that metrics are most valuable when tailored to specific project needs [San Antonio83]. A salient example of the need for tailoring is the collection of data on Ada program design languages, many of which incorporate annotations for addressing issues not supported by Ada.

High portability is a goal, because we want the Data Collection Tool to run in the development environment *whenever possible. Measurement within the development* environment improves turnaround time and simplifies configuration management.

The desired portability can be accomplished by using Ada as the implementation language and avoiding the use of the non-portable features of Ada. The desired flexibility requires a more sophisticated approach, described in the next section.

### 4.1 Compiler Generation Techniques

*We used several compiler generation tools to* automatically generate the Ada implementation from high-level description languages. The conciseness of and the descriptive rather than procedural nature of the high-level description languages makes them easy to modify.

Lexical analysis and parsing of the Ada source code by the Data Collection Tool are supported by well-known compiler generation tools, a scanner generator and parser generator, respectively [Aho78]. The regular expressions, input to the scanner generator, were newly developed based on chapter 2 of the Ada Reference Manual [Ada83]. We found the use of a scanner generator *beneficial for the Ada language, particularly because* the wide variety of numeric literal tokens were easily described using regular expressions. The Ada grammar we used is based on the LALR(1) grammar developed at NYU [Charles82]. Several modifications were made to distinguish between constructs to support measurement of data items.

A pattern recognizer was developed explicitly for recognizing and counting data items. The pattern recognizer reads in a high-level description of data items. Each data item is described by a data item identifier, a counting operator, and a set of recognizing operators. The operators are specified in terms of productions in the LALR(1) grammar. Counting operators specify the value of the data item count for the left-hand-side symbol of all productions, in terms of the data item counts for all symbols of all productions. There are two counting operators in the description language at this time: add, for adding data item counts, and max, for taking the maximum of data item counts.

The recognizing operators specify initialization of data item counts, usually corresponding to the recognition of patterns. The recognizing operators specify the value of the data item count for the left-hand-side symbols of a particular production, in terms of the data item counts for the right-hand-side symbols of that production. Note that although counting operators apply to all productions, recognizing operators apply to a particular production. Essentially, recognizing operators specify when a production is part of a pattern that should be recognized.

There are currently six recognizing operators: "one", "asn", "incr", "comments", "name", and "zero". Each of these is described below.

*one(<production>)*

Specifies that the data item count for the left-hand-side of the production be set to one.

*asn(<production>,<right-hand-side>,<data item>)*

Specifies that the data item count for the left-hand-side of the production be set to the <data item> count of the right-hand-side symbol.

*init(<production>,<right-hand-side>,<data item>)*

Same as asn, except increments the <data item> count by 1.

*comments(<production>,<right-hand-side>)*

Specifies that the data item count for the left-hand-side of the production be set to the number of comments immediately following the right-hand-side symbol.

*name(<production>,<right-hand-side>,<name>)*

Specifies that the data item count for the left-hand-side of the production be set to 1 if the <right-hand-side> equals <name>.

*zero(<production>)*

Specifies that the data item count for the left-hand-side of the production be set to 0.

The following example illustrates the description of a data item for the maximum number of case alternatives "whens" in an Ada case statement.

```
whens_in_case(add);
    one(case_statement_alternative);
    zero(case_statement);

max_whens_in_case(max);
    asa(case_statement,4,whens_in_case);
```

The relevant productions of a case statement are:

```
case_statement : :=
    'case' <express.on> 'is'
        <case_statement_alternative_list>
    'end' 'case' ';' ;
<case_statement_alternative_list> ::=
    <pragma_str> <case_statement_alternative_list>
    <case_statement_alternative> <case_stmt_alt_list>;

<case_stmt_alt_list> ::=
    <case_stmt_alt_list> <case_statement_alternative>
    <eps...on> ;

<case_statement_alternative> ::=
    'when' <choice_list> '=>'
        <sequence_of_statements> ;
```

"whens_in_case" is a data item count of the number of case statement alternatives in a case statement. The "one" operator ensures that each "case_statement_alternative" is counted. The "add" operator specifies that the counts should be added for all of the productions. Hence, the data item count for "case_statement_alternative" (which is 1) is added to the count for the list at "case_stmt_alt_list" and "case_statement_alternative_list". Finally, the "zero" operator zeroes the data item count at the "case_statement", so the number of "case_statement_alternatives" is only accumulated for each case statement, not for the all of the source input.

"max_whens_in_case" is the desired data item count. The "asa" operator collects the number of "case_statement_alternatives" for each case statement. The maximum is computed by "max"ing the "whens_in_case" count over the entire source input.

The implementation of the pattern recognizer can be approached differently, depending on whether the implementation is a product or internal development implementation. During internal development, the data item description language is read in by the Data Collection Tool and incorporated into tables for performing pattern recognition. Using this approach, data items can be added easily without recompilation simply by adding their description to the description language. Alternatively, a generator can read in the data item description language and produce the tables at generation time, to avoid the run time overhead of reading the input and creating the tables. The data item description language can be translated into an S attributed grammar [Lewis74]. Translator writing systems for generating implementations of attribute grammars are available [Farrow82, Payton82].

The data item description language can easily be enhanced by adding more counting and recognition operators. Recognition operators for defining data item counts of right-hand-side symbols can be supported by both implementation approaches. The dynamic evaluation of the data item description language can be supported using techniques described in Fang72, Kennedy79]. Automatic generation of efficient implementations is supported by existing translator writing systems.

## 5. Status

We have developed metrics that address the following criteria of software quality: completeness, generality, independence, self-descriptiveness, simplicity, system clarity, specificity, and functional scope.

The Data Collection Tool supports automated data collection of the basic data items for the above criteria. The Framework Assistant and Quality Analysis Assistant currently implement the capabilities described in the paper.

## 6. References


[Ada83]
"Reference Manual for the Ada Programming Language", ANSI MIL-STD-1815A, United States Department of Defense, Washington D. C., January 1983

[Aho78]
Aho, A., Ullman, J. "Principles of Compiler Design", Addison-Wesley Publishing Company 1978, Reading, Mass. pp. 246

[Booch84]
Booch, Grady, "Dear Ada", Ada Letters, March 1984, pp. 29-32.

[Bowen83]
Bowen, T. P., Post, J. V., Tan, J., Presson, P. E. Schmidt, R. L., "Software Quality Measurement for Distributed Systems Guidebook for Software Quality Measurement", Technical Report RADC-TR-83-175 Volume II, July 1983.

[Dunham83]
Dunham, J. R., Kruesi, E., "The Measurement Task Area", IEEE Computer, November 1983, pp. 47-54

[Charles82]
Charles, P., Fisher, G., "A DALR(1) development grammar for 'Ed' Ada", Ada Letters, III:2 (1982), pp. 34-46

[Fang72]
Fang, I., "FOLDS, A Declarative Formal Language Definition System.", Technical Report STAN-CS-72-329, Stanford University, 1972.

[Farrow82]
Farrow R., "LINGUIST-86: Yet another translating writing system based on attribute grammars", Processing Notice 17, 6 (1982), pp. 161-171

[Kennedy79]
Kennedy, K., Ramanathan, J., "A Deterministic Attribute Grammar Evaluator Based on Dynamic Sequencing", ACM TOPLAS 1,1(79.

[Lewis74]
Lewis, P., Rosenkrantz, D., and Stearns, R., "Attributed translations." Computer and System Science, Vol 9, No. 3, December 1974, pp. 279-307

[Payton82]
Payton T., Keller S., Perkins J., Rowan S., Mardinly, S., "SSASS: A syntax and semantics analysis and generation system", Proceeding of COMPSAC 82(1982), pp. 424-433

[San Antonio83]
San Antonio, R. C., Jackson, K. L., "Application of Software Metrics During Early Program Phases", National Security Industrial Association Conference on Software Test and Evaluation, Washington D. C., February 1983.

[Walsh79]
Walsh, T. J., "A Software Reliability Study using a Complexity Measure", Procedings of the National Computer Conference, New York, 1979


## About the Authors


**Mr. Steven E. Keller manages the Software Research and Development Group at Dynamics Research Corporation. He has a Bachelor of Science degree in Biology from the University of Colorado and Master of Science degrees in Computer Science and Biology from the University of Oregon. Mr. Keller has been involved in the development of compilers for multi-processor scientific computers and was the principal investigator for the development of an ordered-attribute-grammar based translator writing system.**

**Mr. John A. Perkins is a member of the Software Research and Development Group at Dynamics Research Corporation. He has a Bachelor of Science degree in Mathematics from Purdue University and a Master of Science degree in Mathematics from the University of Illinois. Mr. Perkins has been involved in the development of translators for multi-processor scientific computers and in the development of a attribute grammar based translator writing system.**

# TRANSITIONING TO ADA: THE CHALLENGE FOR SOFTWARE ENGINEERING

THOMAS J. WALSH

TELEDYNE BROWN ENGINEERING
TINTON FALLS, NEW JERSEY

## ABSTRACT

Ada* is an evolutionary advance in programming languages and serves as a catalyst for a radical change in the landscape of software engineering. The Ada technologies now emerging have dramatically shifted software engineering's focus from acceptance of any adequate solution representation to a careful examination of the complex problem solving process itself. However, this sudden shift of emphasis has exposed multiple gaps between inflated expectations for the new technologies, and the availability and mastery of these technologies within the software engineering domain. At present, these gaps in technology introduce significant risks which must be addressed by project management. Clearly, software engineering must mature rapidly to allow software projects to derive the full benefits of the Ada technologies.

The key software engineering components effected can be classified in problem solving parlance: development methods, tool availability, and recognition and mastery of problem solving environment. The components are examined highlighting issues and concerns which must be addressed for successful project management. Additional efforts are seen as necessary to ensure Ada's evolutionary advance. Industry especially must commit significant resources to pragmatic research and Ada product development. The planned products of one company (Teledyne Brown Engineering) for 1985 are described.

## INTRODUCTION
## ADA's IMPACT ON THE SOFTWARE CRISIS AND SOFTWARE ENGINEERING

The terms software crisis and software engineering are used extensively in the literature[1-5]. Software crisis has become synonymous with the aggregate of difficulties associated with large software systems. Software engineering is the technical discipline whose mission is to resolve this crisis and allow the production of cost-effective, reliable, and maintainable software products. Although these are very popular terms, there is often some confusion as to the essential natures of each and the relationship between them. Thus, a short discussion of the software crisis and software engineering seems justified to establish the proper perspective for viewing Ada's impact.

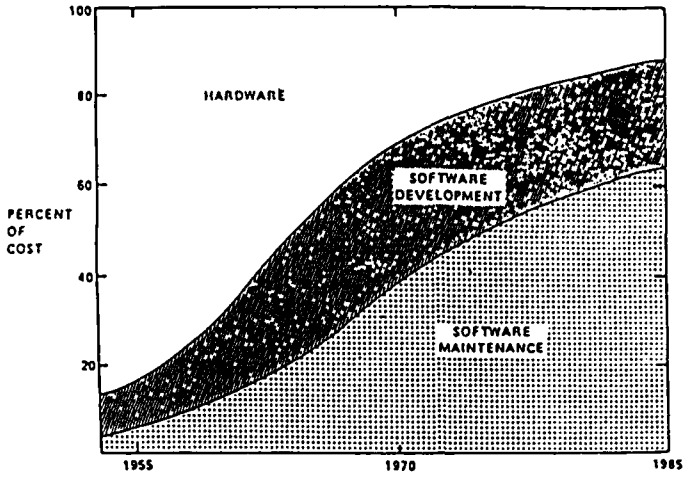
The software crisis has two salient characteristics (quantitative and qualitative), and a number of underlying causes. The quantitative characteristics of the crisis can be well illustrated by two popular graphs. Mr. Boehm's graph[6] nicely illustrates (Figure 1) the inversion of the hardware/software cost relationship over a thirty-year interval. The correlated rise in maintenance costs is also significant and provides supporting testimony to the lack of software engineering rigor. The second exponential graph[7], shown in Figure 2, portends an ominous future if a rigorous and even radical response is not manifested by software engineering. On the positive side, these exponential curves represent powerful economic forces for the rapid evolution of software engineering technology.

Qualitative characteristics or typical complaints include:

- Cost is excessively high and often unpredictable by software cost estimation models.
- Deliverables are often late and suffer a shortfall of capabilities in relation to the perceived original requirements.
- Software products do not inspire confidence in reliability, portability, and maintainability.

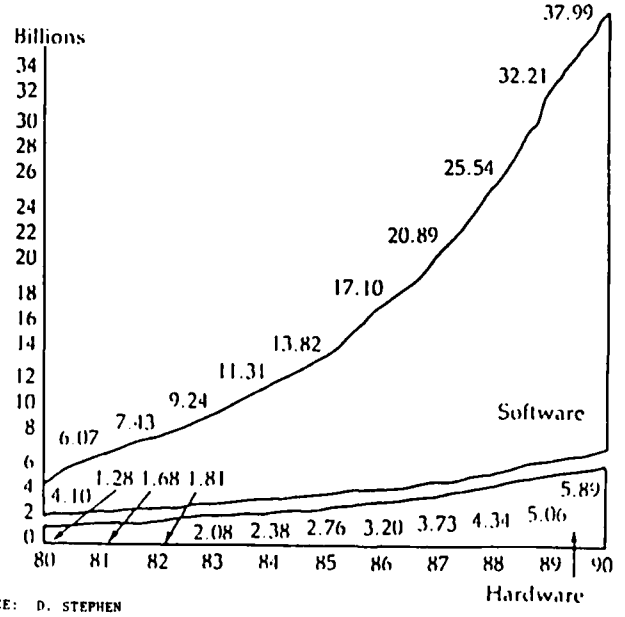
*Ada is a registered trademark of the U.S. Government - Ada Joint Program Office

FIGURE 1 - COST TRENDS FOR SOFTWARE AND HARDWARE




SOURCE: B. BOEHM

FIGURE 2 - FUTURE DOD SOFTWARE AND HARDWARE COSTS




SOURCE: D. STEPHEN

- Software systems are becoming larger, more ambitious and more complex.

- Personnel turns over at a rapid rate (less than 2 years).

Although these characteristics are often cited, the underlying causes also need to be identified. These include:

1) Proliferation of programming languages.

2) Lack of portability in programs, programmers, and technology.

3) Lack of skills in project personnel.

4) Requirements that are ambitious, ambiguous, and often changed.

5) Design that lacks precision, structure, and abstraction.

6) Code that lacks precision, structure and rigor.

7) Testing that does not ensure a minimum level of reliability.

8) Maintenance practices which undermine system reliability.

9) Documentation that is outdated and inadequate.

10) Software products that lack visibility and control.

11) Proposed system missions that are not well defined or excessively complex.

12) Non-recognition of human limitations for precise logical thought.

13) Immature state of software engineering technological foundations.

14) Immature state of software engineering training curriculum.

15) Lack of program support environments.

16) Lack of automated support tools.

Clearly, there are multiple and diverse causes underlying the software crisis. It follows that the standardization on one programming language (no matter how powerful) will not directly solve all these problems. It will need to be supplemented by other software engineering technologies that span the problem solving domain. This is precisely what is happening in the Ada technology area.

Ada's impact on the software crisis and its contribution to software engineering is through the recognition of software as an intellectual activity with many similarities to problem solving as performed in any engineering discipline. Relevant problem solving books are listed as references[8,9,10]. Key ingredients for problem solving include availability of tools, methods of problem representation and solution, and recognition and mastery of the problem environment. This conceptual framework allows us to view the Ada software engineering response to the above underlying causes, as follows:

- Software Tool Availability (1, 15, 16)

- Software Development Methods (4, 5, 6, 7, 8, 9, 10)

- Recognition and Mastery of Problem Environment (2, 3, 11, 12, 13, 14).

## WHY ADA IS AN EVOLUTIONARY ADVANCE IN SOFTWARE LANGUAGES

Ada is an evolutionary advance in programming languages and serves as a catalyst for radical change in the landscape of software engineering. This strong assertion is supported by the following reasons:

- Ada contains a collection of unique advanced features.

- Ada contributes to greater software structure and abstraction.

- Ada features and constructs support its use as a design language.

- Ada shifts software focus from a solution representation to the problem solving process.

- Ada is reforging the technical components of software engineering.

- Ada is inspiring hardware and software methodology convergence.

Ada has the capability (data structures and control structures) of most preceding general purpose high order languages plus a unique set of advanced features. These features[11] are mostly new and include:

- STRONG TYPING

  - Logical partitioning of objects by types

  - Precise set of acceptable values and operations

- o PACKAGES
  - Collection of related entities
  - A building block component analogous to the hardware chip
- o TASKS
  - Concept allowing for problem representation via parallel process execution
  - Communication and synchronization provided via rendezvous mechanism
- o GENERICS
  - Common logic created from a single template independent of type
  - Generic subprogram unit may be a subprogram or a package
- o EXCEPTION HANDLING
  - Ability for software to recover from the unexpected
  - Provide handler routines for system and user defined exceptions

Software languages have traditionally served multiple purposes. First, they contain specific instructions to tell the computer what to do. They also serve as a means of communication among project members and perhaps most importantly, they serve as tools of thought for designers. As a tool of thought, the concepts of abstraction and structure are powerful concepts. There are multiple precedents for software language evolution. As Figure 3 shows, the history of programming languages can be viewed as a continuous process of embedding abstraction and structure capabilities in languages so that human beings would be more comfortable with the communication medium in representing problems and implementing solutions. Barnes[12] refers to evolutionary advances in terms of abstraction (expression abstraction, control abstraction, data abstraction). Shaw[13] provides further elaboration on the value and necessity for abstraction in languages.

Ada incorporates the design principles identified in the sixties and seventies into its features and constructs[14]. Consequently, the Ada programming language is a very attractive vehicle for use as a textual design language. Presently, there are substantial advocates for using Ada as a design language in various phases of the product life cycle including software design, software specifications, software requirements, and system design[15,17].

Ada shifts the software focus from the acceptance of any adequate solution representation to a careful examination of the complex problem solving process itself. This shift is due to the strength of the language (abstraction, structure, and expressive power) and the multiple components of the Ada technology (environment, design language, methodology, and automated tools). Ada is an excellent design language with facilities supporting interface specifications, concurrency and multi-functional black boxes (packages). So pervasive is Ada's impact on software engineering that it is literally reforging the discipline with major emphasis on the following technological areas:

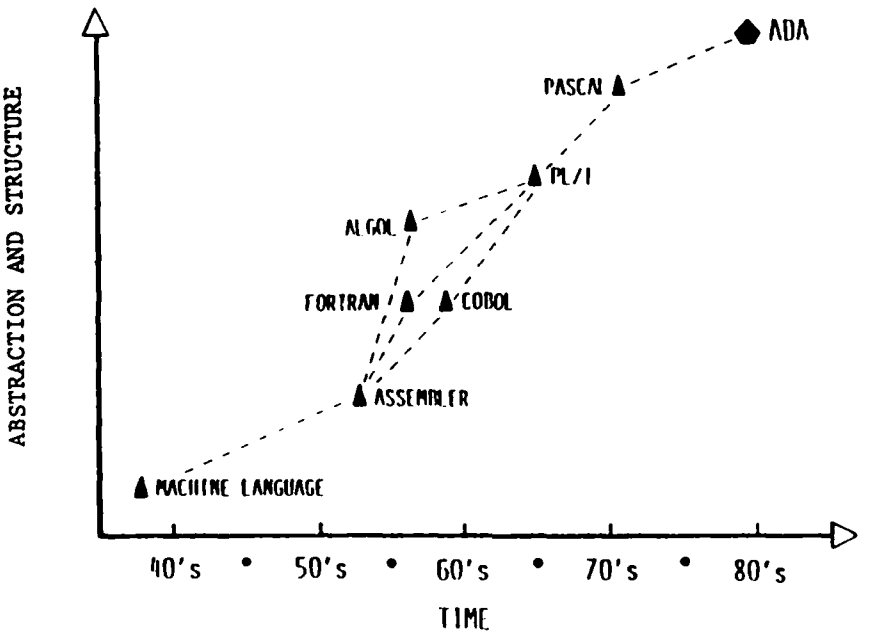
- o Software language features and constructs
- o Sophisticated error checking compilers
- o Programming support environment
- o Design language
- o System development methodology
- o Automated support tools
- o Training and education

Finally, Ada is inspiring hardware and software methodology convergence[15,16]. For the first time, a general purpose programming language supports well-defined interface specifications, package specifications and bodies, concurrency representation with facilities for synchronization, and communication. Ada can be used to represent system design before the hardware/software tradeoffs are evaluated to determine the implementation medium.

With these constructs in the software language, the hardware and software methodologies can converge toward a system methodology. Corresponding graphical and mathematical notation promises to allow computer system design to be far more rigorous and independent from implementation considerations.

In summary, due to the broad spectrum impacted by Ada and its associated technologies, project management must seek a thorough understanding of the Ada technologies and the problem solving domains addressed. This knowledge should be applied to ensure the successful execution of Ada projects.

FIGURE 3 - PROGRAMMING LANGUAGE EVOLUTION PRECEDENTS




THE CHALLENGE FOR SOFTWARE ENGINEERING

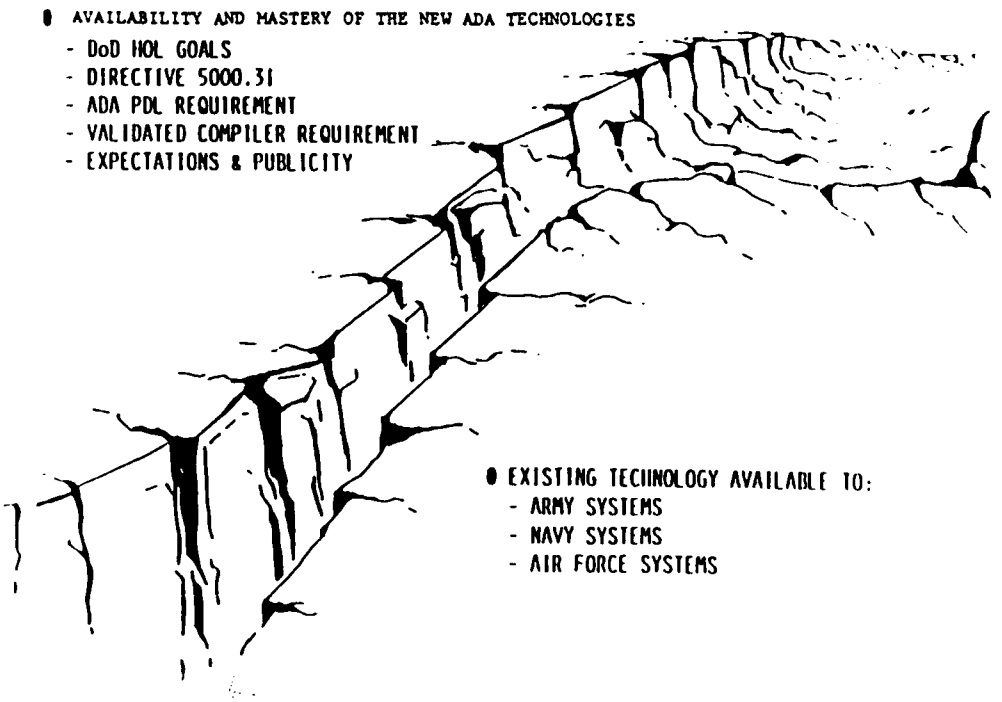
*The challenge for software* engineering is to provide the tools, methods, and pragmatic knowledge of the overall problem environment needed to completely master software's abstract and complex nature. The multi-faceted Ada technologies now emerging represent a challenge of significant scale and dimensions for the software engineering discipline. Ada, unlike preceding software languages, has shifted the software engineering focus from *acceptance of any adequate* solution representation to a careful examination of the complex problem solving process. However, this sudden shift of emphasis has exposed multiple gaps between inflated expectations for the new technologies, and the availability and mastery of these technologies within the software engineering domain (Figure 4). In fact, these gaps in technology introduce significant risks which must be addressed by the project manager for Ada related projects.

Clearly, software engineering must mature rapidly in a number of key technological areas to allow software projects to derive the full benefits of the sophisticated Ada language and its associated technologies. The key technological components effected are: compiler construction, programming support environments, design languages, system development methodologies, automated tools, and the software engineering curriculum to support these essential technologies. These areas can be further classified in problem solving parlance:

- o Tool Availability (Compilers, Support Environments, Automated Tools)
- o Development Methods (Design Languages, System Development Methodologies)
- o Recognition and Mastery of Problem Environment (Software Engineering Curriculum)

The Ada technologies promise many benefits, but also require that these key components of software engineering and the many interrelated issues and concerns be addressed by project management. A key point to be emphasized and understood is that the issues described herein would have existed even if Ada did not emerge. They have been present for some time, but were often eclipsed by other inadequacies inherent with language proliferation, the abstract nature of software, and confusion concerning the problem solving software process. Ada has simply highlighted their presence and illustrated the inadequacy of software engineering today.

FIGURE 4 - THE SOFTWARE ENGINEERING TECHNOLOGY GAP




## ADA PROBLEM SOLVING METHODS

Problem solving capabilities strongly correlate to a knowledge of methods, availability of tools, and accurate perception and mastery of the problem environment. Ideally, problem solving methods should precede and forge the characteristics of the problem solving tools. In practice, however, an iterative process usually replaces such an exact ordering especially with the advent of a new technology such as Ada. This section explores the role of methods, Ada's enhancements to former practices, and unresolved issues.

The role of software problem solving methods is twofold. The first role is to assist in problem analysis and definition; the second role is to assist in an iterative, precise, and eventually complete solution formation. A key point is that methods represent a process that results in a documented product. The recorded product or output from such methods is known as documentation and may take the form of an B-specification, data flow diagram, or design language description. Thus, methods represent the thinking process that influences the final form.

Ada is having a large impact in the area of methods. This is a direct result of the consistent integration and enforcement of the design principles identified in the recent past into Ada. Ada strongly supports many forms of abstraction, information hiding, and modularity[5,14]. Adherence to these principles and the availability of considerable expressive power contribute to Ada's natural use as a design language. There are many benefits that result from using Ada as a design language including:

- o Utilization of Ada's powerful constructs to influence software architecture
- o Enhancement of project communication by using the same language notation in multiple phases of the life cycle
- o Analyzability of design by Ada automated tools (compilers and analyzers)
- o Encouragement of design rather than coding on projects

- Establishment of a bridge between methodology and code

At present, several corporations and universities have developed design languages[18,19]. The number and divergence of these design language definitions emerging represent another possible proliferation problem. Thus, the Institute of Electrical and Electronics Engineers (IEEE), through the Working Group on Ada as a Design Language, is developing a Recommended Practice to provide pragmatic guidance for the project manager[17] in evaluating or developing a design language. This document represents a technical consensus among the working group members from industry and government.

There are many unresolved issues and concerns with the use of Ada as a design language[19,20]. A few deserve comment here since they are acting as serious impediments to progress. They include:

- Lack of a standard for use of Ada as a design language
- Form and content divergence among proposed Ada design languages
- Life cycle phase applicability of an Ada design language
- Relationship between the design language and the implementation language
- Relationship between system design methodology and the design language

The last two points require further elaboration. At present, a number of systems are employing Ada as a design language, but they are utilizing a different implementation language. Such a case implies a lack of one-to-one correspondence between features and constructs of the design language and implementation language. In this situation, a software standards document should describe guidelines for mapping of the design reflected in the design language to the implementation language. The last issue addresses the larger mapping problem between the system design methodology and the design language. The design language must serve as a critical bridge between the project's system design methodology and the implemented source code. Again, the software standards document must address this relationship with guidelines and evaluation criteria.

Ada is also reforging the area known as system design methodology. A system design methodology can be viewed as a set of methods, practices, techniques, and heuristics which govern the way in which the software system is derived. Work in this area is supported by the Ada Joint Program Office. The results are in a document known as METHODMAN[21] and is part of the DOD STARS program[22]. One of the largest impediments to be overcome in this area is the large number of competing methodologies which must be examined. A British Department of Industry study[23] reveals 37 unique methodologies. An examination of the literature and corporate proprietary products reveals hundreds, if not thousands, of unique variations. System design methodologies vary in their form (textual versus graphical), life cycle applicability, machine processibility support factors, and their orientation toward data flow, data structures, control flow, control structures, or some combination thereof.

Issues of the compatibility of Ada to a methodology are often due to the fact that the methodology was derived before the advent of Ada. Accommodation of Ada's unique features such as tasking, data abstraction, and generics must be reflected in design methodology techniques. For instance, structure charts do not usually possess a representation for the tasking construct. Outstanding methodology issues include the following:

- Compatibility with the Ada language
- Proven use on production projects
- Ease of use by managers and programmers
- Support for automated tools
- Applicability to all life-cycle phases
- Compatibility with an Ada design language
- Support for project documentation
- Recognition of inherent methodology limitations

In summary, Ada problem solving methods include Ada as a design language and an Ada compatible system design methodology. These represent the two technological components of software engineering which will be most difficult to mature into a rigorous discipline in support of Ada. Eventually, classes of compatible system design methodologies should be established which view the Ada design language as a critical bridge to reach the Ada implementation code.

## ADA PROBLEM SOLVING TOOLS

Beside the essential tool of a good programming language, software engineers need the assistance of problem solving tools. Ada problem solving tools can be found in the areas of compilers, support environments, and automated tools. Although often presented as separate entities, they are actually quite

interrelated, sharing host/target architecture, data base, and operating system. The role of these tools include the amplification of the human being's capability to manipulate yet another tool - the computer. This section discusses these three areas and highlights unresolved issues.

Ada compilers translate legal Ada syntax (ANSI/MIL-STD-1815A) into machine language for a given architecture. The DOD has established a process of compiler validation to guard against language subsets and extensions. The lack of production quality compilers for a wide variety of machine architectures to date has been the most significant impediment to the widespread usage of the Ada language. The project manager needs to also address the following compiler issues:

- o Completeness of Ada features supported
- o Validation status of selected compiler
- o Speed of compilation and execution
- o Clarity of diagnostic messages
- o Dependence on run-time environment
- o Efficiency and optimization of generated object code
- o Cost effectiveness and reliability factors
- o Limitations and issues of the validation process

This last compiler issue (validation) actually has a set of issues associated with it including:

- o Validation represents only a sample group (presently 2000) of all possible tests of feature combinations.
- o Compilers can only be validated for host. (The original host-target pair concept has not proved workable in practice)
- o ACVC tests do not measure performance or capacity factors.
- o Some legal considerations are unresolved.

The next area of Ada problem solving tools is support environments[24]. The support environment is to serve as a programmer's workbench providing automated tools for reliable system construction from the design to the maintenance phases.

This group of tools at present is mostly system level production tools (Figure 5). It should be recognized, however, that Stoneman is an interpretive document and environments will differ in their available tool and especially in their implementation of these tools. A key concept of the Ada environment is that it is extendable as new tools emerge in the future. The project manager needs to address the following environment issues:

- o Availability of support environments
- o Compatibility with Ada compiler
- o Compatibility with Stoneman requirements
- o Completeness of APSE tool set
- o Lack of well-defined standard interfaces between KAPSE and MAPSE thus preventing program portability
- o Ease of use and training requirements
- o Cost effectiveness and reliability factors

The last area discussed is automated tools. It could be argued that these are actually part of the extended environment. At present, however, these candidate automated tools are closely linked to the syntax of Ada and are more application oriented. Sample automated tools might include the following:

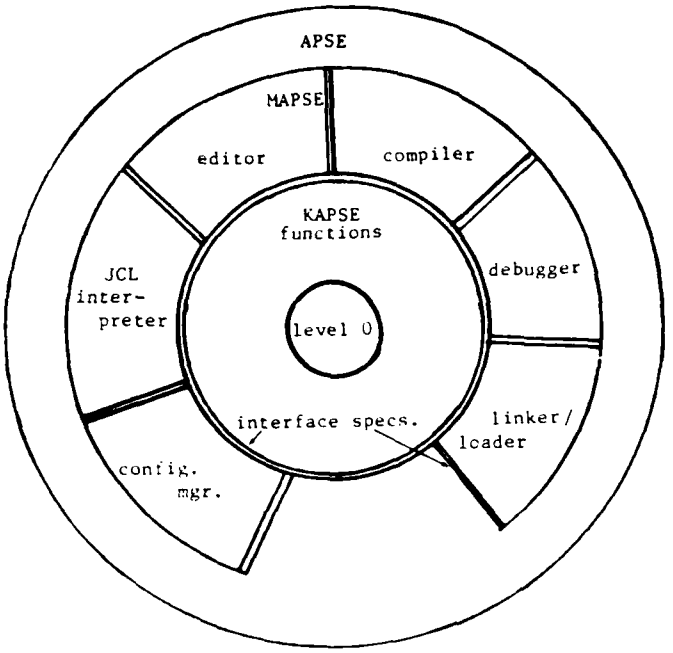
- o Object and Type Cross-Referencer
- o Project Completeness Analyzer
- o Package Dependency Analyzer
- o Automatic Test Case Generator

The issues associated with these automated tools include:

- o Availability of automated tools
- o Compatibility with chosen support environment
- o Ease of use
- o Cost effectiveness and reliability factors
- o Compatibility with host architecture

In summary, Ada problem solving tools are interrelated and have a number of important issues associated with them. The rich syntactical complexity and multiple component parts of Ada contribute to formidable strength, but will also require a significant number of automated tools for human beings to precisely manipulate and control. It can be reasonably expected that application level tools will rapidly mature in the near future.

FIGURE 5 - BASIC ENVIRONMENT AS REQUIRED BY STONEMAN 1980




## ADA SOFTWARE ENGINEERING CURRICULUM

The educational activity required for the effective training of software personnel to master the power of Ada technology is significant. In fact, the educational challenge is to provide the additional training required by the Ada language and its impact in the entire domain of software engineering. Educational training is seen as necessary in at least the following areas:

- o Language Concepts and Pragmatic Usage
- o Ada as a Design Language
- o Ada Compatible System Design Methodology
- o Ada Environment and Automated Tool Usage

The consensus among educators is that hands-on time with a responsive compiler is invaluable.

The educational issues which project management must address include:

- o Availability of training courses
- o Number of courses required for each category of project personnel
- o Amount of time allocated to each course
- o Sequence of courses in Ada curriculum
- o Evaluation criteria to determine cost-effectiveness
- o Hands-on percentage of course and responsiveness of the compiler used
- o Mode of instruction (lecture, films, CAI)

In summary, an Ada software engineering curriculum is essential to support the multi-faceted Ada technologies. The amount of training for Ada and its related technologies will be significant and will, in fact, be similar to that for other engineering disciplines. The software engineering curriculum is the key to the mastery of the problem recognition and problem solving environment.

## SUMMARY AND CONCLUSIONS

Ada is far more than just another programming language and, in fact, represents a significant evolutionary advance in software engineering technology. The unique combina-

tion of advanced language features and the associated technological components identified in this paper have the distinct capability for changing the landscape of software engineering. In addition, Ada's impact cascades into the field of computer systems engineering.

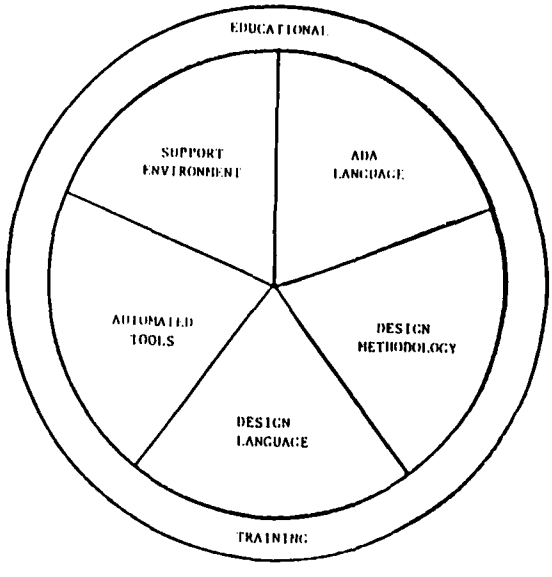
Ada, unlike preceding software languages, has dramatically shifted software engineering's focus from acceptance of any adequate solution representation to a careful examination of the complex problem solving process itself. However, this sudden shift of emphasis has exposed multiple gaps between inflated expectations for the new technologies, and the availability and mastery of these technologies . At present, these gaps in technology introduce significant risks which must be addressed by the project manager for Ada related projects. Clearly, software engineering must mature rapidly in a number of key areas to allow software projects to derive the full benefits of the sophisticated Ada language and its associated technologies.

It has been advocated here that project management utilize problem solving concepts as a framework for viewing the emerging Ada technologies. Key factors include methods, tools, and educational training. Problem solving methods include design languages and system design methodologies. Problem solving tools include compilers, support environment tools, and automated application tools. An Ada software engineering curriculum is essential to support the multi-faceted Ada technologies. In addition to significant training, pragmatic research is also necessary to resolve issues and fully master Ada technologies (Figure 6).

The largest impediments to problem solving are the limitations that problem solvers place on themselves. Software problem solving is no exception. Too often, simply arriving at an adequate minimal solution is the goal of software projects. A direct result of this limited effort is software systems which are not maintainable. With adequate training, methods, tools, and creativity, software people can achieve maximum expectations and make quantum leaps in productivity.

To this author, the Ada technologies represent a unique challenge to forge a disciplined software engineering field with maturity similar to other established engineering fields. This would necessitate more rigorous standards and a firm quantitative basis. Software engineering must rest on precise mathematical foundations. Pre-

FIGURE 6 - PRAGMATIC RESEARCH AREAS FOR SOFTWARE ENGINEERING



viously, there has been considerable work done in advocating and establishing a mathematical basis for software[25-28]. These mathematical foundations need to be extended into the domain of Ada technologies. In addition, software metrics[29], especially those with a firm mathematical basis, need to be applied to Ada software in an effective manner. The importance of mathematical abstraction in software problem solving should not be overlooked. Mathematics can serve as both a tool and as a method.

Ada will be widely utilized beyond defense applications in the commercial sector. Jean Ichbiah[30] cites productivity, reliability, and maintainability as strong motivating factors for commercial acceptance. When this occurs, academic institutions will need to strongly integrate the Ada technologies into their curriculum to prepare students for the workplace of tomorrow. Ada has the potential to help reduce the risk described by the National Commission on Excellence in Education, "the educational foundations of our society are presently being eroded by a rising tide of mediocrity that threatens our very future as a Nation and a people"[31]. The Ada philosophy and technology provides a firm foundation for the software engineering discipline.

Finally, none of the issues identified in this paper should be construed as an invitation to despair, but as a challenge to software engineering to establish firm foundations. Additional efforts and cooperation by industry, government, and academia are necessary to ensure Ada's evolutionary advance. Although each of these sectors has a unique role, industry especially must commit significant resources to pragmatic research and Ada product development in the outlined technological components. This should be done not only to aid software engineering in meeting the Ada challenge, but to ensure the technological lead of the United States in an increasingly competitive world.

## TELEDYNE BROWN ENGINEERING'S ADA INITIATIVES

Teledyne Brown Engineering is committed to a series of Ada Initiatives to build Ada products through internal research and development funds. The primary motivation for this effort is to regenerate and enhance the skills of technical personnel, and to maintain and expand the company's traditional business base. The company sponsored initiatives fall into two main areas, educational courses and automated tools.

The educational courses (each of one week duration) comply to an educational philosophy which includes the following:

- Adhere to the pragmatic approach
- 50% of course hands-on computer
- Proceed from the familiar to the unfamiliar
- Utilize Ada's strong typing mechanism as a learning vehicle
- Emphasize design as well as programming through case studies
- Utilize multiple instructors and lab assistants in delivering courses (typically 4 instructors)

The specific educational courses and expected availability dates are:

- Ada Fundamentals Course (August 1984)
- Ada Advanced Course (2nd Quarter 1985)
- Ada Concurrency Course (3rd Quarter 1985)
- Ada Design Language Course (4th Quarter 1985)
- Ada Real-Time Course (4th Quarter 1985)

The first initiative, the one week Ada Fundamentals: A Pragmatic Approach course has been taught since August, 1984, approximately every two weeks. At the end of 1984, sixty employees from Teledyne Brown Engineering and twenty customers have completed the course. The outline (Figure 7) reflects the above educational philosophy. The student materials include a bound package of all slides and a textbook[32] compatible with the educational approach. The other courses will pursue a similar tack. In addition, seminars (1-2 days) will usually be available before the complete five-day course. This is necessary to meet an urgent need for pragmatic education immediately.

In addition to educational courses, automated tools are very attractive products for realizing the productivity gains promised by Ada. Automated tools for Ada should have the following goals:

- Early detection of software errors
- Increased productivity by project personnel
- Better structuring of software systems
- Decreased software maintenance difficulties
- Increased software product visibility

Teledyne Brown Engineering plans to begin construction in 1985 of a set of automated tools (Figure 8) including the following:

- Dependency Analyzer
- Complexity Analyzer
- Project Completeness Analyzer
- Type and Object Cross-Reference Generator
- Source Code Reformatter

Because Ada is being used as a design language as well as for source code, these tools will accept design language as well as source code input. These tools can be utilized by project development personnel, quality assurance personnel, and project management to better control Ada software development and maintenance.

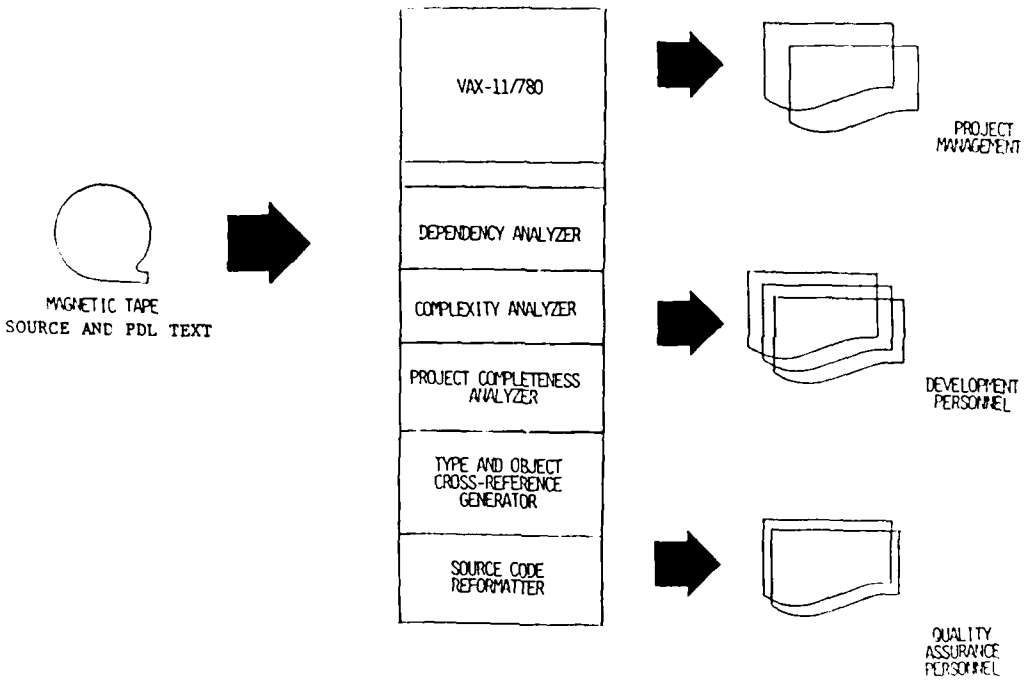
ACKNOWLEDGEMENTS

The author would like to acknowledge the technical support and contributions of Donald R. Clarson and Charles A. Finnell, and the management support and encouragement of Sil Pelosi and Lew Eichert in the preparation of this manuscript and in the pursuit of the Ada Initiatives.

FIGURE 7 - OUTLINE FOR
ADA FUNDAMENTALS: A PRAGMATIC APPROACH

| | |
|---|---|
| **MONDAY** | |
| MODULE 0 | INTRODUCTION |
| MODULE 1 | SOFTWARE CRISIS AND ADA RESPONSE |
| MODULE 2 | ADA LANGUAGE OVERVIEW |
| MODULE 3 | ADA PROGRAM STRUCTURE AND SYNTAX |
| MODULE 4 | ADA PROGRAMMING ENVIRONMENT |
| | PROGRAM WORKSHOP 1 |
| **TUESDAY** | |
| MODULE 5 | INTRODUCTION TO TYPES |
| MODULE 6 | INTRODUCTION TO STATEMENTS |
| MODULE 7 | ADA PROGRAM DESIGN CASE STUDY 1 |
| | PROGRAM WORKSHOP 2 |
| **WEDNESDAY** | |
| MODULE 8 | TYPES REVISITED |
| MODULE 9 | SUBPROGRAMS |
| MODULE 10 | ADA PROGRAM DESIGN CASE STUDY 2 |
| | PROGRAM WORKSHOP 3 |
| **THURSDAY** | |
| MODULE 11 | COMPOSITE TYPES |
| MODULE 12 | PACKAGES |
| MODULE 13 | ADA PROGRAM DESIGN CASE STUDY 3 |
| | PROGRAM WORKSHOP 4 |
| **FRIDAY** | |
| MODULE 14 | INPUT/OUTPUT CONSIDERATIONS |
| MODULE 15 | ADA ADVANCED FEATURES |
| MODULE 16 | ADA TECHNOLOGY EVOLUTION |
| MODULE 17 | SUMMARY |
| | PROGRAM WORKSHOP 5 |

FIGURE 8 - ADA AUTOMATED TOOL PACKAGE INFORMATION YIELD




REFERENCES


1. R. W. Jensen and C.C. Tonies, Software Engineering, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979.

2. B. W. Boehm, "Software Engineering," IEEE Transactions on Computers, Vol. C-25, No. 12, December 1976.

3. R. E. Fainley, P. Freeman, A.I. Wasserman, "Essential Elements of Software Engineering Education," Proceedings of 2nd International Conference on Software Engineering, October 1976.

4. H. D. Mills, "Software Engineering," SCIENCE, Vol. 195, March 1977.

5. G. Booch, Software Engineering With Ada, Benjamin/Cummings Publishing Company, Menlo Park, CA. 1983.

6. B. W. Boehm, "Software and Its Impact: A Quantitative Assessment," Datamation 19, No. 5, May 1973.

7. D. G. Stephen, D. Donis, B. Barbazette, L. Johnson, and B. Munthpy, "DOD Digital Data Processing Study: A Ten Year Forecast," Electronic Industries Association, 1980.

8. G. J. Myers, Software Reliability: Principles and Practices, John Wiley & Sons, 1976.

9. G. Polya, How to Solve It, Princeton University Press, Princeton, N.J., 1971.

10. R. L. Ackoff, The Art of Problem Solving, John Wiley & Sons, Inc., 1978.

11. Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A), United States Department of Defense. Washington, D.C., January 1983.

12. J. G. P. Barnes, Programming in Ada, Addison-Wesley Publishers Limited, 1984.

13. M. Shaw, "Abstraction Techniques in Modern Programming Languages," IEEE Software, October 1984.

14. D. T. Ross, J. B. Goodenough, and C. A. Irvine, "Software Engineering: Process, Principles, and Goals: Computer, May 1975.

15. R. J. A. Buhr, System Design With Ada, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1984.

16. G. W. Cherry, Parallel Programming in ANSI Standard Ada, Reston Publishing Company, Inc., Reston, Virginia, 1984.

17. IEEE Ada as a PDL Working Group, "Ada as a Design Language," (Draft November 1984).

18. "Ada Programming Design Language Survey, Final Report," Naval Avionics Center, October 1982.

19. J. K. Grau and E. R. Comer, "Ada Design Language Concerns," Proceedings of the 2nd Annual Conference on Ada Technology, March 1984.

20. R. M. Blasewitz, "Ada as a Program Design Language - Have the Major Issues Been Addressed and Answered?," Proceedings of the 2nd Annual Conference on Ada Technology, March 1984.

21. Ada Methodologies: Concepts and Requirements, versus Department of Defense (AJPO), November 1982.

22. "The DOD Stars Program," Computer Magazine, November 1983.

23. "Ada-Based System Development Methodology Report," British Department of Industry, September 1981.

24. "Requirements for Ada Programming Support Environments, Stoneman," *Department* of Defense, February 1980.

25. E. W. Dijkstra, "On a Methodology of Design," MC-25 Informatica Symposium, Mathematical Center Tracts, Amsterdam, 1971.

26. C. Bohm and G. Jacopini, "Flow Diagrams, Tuning Machines and Languages with Only Two Formation Rules," CACM, Vol. 9, pp. 366-371, 1966.

27. H. D. Mills, "Mathematical Foundations for Structured Programming," Federal Systems Division, IBM Corporation, Gaithersburg, MD, FSC-72-6012, 1972.

28. R. C. Linger, H. D. Mills, B. I. Witt, Structured Programming, Addison-Wesley Publishing Company, Inc., 1979.

29. A. J. Perlis, F. G. Sayward, M. Shaw, Software Metrics, The MIT Press, Cambridge, MA, 1981.

30. "ADA: Past, Present, Future: An Interview with Jean Ichbiah," Communications of the ACM, Vol. 27, No. 10, October 1984.

31. "A Nation at Risk: The Imperative for Educational Reform," Communications of the ACM, Vol. 26, No. 7, July 1983.

32. K. Shumate, Understanding Ada, Harper & Row Publishers, Inc., New York, NY, 1984.

## BIOGRAPHICAL SKETCH

Thomas J. Walsh is a Principal Systems Analyst in the Ada Systems Department at the Fort Monmouth Office of Teledyne Brown Engineering. He has twelve years of software system experience and has been involved with Ada technologies since 1980. He received a B.S. in Mathematics from Manhattan College and a M.S. in Computer Science from Pratt Institute. Teaching experience includes college (Brookdale Community College) and industrial courses. He has presented an article at the National Computer Conference on software reliability and is published in AFIPS and IEEE press. At Teledyne Brown Engineering, Mr. Walsh is leading the research and development of Ada products associated with the company's Ada Initiatives. These products include five educational Ada courses and an Ada automated tool package. He also is a member and editor of the IEEE Working Group on Ada as a Design Language.

# DEVELOPMENT OF AN EMBEDDED COMPUTER SYSTEM (ECS) APPLICATION IN Ada* - A CASE STUDY

Ronald Rathgeber and Dr. Bruce Burton

Intermetrics, Inc.
Huntington Beach, CA 92649

## ABSTRACT

This case study examined the development of ECS software in Ada. The study was performed by redesigning and reimplementing a portion of the Space Shuttle Fault Detection function in Ada. This case study focused on two main areas: Suitability of current design methodologies to support Ada in ECS applications and Ada Program Design Language (PDL) support for an ECS application. Results of these studies are presented and strategies for future Ada ECS studies based upon our findings are outlined.

## INTRODUCTION

With the completion of production-quality Ada compilers approaching and the DoD directive requiring Ada to be used on all mission-critical DoD projects beginning in 1984, the day when Ada is to be used is here. The primary application area for Ada is intended to be the area of real-time Embedded Computer System (ECS) software development. The utilization of an unproven language such as Ada for ECS applications raises many issues that need to be resolved in order to minimize risk on a large software project that will be implemented in Ada. Along these lines, we at the Intermetrics Engineering Systems Group have committed some of our IR&D funds to the study of Ada in the ECS applications area. In Table 1, the major Ada issues that we are currently addressing are presented along with the objectives of each investigation. The broad objective of these studies is to gain insight on how the Ada language and its tools may be used to achieve the cost-effective development of reliable and efficient ECS applications software.

An important element of our IR&D investigations involves the analysis of the methodologies available for the design and development of Ada real-time programs. In order to examine these methodologies, a case study was performed. This case study focused on the implementation in Ada of a subset of the Fault Detection and Annunciation (FDA) portion of the Space Shuttle Backup Flight System. The implementation was not a simple translation of the HAL/S code into Ada; a complete redesign was performed and documented through two different Ada-oriented Program Design Languages (PDL's). This new design was then implemented in Ada. This report describes the FDA case study and documents our findings.

**TABLE 1. Ada IR&D INVESTIGATION**

| TASK NAME | TASK DESCRIPTION | TASK OBJECTIVE(S) |
|---|---|---|
| Ada FDA Investigation | To redesign and implement in Ada a representative program used on the Space Shuttle | To analyze the strenghts and weaknesses of current methodologies for the design and development of real-time Ada programs |
| Ada Reusability Investigation | To investigate tools and techniques that can be used to facilitate reuse of Ada packages | To identify the short and long term actions that a company should undertake to achieve a high degree of Ada package reuse |
| Ada Reliability Investigation | To study the capabilities of Ada to enhance program reliability and ease of verification | To examine Ada constructs such as exception handling, rendezvous, and others, to evaluate how they affect fault tolerance and verification |
| Ada Real-Time Investigation | To analyze the various problems associated with the use of Ada in real-time applications | To identify and resolve, if possible, several problems associated with the real-time use of Ada. To identify the major advantages/disadvantages in an Ada real-time executive |
| Ada Language Maturity Investigation | To assess the maturity of the Ada language tools that are commercially available | To collect information on quantity and quality of available Ada compilers and associated tools for ECS development |

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

## REQUIREMENTS DEFINITION

The first step in the FDA study was to define the requirements that would be used to guide the redesign. The basis used for creating the requirements was the Program Requirements Documents of the Shuttle Backup Flight System[1].

The functions selected for implementation included a subset of the Fault Detection, Annunciation, Scaling, and Display Task (FDASDT). On the Shuttle, FDASDT performs range checking, limit checking, and annunciation of out-of-limits conditions on a set of predefined parameters. It also scales the parameters for a cockpit display and sets a status indicator for each scaled parameter. The status indicators signify whether each parameter is in-limits, out-of-limits low, out-of-limits high, out-of-range low, out-of-range high, or contains invalid data. If a parameter contains invalid data, then the status indicator is set to invalid data, and the last valid data for that parameter is displayed.

The FDASDT function processes over 150 different parameters. These include both analog and discrete parameters. Limit checking and range checking are not performed on discrete parameters. Instead, each discrete parameter is inspected to determine whether it is in an undesired state. If it is, the parameter's status indicator is set to out-of-limits low, and the error is annunciated. Scaling of the discrete parameters requires only that a display value be set either to true or false. Most of the analog parameters use linear scaling, but a few require cubic scaling.

Since the objective of this case study was to analyze techniques, not to create benchmarks, and due to the limitations of tools currently available, a subset of 40 parameters was selected for processing. These 40 parameters were selected to represent each of the major types of FDASDT parameters. This subset included analog and discrete parameters, with both analog linear and cubic scaling.

In addition to the major types of parameters, four sets of parameters that require special pre-conditioning were also selected. These were IMU Fan, APU Speed, REAC Valve, and Fuel Cell parameters. For the three IMU Fan parameters, fault detection is performed only on the fan that is currently turned on; however, the current state of all three fans is displayed. For the APU Speed parameters, multiple upper limits are checked. In addition, checking of the lower limit is disabled until an APU has been on a specified period of time, or after the APU has exceeded a shutdown limit. For the REAC Valve parameters, the states of two sets of valves (hydrogen and oxygen) are "ANDed" together to determine the state of the REAC valves. For the Fuel Cell parameters, one of three upper limits is selected based on a computed power level.

In addition to defining the specific requirements to be implemented, it was also necessary to define the system under which the FDASDT subset would run. This system definition included a specification of all interfaces between the FDASDT function and the outside world. For consistency and comparison, the system that the FDASDT subset runs under is modeled after the system currently used in the Backup Flight System (BFS), i.e., the Backup Operating System (BOS).

The BOS performs all input/ output for the BFS. All data that is input from or output to a specific hardware device is stored in shared memory areas via HAL/S compools. For example, all data intended for the cockpit displays is stored in the Display compool by the application functions (such as FDASDT). This data is subsequently accessed by the BOS and output to the display.

(A complete definition of the FDASDT requirements used for this case study is given in Intermetrics California Internal Report Number 014.)

## TOP-LEVEL DESIGN

The next step in the study was to perform a top-level design of the FDASDT subset. Several design methodologies were considered for use in performing this design. Among these were Object-Oriented Design[2], Modular Program Construction Using Abstractions[3], and Structured Design[4]. A brief description of these three design methodologies is presented below.

Object-Oriented Design was popularized by Grady Booch as a design methodology specifically targeted for use with Ada. In Object-Oriented Design an effort is made to match design decisions with our view of the real world. With this method, the design is decomposed into a set of software

objects and related operations. These objects become our data, and the operations become procedures that operate on this data. Object-Oriented Design is tailored to specifically take advantage of Ada's package concept, and thus supports information hiding and abstraction concepts.

The Modular Program Construction Using Abstractions method of design was also considered. This methodology is an enhancement of stepwise refinement, which was developed by Wirth[5] and Dijkstra[6]. This methodology performs modular decomposition based on recognition of useful abstractions. It involves an iterative process where each abstraction might be further decomposed by repeating the process.

The Structured Design method developed by Yourdon was our third candidate for a design approach. The first step in developing a structured design is to devise data flow diagrams for the problem. These data flow diagrams are then transformed into structure charts by identifying the afferent nodes (input), efferent nodes (output), and central transforms (processing).

In analyzing the various design methodologies, several evaluation criteria were used. Important criteria included: the ability to model the problem, ease of use, previous successful uses, ability to deal with real-time aspects, and support for advanced Ada concepts, such as data encapsulation. In surveying existing design methodologies, none was found that satisfactorily met all the criteria. Older methodologies, although used successfully on many projects, were developed before Ada and don't provide support for advanced Ada concepts. Newer Ada-based design methodologies have had limited use to date. All of the candidate methodologies were found to be deficient in the area of real-time support.

After a substantial amount of analysis, it was decided to use the Structured Design method for the top-level design. The main reason for the use of the Structured Design technique was that it is already being used successfully to solve a wide variety of design problems. Also, some work has been successfully performed by other companies using this method with Ada. In addition, our personnel had experience with this method and, therefore, no training was needed for either the designers or the reviewers to start this project.

The design was generated with no major problems. The Review Committee suggested several minor modifications, but the overall design was accepted. Due to the limited scope of the problem being implemented, several areas, such as real-time considerations, were not closely examined. Since Structured Design was not specifically designed to support the development of Ada programs, special effort was made to support the design of Ada-like constructs.

## DETAILED DESIGN

The next step in the case study was to perform a detailed design based on the top-level design. In order to provide a comparison of Program Design Languages (PDL's), two separate detailed designs were generated using two different PDL's. The two PDL's used in the study were Byron™[7] and the Software Design and Documentation Language[8] (SDDL) developed by the Jet Propulsion Laboratory.

Byron™ is an Ada-based PDL and, in fact, any legal Ada program is also legal for Byron™. Byron™ provides the user with a set of keywords that expresses additional descriptive design information that can't be written in Ada. The user details a module's detailed design as a collection of Ada-like constructs augmented by the Byron™ keywords. Byron™ performs interface checking (even on separately processed modules) and provides formatted source listings and cross-references. Byron™ will also generate design documents, such as C5 specifications, in a user-specified format.

SDDL is a conventional PDL that allows the user to specify the keywords that it will recognize. For this study, a set of Ada-like keywords was developed. A module's detailed design is then written in pseudo-code using these keywords. SDDL will provide the user with a design document that includes a table of contents, formatted source listings, and a module hierarchy tree. Interface checking, though, is not performed by this tool. In addition, SDDL does not allow nesting of modules, so a workaround was needed to allow procedures and tasks to be nested in a package.

Each of these PDL's has advantages and disadvantages. Byron™ provides the

user with many features that are important on large projects with many programmers. Most important of these is complete interface checking which allows the interfaces to be checked during the design phase. This facilitates detecting design errors before development begins. On the other hand, SDDL is a simple PDL that offers a limited number of features. On a small project, SDDL's ease of use might outweigh its lack of features.

When comparing these PDL's, however, the cost of use and size must also be considered. Byron™ is a large and fairly expensive program. The IBM version of Byron™ that was used required over one megabyte of memory and, therefore, is restricted to running on large computers. Conversely, SDDL is small and inexpensive. SDDL has even been successfully hosted on an IBM Personal Computer.

## CODE IMPLEMENTATION

Next, the detailed design was implemented in Ada and tested. A popular Ada Compiler available on the IBM Personal Computer was used for the implementation. Unfortunately, the compiler used supported only a subset of the Ada language. The version of the compiler that was used did not support many important Ada features, including generics and enumerated I/O.

Only one major implementation problem arose because of the use of Ada. This problem was due to the requirement that all system I/O be done through the BOS. In the BFS this is done by creating I/O compools that serve as the communication region between the BOS and the application programs. In order to follow the spirit of Ada, it was decided to avoid using packages as mere compools. Consequently, packages were set up that contained declarations for all the variables in an I/O compool inside the package body. Procedures were then written that would read from and write to these variables. The procedures were included in the package specification so that programs could use them to access the variables within the package. In a production system, these procedures could use the INLINE pragma for efficiency.

Other problems occurred due to the compiler and environment. The most serious of these was due to the run-time system's lack of garbage collection. The FDASDT module was originally written as an Ada procedure that contained six tasks. This procedure was then executed cyclically. However, due to the lack of garbage collection, each time the tasks were executed they would use a new section of run-time memory. After a few cycles of execution, the run-time memory would overflow. In order to complete testing, it was necessary to execute each task individually. Also, instead of executing the tasks cyclically, each task was rewritten to contain a loop that executed the task until the test was complete.

Another problem encountered with the Ada system used involved the lack of any debugging tools. All debugging had to be done by inserting debug code that would write values, etc., to a file. Other inconveniences included a lack of enumerated I/O and unexplained system crashes.

Except for problems attributable to the limitations of the compiler and run-time system, the implementation of the FDASDT code in Ada went fairly smoothly. Most difficulties were due to lack of knowledge on certain Ada constructs. These problems can be prevented on future projects by providing additional Ada education earlier in a project.

## SUMMARY

The FDA case study has proven useful in examining methodologies for implementing code in Ada. As well as showing the need for extensive training before Ada is used, the study has also shown some areas of software development where more research needs to be done.

Perhaps the area where the most work is needed is in the top-level design. There are many top-level design methodologies, including some that were designed expressly for use with Ada; however, none of these proved to be adequate for general use. When real-time applications are considered, these methodologies are even less acceptable due to their lack of mechanisms for expressing timing considerations.

Detailed design is probably the best supported software development phase. The Program Design Languages (PDL's) currently available should support the users' needs. These PDL's range from fully functional PDL's, like Byron™, that support the Software Development Life Cycle from design to maintenance, to simple, PDL's, such as SDDL. The user can choose among the wide range of

PDL's based on project needs and user tastes.

The main support needed for the implementation phase will be Ada training. Although Ada is similar to other High Order Languages, Ada's new constructs, such as packages and generics, will require user education. For programmers who are inexperienced with High Order Languages, more extensive training will be needed. All programmers, regardless of their previous experience, will need education in the proper use of Ada, including program design methodology.

The test and maintenance phases were not explicitly covered by this case study. Testing of the generated code was done, but only in a primitive manner, since no sophisticated test tools were available. Software development environments must certainly be considered if Ada is to be used on production quality projects, as they play a major role in the development of correct, quality software. No examination was done of the maintenance phase.

Another area that was not explicitly covered by this case study was the collection of "hard" engineering data on our real-time implementation. This collection process was hampered by the fact that the Ada version of the FDA subset was developed on an IBM-PC host with a simulated real-time clock and targeted for the PC. The lack of a realistic embedded processor execution environment precluded the collection of meaningful performance data from the FDA case study. The collection of such performance data and the investigation of Ada run-time support systems for ECS applications constitute major goals for our current IR&D activities. Our continued investigation of Ada compilers and their associated run-time systems should yield information on the code quality, run-time overhead, and performance characteristics for representative real-time applications. Only through the provision of this type of information can a program manager make a quantitative assessment about the risk of commiting to Ada for an ECS applications project.

This case study resulted in a successful reimplementation in Ada of a portion of the Space Shuttle's Fault Detection, Annunciation, Scaling, and Display task. This reimplementation has been useful in examining methodologies of implementing a typical ECS application in Ada. Although it is beyond the scope of this study to define a definitive software development methodology for Ada, the study has provided a knowledge base on software development methodologies involving Ada. This base of knowledge will be useful as further efforts are made to define a comprehensive software development methodology for Ada.

## BIBLIOGRAPHY


1. Bertrand, William, Backup Flight System Management/ Special Processes and Sequencing Program Requirements Document, Rockwell International, 25 October 1982.

2. Booch, Grady, Software Engineering in Ada. The Benjamin/Cummings Publishing Company, Inc., 1983.

3. Liskov, Barbara, Modular Program Construction Using Abstractions. Massachusetts Institute of Technology, Computation Structures Group Memo 184, September, 1979.

4. Yourdon, Edward and Constantine, Larry, Structured Design, Fundamentals of a Discipline of Computer Program and Systems Design. Yourdon Press, 1978.

5. Wirth, N., Program Development by Stepwise Refinement. Communications of the ACM/4, 4, April 1971.

6. Dijkstra, E.W., Notes on Structured Programming. A.P.I.C. Studies in Data Processing No. 8, Academic Press, NY 1972, 1-81.

7. Gordon, Michael, The Byron™ Program Development Language. Intermetrics, Inc., April, 1983.

8. Kleine, Henry, Software Design and Documentation Language. JPL Publication 77-24, Revision 1, August, 1979.

## BIOGRAPHIES

Ronald Rathgeber is a Senior Programmer/Analyst for the California Division of Intermetrics, Inc. and is currenty Deputy Manager of the Software Technology Department with responsibility for Ada Development. Previously, Mr. Rathgeber worked for five years on the Backup Flight System for the Space Shuttle. Mr. Rathgeber received his B.S. in Information and Computer Science from the University of California, Irvine.

Bruce Burton is the Manager of the Software Technology Department at Intermetrics, Inc. He holds an M.S. in Information and Computer Science and a Ph.D in Physical Chemistry from the University of California, Irvine. Dr. Burton is interested in the real-time programming area and in the field of software reuse.

# IMPLEMENTING WATCH DOG TIMERS IN ADA FOR TOLERANCE TO CERTAIN CLASSES OF REAL TIME FAULTS

Christian Wild

Old Dominion University
Department of Computer Science
Norfolk, VA 23508

## Summary

Many projects which use Ada will require high reliability and real-time response. One important class of failures in a real-time system is failure to respond within the required time frame. Because of the difficulty of ensuring real-time response by prior analysis, timing failures are usually detected at run time by watch-dog timers. The provision for a generalized time-out facility will require the restoration of a consistent state before normal processing can continue. Ada provides only a limited time-out facility which is related to the difficulty of providing state restoration. The relationship between watch-dog timers and the more general problem of software fault tolerance is shown and the specification and handling of timing faults is integrated into a uniform methodology for fault tolerant software. The use of standard Ada to implement a generalized time-out facility is also discussed*.

## 1. Introduction

Since Ada was designed to be used in embedded systems, it is likely that there will be a requirement for high reliability in many applications. The achievement of high reliability is made more difficult if these applications also require real-time response. Many of the features of the Ada language and support system were designed to support reliability. However traditional software engineering design and testing methodology has not achieved the degree of reliability desired in critical applications [1]. Present day technologies have been unsuccessful in eliminating software design errors from a complex computer system. Design faults which elude the development and testing phases compromise the reliability of the system. Software fault tolerance attempts to achieve high reliability by detecting and treating residual software design faults.

One important class of failures in a real-time system is failure to respond within the required time frame. The causes for the failure to respond may be due to programming errors, hardware failures or congestion within the system. This class of failures is interesting to study because it deals with the operational aspects of the system rather than its functional characteristics. Issues involved with the specification and verification of performance requirements have received little attention in proof methodology for program correctness. In fact, much of the work in formal verification is limited to "partial" correctness; that is, correctness if the program terminates. Even if a program terminates and functions "correctly", it may fail to meet critical mission objectives if deadlines have been missed. Because of the difficulty of ensuring real-time response by prior analysis, timing failures are usually detected at run time by time-outs (or watch-dog timers). Provision for a generalized time-out mechanism poses several interesting problems. How should the timing specifications be made? What are the semantics associated with a timing fault? And what implementation considerations are there?

Our work integrates the specification and handling of timing faults with a generalized and uniform methodology for fault tolerant software [2,3,4,5]. An important concern is the restoration of a consistent state before normal processing can continue. The need for state restoration is particularly apparent if the system is in a critical section updating a shared variable. Issues involved with state restoration are further discussed in section 2.

Ada provides only a limited time-out facility. These limitations are related to the difficulty of providing state restoration (see section 3). The need for state restoration within Ada has been addressed by other researchers [6]. With a facility for state restoration, Ada

* This work was sponsored in part by the NASA Langley Research Center under grant NAG-1-439.

would be able to implement atomic actions and thus provide a design framework known as Idealized Fault Tolerant Components (see section 4). Section 5 shows how timing fault detection might be integrated into this design framework. We can also show how standard Ada can be used to implement a generalized timing facility through careful control of state changes. The overhead of such an approach is also discussed (section 6). We also note that watch-dog timers are a specific instance of the more general concept of watch-dog processes (section 7).

## 2. Importance of state restoration

The importance of state restoration for software fault tolerance has been noted elsewhere [7]. However since almost none of the existing languages and architectures provide state restoration primitives, it may be useful to discuss the reasons for state restoration. If there were no faults* then there would be no need for state restoration. However after the detection of a fault, it is necessary to restore the system to a consistent state. One way to restore a consistent state is by using forward error recovery in which the current inconsistent state is corrected by compensating for these inconsistencies. Forward error recovery works if the fault is in some sense anticipated (for example a zero-divide fault) and suitable inverse or compensating operations can be programmed. Many times, however, the only sensible strategy is to roll the computation back to a previous (hopefully consistent) state and start from that state.

Because faults do occur, state restoration is provided in some form in every computer system. Checkpoints and audit trails are instances of state restoration. A cold start reboot is an extreme form of state restoration. However most existing practice is limited to restoration in the presence of hardware breakdown. The need for state restoration to tolerate software (and hardware) design faults is less obvious. There is a school of thought which considers tolerance to design faults to be foolhardy and will settle for nothing less than perfection. This school of thought is exemplified by the program proving methodology. We do not wish to take sides in this controversy except to point out that in the case of timing faults, because they cannot yet be eliminated by the techniques of formal verification, it will still be necessary to provide for state restoration.

To understand why state restoration is needed for a generalized approach to the handling of timing faults, let us consider what might happen if a time-out occurs while a process is in a critical section. A critical section is a region of a program which must have exclusive access to some shared resource. While one process is in its critical section, no other process is allowed to enter its critical section to access that shared resource. If a process dies while in a critical section, all other processes which require access to that particular resource would deadlock. Therefore, most existing work on critical sections assumes that a process is "well behaved" when it is in its critical section. For instance, it is generally assumed that a program will not execute an infinite loop while in the critical state. Although it may seem desirable to abort a process which is not well behaved in its critical section, such action is fraught with danger. The state of the shared resource after a process is so arbitrarily aborted may be unknown. In fact many program proving methodologies consider a critical section to be indivisible (also known as an atomic action). Assertions about the state of the resource are only true if there is no process in a critical section on that resource.

Given that the shared resource may be in an inconsistent state, what actions should be taken? Forward error recovery seems difficult since we do not know how far in the computation the offending process has gone. Therefore it seems that backward error recovery by restoring the state to a previous one, namely the state on entrance to the critical section, is the only reasonable course of action. If the shared resource is a data structure, then the easiest way to insure the integrity of the shared data structure is to treat changes to them to be atomic actions [8]. This is equivalent to restoring the state of the data structure to that on entry to the critical section.

## 3. Standard Ada Timing facilities

While Ada does allow some time-outs to be specified using the "delay" statement, a facility for time-outs is not integrated in a general fashion into the language. Only the entry into a rendezvous can be timed out in a reasonable way. Such a time out is allowed because the computation will be in a known state on entry to a rendezvous. Either task involved in a rendezvous can time-out before initiation of a rendezvous (i.e. before the other process reaches the rendezvous point). However the semantics of

* Faults include hardware breakdown, hardware and software design faults and environmental faults. Environmental faults include sensor and actuator breakdown as well as human error.

Ada prevent two processes engaged in a rendezvous from executing simultaneously (this is necessary to ensure the atomic action semantics of critical sections). Because of this the language provides no way for the calling process to terminate a rendezvous once it has begun. If a rendezvous has started and the machine the serving process is on fails, then the calling process will wait in the rendezvous forever. Several researchers have recognized this problem and have proposed a solution [9]. However the problems of software and timing failures have not been addressed. The inability of Ada to deal with time-outs within rendezvous are related to a lack of state restoration primitives necessary for aborting a critical section. It has been suggested [10] that one use the rendezvous only to make service requests and to get service results. The actually processing of the service would not be performed in a rendezvous. This minimizes, but does not eliminate, the problem.

Extensions of Ada's time-out facilities would be useful in real-time systems [11]. For instance it may be desirable to time an intra-process procedure call [12]. This would be useful both for guarding against infinite loops and for meeting deadlines within procedure calls. It should be noted that, in general, state restoration would be required for restoring a consistent internal state for a process if a time-out did occur. Presently timing violations within a process could not be detected within standard Ada. In dealing with timing faults in a real time system, a more general timing mechanism is needed than that provided by the Ada standard.

## 4. Idealized Fault Tolerant Components

One of the issues explored in this study was the relationship between the handling of timing faults and software fault tolerance. In order provide the background for this integration, a brief discussion of software fault tolerance methodology will be given.

Much of the work in software fault tolerance can be integrated into a design framework known as Idealized Fault Tolerant Components (IFTCs). IFTCs are an attempt to incorporate software fault tolerance principles into the accepted practices of hierarchical design and encapsulation with minimum impact on the normal software development process. They are a combination of modular hierarchical structuring techniques, atomic actions and exception handling [2. and 5]. IFTCs provide a unified framework for many of the proposed software fault tolerance techniques [7 or 13]. Using IFTCs, the designer controls the amount of error detection and the kind of error recovery techniques most appropriate for the application. The work of Cristian [4 and 3] investigates properties of correctness and robustness which are applicable to IFTCs.

Figure 1 shows an IFTC.

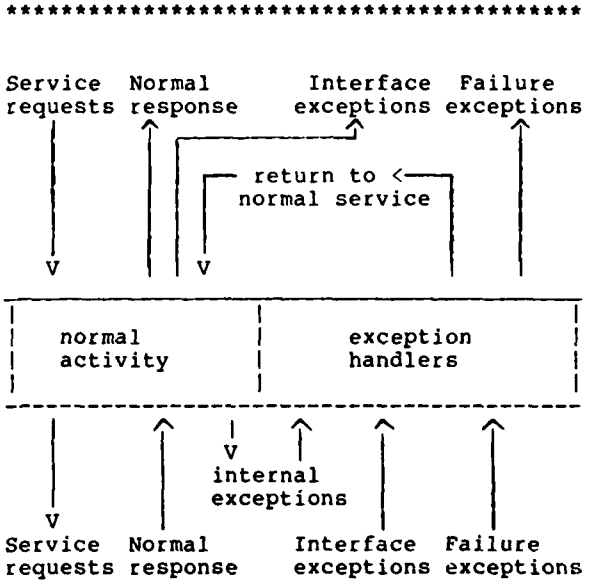



Figure 1. Ideal Fault Tolerant Component

Service requests are made to an IFTC in the normal way (i.e. by procedure call or message, etc.). If the requested service can be provided without faults, the IFTC gives a normal response. During the servicing of the request the IFTC may itself make requests of subcomponents. These subcomponents may also be IFTCs. Thus IFTCs provide a recursive structuring principle.

If a fault occurs during the processing of a request and the error caused by that fault is detected, an exception is raised within the IFTC and an exception handler is called. If the IFTC can handle the error internally, then the fault is masked and activity returns to normal servicing. Error recovery can be forward or backward. Figure 2 shows how forward error recovery can be handled with an IFTC.

```
*****************************************

[inconsistent_assert -> forward_correctl];

or

  F[x -> forward_correct2];

For example

  [ b = 0 -> b = very_small];
  x := a/b;

or

  x := a/b    [zero_divide -> x := MAX];
```

Figure 2: Forward Error Recovery

```
*****************************************
```

The statements between the brackets define the exception handler. The section before the "->" is either an assertion or a propagated exception. The statements after the "->" are the action portion of the exception handler. Internal checks for consistency or interface preconditions are made by assertions ("inconsistent_assert" and "b=0" in figure 2). Subcomponents can also return exceptions (subfunction "F" returns exception "X" and function "/" returns "ZERO_DIVIDE"). In the case shown in figure 2, forward error recovery masks the fault and the computation continues normally.

Since an implicit recovery point is defined on entry to an IFTC, the state can be rolled back to this point by executing a "reset" command. Thus the "reset" command provides for backward error recovery. If the exception handler cannot mask the fault, then the exception can be explicitly propagated to the calling computation where it can be masked or propagated. More information on IFTCs can be found in [7 or 13].

## 5. Integrating Watch-Dog Timers with IFTCs

The problem of integrating watch-dog timers is that of providing the appropriate mechanisms for concurrent monitoring within the hierarchy of IFTCs. IFTCs as defined above only deal with exceptions which occur within one process. The time-out activity is a concurrent process which can cont[illegible] the execution of the process being timed. In the case of a time-out, the timing process must be able to abort the execution of the timed process, restore the run-time environment to a consistent state and return control to the exception handler. Semantically, a time-out should be treated like an assertion or acceptance test which failed [14]. However, the implementation is quite different because of the nature of the concurrent monitoring by the timing process. In the case that the calling and called IFTC are on different processors or computers, extensive housekeeping operations must be performed to restore the system to a consistent run time state. Buffer queues may have to be flushed, late messages will have to be discarded and the serving IFTC will have to be resynchronized.

If a procedure call was timed-out, then the appropriate action would be to abort the computation, execute the "reset" command and signal the exception "time-out". If the computation was a remote procedure call, then the action would be to notify the executing processor of the requested abort operation and have the remote machine abort the offending IFTC, execute a "reset" command and signal the exception "time-out". However because of the delays in communications, it may be necessary to simulate the propagation of the exception "time-out" from the remote procedure call in order to attempt appropriate action on the requesting processor in a timely fashion. This can lead to the following situations.

1) The remote procedure is aborted, state is reset and time-out is propagated. The supporting system on the requesting processor will need to ignore the propagated "time-out" since it has already been simulated.

2) The processor on which the remote procedure call is being executed fails, the underlying system attempts to restart the computation on another processor. The request to abort the remote procedure will have to be redirected to the new processor. This will require that the underlying system have the mechanism for binding names in presence of a changing hardware environment.

3) An abort for time-out is requested but the remote procedure has already finished. The underlying system should recognize that this has occurred and ignore the effect of the remote computation. This may involve the use of time-stamps or other mechanisms to uniquely identify each request for service.

4) Several processes are involved in the computation. This is discussed in [13].

When aborting remote computations with a simulated "time-out" exception, it may be difficult to reset the shared data structure (depending on where such data structures are stored). If the requesting process tries an alternate computation, then the shared data structure must be reset and the "aborted" computation must not be able to access the shared data structure any more. Because of communications delays, assuring that the proper actions take place under the constraints of real-time deadlines can be very difficult.

Another concern with proper timing is whether the calling or called IFTC should specify the timing limit. In the case of a timed recovery block [14], we would like to allow enough time to execute the alternate (or in some cases the primary). This implies we know how much time the alternate might use. If the programmer can define some limit on the number of loop executions (as could be done using Dijkstra's variant function for proving loop termination [15]), then the compiler/run time support system should be able to estimate the time for execution [11]. Thus it may be possible to place loose but useful limits on the execution time of components. This timing would be a function of the called routine. However in real-time systems, the reasonableness of the execution time may not be as important as the meeting of a deadline. The calling routine is usually in a better position to specify the deadline. Another argument for including the timing function with the calling routine is in the case of hardware failures where the calling and called routines are on different machines. Timing on the machine of the calling routine would allow for continued timely operation when the computer supporting the called routine failed.

Figure 3 indicates how timing specification might be achieved using the notation of IFTC.

```
**********************************************

nextpos.compute [time_out(onesecond) ->
        nextpos.approximate];

**********************************************
```

Figure 3 Specifying Timing Constraints

The procedure "nextpos.compute" and watch-dog "time_out" are executed concurrently. The monitor "time_out" can be thought of as making the assertion "nextpos.compute took less than onesecond". If this assertion becomes false, then "nextpos.compute" is aborted, the state is "reset" and the exception "timeout" is propagated. This exception returns control to where the "time_out" monitor appears and thus executes the action following the exception, namely "nextpos.approximate".

While the semantics for watch-dog timers is consistent with an assertion test (although done concurrently) and the syntax for handling exceptions from watch-dog timers can be integrated into IFTC as shown above, the implementation of watch-dog timers is quite different from serially detected exceptions. To execute a computation, the exception handler must be examined for those exceptions propagated by watch-dog timers and those watch-dog timers must be started concurrently with the execution of the monitored computation.

## 6. Implementation in Ada

Ada cannot be used to implement IFTCs directly. The major missing language feature is the lack of state saving and restoration primitives. In addition Ada's exception handling does not implement the single level terminating exception handlers required by IFTCs. Several researchers have suggested modifications to standard Ada to deal with these inadequacies [6]. In this section we will illustrate what can be done to implement watch-dog timers in standard Ada and comment on the efficiency of this approach.

It may be argued that generalized time-out facilities can be achieved in Ada through the use of the "delay" and the "abort" command. Each computation which

is to be timed is executed as a new task instance. The results are returned by a rendezvous with the timing task. The "accept" clause in a "select" statement with a "delay" clause. If the "delay" clause is reached, the timed task is aborted (illustrated in figure 4).

```
*****************************************

TASK TYPE type_a;
TASK timer_of_a IS
   ENTRY output_from_a(...);
END timer_of_a;

TYPE access_a IS ACCESS type_a;
a : access_a;

TASK BODY timer_of_a IS
BEGIN
   copy_input_state;
   a := NEW type_a;
   SELECT
      ACCEPT output_from_a(...)
         update_output_state;
   OR
      DELAY timeout;
         ABORT a;
         -- other actions for
         -- exception handler
   END SELECT;
END timer_of_a;
```

Figure 4 - Timing a Computation

```
*****************************************
```

There are at least three objections to this approach. The first is that timing a computation is awkward to specify since two tasks must be defined. The second is that such a mechanism is likely to introduce considerable overhead and probably would not be appropriate for timing ordinary procedure calls. Thirdly and more seriously is the question of state consistency after aborting the timed process. The topic of state restoration is discussed next.

Recovery Point Primitives

Ada contains no recovery point primitives. This leaves the issue of state restoration up to the programmer. We have already discussed that state restoration is a critical and often ignored aspect of reliable programming. Without help from the language and underlying system, the individual programmer is left to his/her own methods to restore a computation to a consistent state when errors are detected. This restoration is particularly troublesome in the case of time-outs since the timing process is not aware of the progress made by the offending computation. The safest (and perhaps only reasonable) action is to restore the state to its previous value (atomic action rollback). To make matters worse, the values of IN/OUT parameters after an exception occurs is implementation dependent. The Ada run time support system can choose either call by reference or call by value with copy out of the updated value. Thus the state of an IN/OUT variable in the calling procedure when an exception occurs depends on the implementation chosen ([16] argues that the choice should be under programmer control).

In the absence of system provided primitives for recovery points, the programmer can provide for atomic actions by a copy-in copy-out discipline for those items of the global state which are to be modified. Making changes only to copies of the state will allow those changes to be easily abandoned if an error is detected. If the computation terminates normally (with no residual errors detected), then the changed values can be used to update the global state. Note however that the state update is a critical section and must not be interrupted. This implies that timeout must be turned off when the computation enters the update phase. Figure 4 illustrates this form of programmed state saving and updating. One objection to this scheme is that it can considerably slow down normal processing* In fact the time for update of the state on normal termination might cause the missing of a deadline which would not have happened otherwise! If the computation has a permanent internal state, this would have to be copied also.

Although it is possible to perform state restoration purely by programmed actions with no help from the underlying system, such an approach seems fraught with danger and a source of unreliability. The performance of these programmed state restoration methods may not be adequate for real-time systems. It would seem that a system supported (automatic) state recovery mechanism should be made available.

## 7. Watch-Dog Processes

* Ideally we would like software fault tolerance implementations to have minimal impact on the normal processing activity. Efficiency for abnormal processing, which should occur rarely, is not so much a concern.

A watch-dog process can be defined as a process which monitors the behavior of some other process for the intention of detecting faults in the monitored process. One reason for studying watch-dog timers is that they represent a specific but important subclass of the general concept of watch-dog processes. Other classes of watch-dog processes have received less attention. Perhaps this is due to the constraints imposed both by current computer languages and architectures. With the declining cost of hardware, there is increased potential for watch-dog processing. For example, it would be possible to interface a special processor to the memory bus to monitor memory address references [7]. This could be used to detect erroneous control flow sequences [17 and 18]. Other possible uses of watch-dog processing include monitoring resource usage for adherence to resource conversation laws (e.g. a consumer can't consume more than the producer produces). In addition simulation could be used to predict gross behavioral patterns of the system under observation.

One important issue which arises with the use of watch-dog processes is the manner in which the monitored and monitoring process interact. What access should the monitoring process have to the monitored process state? Should their interaction be synchronous or asynchronous? What should happen to the monitored process if the monitoring process fails? How should watch-dog processing be implemented on multi-processor and distributed systems? What should be done with the monitoring process if the monitored process terminates either normally or abnormally? Clearly there are many questions with the implementation of general watch-dog processing which have not been extensively studied**. The complications of implementation on distributed systems have only been touched upon. However, preliminary observations indicate that watch-dog processes can be specified in a manner similar to that suggested earlier for watch-dog timers and that the semantics of a watch-dog process should be the same as a concurrently executing assertion.

## 8. Conclusions

In this paper we have discussed the implications of providing watch-dog timers in Ada. We have shown that the provision for a generalized time-out facility (i.e. the ability to abort a process in its critical section) will require state restoration. We have shown that the limited facilities in Ada for time-outs is related to the lack of state restoration primitives in the standard language. While not necessarily advocating the use of software fault tolerance, we noted that a generalized time-out facility could be integrated with the design framework of an Idealized Fault Tolerant Component. The implementation of Idealized Fault Tolerant Components will require modifications to Ada. However through the careful control of state changes, standard Ada can be used to simulate Idealized Fault Tolerant Components. Whether this simulation will be feasible for real-time applications will depend on the application.

** In fact one might observe that the idea of a watch-dog process is antithetical to the principal of encapsulation and information hiding. However faults rarely observe such principles.

## References


1. A. Avizienis, "Fault-Tolerant Systems," IEEE Trans. on Computers Vol. C-25(12), pp.1304-1312 (Dec. 1976).

2. B. Randell, "Fault Tolerance and System Structuring," 4th Jerusalem Conference on Information Technology (TBGiven), Computing Laboratory, University of Newcastle upon Tyne (Dec. 2, 1983).

3. Flaviu Cristian, "Correct and Robust Programs," IEEE Trans. on Soft. Eng. Vol. SE-10(2), pp.163-174 (March 1984).

4. Flaviu Cristian, "Exception Handling and Software Fault Tolerance," IEEE Trans. on Computers Vol. C-31(6), pp.531-540 (June, 1982).

5. P.A. Lee, "Structuring Software Systems for Fault Tolerance," AIAA Computers in Aerospace IV Conference, pp.30-36 (Oct. 1983).

6. M. Di Santo, L. Nigro, & W. Russo, "Programming Reliable and Robust Software in ADA," FTCS-83??, pp.196-203, IEEE (1983).

7. Tom Slivinski, Jack Goldberg, Tom Anderson, John Kelly, Ellis Hitt, Jeff Webb, Christian Wild, & Karl Levitt, Study of Fault Tolerant Software, Mandex, Inc. (May 1984).

8. Flaviu Cristian, "Robust Data Types," Acta Informatica Vol. 17, pp.365-397 (1982).

9. John Knight & John Urquhart, "Fault Tolerant Distributed Systems Using Ada," AIAA Computers in Aerospace IV Conference, pp.37-44 (Oct. 1983).

10. Valerie A. Downes & Stephen J. Goldsack, Programming Embedded Systems with Ada, Prentice-Hall International, Englewood Cliffs, NJ (1982).

11. Anthony Wei, "Real-Time Programming with Fault-Tolerance," PhD Thesis(N82-20897), Univ. of Illinois (1981).

12. W. Jessop, "Ada Packages and Distributed Systems," SIGPLAN Vol. 17(2), pp.28-36 (Feb. 1982).

13. Christian Wild, Programming Language Support for Software Fault Tolerance, Dept. Computer Science, Old Dominion University (May 1984).

14. Roy Campbell, Kurt Horton, & Geneva Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," Proc. of the 9th Annual International Sym. on Fault Tolerant Computing, pp.95-101, IEEE (June 20, 1979).

15. Edsger Dijkstra, A Discipline of Programming, Prentice-Hall, Englewoods Cliffs, NJ (1976).

16. Douglas Jones, "Tasking and Parameters: A Problem Area in Ada," SIGPLAN Vol. 15(5), pp.37-40 (May 1980).

17. J.M. Ayache, P. Azema, & M. Diaz, "Observer: A Concept fo On-Line Detection of Control Errors in Concurrent Systems," FTCS - ??, pp.79-??, IEEE (1979).

18. Stephen Yau & Fu-Chung Chen, "An Approach to Concurrent Control Flow Checking," IEEE Trans. on Soft. Eng. Vol. SE-6(2), pp.126-137 (March, 1980).

Dr. Christian Wild worked for eight years at Bell Telephone Laboratories in the areas of CAD, graphics, computer vision and computer networks. He received his PhD degree from Rutgers University in 1977 in Computer Science. After taking a two year sabbatical to go sailing, he joined the faculty in the department of Computer Science at Old Dominion University in 1980. Dr. Wild's research interest include artificial intelligence, automatic programming, software fault tolerance and advanced computer architectures.

# WHY NOT UNIX*? THE CASE FOR THE ARMY SECURE OPERATING SYSTEM

Eric R. Anderson
Ruth M. Hart

TRW Defense Systems Group
Redondo Beach, CA 90278

## Abstract

The Army Secure Operating System (ASOS) project is building a set of secure operating systems which will support tactical applications written in Ada**. We have sometimes been asked to justify the need for such an operating system; in particular, to explain why an existing operating system such as UNIX could not be used instead. This paper answers this question from the perspective of the Army; it examines the Army's needs in the area of support for Ada, performance, functionality, and security, and shows how ASOS meets those needs and why UNIX does not.

## Background

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. Both operating systems are to be written in Ada and are to be designed to support tactical Ada applications. The Dedicated Secure Operating System is intended to be run in either a dedicated or systems high mode, and is to be optimized for efficiency, while the Multilevel Secure Operating System is to be designed to support multilevel secure applications. At present, we are prototyping a subset of the Dedicated Secure Operating System.

As our design has progressed, we have sometimes been asked to justify the need for these operating systems. In particular, we have been asked to explain why an existing operating system such as UNIX could not be used instead. This paper answers that question.

## What Does the Army Need?

The Army needs operating systems that will support their future tactical applications systems. Such systems have four basic characteristics. First, they will be coded in Ada. Ada is the language of choice for all future Army tactical systems. Second, they require real time response. Battlefield operations must be able to respond to events quickly or they lose their usefulness. Third, many of these systems will need to be multilevel secure. A number of battlefield applications must handle multilevel data, with cleared and uncleared Military Operational Specialty (MOS) types of troops. Finally, such systems must execute on Army approved architectures.

One other characteristic of Army tactical systems is that, in general, they do not execute in a program development environment. Thus, these operating systems do not need to support such activities as compiling, assembling, and linking. In fact, program development features must often be removed from fielded systems. One consequence of this is that while a command interpreter may be part of such a system, it is likely to be very application dependent, and thus is provided as part of the application rather than part of the operating system. It is only necessary that the operating system support execution of command interpreters.

The rest of this paper examines how UNIX and ASOS provide support for these basic characteristics.

## UNIX

The following subsections look at UNIX as a possible operating system for supporting Army tactical applications. This is somewhat difficult to do, because UNIX is really not a single operating system, but rather a collection of operating systems which share a common kernel design. When the ASOS project moved from a computer supporting Berkeley UNIX 4.1 to a machine supporting Berkeley UNIX 4.2, many of the tools that we considered part of the operating system no longer worked in the same way. Thus, while some of the assertions in the following subsections may not be true for *all* versions of UNIX, they all are true for *some* versions of UNIX.

---

*UNIX is a trademark of Bell Laboratories

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

### Support for Ada

There is nothing to prevent Ada programs from running under UNIX. In fact, several Ada translators, including the AdaEd interpreter and the Telesoft and Irvine Computer Sciences Corporation (ICSC) compilers are currently hosted on UNIX. In these compilers, the Ada Runtime Support Library (RSL) executes on top of UNIX, as pictured in Figure 1.

Each Ada program (where a program is simply a collection of Ada compilation units that have been linked together) is equivalent to a UNIX process. This has the following consequences:

1. A two level scheduler is required. That is, first a particular operating system process running an Ada program is scheduled, and then an individual task within that program. As a result, if one task in a given Ada program is waiting for I/O, all other tasks within the program are suspended as well.

2. A separate RSL (code and data) may be required for each Ada program.

3. In order to transition from Ada to the hardware, it is necessary to traverse many layers of software.

Thus, although there is no conceptual problem with executing an Ada application under UNIX, it is not efficient to do so. In addition, because there is no hardware separation between the Ada program and its associated RSL, the integrity of the RSL may be compromised.

### Real Time Response

UNIX was designed as an operating system that would execute in a timesharing environment. It is widely recognized as having superior program development features (although not a good user interface). As pointed out earlier, however, these features are not important in an operating system that supports Army tactical applications. Instead, it is important that such an operating system provide real time response. In this respect, UNIX has several problems.

One such problem is the granularity of the system clock. Although different versions of UNIX vary in this respect, some versions of UNIX only support a one second clock granularity. This will severely degrade the response to an Ada **delay** statement, for example. A second problem is that the basic UNIX kernel supports only one type of file, namely a stream file of characters. Although it is certainly possible to construct record files and indexed files on top of this basic file structure, doing so obviously imposes a performance penalty. As mentioned above, in order to support Ada on UNIX, tasks are scheduled secondarily to programs, which means that all tasks within a program halt if one task within the program is waiting for I/O. Finally, having many layers of software between the applications program and the hardware causes substantial execution overhead.

### Security

UNIX security leaves much to be desired. In his paper "On the Security of UNIX,"[1] Dennis Ritchie, one of the original UNIX developers, points out that UNIX was not developed with security in mind. In particular, UNIX is weakest in protecting against crashing or crippling the operation of the system, for example, through excessive consumption of resources such as disk storage, swap space, or processes. In addition, Ritchie points out that it is possible for any knowledgeable user to assume superuser status and thus sabotage the system.

With regard to the Trusted Computer System Evaluation Criteria[2] established by the Department of Defense Computer Security Evaluation Center, UNIX could not even be certified as a C2 system, let alone be certified as multilevel secure. Not only does UNIX not have any mandatory security, its discretionary security is not sufficient to be certified as C2. This is because C2 security requires access lists, e.g., a file would have associated with it a list of users allowed to read it, another list of users with permission to write it, and a third list of users allowed to execute it. All other users would have no access to the file. In UNIX, on the other hand, discretionary security is performed at a coarser granularity. In particular, the list of possible users is divided into mutually exclusive groups. Different permissions may be given to the owner of a file, to all other users within the owner's group, and to everyone else, but it is not possible, for example, to give a user outside the owner's group read permission to a file unless every other user outside his group is given read permission as well.

## ASOS

### Support for Ada

TRW's ASOS is an Ada operating system. By this we mean that it is specifically tailored to support Ada applications. In particular, ASOS has a single level scheduler, one in which an Ada task is the same as an ASOS task. Thus, tasks are scheduled independently of the programs in which they exist. Figure 2 depicts the ASOS three domain architecture. The operating system, which is a superset of the Ada Runtime Support Library, is implemented in two domains, the kernel and the supervisor, which contain the security related and non-security related portions of ASOS, respectively. Each Ada program has its own copy of supervisor data, but the

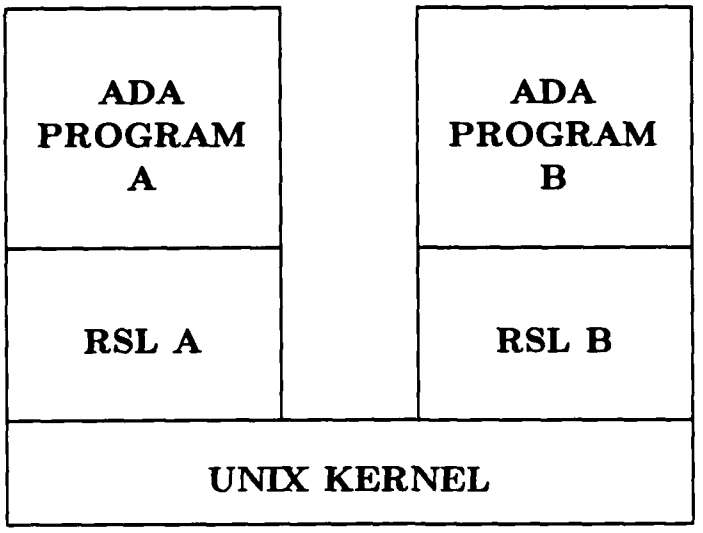



**Figure 1:** UNIX Ada Architecture

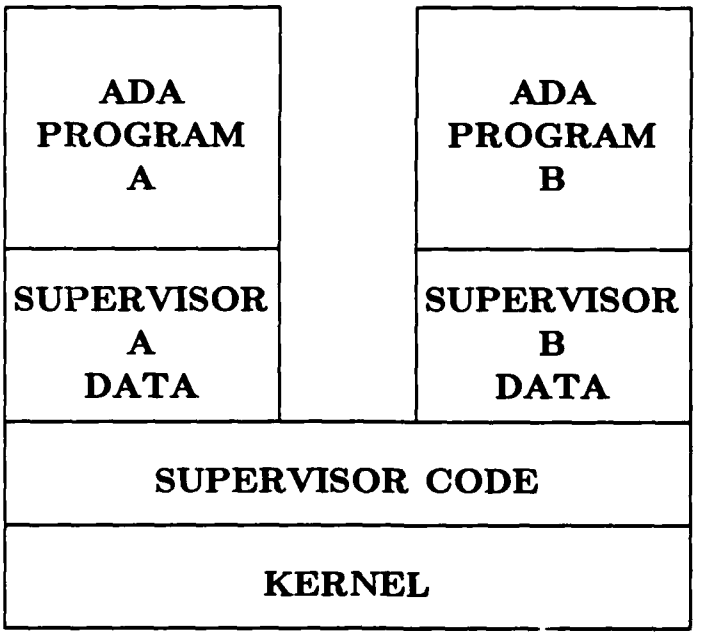



**Figure 2:** ASOS Architecture

supervisor code is shared among all Ada programs. Furthermore, unlike UNIX, there is hardware separation between the application and the RSL, which aids security.

An applications program can interface with ASOS in three different ways: implicitly, through Ada language semantics such as tasking or exception handling, and explicitly, through Ada standard packages such as those for input/output and through ASOS packages which provide capabilities beyond those which are part of Ada, such as those described in the next subsection. Therefore, ASOS serves as a replacement for and extension of the Ada RSL. The real time and security requirements imposed on ASOS dictate that much of the RSL be replaced.

### Real Time Response

ASOS is designed to support real time applications, not program development. This support takes two forms. First, we have added functionality to the operating system in the areas of deadline scheduling, semaphores, and keyed I/O. The user accesses this functionality by issuing Ada subprogram calls from within his applications program.

Unlike many current Ada systems, ASOS provides support for communication between tasks in separately linked programs. Our interprogram communication facilities are based, insofar as possible and appropriate, on the Common Ada Programming Support Environment (APSE) Interface Set (CAIS) being developed by the Navy's Kernel APSE Interface Team (KIT). Our multiple program environment will enable ASOS to be easily extended to a distributed computing environment.

### Security

ASOS will support both dedicated secure and multilevel secure applications. In particular, the Dedicated Secure ASOS has been designed to meet the requirements of a C1 system as defined by the Trusted Computer System Evaluation Criteria[2], while the Multilevel Secure ASOS has been designed to meet the requirements of an A1 system.

## Summary

ASOS functionality supports tactical applications rather than program development. Its response is designed to support real time requirements rather than timesharing. Finally, it will provide C1 level security for dedicated secure and systems high applications and A1 level security for multilevel secure applications. In summary, with regard to each of the characteristics required of an operating system to support future Army tactical systems, ASOS is superior to UNIX.

## References


[1] Ritchie, Dennis, "On the Security of UNIX," UNIX Programmer's Manual, Section 2, AT&T Bell Laboratories.

[2] Department of Defense Trusted Computer Systems Evaluation Criteria, CSC-STD-001983, 15 August 1983.


## Acknowledgements


This paper is the result of research conducted as part of the Army Secure Operating System (ASOS) project, which is sponsored by the United States Army Communications-Electronics Command (USACECOM) and the Department of Defense Computer Security Center, under Contract No. DAAB07-84-C-K541.



Eric Anderson has 15 years experience at TRW, as both a software manager and developer. His areas of expertise include real time operating systems and computer security. Currently, he is the TRW ASOS Project Manager. In the ASOS concept definition phase he was Deputy Project Manager and chief designer of both the Dedicated Secure Operating System and the Multilevel Secure Operating System, and designed the Task Management portions of both systems. He previously managed the "Security Kernel for Secure Operating Systems" IR&D project. He was a subproject manager on the SENTRY project operating system and a work package manager on the MIFASS project real time operating system. Previously, he was the project manager of the Kernelized Relational Information and Storage System (KRISS), and a work package manager of the Kernelized Secure Operating System (KSOS). He has an A.B. degree in Computer Science from the University of California at Berkeley and an M.S. degree in Computer Science from the University of Southern California.

Ruth Hart has ten years of experience at TRW in the design and development of operating systems and support software, as well in the analysis of high order languages. She has worked on the ASOS project since its inception, and is currently Deputy Project Manager and Work Package Manager for the Interim Operating System. During the concept definition phase, she was in charge of the Ada Language System Extensions (ALSE), coordinated the writing of formal specifications for the Multilevel Secure Operating System Kernel, and wrote Users Manuals for the Dedicated Secure Operating System and the Multilevel Secure Operating System. Previously, she coordinated Integration and Test of the prototype Engagement Controller Operating System (DEMO ECOS) on the SENTRY project. She was the Project Manager of the JOVIAL(J73) Validator project at TRW, and developed formal specifications in SP-EUCLID for the Kernelized Secure Operating System (KSOS). She has an A.B. degree in Mathematics from Cornell University and an M.S. degree in Computer Sciences from Purdue University, where she taught computer science for six years.

# Integrating Ada into Multi-lingual Systems: Issues and Approaches

**Michael J. Horton and Teri F. Payton**

**System Development Corporation**
**Research and Development**
**Paoli, Pa**

Abstract

This paper summarizes one particular Ada* transition study and extrapolates the results to identify areas of common concern in multi-lingual systems with Ada. It provides a discussion on modeling existing tasking and data access behavior with Ada constructs. Characteristics of the scheduler/dispatcher of an Ada run-time system are discussed. Of particular concern is the dispatching/preemption strategy for tasks of the same static priority which is left undefined by the Ada reference manual. Crucial areas for interfacing Ada and other languages are explored. This includes areas of concern at system generation time for linkage and data configuration, in the program level interface for data sharing and subprogram invocation, and in the system-level interface to effectively utilize the Ada run-time system. These areas are explored with respect to both importing foreign code into an Ada system and importing Ada into existing systems.

## I. INTRODUCTION

### Overview of the Transitioning Problem and Transitioning Models

Ada – the DoD common HOL – possesses many features supportive of software engineering principles such as packages, generics, and exception handling that the Ada software community believes will lead to reliable, maintainable software systems. Ada addresses the problem of programming-in-the-large and there are many existing large DoD software systems fielded or under development in different programming languages that may take advantage of Ada. Various transitioning approaches have been discussed in past studies [1,2]. Approaches to transitioning include:

a) Complete redesign/rewrite

b) Automatic translation

c) Incremental rewrite which may involve a multi-lingual system capability

The complete redesign/rewrite approach provides the most benefit from the use of Ada features yet may be the most costly to implement. Automatic translation runs awry when system executive calls, multi-tasking and direct code are involved. Additionally, an automatic translation approach only serves to take the solution that was suitable for a particular implementation language and perform a literal translation to Ada. It would not typically take full advantage of Ada's strong typing, encapsulation facilities, generics or other capabilities built into the language to facilitate maintainability and reliability. The third approach of incremental rewrite permits a phased migration from existing languages to Ada. Several strategies have been suggested for this approach such as:

a) rewriting or translating each foreign language procedure to its Ada equivalent maintaining the basic program structure

b) defining major logical sections and redesigning/rewriting these sections in Ada

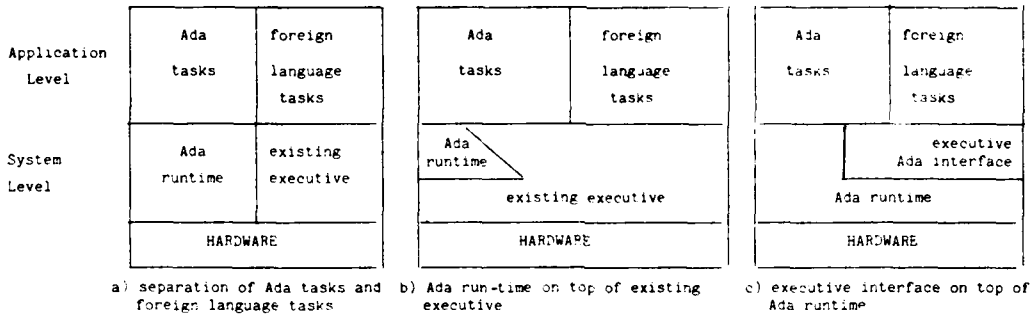
c) maintain existing code in the foreign language and implement new code (major upgrades) in Ada

In any of these approaches the degree and means of interfacing Ada with the foreign language must be addressed. Issues such as program structure, tasking models and data sharing arise. Among the most interesting existing systems to examine are the real-time, multi-tasking applications. Historically, each major development project specified its own executive closely coupled to the application and application designers view of tasking. Such real-time programs were often characterized as a set of cooperating yet independent tasks where scheduling techniques are a critical component and the sequence of task schedulings may be dependent on external stimuli. Typical implementation languages had no direct provision for a tasking model and the executive provided the means for scheduling, inter-task communication and interrupt handling. Much of the functionality of such basic run-time executives is inherent in the Ada language itself eliminating the need for an executive to drive the system.

Thus in the incremental rewrite or multi-lingual approach several views (as depicted in figure I-1) of the Ada system, the foreign code and the executive are possible. Figure I-a portrays the view in which Ada tasks are fairly well separated from foreign language tasks. The Ada tasks utilize the Ada run-time system; the foreign language tasks utilize the existing executive and the degree of interaction may be minimized. Such a view is feasible when one is rewriting major logical sections of the code.

In Figure I-b, the Ada run-time support may be built on top of the existing executive. Such a view may be extremely useful if we consider the strategy

*Ada is a registered trademark of the U.S. Dept. of Defense




a) separation of Ada tasks and foreign language tasks

b) Ada run-time on top of existing executive

c) executive interface on top of Ada runtime

Figure I-1: Views of Application/System level interfaces in Multi-lingual Systems

of using multiple Ada programs as the basic items interfacing with foreign language tasks as opposed to multiple Ada tasks within a single Ada program. The underlying executive would then continue to provide the necessary "tasking" semantics. This view would require validation of the Ada system utilizing the existing executive.

Figure I-c depicts a viewpoint where the Ada run-time support provides the basic executive capabilities and an interface is provided which maps existing systems functions into Ada primitives. The foreign language tasks need no modification and can retain their existent system level calls through this interface. The Ada compiler and run-time would not necessarily require modification and thus if validated on the particular hardware target, no revalidation would be required for use in the multi-lingual application.

These three views differ in the level of interaction between Ada tasks and foreign language tasks and the system-level interfaces. An Intermetrics study [1] recommends disallowing the intermixing of Ada and existing CMS-2 tasking code due to considerable technical difficulties. This paper summarizes an in-depth study of one particular CMS-2 multi-tasking system RNTDS (Restructured Naval Tactical Data System) in an attempt to qualify these technical difficulties and determine the technical feasibility of intermixing of tasks in the two languages while maintaining the existing tasking model.

During 1983, we conducted a study [3] for the Navy FLTCOMBATDIRSSACT, Dam Neck, Virginia concerning transitioning RNTDS to Ada. The opinions and conclusions expressed in this paper are those of the authors and do not necessarily reflect the official position of FLTCOMBATDIRSSACT. That study focused on two major areas:

1) the development of an Ada strategy for modeling full RNTDS tasking and data access behavior addressing all RNTDS requirements in an integrated approach.
2) the development of strategies for integrating Ada and CMS-2 in RNTDS programs.

These two areas of investigation addressed an overall goal of insuring the feasibility of transitioning RNTDS to an Ada framework while maintaining the behavioral characteristics of an RNTDS program execution. The viewpoint from which this study was conducted considers an incremental transitioning approach rather than a redesign or translation of RNTDS in Ada. It was assumed that existing RNTDS tasks written in CMS-2 must be viable without source changes in the combined CMS-2/Ada environment. Thus existing CMS-2 tasks retain the same interface to visible executive functions to handle executive service requests.

Two approaches were analyzed – the multiprogramming approach with each task set an Ada program and the viewpoint expressed in figure I-1c with the Ada run-time system serving as the underlying executive. A summary of this case study is provided in Section II. We believe the case study raises several issues with respect to:

- Ada runtime decisions that are not prescribed by the language reference manual
- Intertask communication and data sharing
- Task dispatch and termination
- System level and program level interfaces in a multi-lingual environment.

Section III addresses these and other issues of importance in integrating Ada in multi-lingual systems.

## II. RNTDS Case Study

The Restructured Naval Tactical Data System (RNTDS) is an existing Navy system written primarily in CMS-2. It runs on an AN/UYK-7 computer under the NTEP-7 run-time (operating) system.

RNTDS includes a large collection (library) of building blocks – individual appropriate for the various types of Naval ships based upon hull-specific information. It is instructive to look at RNTDS to see to what extent, if any, the use of Ada can be introduced into the RNTDS program, since, like many other existing programs written in languages other than Ada, it has its own tasking model which differs from Ada's, its own concept of data sharing, and its own method for constructing systems.

### RNTDS Tasking Model

The basic unit of work in RNTDS is the (single purpose) task. There are three categories of tasks: application, I/O, and system control.

The bulk of RNTDS tasks are application tasks which perform functions related to carrying out the program's mission: the processing, evaluation, display, etc. of tactical data.

I/O tasks perform I/O operations on behalf of application tasks for a specific peripheral device.

There are several categories of system control tasks which include executive tasks called executive service routines (ESRs) which perform system services such as task scheduling and dispatching; error processor tasks, which respond to system or application-detected error conditions; and Common Service Routines (CSRs), which in general provide system support (such as performing mathematical calculations), but could also be used to share tactical-oriented code between separate application tasks.

In RNTDS, application tasks are scheduled for execution in groups of tasks called *task set chains (TSCs)*, rather than as independent tasks. The order in which the tasks within the TSC are dispatched (executed) is built into the definitions of the individual TSCs.

The set of TSCs in any given RNTDS program, as well as makeup and ordering of each individual TSC, is static and is specified as part of the program specification process. The RNTDS system designers have complete freedom to group application tasks out of its library together into "complex" task set chains since application tasks are, by design, totally oblivious of what the other tasks are in any TSC, and since the same application task may be included in any number of different TSCs.

Within an individual TSC, the tasks are linked together, possibly including loops, and are dispatched and executed in the order in which they are linked together. The component tasks of a TSC are partitioned into *sets* of tasks, and the sets are further partitioned into *subsets*. The subset boundaries are significant because the tasks within a particular subset are allowed to execute in parallel (on separate processors) if a sufficient number of processors are available. The set boundaries are significant because these represent points at which the execution of a TSC can be terminated: Any task within a TSC can issue a call to the *Terminate TSC* executive service routine(ESR). When such a call is made, the TSC execution will be terminated at the end of the "current" set of tasks – where the current set is defined as the set which includes the task making the terminate request.

Figure II-1 is an example of a TSC, which includes two sets, one of which includes two subsets. This example also illustrates a loop – meaning that potentially, this TSC would continue to execute "forever" once it is scheduled – unless one of the tasks within the chain calls the *Terminate TSC* ESR.

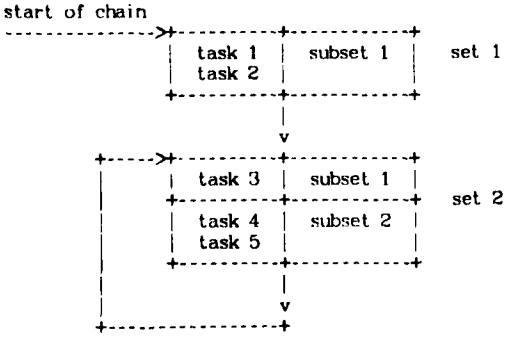



Figure II-1 Example of Task Set Chain

**Scheduling TSCs**

A TSC may be explicitly scheduled by another task, possibly with a delay, via a call to an executive service routine, or can be implicitly scheduled upon the detection of a particular type of event such as the detection of an error condition or an interrupt. Time-dependent tasks can also be scheduled at regular intervals – triggered by real-time clock interrupts.

Each TSC is scheduled at a particular dynamic priority level which indicates the "tactical significance" – and, therefore, relative importance of – that particular scheduling instance of the TSC. That dynamic priority is "inherited" by all tasks in the chain and facilitates task dispatching. Each scheduling of a task set chain may be at a different priority level.

TSCs which are explicitly scheduled by another TSC's task are added by the run-time system to a system queue for later dispatching. After that point, the scheduling and scheduled TSCs are totally independent of one another: they will compete against one another for system resources (including CPU resources) and can terminate in any order.

The same TSC may be scheduled any number of times concurrently – each instance possibly at a different priority – the only practical limitation being determined by the size of the run-time system's scheduling queue. The run-time system supports "planned prioritized degradation" whereby it disregards previously scheduled, low priority TSCs in favor of higher priority TSCs when it encounters overload situations.

**Task Dispatching, Prioritization**

The RNTDS run-time system supports three levels of execution: *Normal Dispatch* (the lowest level), *Interrupt*, and *Error* (the highest level). The vast majority of RNTDS tasks execute at the Normal Dispatch level. The Interrupt and Error levels are primarily reserved for the run-time system's initial response to the detection of an interrupt or error condition, respectively; most typically, the run-time system's response is to schedule the appropriate interrupt handler task or error processor tasks for later execution (dispatch) at the Normal Dispatch level.

The run-time system maintains a queue of tasks which are ready and waiting to be dispatched at each execution level. Each task in the dispatch queue has a priority associated with it; more precisely, each instance of a task within the queue has a priority associated with it, since the same task may be in the dispatch queue any number of times. Whenever a CPU becomes available because the task executing on it completed, the run-time system assigns the highest priority task within that queue to that freed up processor.

One important requirement of the run-time system is that individual tasks will, in general, be allowed to "run to completion" without being preempted by any other task at the same execution level. This means that when a task within a "low" priority TSC schedules a higher priority TSC, the scheduling task will regain control of the CPU and will go to completion without being preempted by the first task(s) in the newly scheduled TSC. Another requirement is that no individual task will be execut-

ing concurrently on two or more CPUs. The *only* CPU preemption that the RNTDS run-time system supports is the preemption of a task executing in the Normal Dispatch level when an interrupt or error is detected, or the preemption of a task executing in the Interrupt level when an error is detected.

**Intertask Data Communication**

RNTDS tasks are, by design, rather limited in their ability to communicate with one another.

The primary data communications medium among RNTDS is the common data base (CDB). All data which is valid beyond the execution of any individual task (i.e., all residual data) lives in the CDB. It is a shared memory resource available to every RNTDS task. In addition, a task which is scheduling another TSC can pass a data packet containing information about that specific invocation of the scheduled TSC. That packet will be available to each task within the scheduled TSC.

## III. Transitioning Issues and Approaches

### Run-time Support Issues in Transitioning to Ada

In our RNTDS case study we encountered several aspects of its run-time model which we found at odds with or undefined in (i.e., left to the implementor's choice in) the Ada model. These aspects included:

1) task scheduling,
2) task prioritization and dispatching,
3) "graceful degradation" under overload situations.
4) intertask communications,

The following subsections will examine these issues more closely and will, whenever possible, propose possible ways of modelling these aspects of RNTDS in an Ada framework – i.e., though interaction with the Ada run-time support.

**Task Scheduling**

In RNTDS, application tasks are always scheduled in groups within a task set chain. This aspect was quite easy to model in Ada: Each RNTDS task could be defined as an Ada task type, with the task body performing the intended function. The task set chain could also be defined as an Ada task type, whose body would control the order in which tasks were executed by activating them in the appropriate order and by making use of Ada tasking semantics which define points at which one task would await the completion of other task objects which it activated. The use of task types in representing RNTDS tasks and task set chains would easily permit one to create any number of separate "instantiations" of the RNTDS objects concurrently as required in the RNTDS model.

Figure III-1 illustrates an example of what a task set chain description could look like in Ada, modelling the behavior of the task set chain described in figure II-1. Note that the requirement that RNTDS tasks within the same subset are potentially executable in parallel is easily encapsulated in Ada by activating all of the tasks in the same subset within the same Ada **block** statement.

```
task body typical_tsc is
    ...
begin
    ...
 begin                          -- first set
   declare                      -- first subset in first set
    t1: PKG1.task_1;
    t2: PKG2.task_2;
   begin ... end;
 end;

 while ESR_TERMINATE.has_not_been_requested loop
  begin                 -- second set
    declare             -- first subset in second set
     t3 : PKG3.task_3;
    begin ... end;

    declare             --second subset in second set
     t4 : PKG4.task_4;
     t5 : PKG5.task_5;
    begin ... end;
  end;
 end loop;
end typical_tsc;
```

Figure III-1 Ada Code Fragment Describing Sample TSC

An aspect that cannot be as simply modelled in Ada is the *delayed* activation of task set chains, where one task can schedule another set of tasks for later activation. Ada does support the **delay** statement; however, such a statement cannot be used in a task which wants to delay the activation of another task set chain, because the activating as well as activated task(s) would be delayed, violating one of the RNTDS tasking requirements that application tasks, once started, are supposed to run quickly to completion without delay (i.e., without any preemption).

However, one can model the scheduling delay by putting the burden for initiating the delay onto the scheduled task set – by putting a **delay** statement at the beginning of the task set chain body, and by having the amount (if any) of the delay to be passed (via rendezvous) to the scheduled task set chain.

**Task Prioritization and Dispatching**

RNTDS (and many other existing systems written in languages other than Ada) assumes a particular model for task dispatching that differs from Ada's model. Repeating some of the characteristics of the RNTDS run-time system:

- dynamic priorities
- multiple scheduling of task set chains
- task lock-out – the same task will not be executed concurrently on different CPUs.
- tasks of a given execution level (normal, I/O or interrupt) are non-preemptable by tasks at the same execution level.

The Ada run-time characteristics as specified in the Ada reference manual include the following:

- *Static* priorities may be assigned to an Ada task type and are associated with every activation of a task object of that type.
- There is no guarantee (i.e., policy) defined in the Ada standard that would ensure that objects of that same task type could not be dispatched on different CPUs concurrently.
- The Ada standard leaves the CPU preemption policy up to the run-time implementor's choice.

We do not see any way to use Ada's static priority scheme to support run-time requirements such as RNTDS's, except to the extent that static priorities could be used to distinguish application, I/O and system control tasks from on another. If Ada is to be transitioned into systems with similar requirements, we see the need for at least the necessity of implementing a customized scheduler/dispatcher into the Ada run-time system, possibly going as far as implementing a multi-programming system which could allow multiple Ada programs to be executing concurrently.

A customized scheduler/dispatcher would be needed to guarantee that an existing system's CPU preemption policy were enforced – at least for those tasks already written in the "foreign" (i.e., non-Ada) language.

To implement a dynamic priority scheduling policy similar to that built into RNTDS would take more dramatic action:

If one tried to support dynamic priorities in the context of a single Ada program, one could add a system-level package similar to that specified in figure III-2. Such a package, which would have to be accessible to applications code, would allow that "non-standard" attribute to be passed down to the run-time system. A routine such as *set_priority* could be interpreted, for example, to assign a "dynamic priority" to the Ada task making the call, or perhaps, to tasks which the current task would later activate. Within the same ***static*** priority level, the customized dispatcher could use this new attribute of a task in basing its dispatching decisions.

```
package SCHEDULER is
  max_priority: constant integer := impl. defined;
  type priority_level is integer range 0..max_priority;
  procedure set_priority(dyn_priority: priority_level);
     ...      -- additional scheduler specs go here
end SCHEDULER;
```

Figure III-2 Specification of SCHEDULER Package

In a multi-programming environment which allowed one program to initiate the execution of another Ada program, one could presumably include a "priority" to be associated with that newly scheduled program. Such a priority-scheduling policy is very common in multi-programming environments, and is outside the scope of the Ada standard.

**Graceful Degradation**

In RNTDS, when the system load becomes "too high" and the run-time queues are full, the run-time system is allowed to terminate the execution of lower priority task set chains in order to permit higher priority tasks to continue to run.

The Ada standard does not specify behavior under similar overload conditions, yet it is an issue with implementations for at least small, memory-limited embedded systems. We believe that a run-time system could be implemented which could abort or raise a system-defined exception in low priority tasks when it was unable to activate a higher priority task because of a "system overload" situation.

**Intertask Communications**

RNTDS has an intertask communications scheme which passes packets of information between tasks and which shares data amongst tasks in a "common database". We are convinced, based upon our earlier study, that such communications schemes can be modelled in Ada using the rendezvous and shared data.

There is, however, an aspect of RNTDS's "intertask communications" that is not easily expressed in standard Ada terms: One task (in a task set chain) is able to signal the system (and indirectly, the other tasks in that same chain) that execution of the tasks in that chain is to terminate ***after all the tasks in the current set are finished executing.*** An RNTDS task does so by calling upon a system service routine and it does so "anonymously", i.e., without identifying itself to the system service routine – relying on the fact that the system knows which task set chain it is a part of.

In figure III-1 we show an example of an Ada model for a task set chain "type". For a "child" application task to signal the parent that it is to terminate itself at the next set boundary – and do so anonymously, without identifying its parent – would require direct support from the run-time system, again in the form of a system-level, applications-visible package.

**Logistics of Interfacing Ada with Other Languages**

The evolutionary transition of large DoD multi-tasking applications from a foreign language/executive combination to Ada involves interfaces at several levels. These include:

a) system generation – linkage and data configuration level

b) the programming level

c) *the system level.*

At the linkage and data configuration level, several views are presented regarding the integration of existing system generation capability (such as exhibited in RNTDS) with standard Ada tools. At the programming level, the need for sharing data and interfacing calls between components of the various languages is discussed. At the system level, an executive/Ada interface is explored to interface between service calls by the foreign language routines and an Ada run-time support. The executive/Ada interface is considered as emulating necessary executive functions by transforming executive actions into actions supported directly by the Ada run-time support library.

**System Generation – linkage and data config.**

The primary aspects of this phase considered in our case study are:

1) the linkage of object modules of different languages – assembly, Ada and other high level languages

2) the determination of segments

3) the modification of code to utilize appropriate base registers.

These aspects of the system may be addressed from two points of view:

1) system generation of the foreign code system prior to (separate from) Ada linkage

2) system generation of a single integrated application system.

## Generation and Link Time Considerations

In our case study the existing system generation process utilizes the RNTDS information bases to configure the system, selecting appropriate library tasks and determining program and data segmentation. Once the designer has selected task sets, the process automatically performs the remainder of the functions for generation – link, program and data segmentation and base register setup. Two approaches were considered:

1) use the existing system generation process for existing task sets whose tasks are all foreign language routines and to rely on the Ada linker to select appropriate tasks from the library and perform the segmentation for new task sets implemented in Ada.
2) a single integrated system generation regardless of implementation language.

Figure III-3 depicts the point of view of the system generation process preceding linkage with Ada. This is the view postulated in the early RFI[4] for ALS/N for integration of MTASS. Note: the total view taken in the ALS/N and in the Army's ALS[5] is that if an interface to foreign code is permitted on a particular system, then the foreign code is linked into one unit in which all external references have been resolved.

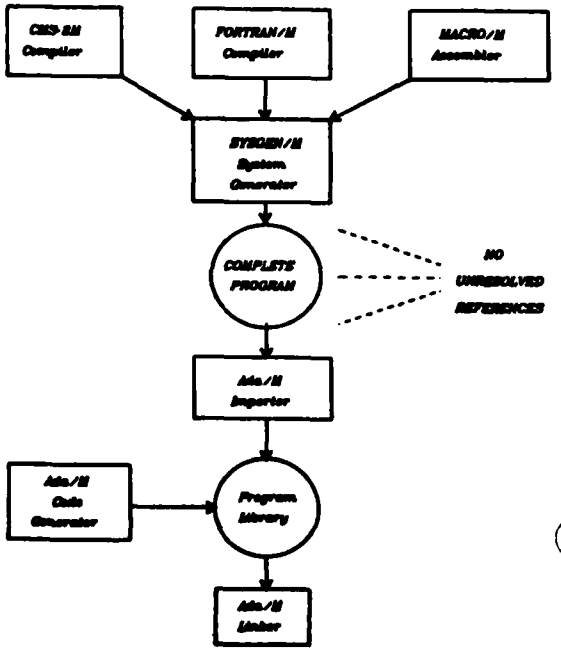



Figure III-3
ALS/N RFI Generation Relationships

The advantages of this approach are:

a) little if any modification would be required for either the existing system generation process or the standard Ada linker
b) since the Ada tools remain intact for handling Ada, no revalidation of the Ada system would be required

The disadvantages which appear to make this approach untenable are:

a) no foreign language task or assembler routine may call on any Ada subprogram
b) since no foreign task could cause the initiation of an Ada task set, the designer must consider the implementation language when determining processing paths
c) the system generation process itself must be cognizant of the language associated with each set of tasks.

This view treats Ada as a special language, the primary language of the system, an approach which may not be suitable for an incremental transition strategy.

System generation is a process to be applied to an entire application *system* and thus this separation of activity based solely on choice of implementation language should be questioned as a viable mechanism for transitioning. The construction of systems (the generation process) should be independent of the implementation language of subcomponents of the system. At the post-compilation phase, modules are available in relocatable object code destined for the same target machine. The format of the relocatable object module may differ depending on the compilation/link system used and thus require reformatting to "import" the code into the system. We suggest an integrated approach whereby multiple segments of object code may be imported prior to the resolution of external references. As depicted in Figure III-4, this can be viewed in two veins:

1) importing foreign language code into an Ada system
2) importing Ada code into the existing system generation process.

The first approach is consistent with Ada language requirements. The language reference manual[6] does not specify the linking procedures. We advocate this approach when the view of a single Ada program as the encapsulating system structure is followed.

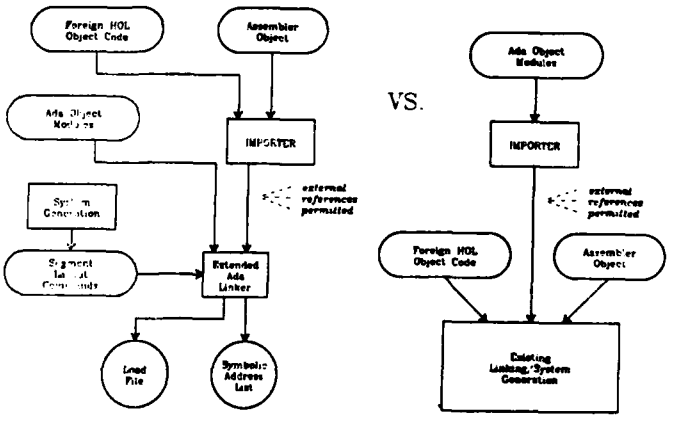



Figure III-4
LINKING SCENARIOS

This second approach is extremely reasonable if the multiprogramming Ada model is followed. The advantages of this approach are:

a) Ada is viewed as an implementation language on the same level as the other languages used in the system

b) the system generation process continues to have complete control of data and program segmentation

The disadvantage of this approach is:

a) A validated Ada compiler consists of the compiler/linker and run-time support. This approach provides an alternative mechanism for linking and may require re-validation of the Ada system although in the case of a multiprogramming model the issue of combining multiple Ada programs to form a system has not been addressed with respect to Ada validation.

**Machine Dependent Characteristics:**

**Segmentation, Base Registers**

On small machines with limited memory, issues such as segmentation become important. In our case study, a tool – the Configuration Control File Generator – determines the segmentation of data and programs and the setup of appropriate base register information.

A standard Ada compiler/linker will need to utilize the available base registers. In RNTDS, the determination of segments and choice of base registers is primarily a post-compilation activity. An Ada system will need to support comparable notions. The question is the level of control available to the user or automatic system generator. Various options are possible:

a) complete control of the segmentation and base register determination is left to the compiler/linker

b) the compiler supports an implementation-dependent pragma for control of segmentation

c) the user (either by hand or automatically) includes representation specifications for data and tasks (this may necessitate an additional compilation after the generation process which determines the segmentation)

d) the linker accepts user commands to assist in this determination.

To uphold existing system semantics, we believe linker commands are the appropriate approach. The segmentation information is truly a system-wide activity and thus in addressing separate compilation and programming-in-the-large, it is a post-compilation activity.

**Program Level Interface**

At the source program level, aspects of both data sharing and program invocation must be addressed.

**Data Sharing**

The RNTDS model, similar to many real-time multi-tasking DoD systems, incorporates the notion of a common data base (CDB) which is accessible by different, independent tasks. This notion compromises the reliability of the system as compared with the more controlled access features inherent in Ada's scope and visibility rules. Yet, the common data base is a prime component in the underlying design of many existing systems and cannot easily be avoided in an incremental transition.

The shared data areas act implicitly as a file to serve as a storage area between different accessing agents. Ada packages describe the CDB in a manner analogous to existing CMS-2 sys_dds constructs. When foreign language routines are invoked from Ada the Ada global data is not visible to the foreign routines unless data items are explicitly passed to the foreign subprograms as dictated by the Ada reference manual.

In a multi-lingual system, routines in either language may require access to the same data segments. For example, sys_dd descriptions can be utilized in a CMS-2 task while equivalent Ada packages are utilized in the Ada tasks. The CMS-2 compiler utilizes the sys_dds. The Ada compiler uses the package. The system generation process can utilize either. It is important to insure that Ada language rules have not been violated. This approach provides an aliasing capability between the languages. Yet, it can be viewed as similar to the legal case where representation specifications are utilized to access a particular location which may also be accessed outside the scope of the Ada program. Indeed, after the system generation process, representation specifications could be automatically inserted in the Ada code to achieve the same effect in a perhaps less clear manner. In the multiprogramming view, the CDB can be viewed as an in-core file that is accessed by multiple programs, thus there is no conflict with Ada semantics.

It is important to note that this model of data sharing does not explicitly enforce synchronization. Indeed, in RNTDS, synchronization is given by the task set structure (signaling the initiation of sets of tasks when data is available) and not via explicit locks on the data. The stimulus-response paths defined by RNTDS task sets are retained in the multilingual model.

**Subprogram Invocation**

The invocation of subprograms from the foreign language must be examined in the context in which they could logically occur. In our case study, we determined it was not necessary for application-level CMS-2 tasks to directly invoke application-level Ada subprograms or tasks. In this case, each task was a small, indivisible sequence of instructions (100-200 source lines) that is single-purpose. We recommended that while we support the intermixing of tasks written in different languages to form specific task sets, intermixing high-level languages within a task (calls) did not seem necessary. A minimal level of effort would be required to convert the entire task. Note: in our case study, activation of other tasks was accomplished via system level calls. We recommend foreign language application tasks be permitted to invoke Ada system-level routines or foreign language coded system interfaces be allowed to call Ada system-level routines.

**System Level Interface**

In the modeling of RNTDS behavior in Ada that was undertaken in the case study, no modification was to be required of existing foreign language application tasks and thus the existing tasks continue to

utilize executive calls to perform system functions. Ada tasks will rely primarily on the Ada run-time support library. The executive and the Ada run-time support each provide the necessary task management functions. To support a single application system, we require a common underlying task management system or data structures. To achieve this mechanism, we studied interfacing the existing executive to the Ada run-time system– the executive being an interface between the foreign language tasks and the Ada run-time system. The application system then consists of both Ada and foreign language tasks executing in a cohesive environment. The base of this environment is the Ada run-time system. The "glue" of this environment, providing the capability to interface existing tasks, is the executive/Ada interface.

A crucial issue to the implementation of the executive/Ada interface is how one envisions that interface. It can be conceived of as:

a) an interface sitting strictly on top of the Ada run-time support (i.e. no special access – simply an Ada program). It would not directly be involved in the task scheduling/dispatch process but would act as a buffer task indirectly causing the scheduling of desired tasks through the Ada run-time system.

b) an extension of the system level capabilities which is permitted access to a selected set of run-time support routines and data structures.

Both these views are forms of the approach given in Figure I-1 in which the executive/Ada interface sits on top of the Ada run-time system. We believe that the interface implementation as well as the Ada run-time system must both have access to the system data structures for efficient task management purposes. This can be accomplished in several ways:

1) direct code to access specific memory locations
2) system data structures are visible to the executive/Ada interface
3) Ada run-time procedures or entries are visible to the executive/Ada interface.

Case (3) is the cleanest and most preferable mechanism. It fosters the Ada concept of an abstract data type. Rather than choosing case (2) and giving the interface direct access to the data structures,an Ada package can be provided encapsulating the data structures and the visible operations on those structures. As in an abstract data type, the data structures themselves are hidden and the visible subprograms provide a controlled access mechanism to system structures. The disadvantage of this approach is that the design and packaging of the standard Ada run-time system becomes important. The granularity of the Ada run-time system should be such that necessary primitive operations may be made visible to this extended system interface. The executive/Ada interface must be able to engage in activities such as:

- scheduling a task
- determining the parent of a task
- associating bits of data such as flags for termination of task chains or dynamic priorities with task control blocks
- scheduling an I/O request
- modifying base registers to change segment access.

## IV. Summary and Conclusions

The issues raised in this report can be largely handled within the bounds of Ada semantics. Yet, a particular compiler system may thwart transitioning attempts as the Ada language reference manual permits the implementor a high degree of freedom in choosing strategies that effect important transition topics such as task management, linking and interface to foreign code. The following paragraphs outline issues in these areas that must be examined in the particular compilation system for incremental transition and potentially for new development of large DoD application systems.

### Task Management

With respect to the Ada run-time system two areas are critical to transitioning:

1) the structure of the run-time system
2) the scheduling/dispatch algorithms for tasks of the same static priority.

The run-time support system should be structured in a manner facilitating the interfacing with other system-level functions, permitting system-level functions external to the Ada run-time support library to invoke system primitives. Such primitives might include:

scheduling of a task or an I/O request

determining the parent of a task

setting and interrogating executive-specific flags associated with a task control block (dynamic priority, task set termination)

modifying base registers to change segment access.

The Ada language does not prescribe the behavior for scheduling tasks of the same static priority. The scheduling algorithms for dealing with tasks at the same static priority level should be a replaceable unit - the chosen implementation controlled by the application manager. In particular, task lock-out, time-slicing, task pre-emption and task selection from a list of ready tasks are not addressed. Existing systems in transition may require or forbid the strategy chosen by the implementor. This may have ramifications with respect to validation. With respect to priorities, the language reference manual does not prescribe a range of priorities. The range permitted by a particular Ada implementation may not be [illegible] tion system model under transition.

### Linking/Interface to Foreign Code

The implementor has complete [illegible] choice as far as user commands to the linker and the level (if any) of support for foreign languages. Implementation of the INTERFACE pragma is not required. The implementation is also permitted to place restrictions on the allowable forms and placement of parameters. Thus an implementation which *does* support the INTERFACE pragma could restrict the user from passing any information and allow the call to act simply as a trigger of the foreign language tasks. The Ada language reference manual does not provide for sign code from invoking Ada routines nor does it state a means of doing so. Conventions for such call are left to be defined by other language processors.

Thus, with respect to linking and interface to foreign code, the designers of a major application transition must determine in the context of the application what is required and seek an implementation that is sufficiently flexible to meet the requirements.

## BIBLIOGRAPHY


[1] CMS-2 to Ada Transition Plan. Draft. Intermetrics. October 1981.

[2] CMS-2 to Ada Transition Plan. Draft. Softech. October 1981.

[3] Study Report on the Transition of RNTDS to Ada. SDC. May 1983.

[4] Draft RFI, Ada Language System/Navy. Navy Sea Systems Command. October 15, 1982.

[5] Draft. Ada Language System Specification. CR-CP-005 9-A00. Softech. June 1981.

[6] Reference Manual for the Ada Programming Language. ANSI/Mil-Std-1815 A. U.S.Department of Defense. January 1983.


### Biographical Information

Michael Horton is a member of SDC's research staff in Paoli, Pa., where he is the principal investigator of the Environment project which is developing a common programming environment for SDC. His research interests include software and microprogramming environments. he has a MSE in computer science from Univ. of Penn. and a BSEE from CCNY. He is a member of the ACM and IEEE Computer Society.

Teri Payton is Manager of the Software Technology department in SDC's Research and Development Division. She is leading R&D efforts towards the development of a UNIX-based, Ada-supportive environment. Ms. Payton has been active in the Ada community since 1980 and served as former Secretary and current co-chair of the Environments Subcommittee of SIGAda. Her major research interests are language design, compilation techniques and software environments. She holds an MS in computer science from Villanova and a MS in Mathema[illegible] Computer Science. She is a member of the ACM and IEEE Computer Society.

# EXPERIENCES IN ACQUIRING AND APPLYING ADA TO THE SUBACS PROJECT

Oliver Cole and Steven North

OC Systems, Inc., 119 S. Ingram St., Alexandria VA 22304

## Abstract

Ada (1) is being used on the DoD Submarine Advanced Combat System (SubACS) project, a large-scale, embedded DoD system. The intent of this paper is to give a high-level view of the Ada being used on SubACS, and to share some of the experience that we have gained from our effort. The SubACS project is briefly described, followed by a description of the Ada being used. As of February 1985, SubACS Ada will be in production use by thirty programmers. Experiences acquired during the production use of Ada on the SubACS project will be published at a later time; the results so far are encouraging.

## SubACS

The SubACS project will integrate the combat and acoustic subsystems of a submarine into a single system capable of making combat related decisions based upon sensor data. It is a large project, requiring hundreds of man-years of effort in software and hardware development.

The system will consist of AN/UYS-1 and other signal processors communicating acoustic processing data to AN/UYK-44 and Motorola 68000s. The primary means of communication between the processors is a redundant fibre optic bus running the "length" of the submarine.

The system is fault tolerant. An error in one portion of the system does not propagate. Hardware and software controls ensure that a faulty program cannot crash the system. The runtime environment includes multiprogramming and multitasking. Multitasking is required by the nature of the real-time SubACS application, and multiprogramming is used as part of the software architecture to support the isolation of major functions.

The system is distributed. Programs are dynamically assigned processors and memory resources as they are run. Lastly, the system is multilingual. SubACS is using Pascal, CMS-2M, Ada, SPL/I and 3 different assembly languages.

## The Ada

The Ada used by SubACS is vanilla Ada. There are no implementation-specific features, such as representation specs or address clauses. There is a single implementation defined pragma: an interface pragma. The Ada on the AN/UYK-44 is virtually identical to the Ada for the 68000 except for the target operating system calls, which the user accesses via an interface pragma.

Currently, SubACS is using a subset of Ada. This subset excludes floating-point, exceptions, and much of tasking. The syntax of single tasks is retained by the subset. As a result, SubACS tasks are declared as Ada tasks and share data and code according to the visibility rules of Ada. The alternatives to using the Ada single task construct were ungainly.

The Ada being used by SubACS consists of the TeleSoft-Ada Front-End, and three code-generators: one for the System/370, one for the AN/UYK-44, and one for the Motorola 68000. There is single Ada Front-End for all three targets and it comprises about 70 percent of the bulk of the compiler. There is no AN/UYS-1 Ada code

generator. The AN/UYK-44 and 68000 are being developed as targets; the System/370 is used as a host.

The three different code-generators were developed in parallel by different sub-contractors and have different designs. The AN/UYK-44 code generator generates assembly language for an existing set of support software called MTASS/M. The code generator is broken into two halves: the first half transforms the graph representation output by the front-end (called Low Form) into a sequential, pre-fix, largely machine-independent representation (called Sequential Low Form) form. The second half of the AN/UYK-44 code generator generates the AN/UYK-44 assembly language from the Sequential Low Form. This design has proved to be lowest risk and conceptually the simplest. Most of the sequentializing half of the code generator can be re-used for future retargets.

The 68000 code generator is monolithic; it is invoked once during the program build process, and generates code for all library units at that time. Because of this design, a number of optimizations are available, such as eliminating un-called procedures, using minimum sized addressing, and rearranging elaboration code to produce a single "procedure" to elaborate the main program. Additionally, to support the requirements for reconfigurability, the machine code for the 68000s must be position independent. The 68000 does not allow 32-bit displacement in the position independent addressing modes, but the monolithic code generator style can easily handle full 32-bit code displacement when it occurs, whereas a non-monolithic code generator cannot.

Unfortunately, the monolithic style forces code generation to occur on all code, even if unchanged. Program re-builds are often desired after relatively minor code modifications and it is unnecessary to require such extensive recodegeneration in such cases (Ada compilers are slow enough already).

Listings are also a problem with the monolithic style of code generators. Much debugging will be performed at the hardware level in SubACS due to the lack of good source code debuggers, and good machine-code/source-code listings are needed. Unfortunately, the attractive optimizations in a monolithic code generator result in a great deal of code-motion, making the listings substandard. For example, the monolithic code generator emits the library package body statements and main program elaboration code in a single procedure, saving the procedure linkages that are required in a non-monolithic code generator. The resulting machine-code is difficult to correlate to the original source, however.

The last of the three code generators, the System/370 code generator, transforms the Low Form into System/370 code without an explicit sequentializing stage. The code generator makes a recursive descent over the Low Form, and outputs assembly language or object code.

Overall, the sequentializing non-monolithic style of code generation seems to have turned out best. It is easiest to understand and to re-target, and is producing good quality code. The monolith style has advantages in some types of optimizations, and the overall generated code organization is simpler, but the added cost of unnecessary code generation seems to outweigh the other advantages. It is still early, however.

## Testing

IBM has put significant effort into testing these compilers. The intent of IBM's testing is to provide SubACS with a useable Ada, not necessarily a quick AJPO validation. Issues such as performance and capacities take precedence over the implementation of unneeded features.

SubACS Ada is intended for embedded processors and has no need for human readable I/O. The target hardware for SubACS Ada is not completed, and, in any case, is not intended to be a software testbed. Testing for the 68000 and AN/UYK-44, therefore, is done with hardware simulators running on a System/370. The tests for the System/370 compiler are run on the System/370 directly.

Because of the multiple testing environments, a specific commenting guideline was adopted for all tests to allow simple test usage. Special comments are included in all tests, and are defined as part of the guideline as follows:

--www Test name

Test name is the identifier of another test. The --www comment is a directive to compile the named test before this one.

--ppp

A --ppp comment specifies that this file should be ignored until it is mentioned in a --www comment by another test. A --ppp test has no meaning, except in the context of another test. When mentioned specifically in a --www comment, then this test is compiled normally, and any special comments (except the --ppp) are honored.

--xxx integer

All tests are written as functions that return an integer. The test, when run, must return the same integer as the --xxx comment, or the test has failed. The test itself may be arbitrarily complicated, but the result itself, is a simple pass/fail.

--date Month Day Year

A --date comment describes the last date the test was changed.

--### < text >

A --### is a single line description of the test.

Additionally, a file naming convention is used. The first letter of each file name determines the type of a test: an X indicates that the test should compile and execute, a P indicates that the test should fail in compilation, a K indicates that the test need only compile to pass. These character assignments have been chosen so as not to conflict with the ACVC naming conventions.

For example, the following comments:

```
--date August 17, 1984
--xxx 7
--www HARRY
--ppp
--### test problem report 541
```

mean that this test was last modified August 17, 1984. This test returns a 7, if successful. The package HARRY must be compiled before this test. This test need not be compiled, it will be mentioned by a --www comment in another test.

By adopting this strategy, a simple command file can be written, for any host, which runs all the tests automatically by looking for special comments. The simplicity of this scheme has greatly eased the maintenance of our continually evolving test library; we would soon be overwhelmed without it.

Early in the development of the compilers, simple tests were written to test basic compiler functions. These tests rely heavily on code scraps from the Ada reference manual, are typically a few lines in length, and test a single function. The compilers have since progressed beyond these tests, and they are no longer used; larger test cases have replaced them. These tests will be useful for initial testing on new code generators, however.

Compiler capacities were the subject of much debate. Specific capacity requirements for the compilers are not known in detail; the only requirement is to compile SubACS. In general, capacities were overemphsized for fear of running into a brick wall. Over-sized capacities will be adjusted during maintenance, as part of an ongoing effort to increase compile-time efficiency. A number of the tests are designed to validate our "best guess" of the required capacities.

There are hundreds of tests in the current test library. Some are from the ACVC test suite, most have been written to test specific functions or capacities that are critical to SubACS. It is our understanding that IBM will make these tests publicly available.

Operational Prototypes were developed to test the operability of the two Ada targets. Multitasking SubACS "applications" were written, and run on SubACS prototype hardware. There operational tests are run on each new compiler delivery and have become de-facto acceptance tests of new compiler deliveries. The prototypes are not large, but they are good tests. Because SubACS defines many pre-runtime and run-time interfaces over and above those of Ada, the early development of operational prototypes was critical to the testing of SubACS Ada.

A Heap Sort program is used as a rough measure to gauge code quality as the compilers progress. The Ada-Europe Guidelines tests for code quality were implemented and are used to guide optimization efforts. Currently, the Ada compiler for the AN/UYK-44 generates code of quality better than that of the CMS-2M compiler. The 68000 produces code of quality approximately equal to that of the Language Resources Pascal compiler, at this time.

In the most ambitious testing scheme, IBM wrote a random test case generator. The test case generator produces random, yet equivalent, Pascal and Ada tests. The two tests are run and the results are compared. If the two are equal, the test case passed. If the two are not equal, something, somewhere, is wrong. The tests are constructed from templates and quite a number have been generated; enough to effectively exhaust simple control flow related bugs.

It seems that a good testing environment is the first step towards a good development environment. Testing uncovers deficiencies in the development tools as well as finding language bugs. It is absolutely essential to automate the running of tests during compiler acceptance and testing.

## Co-existence

Although IBM made an early committment to use Ada in SubACS, other languages were included for two reasons: 1) it was unreasonable to link the success of the SubACS project to the success of an Ada compiler implementation, and 2) the SubACS project could not afford to wait; critical-path pieces of software could not wait until Ada became available. It was necessary, then, to choose other languages to use until Ada could replace them. Pascal and CMS-2M, are being used; Pascal was chosen for the 68000, and CMS-2M for the AN/UYK-44. SubACS Ada co-exists with Pascal on the 68000, and with CMS-2M on the AN/UYK-44.

Normally, this co-existence is at the program level, i.e., a single SubACS program (of which there are many) contains only one language: Ada, CMS-2M or Pascal. At run-time, however, a single 68000 (or AN/UYK-44) will likely be running one or more Ada and Pascal (or CMS-2M) programs, simultaneously. Logical communication links between the programs are established at runtime. These inter-program communications primitives perform block transfers between programs and are language independent. The pre-runtime software does not realize that these transfers cross language boundaries.

Co-existence can also occur within a single program. An Ada program can call a Pascal (or CMS-2M) procedure, but a Pascal (or CMS-2M) program cannot call an Ada procedure. An Ada procedure environment is much richer than that of a Pascal (or CMS-2M) procedure environment; supporting that environment was not feasible using the existing Pascal (or CMS-2M) implementations. The interfaces are mainly intended to provide access to machine code and to reduce the overall risk with an immature Ada compiler; interfaced procedure calls are discouraged.

An interface pragma (as defined in the Ada Reference Manual) is used to "co-exist" with Pascal on the 68000. One of two languages are allowed as an argument to the SubACS 68000 interface pragma: LRS_Pascal or 68KOS. The LRS_Pascal interface pragm allows access to procedures written in Language Resources Pascal (or assembly language). For each LRS_Pascal procedure, the compiler emits a symbolic reference into the machine code which must be resolved at a later time. The other interface, the 68KOS interface, is designed to give fast easy access to the underlying non-Ada operating system by generating a TRAP

instruction in-line.

The AN/UYK-44 compiler produces relocatable assembly language as its output; the package bodies that represent the interfaced procedures can be swapped before being assembled, so the interfaced procedure becomes "part of" the Ada program.

The SubACS Ada run-time environment will co-exist with the current Pascal (or CMS-2M) environment by providing a common task priority definition for all tasks, regardless of language. Ada tasks will compete for resources on an equal basis with Pascal (or CMS-2M) tasks and if possible, Ada rendezvous will co-exist with existing multitasking primitives. Unfortunately, the two targets have dissimilar executives; the executive on the 68000 is preemptive and provides a WAIT/POST event mechanism for task synchronization, whereas the AN/UYK-44 executive is cyclic and non-preemptive in nature, consistent with the Navy RNTDS philosophy.

The existence of multiple languages in SubACS has complicated the interfaces, but has reduced the overall risk and shortened the schedule. If SubACS was starting today, it would still be beneficial to introduce different languages to allow an objective measurement of Ada in terms of existing languages.

## Distributed

The SubACS application is distributed over a number of 68000 and AN/UYK-44 processors. The distribution is intended to increase reliability, availability and computing power.

A "single" operating system, called the Network Operating System (NOS), controls the distributed system. NOS assigns jobs (called processes) on a dynamic basis to local operating systems for execution. The local operating systems are largely autonomous; NOS is itself largely implemented as a number of NOS processes. These NOS processes use Process_Create and Process_Delete primitives to start and stop other NOS processes. CPU time limits and strict interprocess memory protection provide fault tolerance.

A single Ada program corresponds exactly to a single NOS process. An Ada program is not distributed. Many Ada programs may be executing on a single processor, but a single Ada program will not run on more than one processor at a time i.e., all tasks contained within a program execute on the same CPU and share memory. Ada programs synchronize and communicate amongst themselves by NOS primitives. NOS primitives provide low-level I/O capabilities, such as OPEN_PORT, CLOSE_PORT and TRANSFER DATA between Ada programs. Calls to NOS to perform I/O may be synchronous. If one task in an Ada program is waiting for a synchronous NOS call to complete, other tasks in the same program may be dispatched by the local operating system.

## Observations and Recommendations

Our overall impression of the Ada language is extremely favorable. The Ada compiler is immature and slow, but Ada is wholly sufficient for the requirements of SubACS, and seems to be integrating well into the complex SubACS environment. It is still early in the life-cycle of SubACS, however.

We are impressed with the software engineering facilities provided by Ada, especially packages. Smart Ada consumers will emphasize the package concept. The package is the basic building block of programs and the basic unit of re-usability. When purchasing Ada software, contractually specify the existence of specific useful, re-useable packages in the deliverable product. With a little foresight, an Ada package store will develop naturally as each new software project adds one or two re-useable packages. Think packages.

## Acknowledgements

We would like to thank the following people for their reviews of this paper: Dick Drake, Lori Govelitz, Paul Kohlbrenner, Paul Popick, and Nancy Vesper.

## References

The "Ada-Europe Guidelines" is available from the National Physical

Laboratory as NPL Report DITC 10/82, ISSN 0262-5369. It was also reprinted in Ada Letters, Vol. III, No. 1 (July/August 1983), pp.37-50. We highly recommend this for anyone involved with Ada acquisition.

Oliver Cole received his B.A.Sc. from the University of Toronto in 1980.

He worked for SofTech, Inc. from graduation until 1983 maintaining real-time high-level language systems for the U.S. Navy. Since October 1983 he has worked for OC Systems, Inc.

Steven North received his B.S. in Computer Science from Yale University in 1981.

He worked for SofTech, Inc. from graduation until 1984 on a number of real-time high-level language systems including the Army Ada Languge System. Since January 1984 he has worked primarily on SubACS Ada.

---

1. Ada is a registered trademark of the United States Department of Defense, Ada Joint Program Office.

# AUTHORS INDEX

# 4th Annual National Conference on Ada Technology

**Sponsored by U.S. Army Center for Tactical Computer Systems (CENTACS)**
**Fort Monmouth, New Jersey**

**19-20 March 1986**

**Hyatt Regency Hotel, Atlanta, GA**

Please provide in the space below a 100-500 word abstract (20 copies) of proposed technical paper on Ada supports such as applications, education, training, research, programming support, methodology, tools/techniques, and other areas of interest to industry, academia, and government. The paper submitted for acceptance should not have been previously published or presented. Abstracts should be submitted no later than September 6, 1985 to the U.S. Army Communication-Electronics Command, Attn: AMSEL-TCS-SA, Fort Monmouth, N.J. 07703.

TITLE_____

AUTHORS_____

COMPANY_____

ADDRESS_____

END

FILMED

3-86

DTIC