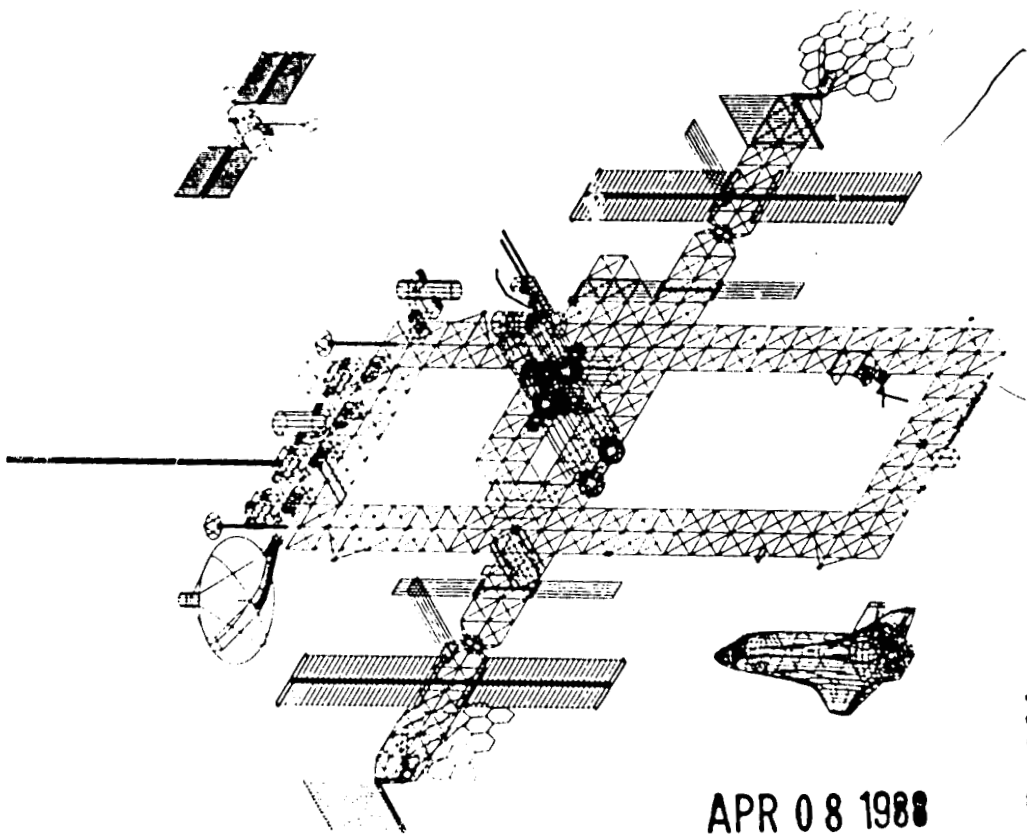
PROCEEDINGS

First International Conference on Ada® Programming Language Applications For The NASA Space Station



June 2 - 5, 1986

Hosted by:

University of Houston-Clear Lake
School of Sciences and Technologies
High Technologies Laboratory

NASA Lyndon B. Johnson Space Center

In Cooperation with Local Contractors

PROCEEDINGS

FIRST INTERNATIONAL CONFERENCE ON ADA* PROGRAMMING LANGUAGE APPLICATIONS FOR THE NASA SPACE STATION

Edited by

Rodney L. Bown
High Technologies Laboratories
University of Houston-Clear Lake

A Conference Cohosted by:

NASA Lyndon B. Johnson Space Center
University of Houston-Clear Lake
School of Sciences and Technologies
High Technologies Laboratories
In Cooperation with Local Contractors.

Houston, Texas
June 2-5, 1986

NASA

National Aeronautics and
Space Administration

Lyndon B. Johnson Space Center
Houston, Texas
77058

University of Houston

Clear Lake

Houston, Texas 77058-1068

WELCOME TO TEXAS

The NASA space station will be the vehicle that will enable man to have a permanent presence in space. The First International Conference on Ada* Programming Language Applications for the NASA Space Station has provided an opportunity for the government, industry, and university to engage in a lively technical discussion related to the global network of information system resources for this new vehicle and its new world environment.

The Lyndon B. Johnson Space Center, the University of Houston-Clear Lake, and supporting contractors welcome all out-of-town attendees to Clear Lake during the Texas sesquicentennial year. The conference committee is committed to providing a quality technical conference and a friendly Texas experience for all attendees and their guests. The conference has been organized to provide multiple technical sessions/panels/activities on a variety of issues. In addition, the local arrangements committee will assist all attendees to plan and schedule non-conference activities that will provide an opportunity for everyone to enjoy the Houston/Bay Area/Galveston attractions.

We believe the contents of this volume will provide a valuable technical resource for future research and development efforts directed to the support of NASA space activities. We are proud of the organizations and their people who have contributed to the success of the conference.

Jack Garman, Deputy Director
Mission Support

Edward T. Dickerson, Dean
Sciences and Technologies

Ed Chevers, Deputy Chief
Avionics Systems Division

NASA Lyndon B. Johnson
Space Center

Charles W. McKay, Director
High Technologies Laboratories

University of Houston-
Clear Lake

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

ACKNOWLEDGEMENTS

Steering Committee

Jack Garman	Lyndon B. Johnson Space Center
Ed Chevers	Lyndon B. Johnson Space Center
Edward T. Dickerson	University of Houston-Clear Lake
Charles W. McKay	University of Houston-Clear Lake

Conference Chair

Rodney L. Bown	University of Houston-Clear Lake

Executive Coordinators

Bob MacDonald	Lyndon B. Johnson Space Center
Carol Kasworm	University of Houston-Clear Lake

Publicity

Becky Schergens	University of Houston-Clear Lake

Technical Committee

Richard Kessinger	Softech
Kathy Rogers	Rockwell International and Vice-Chairperson of the Clear Lake Chapter of SIGAda

Local Arrangements

Al Mandelin	IBM, Federal Systems Division
Ed Monteiro	McDonnell-Douglas

NASA Exhibits

Roger Bilstein	University of Houston-Clear Lake

International Host

Steve Brody	Lyndon B. Johnson Space Center

Members of the Support Team

Steve Gorman	Lyndon B. Johnson Space Center
Ken Goodwin	Charles Stark Draper Labs.
Pat Rogers	University of Houston-Clear Lake
Cathie Duffy	University of Houston-Clear Lake
Sue LeGrand	Softech
Paul Brown	Hikkok
Gary Barber	Intermetrics
Mark Denson	McDonnell-Douglas and Chairman of the Clear Lake Chapter of SIGAda
Richard Lehman	Lockheed
Charlie Randall	GHG Corp.

Special Acknowlegement

The entire organization extends a special message of gratitude to Mss. Vickie Gilliland, Mary Ann Pollard, Karen Gunter, Sheri Lindelsee, and Janice Fisher for their friendly office support.

TABLE OF CONTENTS

Note: Page numbers are listed in the following sequence: session letter. (session number.) paper number. page number.

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

SESSION A — **OPENING PLENARY SESSION**
Nassau Bay Hilton Hotel
Monday Morning 9:00 to 12:00

Welcome to the Conference
Jack Garman, Director
Mission Support
NASA Lyndon B. Johnson Space Center

Welcome to the Johnson Space Center
Jess Moore, Director
NASA Lyndon B. Johnson Space Center

Welcome to the University of Houston-Clear Lake
Dr. E. T. Dickerson, Dean
Sciences and Technologies

Welcome by the NASA Space Station Program Office

Ada Joint Program Office
Virginia Castor, Director

Space Station Computing

Ada International
Commission of European Countries
Rudy Meijer is being represented by
Pierre Vielcanet
Informatique International
Toulouse, France

NASA/Johnson Space Center International Office
William Rice

MONDAY EVENING SESSION
University of Houston-Clear Lake Bayou Building

Reception
Keynote Speech - Software Engineering
Dr. John Manley, Director
Software Engineering Institute
Carnegie Mellon University
Pittsburg, PA

DEBUGGING TASKED ADA PROGRAMS

by

R.G. Fainter
Virginia Tech,
and
T.E. Lindquist
Arizona State University

Abstract

The applications for which Ada was developed require distributed implementations of the language and extensive use of tasking facilities. Debugging and testing technology as it applies to parallel features of languages currently falls short of needs. Thus, the development of embedded systems using Ada poses special challenges to the software engineer. Techniques for distributing Ada programs, support for simulating distributed target machines, testing facilities for tasked programs, and debugging support applicable to simulated and to real targets all need to be addressed. This paper presents a technique for debugging Ada programs that use tasking and it describes a debugger, called AdaTAD, to support the technique. The debugging technique is presented together with the user interface to AdaTAD. The component of AdaTAD that monitors and controls communication among tasks has been designed in Ada and is presented through an example with a simple tasked program.

1. INTRODUCTION

Because of the distributed nature of the Space Station and its unmanned platforms, software that the Space Station uses must be highly distributed. This implies, therefore, that the task will be used extensively in Space Station software. Because of the difficulties associated with locating errors in tasked programs and because of the cost of programming errors in Space Station software, tools to aid in the production of correct programs must be developed. Such a tool is currently under development and is described in this paper.

One view of program testing [1] indicates that a program has been tested when every statement in the program has been executed at least once and every possible outcome of each program predicate has occurred at least once. Considerable literature addressing techniques for testing software reflects a view of testing that is consistent with this definition. Although this definition does not naturally extent to tasked programs, it is indicative of the view that testing occurs late in software development and is oriented toward validation.

In contrast, debuggers have traditionally had utility in earlier software development activities. Accordingly, debuggers are used as automated support for locating errors and determining what is needed to correct errors. Ideally, testing is used to identify the presence of errors and debuggers to support location and correction. When tasking facilities are included in a language, however, the software designer is left without good testing techniques, and debugging must enter into the process of identifying the existence of errors.

Helmbold [2] suggests that "Debuggers for parallel programs have to be more than passive information gatherers--they should automatically detect errors". When tasking errors directly depend on the semantics of the language, a debugger is able to actively aid in detecting errors. More commonly, errors are also dependent on the specific logic of task interaction and the use of the language.To take an active role in identifying this more complex type of errors, the debugger must include facilities to analyze the logic of the program. Helmbold distinguishes types of tasking errors as "Task Sequencing Errors" and "Deadness". AdaTAD provides task information that may be used to detect either type of tasking errors, although it does not actively detect errors.

AdaTAD is a debugger whose capabilities are specific to the problems of concurrent programs. The name AdaTAD is an acronym for Ada Task Debugger. Most debuggers allow the user to trace the execution of a program, but the program remains under control of the operating system. AdaTAD differs from other debuggers by exercising direct control over the execution of a

program's tasks. The user is able to specify which tasks run when, at what rate and for how long. Of course, to emulate more closely the environment in which a program is to execute, the user may defer these decisions to the runtime system, and simply monitor task synchronization and communication. AdaTAD combines typical debugging facilities with others specific to supporting the Ada constructs for rendezvous.

Space Station software may be configured in many different ways. One possible scenario might involve an Ada program with tasks running on an Earth-based computer, on one or more computers aboard the main station and on computers on one or more unmanned platforms. AdaTAD has the capability to allow the software engineer to debug such a program in at least two different ways. Firstly, the software engineer may construct, solely on ground based computers, an environment similar to that which exists on the Space Station for debugging purposes. Secondly, because AdaTAD itself may be distributed, the program may be run under AdaTAD in the actual Space Station environment. This allows the software engineer a great deal of flexibility in exercising the program under a variety of conditions.

A method for debugging tasked Ada programs and AdaTAD are presented jointly in this paper. Our approach to task debugging centers on removing task errors from three successive levels of consideration. Errors within tasks, which are principally independent of other tasks, are first addressed. Next, the communication and synchronization structure among tasks is addressed, and finally, any application specific concerns are addressed. AdaTAD, as it relates to these levels, is discussed in the following three sections together with a discussion of our approach to debugging. A subsequent section addresses the design of AdaTAD. Ada is used in the design to allow increased effectiveness on multiprocessor applications, and to show how the rendezvous constructs can be used to control the execution of tasked Ada programs. An Ada implementation of AdaTAD would require emitting special code from the compiler for synchronization with AdaTAD.

2. LOGIC ERRORS WITHIN A TASK

The first level of usage for the debugger is to address logic errors within each of a program's tasks. These errors are exclusive of intertask communication and synchronization. Removing them is synonomous to removing errors detected during unit testing of software. At this level, we assume that interactions with other tasks are correct and examine the activities of the task itself. Testing and debugging at this stage considers a piece of software in absence of all elements of its environment except any procedures or functions it calls. For example, a task may use information

obtained from other tasks to retrieve and update information in a database. Task logic to perform operations on the database is considered, at this level, exclusive of synchronization with other tasks.

AdaTAD facilities are used in conjunction with a testing strategy in which some form of code analysis may be performed. AdaTAD is designed to aid in executing test cases and in removing any errors subsequently found.

2.1 User's View of AdaTAD

AdaTAD provides many facilities which are common to source level debuggers in addition to those specific to tasks. After introducing the manner in which AdaTAD includes common functions, facilities specific to removing logic errors from tasks are presented.

Command Entry

In accordance with the findings of Wixon [3], AdaTAD is designed to use command driven user input instead of either a menu or iconic input. Commands exist to control the initiation, configuration, and completion of an AdaTAD session as well as to control the execution of the task being debugged. Arguments to commands are entered as parameters to the command line itself. Each task has a keyboard assigned to it for interactive input. When a task is the current task its keyboard is the physical terminal to which the task has been assigned.

Information Display

Since so much information is made available to the user of AdaTAD, a well engineered display is critical. We have designed an interface that combines textual and graphical status information in a windowing framework. The concept of windowing has recently received much attention. Windows allow a process to assume that it has a dedicated output device, independent of whether the window is being viewed. Assignment of screen geography can vary dynamically under user control to allow variable presentation of information.

The AdaTAD display consists of a set of task windows and a task interaction status display. The user may configure windows on the screen by using the **WINDOW DEFINITION** command. Figure 1 shows a task window and the panes that are included (the task interaction status display is presented in the next section). The panes display information about the current

execution state of the task, information on designated variables, the source code context and task output.

AdaTAD control commands manage the appearance of the debugger to the user and perform basic initiation and termination of users programs. The commands include:

EXECUTE	--initiate program and enable execution
DEFINE_WINDOW	--specify size and location of a window
ZOOM	--alter the size of a window
CURRENT_TASK	--task to which taskless commands apply
ASSIGN	--associate i/o device with a task
TERMINATE	--complete the interactive session

Although these commands are not specific to a particular task, they are needed in tailoring a specific debugging session for logic errors.

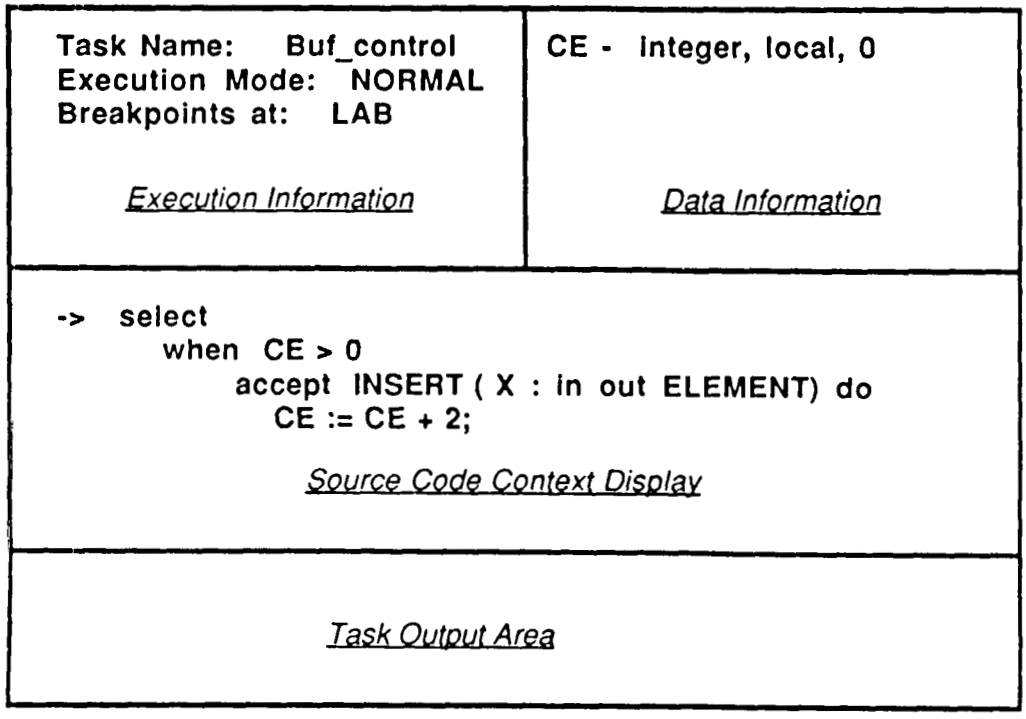


Figure 1. Task Window Format.

Task Execution Information and Control

Two breakpoint facilites are provided for controlling the execution of statements within a task. Assertion breakpoints may be placed within the a task by adding an **ASSERT** statement to the program, and unconditional breakpoints may be associated with any statement of a task. Since several allocated tasks may have the same task body, breakpoints cause breaks to occur in all tasks having the body.

Four modes of task execution are provided to accommodate various debugging techniques.

NORMAL	--execute until encountering break
SINGLE STEP	--user initiated statement execution
TIMED	--execute statements at a given rate
WAIT	--suspend task execution

When a task halts execution at a true or unconditional breakpoint, the task is placed in a **WAIT** mode of execution. Execution is resumed by explicitly placing the task in another execution mode (**NORMAL, SINGLE_STEP,** or **TIMED**).

Examination of Data.

AdaTAD provides facilities for viewing or altering the values of program objects by the object's source code name. If tasks communicate via shared variables, then AdaTAD aids in detecting any attempt to violate the assumptions described in section 9.11 of the Ada Language Reference Manual [4].

2.2 Using AdaTAD to Remove Logic Errors

Testing and debugging the logic errors within tasks can best be done by removing the influence exerted by the task's environment. The environment must be specified by the test case and controlled by the debugger. All interactions with other tasks, such as entry calls to the tested task, accepts of calls made by the tested task, or the use of shared variables are controlled during testing and debugging by AdaTAD stub facilities.

The test cases for this phase can be characterized as including input, environment and expected results. When the task is initiated in a state satisfying the input condition and executed in the environment specified then it should exhibit the expected results. The input condition describes

the values of inputs to the task. These may include the initial state of a database used by the task or of objects obtained through input.

The environment specification must describe the necessary interactions with other tasks to carry out a test case. When selective waits or conditional or timed entry calls are contained in the task, the specification indicates specific paths through the constructs relevant to the test. For example, a test case that is to examine a specific delay alternative must specify in its environment section conditions causing that delay to be executed. Further, to obtain the information needed for a test case it may be necessary to specify which task is to call a specific entry to the tested task.

The anticipated results of executing a test case may not be as simply expressed as an output condition to be true when execution completes. Tasks may execute indefinitely, may terminate in synchronization with others, or may transmit their results to other tasks through entry parameters. Accordingly, the anticipated result may be a condition to be true at a specific point during execution of the task (possibly within an iteration).

AdaTAD Support

AdaTAD facilities are used to execute test cases, and debugging can be done in conjunction with testing if needed. Facilities supporting the execution of test cases can be compared to those of other debuggers for handling procedure stubs. In AdaTAD, these facilites include commands to:

1. Cause a terminate condition to evaluate true,
2. Provide a dummy entry call to a task with specific arguments,
3. Cause an entry call to another task to be accepted and out parameters from that call to be set,
4. Deterministically select an alternative in a nondeterministic selective wait,
5. Selectively satisfy durations on delay statements.

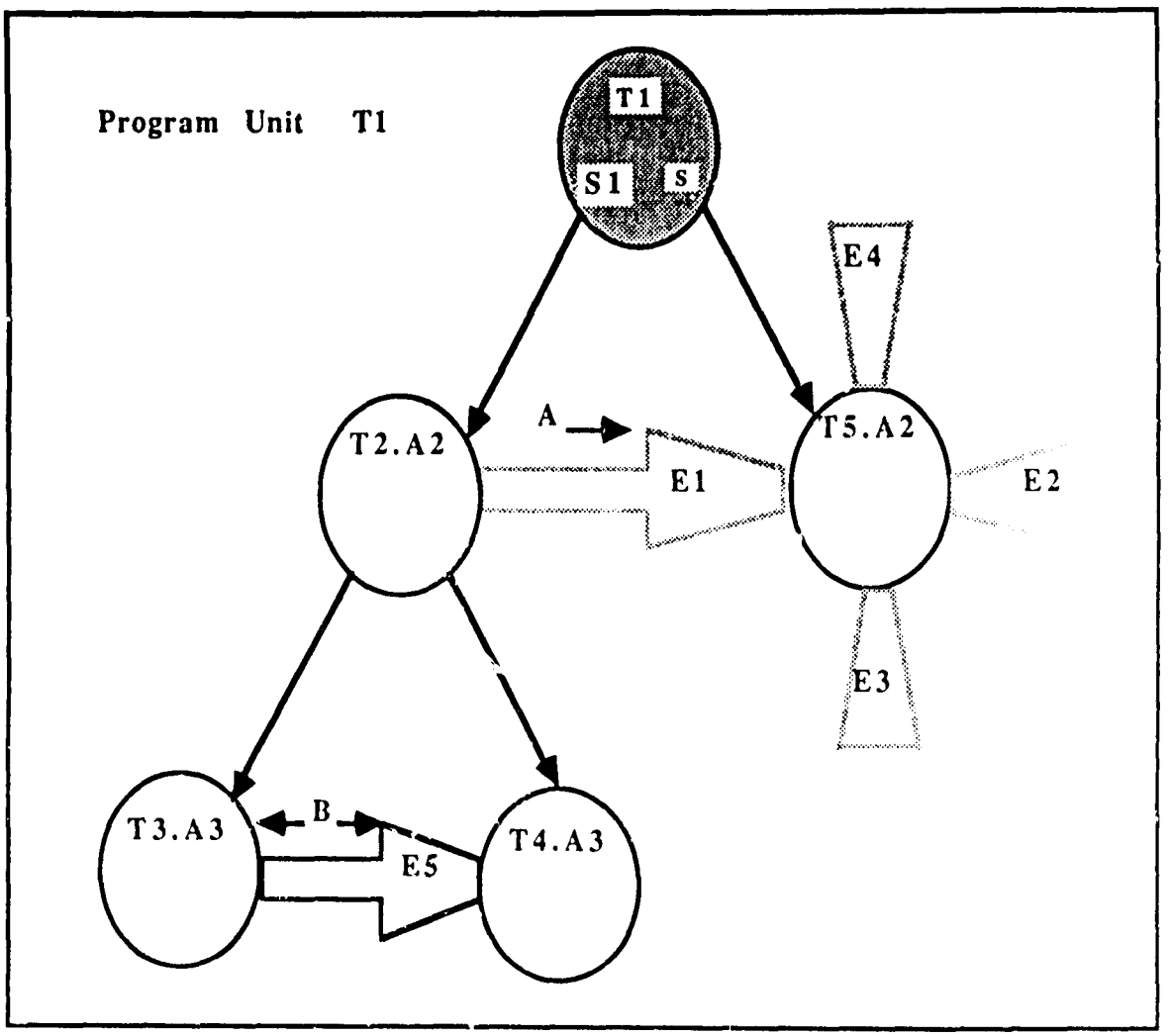
3. SYNCHRONIZATION AMONG TASKS

After checking the logic within a task, the communication and synchronization among tasks is considered. This step is analogous to integration testing in that the cooperation among possibly several tasks is addressed. Data flow and control flow through tasks of the program are observed at this level of testing and debugging. From the perspective of a single task, this level checks, in a rudimentary manner, the task's tasking environment. Subtle timing interactions and interactions with the operating environment are left to the final level of checking.

The scenario for testing and debugging follows the same approach as with task logic. Test cases are identified using source code analysis. Test cases are run using AdaTAD support, and errors are located and removed using AdaTAD debugging facilities. Test cases focus on task interaction. Input conditions and expected results are included, but no specific information describing task execution constraints is included.

3.1 Task Interaction Status

AdaTAD's Task Interaction Status window depicts the state of rendezvous and consequently is particularly useful for synchronization testing. Within the window a graph is used to represent tasks of the program and relationships between tasks. Each task has a corresponding node in the graph, and relationships such as "**depends on**" and "**is in rendezvous with**" are depicted by directed edges from one task to another. Figure 2 shows a hypothetical program unit, called **T1**, at some point of execution, and Figure 3 is the legend for the task status area. **T1** has four subordinate tasks, **T2, T3, T4** and **T5**. Each of these subordinates has an underlying task type: **A2** for **T2, T5** and **A3** for **T3, T4**. Arcs with solid arrow heads indicate the dependent relation among tasks. Thus in this example, **T1** has caused initiation of **T2** and **T5**. Rendezvous and communication status is conveyed through double-line arcs. The arc from **T2** to **T5**, with shaded lines, indicates that **T2** is waiting at an unaccepted entry call to **T5**'s entry **E1**. **E1** has a single input (IN) parameter, and for this call **A** is the argument.



Task Interaction Status Window

The large shaded arrowheads (without bodies) pointing at **T5** indicate the task will not be immediately accepting the call to **E1. T5** is waiting at a selective-wait with three open accepts (**E2, E3, E4**). The large solid arc from **T3** to **T4** indicates that these two tasks are currently in rendezvous. **T3** is the calling task and **T4** is the accepting task, as indicated by the arrow-head. For entry **E5** the argument is B, which is an IN OUT parameter.

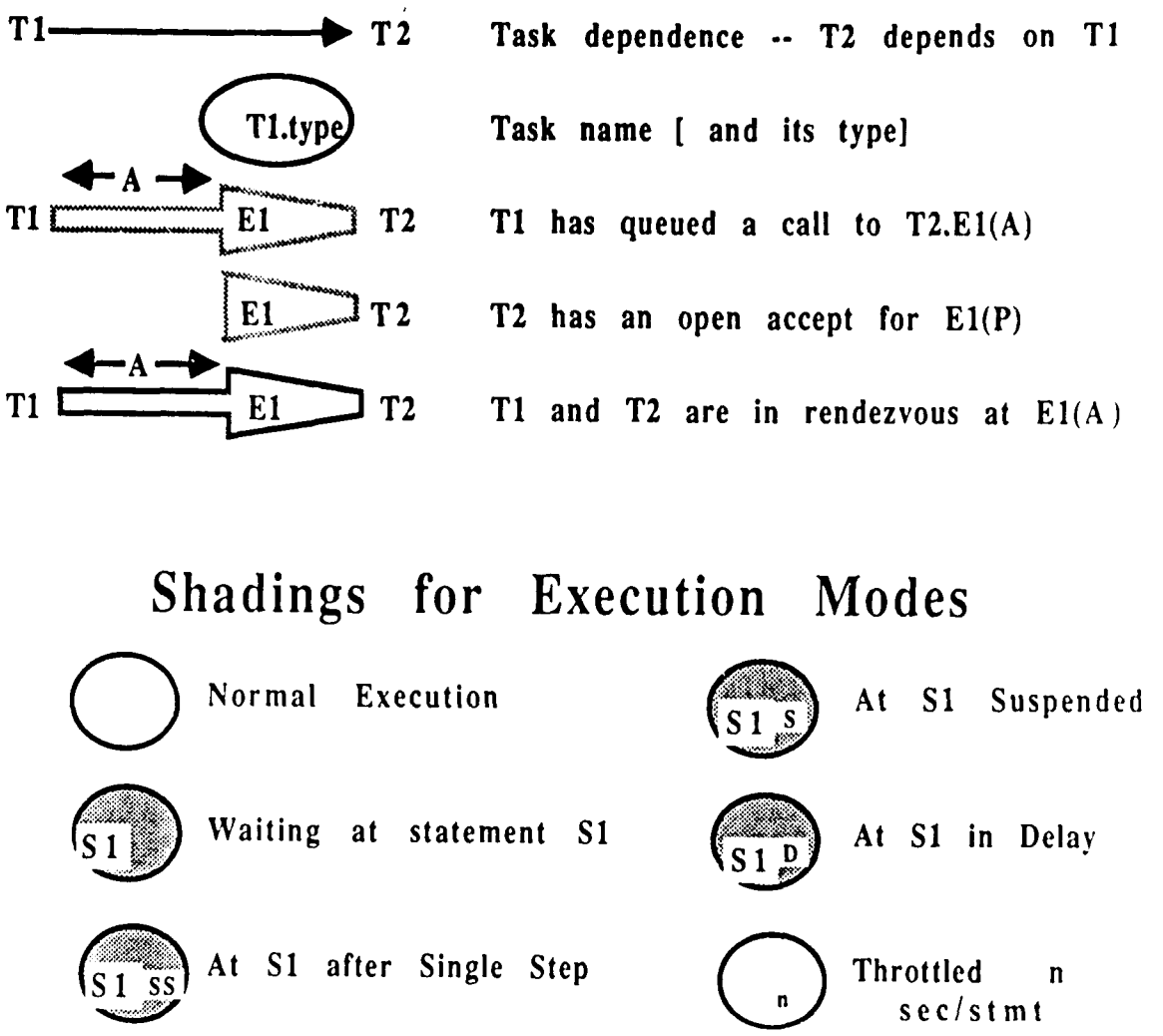


Figure 3. Legend for the task interaction status display

The main program unit, **T1**, is currently in a **WAIT** state of execution, as indicated by the shaded task node for **T1**. The small s in the lower right of the task node indicates the task is

waiting because it was suspended.

3.2 Execution Control For Checking Interactions

The displays generated by AdaTAD for checking task interactions are the same as those for logic checking within a task, but the capabilities available to the user differ. When checking task interactions, AdaTAD does not allow the user to:

1. Provide a dummy entry call to another task, or
2. Provide a dummy accept of an outstanding entry call.

Additional facilites are provided to specifically aid in debugging task interactions. These include:

1. Break at rendezvous beginning/completion,
2. Examine the calling queue for an entry,
3. Reorder the calling queue for an entry
4. Examine/alter arguments to an entry call.

Rendezvous breakpoints provide a means for control to return to the user at the boundaries of a rendezvous. When both tasks reach the synchronization point, the user may need to examine assertions, arguments, or results to determine correct communication between tasks. Rendezvous breakpoints may be associated with either pairs of tasks or with entries within a task. In one situation, the user may be interested in examining communication between tasks **T1** and **T2** each time they rendezvous, indepenf of the entry at which rendezvous occurs. In another situation, a user may need to know parameter information each time that a specific entry within a task in called, independent of what task is calling.

4. APPLICATION SPECIFIC USES OF ADATAD

The final stage of debugging considers the operating environment in which the tasks must execute. For an embedded system, this may include operating within a set of heterogeneous processors, each with different resources and capabilities. Testing and debugging at this level is

often accomplished with a simulation of the operating environment. While specific tools are necessary to support this activity, AdaTAD provides facilities that are useful in a general manner to the problem of addressing the operating environment.

The problems that may arise in this phase of testing include timing inconsistencies among tasks, space requirements of a task, or resource contention caused by task interaction. Device interactions for special purpose input or output may be one cause. Another cause may be constraints imposed on the program by task distribution or the interaction between the task scheduling strategy and the operating environment.

AdaTAD provides facilities that allow the user to monitor program elements that will reveal these environment related problems. Ultimately, we recognize that the program under observation may to some extent be perturbed by the debugger. Nonetheless, a certain amount of debugging can be useful in this phase. To a large degree, testing technology is not appropriate for revealing application specific errors. This is an area in which ad hoc stress testing has been most successfully applied.

The capabilities that support this aspect of testing include:

1. Call Queue Display,
2. Entry Call Frequency,
3. Accept Entry Frequency,
5. Statement Execution Frequency,
6. Object Update Frequency.

The user can request that certain entry call queues be displayed automatically when modified. This provides a monitoring ability for a service rendezvous that is used by several tasks. The frequency displays allow the user to selectively obtain information that will show the contention points in a program. Entry call frequency may be obtained in two forms, entry call by any task and entry call by a named task. Statement and Object frequency information is useful in determining the dynamic space requirements of a task. One can observe executions of allocator statements or updates to objects detailing the size of dynamic structures. Although these facilities do not directly support monitoring interactions with the external environment, often internal objects or statements reflect their status.

5. THE DESIGN OF ADATAD

As with any debugger, AdaTAD requires specific modifications to the compiler and linker. To allow the debugger itself to be designed and implemented in Ada, source code changes are made to provide synchronization through AdaTAD entries. AdaTAD is, itself, a set of Ada tasks. There

are four major cooperating tasks including:

1. AdaTAD Coordinator,
2. Data Base Monitor,
3. Command Processor, and
4. Terminal Communicator.

There are also two arrays of tasks, including:

1. Logical Processor Tasks and
2. Terminal Drivers.

Additionally, there is a task to handle input/output between the user's program and non-terminal input/output devices. Figure 4 is a diagram of the overall structure of AdaTAD.

AdaTAD tasks communicate via the rendezvous and a shared variable. The data base stores execution information about the user's tasks. AdaTAD effectively makes each user task part of a logical processor task, which controls its exeuction. The terminal communicator is responsible for receiving user commands and updating task displays. The data base monitor provides operations that both synchronize access to the data and perform data storage and retrieval functions. The coordinator mediates communication among logical processors whose user tasks are synchronizing. The coordinator is also responsible for directing parsed user commands to the appropriate logical processor.

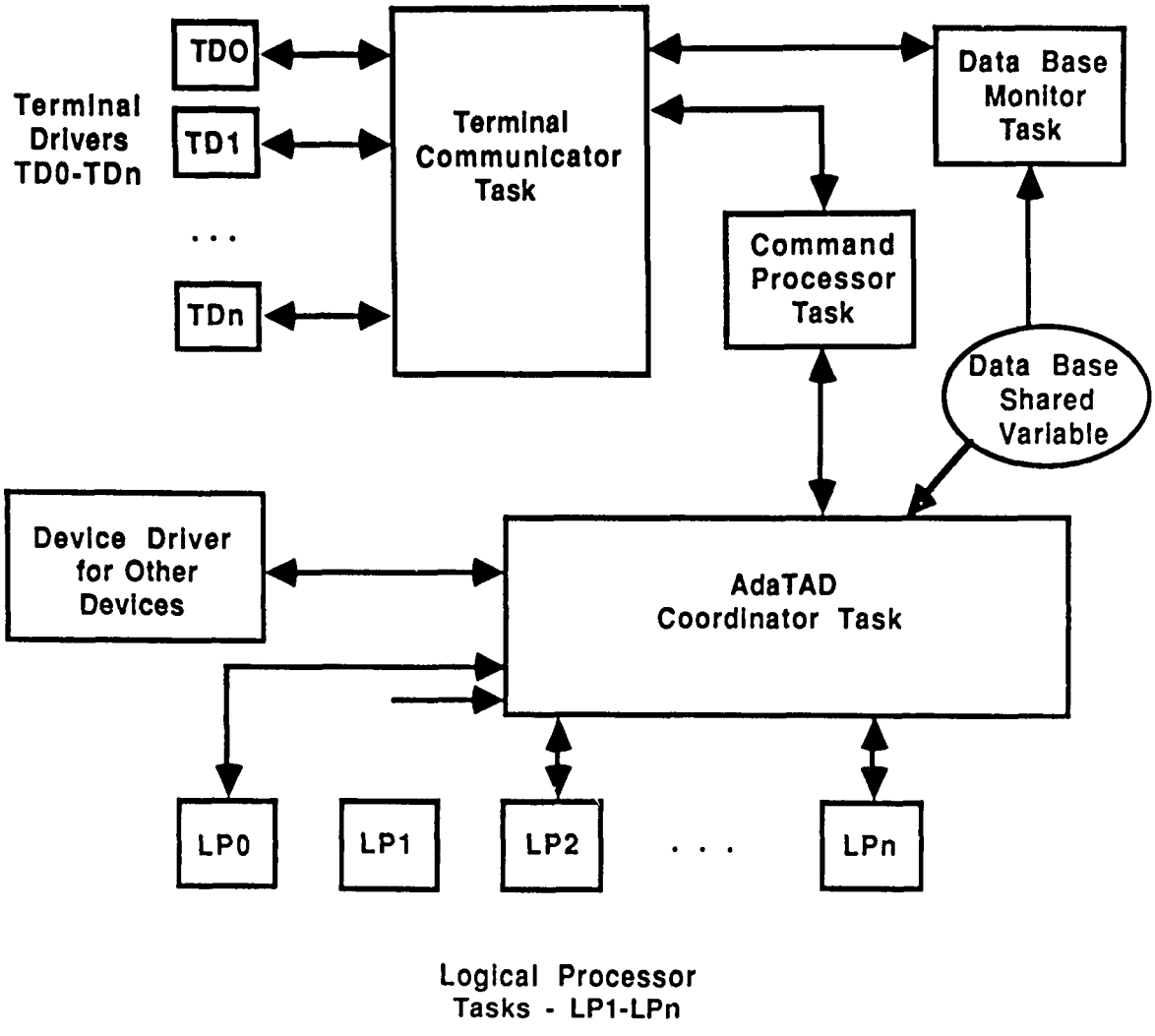


Figure 4. Task structure of AdaTAD.

5.1 Design of the Logical Processors

Logical processors are the most complex tasks in AdaTAD, because they monitor and control the synchronization among user tasks. Synchronization with other AdaTAD tasks is used to communicate the current state of execution to the data base maintained by the Coordinator. Logical processors have four entries for receiving input from the command interpreter, for servicing rendezvous requests from user tasks, for notifying rendezvous completion from servicing tasks, and for notifying task termination from other logical processors. Three tasks are defined within

each Logical Processor. The EXECUTOR task directly controls environment for the user task, the TRANSMITTER task serves as a funnel for messages to the coordinator, and the EXECUTION_AREA_MONITOR maintains the variables which reflect the current execution state of the user task. Although the presence of three tasks complicates the Logical Processor, it allows for maximal parallelism in the execution of the Logical Processor, and it minimizes the time spent by the user task in synchronization with AdaTAD.

Receive_User_Command

Through this entry, the logical processor is called by the coordinator when a user command is to be executed by the logical processor. A case statement within this entry selects the proper code to implement the command. With only two exceptions, the implementation of the commands at this level involve setting values in the execution data base. For example, if the user wants to change the execution state of a task, the command is channeled to the appropriate logical processor and the execution state variable is changed.

Receive_Rendezvous_Completion.

When a rendezvous between two user tasks completes, the calling task must be released for further execution. To do this, the AdaTAD coordinator calls Receive_Rendezvous_Request. The call indicates that a rendezvous requested by the task running on the logical processor has been completed. The entry updates the local data base so that the user task can continue execution. Any arguments which were changed by the rendezvous exist in the argument list and are copied to the appropriate area.

The Executor Task

This task directly controls execution of the user's task. The compilation system modifies the user's task to physically nest it within the Executor. The Executor has one entry which is called when another user task has issurd an entry call to this task. The call is forwarded to the Executor by the logical processor's Receive_Rendezvous_Request entry when the coordinator sends an entry call. The compilation system converts rendezvous code into procedures that may be called to perform the rendezvous code. Thus, when the user task is ready to accept the call, the appropriate procedure is called.

Transmitter Task

The Transmitter sends messages to the AdaTAD coordinator. It is called by the user's task to request an input/output service or to inform the coordinator that a rendezvous has begun or completed. Transmitter is called by the execution area monitor to send the current state of the data base to the coordinator.

Execution Area Monitor Task

Since the execution data base is a shared variable that must be accessed by the Executor, the Transmitter and the Logical Processor itself, synchronization to the information is provided by the Execution_Area_Monitor. Tasks requiring information from the data base get the information by making an entry call to the monitor. The task services the following entries.

Sing_step_rel: Called by the logical processor after the coordinator has signaled that the user has pressed a key to cause execution of the next statement in single step mode. The entry enables execution of the next statement.

Set_bk_state: Called by the Logical Processor to enable or disable breakpoint checking.

Set_ex_md: Called by the Logical Processor whenever the execution mode is to be changed.

Set_ex_rt: Called by the Logical Processor to set the rate for timed execution.

Set_timed: Called by the Logical Processor to enter timed execution mode.

Examine_exe: Called by the statement prologue to see whether statement execution is enabled.

When there are no outstanding entry calls to the monitor, the current execution mode is determined and the appropriate action is taken. If the execution mode is TIMED, the monitor determines whether it is time to execute the next statement.

5.2 The Coordinator Task

The AdaTAD coordinator mediates communication among AdaTAD tasks. When user tasks rendezvous, the coordinator handles communication among their Logical Processors. This mediation occurs when a rendezvous is requested, when a rendezvous completes and when a rendezvous begins. The Coordinator also mediates input/output requests for user tasks. To allow all appropriate information regarding the execution status of tasks, all communication with the underlying operating system must be recorded. This is done when a user's task requests service and when control returns from the operating system facility. Two further functions of the

Coordinator are to dispatch AdaTAD user commands to the appropriate Logical Processor and to collect status information for data base modifications. The coordinator interactively accepts entry calls to its entries in the order in which they arrive. We now describe the Coordinator in terms of its entries.

Rendezvous_request.

When one user task requests a rendezvous with another, the requesting task's Logical Processor makes a call to this entry of the Coordinator to initiate the rendezvous. The coordinator, in executing the call, looks up the Logical Processor for the called task. The name of the called task is taken from a descriptor list which also includes parameters for the call. Before making an entry call to the Logical Processor of the called task, Coordinator sets an indicator to show that the calling task is awaiting synchronization.

Rendezvous_begin.

When a rendezvous begins, the called task calls this entry with the names of the two synchronized tasks. The entry updates the synchronization information for the two tasks. It clears the **waiting** indicator, sets the **is_synchronized** indicator and records the names of the called and calling tasks in the synchronization data base.

Rendezvous_completion.

When the called task completes its rendezvous code, its logical procesor calls this entry. This occurs when the servicing task either terminates or encounters the end of the synchronized code of an accept statement. The entry updates the synchronization data base to reflect the rendezvous has completed. Further, an entry call is made to the logical processor running the served task so it may continue execution. The single parameter for this entry is the name of the task which has been served.

Data_base_update.

Each logical processor has local data that controls the execution of the user's task. When that data changes, the central data base is periodically informed through calls to this entry by Logical

Processors. Parameters convey the task name and its execution state. A local procedure, which the entry uses to perform the update, blocks the data base monitor task from looking at the data base while doing the update.

5.3 Data Base Monitor Task

The data base monitor is used to implement exclusive modification of the data base and to drive terminal updates of task status. An AdaTAD task acquires exclusive access to the data base through the Monitor's Hold and Release entries (P and V). For example, Hold is called by the Coordinator prior to making data base modifications required by a user command. After completing the modifications, Release is called.

The current state of the data base is transmitted to the Terminal Communicator task for display when no other task is modifying the data base. This is accomplished with an else clause on the selective wait for the Monitor's Hold entry. If no AdaTAD task has queued a call to Hold when the selective wait is encountered, then the else clause is executed and information is sent to the Terminal Communicator.

5.4 The Command Processor Task

The Command Processor analyzes the user commands. When a command is successfully parsed, it is dispatched, along with its parameters, to the AdaTAD Coordinator for execution. Even commands which affect information display are executed by the Coordinator. If a command is erroneous, nothing is sent to the Coordinator, and an error message is sent back to the Terminal Communicator. The internal procedure Analyze_Command does the lexical and syntactic analysis of the command.

Parse is the only entry into the Command Processor. Parse is called by the Terminal Communicator when unsolicited input occurs on a terminal.

5.5 The Terminal Communicator Task

A task's terminal input and output is controlled by the logical processor, through the mediation of the Terminal Communicator. The Terminal Communicator also provides the intelligence for display of the AdaTAD data base. The Terminal Communicator manages the windowing capability of AdaTAD. The five entries in this task receive information from the

Coordinator, the terminals, the Data Base Monitor and the Command Processor.

From_Terminal and From_Coordinator

The Terminal Communicator task has two accept statements for the From_Terminal entry. The first handles unsolicited input from a terminal. Assuming that unsolicited input is a command, the first accept receives an information string and passes that string along to the Command Processor. For example, when the user enters the string **"set wait"**, the Terminal Communicator assumes that this is a command and sends it to the Command Processor.

The From_Coordinator entry is called by the Coordinator when a user task requires input or output. We call this solicited input or output. The second accept for the From_Terminal entry is used for input of solicited information. From_Terminal is accepted after accepting the From_Coordinator entry. These entries are called when a user task has requested terminal input.

From_Command_Processor

This entry is called by the Command Processor when it has detected an error in a user command. This entry displays the error message on the terminal from which the command was entered.

5.6 The Terminal Drivers

The Terminal Drivers are an array of tasks that handle the transmission of data between the physical terminals and the Terminal Communicator. The Terminal Driver has four entries and one internal task which has no entries.

The Output entry is called by the Terminal Communicator to write a string on a terminal. It then calls the Output entry in the Terminal Driver. Output accepts the string and writes it on the device through the appropriate Terminal Driver. The Input entry passes the string and the Terminal Driver number to the Terminal Communicator.

Terminal_watcher

Internal to the Terminal Driver is a task whose sole job is to wait for an input string from the terminal. When a string is received, as indicated by a terminal character, the task makes an entry

call to the terminal driver's input entry, passing the string. The identify entry is called by the Terminal Communicator as soon as the driver begins execution, to assign the driver a number, which is used in all communication.

5.7 An Example of Synchronization Among User Tasks

Controlling the synchronization of user tasks is the most complex of actions that AdaTAD performs. AdaTAD must intervene when a rendezvous request is made, when the rendezvous begins, and again when the rendezvous ends. To keep track of these interactions, the compiler converts user entry calls to calls of AdaTAD task entries. The compiler also generates code to inform AdaTAD when a rendezvous actually starts and when it completes. In this manner, AdaTAD can record the status of all user task synchronization. These actions occur whenever a rendezvous request is made, but they are normally transparent to the user. The following paragraphs describe what occurs in each case of AdaTAD intervention.

As an example of how AdaTAD controls execution, assume that two tasks (A and B) are running. Assume that task A wants to make an entry call to task B's entry named E1. Since the example is concerned with synchronization only, we assume that no data are passed during the rendezvous. Assume further, task A is running on logical processor one and task B is running on logical processor two.

Rendezvous Request

Task A has an entry call statement of the form **B.E1**. For this call, the compiler generates code to produce an empty argument list (alist), which consists only of the head node. This node names the calling task, the called task and the called entry. The compiler converts the statement B.E1 into:

```
TRANSMITTER.SEND_RENDEZVOUS_REQUEST(alist);
```

The first action that takes place at execution time when task A is ready to make this rendezvous is that the transmitter is invoked. The transmitter's send_rendezvous_request entry accepts the call and immediately sets task A's execution mode to **wait**. Then, the transmitter makes an entry call to the coordinator, passing the argument list along unchanged.

The request for rendezvous arrives at the coordinator's Rendezvous_Request entry. The

coordinator looks in the argument list, to get the name of the called task, in this example, B, and gets the number of the logical processor that is running the called task. The coordinator then looks up the called entry name in the task data base. In this example, the entry is E1. The coordinator uses the number to index the array of tasks which implement the logical processors. Next, the coordinator makes an entry call to the Receive_Rendezvous_Request entry of the appropriate logical processor. At this point, the synchronization information on the calling task, A, will be updated to reflect that it is waiting for a rendezvous, and AdaTAD knows that a rendezvous request has been made and that the calling task is in a wait state for that rendezvous. Further, the user notices on the display that the calling task has entered a wait state awaiting a rendezvous. The display also indicates the task being called, the state of the calling task and any other tasks awaiting rendezvous.

When the Logical Processor accepts the rendezvous request, it passes the argument list to the executor running task B. The Executor receives the request at its Rendezvous entry and extracts the name of the called task and entry from the argument list. The name of the calling task (A) is used later to tell the coordinator that the rendezvous is in progress. The name of the entry allows the executor to request the proper entry into the user's task. If appropriate, the Executor calls the procedure written by the compiler for the receiving task. This procedure decodes the argument list and executes the entry call into the user's task. Assuming that the called task, B, is waiting at the entry being called, the Executor's entry call is answered immediately and the user's task begins execution.

Accepting a Rendezvous Request

The first thing that the user task's accept statement for E1 does is make an entry call to the transmitter with the name of the calling task. This entry call is to the Send_Rendezvous_Beginning entry. The entry sets the called task's execution data base to reflect that the called task is now running, and then the coordinator is informed that the rendezvous is beginning. The coordinator acts on this information by updating its synchronization information data base. The user would now see that the rendezvous is in process in the display area. After the user's task indicates that the rendezvous has been accepted, AdaTAD does not intervene. A user observing the synchronized behavior of the tasks would see that they obey the rules of synchronization prescribed by Ada.

When the rendezvous between A and B is complete, the servicing task, B, encounters a call to the Transmitter's entry Send_Rendezvous_Completion. The servicing task remains in a running

state until it reaches a point where it must wait for another rendezvous. The Transmitter sends a message to the coordinator that the rendezvous is complete. As far as the servicing task's logical processor is concerned, the rendezvous is now over. However, there is still work for the coordinator to do. Upon receiving notification of the termination of the rendezvous, the coordinator updates its synchronization data base to reflect the end of the rendezvous. As far as the coordinator is concerned, the rendezvous is now over, as indicated by calling Receive_Rendezvous_Completion in the logical processor running the calling task. When the calling task's logical processor receives this message, the calling task's execution mode is set to run so it can proceed.

Wait for Synchronization

If the called task in the above scenario is not waiting at the entry, it would not immediately inform the coordinator that the rendezvous had begun. Thus, the coordinator would reflect the wait in its data base. The user would be able to see the called task executing elsewhere and the calling task waiting.

6. SUMMARY

The problem of testing and debugging Ada programs that make extensive use of tasking facilities has been addressed in this paper. We have considered an approach to debugging tasks that is similar to the scenario in which software units are first considered. Following units, interactions among units are addressed. Our approach recommends a three tier approach to debugging tasked programs. The first tier considers the logic of tasks independent of their interactions. The second tier addresses interactions among tasks that take place through rendezvous and synchronized access to shared data. The final tier deals with application specific concerns. Here, the subtlities of the interactions between a tasked program and its operating environment are considered.

We have presented the design of a debugger suitable for applying this methodology. AdaTAD , which stands for Ada TAsk Debugger, includes facilities specific to each of the tiers. When used in conjunction with a testing methodology, AdaTAD supports the execution of test cases and the process of locating and fixing errors uncovered through testing. We have presented the user interface to AdaTAD in conjunction with an explanation of the three tiered approach to debugging tasked programs.

The applications for which Ada is intended require a level of technology that currently doesn't exist in today's Ada compilation systems. For embedded real-time systems, a compiler must support the distribution of an Ada program across a set of possibly hetergeneous processors. When such compilation systems appear, we will immediately be faced with the challenge of demonstrating the reliability of Ada software. In addition to modifying existing testing and debugging methodologies, special purpose tools such as AdaTAD will be required. To ease the implementability of a system such as AdaTAD, we have designed the bulk of the system in Ada. While an Ada design certainly compromises execution efficiency, it also eases implementations. The final section of this paper has presented the Ada design of AdaTAD together with an example of how synchronization can be controlled and monitored using Ada primitives.

7. REFERENCES

1. Miller, E.; et.al. Program Testing, IEEE Computer, Vol. 11, No. 4, April 1978 pp. 10-12.

2. Helmbold and Luckham, "Debugging Ada Tasking Programs," IEEE Software, March 1985.

3. Whiteside, J., Jones, S., Levy, P. and Wixon, D. "User Performance with Command, Menu, and Iconic Interfaces," in Proc. CHI '85 Human Factors in Computer Systems, (San Francisco, April 14-18, 1985), ACM, New York, pp. 185-191.

4. Ada Language Reference Manual.

Software Unit Testing in an Ada Environment

Glenn Warnock
PRIOR Data Sciences

Introduction:

PRIOR Data Sciences is currently developing a validation procedure for the Ada binding of the Graphical Kernel System (GKS). PRIOR is also producing its own version of GKS written in Ada. These major software engineering projects will provide an opportunity for PRIOR to demonstrate a sound approach for software testing in an Ada environment.

PRIOR's GKS/Ada validation capability will be a collection of test programs and data, and test management guidelines. These products will be used to assess the correctness, completeness, and efficiency of any GKS/Ada implementation. GKS/Ada developers will be able to obtain the validation software for their own use. PRIOR anticipates that this validation software will eventually be taken over by an independent standards body to provide objective assessments of GKS/Ada implementations, using an approach similar to the validation testing currently applied to Ada compilers. In the meantime PRIOR will, if requested, use this validation software to assess GKS/Ada products. This project will require PRIOR to offer a well organized, thorough, and practical method for high level product testing.

The second project, PRIOR's implementation of GKS using the Ada language, is a conventional software engineering task. It represents a large body of Ada code and has some interesting testing problems associated with automated testing of graphics routines. Here PRIOR's normal test practices which include automated regression testing, independent quality assurance, test configuration management, and the application of software quality metrics will be employed.

PRIOR's software testing methods emphasize quality enhancement and automated procedures. These general methods apply to software written in any programming language. Ada makes some aspects of testing easier, and introduces some new concerns. These issues are addressed below.

The Goals of Unit Testing:

The goal of a test plan is the discovery of the maximum number of errors within a reasonable cost limit. Costs may be measured in dollars or in elapsed time, and will have different limits depending on the nature of the software being tested. For example, PRIOR is aiming to be able to validate a GKS/Ada implementation within a period of less than one week. To achieve this PRIOR's GKS/Ada test

suite will have to be carefully organized so that it is both robust, and yet still easy to use.

Testing of GKS/Ada provides an excellent example for our examination of Ada unit testing. Comprehensive and sophisticated unit tests are required to test the complex functionality. The requirements are well defined by the GKS standard, while the design specifications are covered by the proposed standard Ada binding for GKS. A unit test plan should test both the GKS requirements, and the GKS/Ada binding characteristics.

Testing Techniques:

Essentially, the purpose of unit testing is to exercise the module under test to verify that it performs correctly without producing undesirable side effects. PRIOR has developed TESTWARE, a collection of tools which provide a standard methodology to exercise and validate software modules. TESTWARE is used to initialize the appropriate global data areas and call the module to be tested with the appropriate input parameters. The returned parameters and results are then verified.

The use of a tool such as TESTWARE results in a suite of test cases which has significant value for the full life of the associated software module. An additional benefit of such a methodology is the ability to measure the degree of test coverage, to track the progression of testing, and to schedule software projects with greater accuracy.

The basic component of PRIOR's TESTWARE is the test driver. The test driver provides the framework necessary to run the tests and log the results. For each test, the necessary initializations of global data and input parameters are performed by the test driver. The module under test is called, executes and returns. The test driver must verify the return parameters and validate the global data.

In the course of execution of the module, some stubs may be necessary to "feed" the module with the necessary output parameters. It is often desirable to verify that the correct stubs are called and the appropriate input parameters passed to them. For these testing activities it would be very convenient to have an Ada compilation system that treated every call to an uncompiled subprogram as a request to interact with the test operator. The Ada system should make known the parameter values passed in, and permit the operator to supply values to be returned. We are currently writing stub routines to do this, but it would be more efficient to have this done automatically. Ada compilation systems with this capability will be very useful.

Every module to be tested requires a unique test driver. Therefore, the production of the test driver must be as automated as possible. Working from a standard

template, the test developer uses standard utilities and adds specialized code to perform the necessary initializations and verifications.

The test driver is actually driven by the test data. Data is required for initializing the global data and specifying the input parameters. Stub data is comprised of stub names, expected input parameters, and the required output parameters. Additional data describes the expected output parameters and specifies expected changes to global data. The separation of data from the test program eliminates the need to recompile the software when test data must be changed. An unlimited number of test cases can be defined in a single test data file.

Standard utilities are used to provide the translation from data to test case. The greater the flexibility available in describing test data, the more powerful and easy to use will be the testing tool. The tester should be able to easily specify enumerated types, character strings, and floating and fixed point real numbers. A range or allowable delta must be available for specifying expected output values such as floating point reals.

A variety of automated test tools such as TESTWARE have been developed for languages such as Pascal, C, and FORTRAN. These often test for errors which will not occur in Ada due to the strong typing, interface checking and run time error checking. However, additional testing difficulties arise which relate specifically to the Ada language. Testing of tasking operations is necessary to identify deadlock and starvation. Procedures for testing generic packages are required. Run time performance must also be assesed.

The GKS/Ada validation suite poses some additional problems. GKS output is often of a form which is most easily validated interactively. As an example, one test case may cause a green duck to be drawn upside down in the lower left corner. An important aspect of effective testing is that the test itself should validate the results. If the test procedure simply describes the correct display the operator may not notice if the green duck actually appears in the lower right corner. It is preferable to have the test software ask: "What colour is the duck?" (Green). "Is it upside down?" (Yes). "Is it in the lover left corner?" (No). It can be seen from this example that the task of supplying effective test software is a significant one.

The overall consideration in the design of TESTWARE is that the tester have the necessary tools to easily create the appropriate environment for running the unit under test and to be able to verify its actions and results. At the same time he must not be required to provide tedious amounts of data which are not directly related to the test.

Project Management:

Often the test portion of a software project is not given the attention or importance it deserves. Testing is usually viewed as something like "the process of

demonstrating that errors are not present" when actually errors are inherent in software. When software is tested by the person or group which developed it, with this attitude, it is not surprising that many errors go undiscovered.

To be successful testing should be approached with the philosophy expressed by Glenford Meyers. "Testing is the process of executing a program with the intent of finding errors". Testing is really a destructive process. The implementation schedule should reflect this and allow the necessary time for testing and corrections. The evaluation of test effectiveness should be based on the number of errors discovered. To be most effective it is best to have an independent test team.

Significant responsibilities must rest on the test authority. Developing a unit test for every module in the system is often not appropriate so the test authority must determine which modules should be tested and in which combination and order. The selection of appropriate test cases is critical to the success of testing.

Testing can be performed in an incremental or non-incremental manner. In the non-incremental method, all modules are tested seperately, with calls to lower modules replaced by stubs. When all modules have been tested, they are integrated and tested as a system. This method allows for greater parallelism in the unit testing process.

With incremental testing, the previously tested modules are used by the module under test, when available, instead of stubs. This provides more test coverage as the earlier modules are more extensively exercised. Also, integration and interface errors are discovered earlier and are easier and less expensive to correct.

Although top down design is often the preferred method of large system design, top down implementation and testing are not always preferrable. It is difficult to use an incremental method of testing if top-down implementation is used, as it becomes increasingly more difficult to provide the necessary input parameters to drive the test cases for the lower level modules as they are added. In addition a large number of stubs are required. With bottom up incremental testing, fewer stubs are necessary and the test driver is directly calling the module under test so that it is easier to force the test conditions.

Test cases can be generated by studying the internal logic and paths of the module (white box techniques) and by studying boundary conditions and combinations of input classes (black box techniques). Automated tools can also be helpful for this.

The real effectiveness of an automated test environment will be determined by its degree of integration into the software development environment. Test modules have to be associated with the appropriate software modules in the library. Commands should be available to permit the library manager to automatically retest appropriate modules. It is very important to track errors discovered and to have the ability to generate statistics and status information concerning the test process.

Coordination of Test Development:

A number of GKS test routines have already been written by groups in Europe and in the U.S.A. . PRIOR intends to include these in its test suite, and then extend it to cover new areas. By making this activity as visible as possible we hope to avoid any duplication of effort.

Formally Verifying Ada Programs which use Real Number Types

David Sutherland

Odyssey Research Associates

Table of Contents

We wish to apply formal verification to programs which use real number arithmetic operations (hereinafter referred to as mathematical programs). Formal verification of a program P consists of (1) creating a mathematical model of P, (2) stating the desired properties of P in a formal logical language, and (3) proving that the mathematical model has the desired properties of step 2 using a formal proof calculus. If the model faithfully embodies P, and the properties of step 2 are a correct formalization of the desired properties of P, the formal verification provides a high degree of assurance that P is correct.

There are two principal difficulties in formally verifying mathematical programs:

1. How to model inexact machine arithmetic operations

2. How to state the desired properties of mathematical programs in view of the fact that such programs in general deliver inexact results (e.g. a square root program does not compute the exact square root)

1 Modeling Machine Arithmetic

Our starting assumption is that machine arithmetic operations can be represented as the ideal real number operations followed by rounding. The operation of rounding is modeled by a cropping function, CR, from the real numbers (denoted by $\underline{R}$) to $\underline{R}$. The range of CR represents the machine real numbers, sometimes called the model numbers. This was the approach taken in [1], [2], and [3] and is consistent with the proposed IEEE standard for floating point arithmetic [4].

We will assume CR satisfies the following axioms, hereinafter referred to as "the cropping function axioms":

- Axiom 1: The range of CR is finite.

---

1. Mansfield, R., A Complete Axiomatization of Computer Arithmetic I to appear in the Journal of Mathematics and Computation

2. Holm, John, Floating Point Arithmetic and Program Correctness Proofs, Ph. D. thesis, Department of Computer Science, Cornell University, August 1980

3. Brown, W. S., A Simple but Realistic Model of Floating-Point Computations, Computing Science Technical Report No. 83, Bell Laboratories, April 1981

4. A Proposed Standard for Binary Floating Point Arithmetic, Draft 10.0 of IEEE Task P754, Dec. 1982

- Axiom 2: CR(CR(x)) = CR(x)

- Axiom 3: CR(0) = 0

- Axiom 4: [x <= y <= z & CR(x) = CR(z)] -> CR(x) = CR(y)

The first axiom expresses the fact that there are only finitely many machine real numbers. The second axiom says that the result of a rounding operation (i.e. a machine real number) is unaffected by further rounding. Note that the second axiom implies that the range of CR and the set of fixed points of CR are the same. The third axiom says that 0 is a fixed point of CR, i.e. that 0 is a machine real number. The fourth axiom says that if x and z round to the same number and y is between x and z then y rounds to the same number as x and z. As usual when stating axioms in first order logic there are implicit universal quantifiers in front of the formulas displayed as Axioms 2 through 4.

The cropping function axioms are consistent with the four rounding modes which the proposed IEEE Standard would require to be supported, namely rounding to the nearest machine real number, rounding towards 0, rounding towards plus infinity and rounding towards minus infinity. They are also consistent with rounding away from zero, a mode which is not mentioned in the proposed IEEE Standard.

We can derive some useful consequences of the above axioms:

1. CR is monotone, i.e. x <= y -> CR(x) <= CR(y)

2. There is no machine real between x and CR(x).

3. 0 <= x -> 0 <= CR(x) and x <= 0 -> CR(x) <= 0.

Note that the second statement does not imply that there is no machine real that is closer to x than CR(x). Again, we do not wish to require this because the proposed IEEE Standard would require other rounding modes than rounding to the nearest machine real.

2 Modeling Program Execution

We must embed the above ideas about modeling machine arithmetic into a larger model of program execution. We base our formal model of execution on a simple informal picture of program execution. We think of the program as executing a step at a time. At each point in time, the program (or the machine it is running on) is completely described by (1) the "point" in the program where control currently is, and (2) the values of each of the program variables. The program code determines the relationship between the values of variables and the point of control before a given step and after that step. We will assume, for the sake of simplicity, that all variables have a defined value initially, but this value will be unspecified by the execution model. In

addition, we will assume that the result of attempting to perform a computation which is undefined (e.g. division by 0) has a completely unspecified effect. To use the model, it will usually be necessary to prove that no undefined computations are attempted, and that the values of program variables are not referenced before they are assigned to.

How do we represent the above informal picture mathematically? We will represent "time" by the non-negative integers (which we will hereinafter refer to as the natural numbers). The "points" where control can reside will be represented simply by a finite set. The data types of program variables other than real number variables will be represented by the corresponding mathematical objects, e.g. the data type of integers will be represented as the mathematical integers. The real data type will be represented by the range of CR.

The execution of the program will be represented by a collection of functions giving the history of the flow of control in the program and the histories of the values of the program variables. Thus, there will be a function from time (i.e. the natural numbers) into the set of control points (which we will denote by PC), and for each program variable v, a function from time into the data type of v.

The functions representing histories will be required to satisfy certain conditions derived from the program. For example, if X, Y and Z are integer program variables, FX, FY and FZ the corresponding history functions, and at a certain time t control is at a program instruction

$$X := Y + Z$$

then the functions must satisfy the condition

$$FX(t + 1) = FY(t) + FZ(t)$$

For real variables, all operations are the ideal real operations followed by cropping. For example, if A, B and C are real program variables, FA, FB and FC the corresponding history functions, and at a certain time t control is at a statement

$$A := B + C$$

then the functions must satisfy the condition

$$FA(t + 1) = CR(FB(t) + FC(t))$$

3 Error Magnitude in the Model

The cropping function axioms capture certain qualitative properties of CR. They are not enough to do useful verification, however, because they say nothing about the size of the error introduced by CR. For example, the cropping function axioms are satisfied by the zero function. Thus, any program which we could verify using only the cropping function axioms would have to be correct even when running on a machine which used the zero function as its cropping function. Very few useful mathematical programs would be correct in any sense on such a machine, and thus we could not be able to verify such programs solely on the basis of the cropping function axioms. We need some additional axioms on the size of the error introduced by CR.

It is not clear, however, what kind of axioms to add. If we add axioms which give specific numerical bounds on the size of the error in a certain range, then any verification we do will only apply to machines that meet these numerical conditions. For a machine that did not meet the conditions, any verification done on the basis of the conditions would be invalid, despite the fact that many programs might still run correctly on the machine. On the other hand, some machines which met the conditions would probably actually meet much more demanding conditions. There could be programs which run correctly on such machines which we cannot prove correct because our axioms do not reflect the high degree of accuracy in the machine.

One solution to this dilemma would be to add non-specific numerical bounds on the error. In other words, add a symbol (say, "e") and add an axiom like "the percentage error between x and CR(x) is always less than e." One could then verify statements about the accuracy of mathematical programs in terms of e. For example, if P were a program to compute square roots, one might try to verify a statement like "the percentage error between P(x) and the square root of x is 5*e." If one then wanted a certain degree of accuracy from P, one could solve for the degree of accuracy in CR that would be necessary to achieve the desired accuracy from P.

There are several problems with this approach. First of all, it is very costly. With present technology in automatic theorem proving, the problem of generating and proving statements of the kind mentioned above in a mechanical proof system is intractable in terms of both the amount of computational power and the amount of human input required. Second, in some situations it forces us to do an analysis that is more detailed than necessary. Many errors in mathematical programs occur at a much lower level of numerical complexity. For example, ZBRENT is a Fortran subroutine from the IMSL library which is supposed to find a zero of a user-defined function F given a pair of endpoints A and B such that the values of F at A and B are of opposite sign. It does this by gradually moving the endpoints inward, always making sure that the values of F at the current endpoints are of opposite sign. In the process of the computation, it generates various pairs of real values X and Y which it must test to see if F(X) and F(Y) are of opposite sign. It does so by

multiplying F(X) and F(Y) together and testing whether the result is negative or not. This is an incorrect (not to mention inefficient) test, since it is possible to have F(X) and F(Y) be small numbers of opposite sign whose product is so small that underflow causes the machine to compute 0 for their product. This causes ZBRENT to act as if F(X) and F(Y) are of the same sign, giving incorrect results in some cases. This programming error is not "numerical" in nature, but is inherent in the notion of inexact (although "close") computation.

What we would like is a model of machine arithmetic which captures the idea of "close" but inexact computation without referring to specific numerical constants. In the next section we present such a model. The model is based on an alternate approach to real analysis called non-standard analysis.

4 Non-standard Analysis

Calculus was developed in the eighteenth century based on the notion of infinitesimals. These were positive entities dx smaller than any actual positive real but not 0. Furthermore, they obeyed the laws of ordinary real arithmetic so that one could carry out ordinary algebraic manipulations like

$$y = x^2$$

$$y + dy = (x + dx)^2$$

$$(x + dx)^2 = x^2 + 2 * x * dx + (dx)^2$$

$$dy = 2 * x * dx + (dx)^2$$

$$dy/dx = 2 * x + dx$$

In particular the derivative, dy/dx, was the actual quotient of two infinitesimals.

Attempts in the nineteenth century to justify working with these extended reals were not successful and a different approach and proof technique in terms of limits was adopted instead (the so-called epsilon/delta method.)

In the early 60's logicians showed how to justify working with actual infinitesimals using so-called "non-standard models of the reals." These models are ordered algebraic structures which have all the same algebraic and ordering properties of the standard real numbers, and which contain the standard real numbers, but which also contain additional, non-standard numbers. Doing real analysis by means of such non-standard models is called non-standard analysis.

4.1 Non-standard Models

What exactly do we mean by a "non-standard model" of some mathematical object like the real numbers? First of all, by "mathematical object" we will just mean a non-empty set. Before we give a precise statement of "non-standard model", we must discuss the notion of a first-order statement about a mathematical object.

Suppose we have a mathematical object M. A term of M is an expression which is of one of the following forms:

1. An element e of M
2. f(t1,...,tn) where f is an n-ary function from M into M and t1,...,tn are previously defined terms of M.

Thus, if M is the real numbers, then 0, 1 and 1 + exp(5) are terms on M (where exp stands for the "e-to-the-x" function and + is the usual addition function, written infix).

A first-order statement about M is a statement of one of the following forms:

1. p(t1,...,tn) where p is an n-ary predicate on M
2. A statement built up from finitely many previously constructed first-order statements by the use of logical connectives (e.g. "not", "and", "or", "if-then-else", etc.)
3. A statement of the form "for all x in M, ..." where ... is a previously constructed first-order statement involving the variable x.
4. A statement of the form "there exists x in M such that ..." where ... is a previously constructed first-order statement involving the variable x.

The following are first-order statements about the real numbers:

$$0 < 1$$

$$\text{not } (5 = 1)$$

for all x in the real numbers, for all y in the real numbers,
$$x + y = y + x$$

there exists x in the real numbers such that for all y in the real numbers, $x*y = y$

there exists x in the real numbers such that $x*x = -1$

Notice that the first four statements are true of the real numbers, whereas the fourth is false of the real numbers. A first-order statement about M need not be a true statement about M; it need merely be of a certain form.

In general, there will be some facts about a given mathematical object M which can be expressed as first-order statements and some which cannot. The first four examples above are facts about the real numbers which are expressible as first-order statements. A fact about the real numbers which is not expressible as a first-order statement is the fact that every non-empty set of real numbers which has an upper bound has a least upper bound (this property is called completeness). This statement is not a first-order statement as written because it refers to sets of reals rather than just individual reals. Some statements which refer to sets of elements or other higher-order structures turn out to be equivalent to first-order statements. For example, the statement "for every bounded set S of real numbers, there is a real number x that is not in S" is not in the form of a first-order statement, but it is equivalent to the first-order statement "for all x in the real numbers, there exists y in the real numbers such that $x < y$." It can be shown that the completeness property is not equivalent to any first-order statement.

We will now define what we mean by a non-standard model. Suppose we have some set M (e.g. the set of real numbers). A non-standard model of M consists of:

1. A set M'
2. For each element e of M, a corresponding element e' of M'
3. For each n-ary function from M into M, a corresponding n-ary function f' from M' into M'
4. For each n-ary predicate p on M, a corresponding n-ary predicate p' on M'

such that every first-order statement which is true of M is true of M' when the elements, functions and predicates in the statement are interpreted as the corresponding elements, functions and predicates of M'. For example, suppose R' is a non-standard model of the reals. Let +' denote the binary function on R' corresponding to the addition function on the reals. Since + is commutative, and since commutativity of + is expressible as a first-order statement (see the examples above), +' must be commutative on R'. On the other hand, R' need not have the completeness property, and there are non-standard models of the reals which are not complete.

We will call the elements of M' which correspond to elements of M the standard elements of M'. We can identify elements of M with their ocrresponding elements of M', and thus speak of M as being a subset of M'. Under this identification, for each function f and each predicate p on M, the corresponding f' and p' on M' is extends f and p respectively. We will call a non-standard model M' of a mathematical object M a proper non-standard model of M if there is an element x of M' which is not in M.

It can be shown (we will not give the proof here) that every infinite mathematical object M has a proper non-standard model M'. The same does not

hold for finite mathematical objects. The reason is simple. Suppose M = {e1,...,en}, and M' is a non-standard model of M. It is a true first-order statement about M that "for all x in M, x = e1 or x = e2 or ... or x = en" (the conjunction is finite). Therefore, the statement "for all x in M', x = e1' or x = e2' or ... or x = en'" is true of M', but this says that the only elements of M' are the standard elements.

4.2 Non-standard Models of the Reals

What does a proper non-standard model of the reals look like? It can be shown that every proper non-standard model of the reals consists of the standard real numbers plus the following three kinds of non-standard numbers:

1. Infinitesimals. These are numbers which are not 0 but which are smaller than any standard non-zero real number.

2. Infinite Numbers. These are numbers which are larger than any standard real number. There are both positive and negative infinite numbers. Every proper non-standard model of the reals must have infinite numbers as well as infinitesimal numbers in order to satisfy the algebraic property that every non-zero number has a multiplicative inverse. The multiplicative inverse of a non-zero infinitesimal is an infinite number.

3. Finite Non-standard Numbers. These are numbers of the form x + i where x is a non-zero standard real and i is an infinitesimal. Such numbers are neither infinitesimal nor infinite, but are not standard either.

In the original formulation of calculus, infinitesimals were informally thought of as non-zero real numbers which were in some sense "arbitrarily small". Thus, the notion of infinitesimals lends itself very well to modeling computation which is inexact, but whose inexactness can be taken to be arbitrarily small.

5 Non-standard Models of Execution

We will incorporate the idea of machine real operations which differ infinitesimally from the ideal operations by using non-standard execution models. A non-standard execution model will be a representation of program execution like that described in section 2, but with the standard mathematical objects replaced by non-standard objects. What exactly does this mean?

First, time will be represented by a proper non-standard model of the natural numbers. A proper non-standard model of the natural numbers consists of the standard natural numbers with infinite elements added. Thus, the history functions will be functions whose domain is a proper non-standard model of the natural numbers.

Second, all data types of program variables other than real variables will be represented by proper non-standard models of the standard data types (if proper non-standard models exist. For example, the data type "boolean" is finite and therefore has no proper non-standard models. Finite data types will be represented in non-standard models of execution by the standard model of the data type). For example, the data type consisting of the positive and negative integers must be represented by a proper non-standard model of the integers (which just looks like the standard integers with both positive and negative infinite numbers added).

What about the data type of machine real numbers? In section 2 we obtained the machine real data type by choosing a cropping function on the ideal reals and taking its range. We cannot replace this type by a proper non-standard of itself, because by the first cropping function axiom, this set is finite and so has no proper non-standard models. Suppose instead that we start with a proper non-standard model of the reals R' and a function CR from R' into R' satisfying the cropping function axioms and the additional axiom (called the "error axiom") that for all finite x in R', CR(x) - x is infinitesimal. This axiom formalizes the statement that on all numbers that are not "large" (i.e. not infinite), the roundoff error is "small" (i.e. infinitesimal). We will use the notation "x == y" to stand for "x - y is infinitesimal."

Unfortunately, there are no such cropping functions. In order for the error axiom to be met, the range of CR must be infinite, which contradicts the first cropping function axiom.

How can we resolve this inconsistency? There are definite cases in which we make use of the first cropping function axiom in verification, so we cannot simply abandon it. What we will do instead is, rather than assuming that CR satisfies the first cropping function axiom, assume that CR satisfies all first-order statements implied by the first cropping function axiom. It can be shown that the first cropping function axiom is not equivalent to any first-order statement, so this is a true weakening of our set of axioms. In addition, it can be shown that the resulting weaker set of axioms is consistent. The first-order consequences of the first cropping function axiom will be more than enough to verify most mathematical programs. In summary, we will represent the machine real data type in a non-standard model of execution as the range of a function CR from a non-standard model of the reals into itself such that CR satisfies cropping function axioms 2 through 4, the error axiom given above, and all first-order statements implied by the first cropping function axiom.

6 Specifying Mathematical Programs

How do we state the properties of mathematical programs we want to prove? Suppose we restrict ourselves to considering programs whose purpose is just to compute some real-valued function. If f is a real-values function of n arguments, and P is a program to compute f with parameters A1,...,An, we can state the specification of P in terms of the above formalism simply as "for

all inputs x1,...,xn, P(x1,...,xn) == f(x1,...,xn)" or, in slightly more detail, "for all inputs x1,...,xn, if P is executed with the initial values of A1,...,An being x1,...,xn respectively, then P will eventually terminate with output == f(x1,...,xn)." In terms of the above formalism, P has terminated at a time t if PC(t) = stop where "stop" is a control point at the end of the program.

7 An Example Verification

To illustrate the use of the model, we will verify a program which computes the square root function by Newton's method. The proof will be informal. We will denote the ideal square root of a number x by root(x).

Newton's method begins with an initial "guess" at the square root. The guess is then refined by an iterative process. At each step, the current guess g is replaced by (g + (x/g))/2 (where x is the number whose square root is being computed). The only facts about Newton's method we will need to know for the verification are that if x is non-negative and the initial guess is bigger than root(x), then:

1. All succeeding guesses will be bigger than root(x).

2. Each new guess will be less than the previous guess.

We now give the program. We will adopt the convention of writing the symbols for machine real operations "doubled", e.g. machine real addition will be denoted by "++", to distinguish machine operations from ideal operations (which will be denoted by the usual "undoubled" symbols). The value in RESULT is output when the program terminates. The program is:

```
SQRT(X:REAL):REAL

   RESULT := X ++ 1

   LOOP

      IF RESULT ** RESULT <= X

         THEN LEAVE

      IF RESULT <= (RESULT ++ (X//RESULT))//2

         THEN LEAVE

      RESULT := (RESULT ++ (X//RESULT))//2

      END
```

END

Note that the conditions for leaving the loop are not the kind of conditions one usually sees in programs of this type. The usual approach to terminating iterative processes of this type involves either terminating when a certain degree of accuracy is reached, or when a certain bound on the number of iterations is reached, or both. In SQRT, the iteration is terminated when the iterative process in the machine ceases to act like the ideal Newton's method in one of the two ways mentioned above.

We will now verify that if SQRT is executed with the initial value of X non-negative and finite, then execution eventually terminates with

RESULT == root(the initial value of x)

We will perform the verification by establishing a series of lemmas, leading up to the result we want.

Lemma 0: if x and y are non-negative and x == y, then root(x) == root(y).

Proof: the proof breaks into 2 cases:

Case 1: x and y are infinitesimal. The square of a non-infinitesimal number is non-infinitesimal, so root(x) and root(y) must therefore be infinitesimal, and thus the difference between them is also infinitesimal.

Case 2: either x or y is not infinitesimal. Since the two numbers differ by an infinitesimal, if one is not infinitesimal the other is also not. Since the square of an infinitesimal is infinitesimal, root(x) and root(y) are also non-infinitesimal. By algebra, we have

x - y = (root(x) + root(y))*(root(x) - root(y))

Since the left side is infinitesimal and the first factor of the right side is not, the second factor of the right side must be infinitesimal.

Lemma 1: Whenever (RESULT ++ (X//RESULT))//2 is computed, RESULT is not 0.

Proof: Suppose not. Let t be the earliest time such that PC(t) is at a statement where (RESULT ++ (X//RESULT))//2 is computed and RESULT = 0 at time t. Prior to t, the program must have been executing normally, since division by 0 is the only exceptional condition that can arise (we are ignoring exceptional conditions such as STORAGE_ERROR or overflow which cannot be analyzed on the basis of the program's text).

The only points in the program where (RESULT ++ (X//RESULT))//2 is computed are in the second conditional inside the loop and in the subsequent assignment statement. Since t is the earliest time when a division by 0 is attempted, and program execution before t is normal, we can conclude that:

1. Control at time t must be at the second conditional.

2. Control at time t - 1 must be at the first conditional with RESULT = 0.

3. X at time t - 1 must be negative (by cropping function axiom 3, if RESULT is 0 then RESULT ** RESULT is also 0).

But X is assumed to be non-negative initially, and since no assignments to X can have taken place in the course of normal execution prior to t, X must be non-negative at time t - 1, a contradiction.

We can therefore assume for the rest of the lemmas that the program executes normally at all times.

Lemma 2: The value of X is always the same as the initial value.

Proof: trivial, since there are no steps in the program which assign to X.

Lemma 3: SQRT halts.

Proof: Suppose not. In this case, the set of times t where the value of RESULT decreases from time t to time t + 1 has no upper bound (else at some point control would leave the loop at the second conditional). This fact can be expressed as a first-order statement using the history function for RESULT (call it FRESULT) as follows: "for all times t there exists a time t' such that t < t' and FRESULT(t' + 1) < FRESULT(t')." Howvere, the negation of this statement is a first-order statement which is implied by the first cropping function axiom, a contradiction.

Lemma 4: After the initial assignment to RESULT, the value of RESULT is always >= 0 and <= X ++ 1.

Proof: The proof is by induction on time (i.e induction on the number of steps that have been executed). Immediately after the initial assignment to RESULT, RESULT = X ++ 1 so certainly RESULT <= X ++ 1. We must therefore establish that 0 <= X ++ 1.

Since X and 1 are finite, X + 1 is finite and so by the error axiom, X ++ 1 = CR(X + 1) == X + 1. 1 is not an infinitesimal, and X is non-negative, so X + 1 is at least distance 1 from 0. Since rounding only introduces an infinitesimal error, and the distance between X + 1 and 0 is not infinitesimal, X ++ 1 cannot be 0.

To complete the induction, we must show that at every step in execution, if 0 <= RESULT <= X ++ 1 is true before the step, then it is true after. For execution steps which are not executions of the assignment statement inside the loop, this is trivial, since no other statement changes the value of RESULT. Suppose a given step is an execution of the assignment statement inside the loop. First of all, this means that control must have passed throught the preceding conditional, so the next value of RESULT must be less than the previous value, so if RESULT is <= X ++ 1 before the assignment then the same is true after. Second, as shown in Lemma 2, in order for control to have reached this statement at all, RESULT must be non-zero, so it is strictly

positive. The value of X is non-negative. Therefore, since CR of a non-negative number is non-negative, (RESULT ++ (X//RESULT))//2 must be non-negative. This completes the induction.

Lemma 5: RESULT is always finite.

Proof: Since 0 and X ++ 1 are finite and RESULT is always between them, RESULT is also finite.

Lemma 6: When SQRT terminates, RESULT == root(initial value of X).

Proof: We will denote the value of RESULT at termination by R. The proof breaks into three cases:

Case 1: R ** R = X. By Lemma 5, R is finite so by the error axiom, R * R == R ** R = X = initial value of X and the conclusion follows by Lemma 0.

Case 2: R ** R < X. Claim: R * R < X. If not, then R * R >= X, so by the montonicity of CR, R ** R = CR(R * R) <= CR(X) = X, a contradiction. The initial value of RESULT has square > X, so the assignment statement inside the loop must have been executed at least once before termination. Therefore, there exists a previous value of RESULT, call it RP, such that R = (RP ++ (X//RP))//2 <. RP and RP ** RP > X. By the same reasoning as above, the second statement implies that RP * RP > X. Therefore 0 < X/RP < RP so X/RP is finite, so (RP ++ (X//RP))//2 == (RP + (X/RP))/2. But the left side is less than root(x), while the right side is greater than root(x) by property of (ideal) Newton's method. When two numbers which differ by an infinitesimal are on either side of a fixed number, they each differ from that fixed number by an infinitesimal. This establishs the conclusion.

Case 3: R ** R > X. In this case, the program must have terminated because R <= (R ++ (X//R))//2. The assumption of the case implies that R * R > X as above, so 0 < X/R < R so X/R is finite, so (R ++ (X//R))//2 == (R + (X/R))/2. The left side is >= R, while the right side is < R by property of (ideal) Newton's method. Therefore, R - ((R + (X/R))/2) is infinitesimal. Rearranging algebraically, we get (R*R - X)/(2*R*R) is infinitesimal. The denominator is finite, so the numerator must be infinitesimal. The conclusion follows from Lemma 0.

8 The Asymptotic Interpretation

What does verification of a mathematical program executing over a non-standard model of the reals tell us about actual execution on a standard machine? This question is similar to the question "what does a proof in non-standard analysis involving infinitesimals show about analysis in the standard reals?" We will explain heuristically how non-standard analysis proofs relate to standard analysis, and argue by analogy that the same relation holds between verification of non-standard execution and execution on a standard machine.

It can (and has) been proved that the analogy is actually valid, but the proof is beyond the scope of this paper.

Consider the non-standard analysis proof that the derivative of the x^2 function is 2*x. It goes as follows: take an arbitrary infinitesimal i and compute ((x + i)^2 - x^2)/i. The result is 2*x + i. Thus, the value of the difference quotient for any infinitesimal is only infinitesimally different from 2*x. This is actually a proof that the standard x^2 function has derivative 2*x in the usual sense, although it takes some mathematical logic to prove the connection.

What does it mean to say that the derivative of x^2 is 2*x in standard analysis? It means that the limit of the expression (x + h)^2 - x^2)/h as h goes to 0 is 2*x. Thus, a non-standard analysis proof about numbers being infinitesimally different establishes a standard fact about behavior of an expression as a certain quantity gets smaller and smaller.

The same relation holds between non-standard and standard execution. Our proof that if x is non-negative and finite then SQRT(x) == root(x) actually establishes that if we run SQRT on a sequence of machines whose CR is more and more precise, the output of SQRT(x) will converge to root(x). More generally, if we have any real-valued ideal function f and a program F and we can prove in the non-standard formalism that for all finite x in the domain of f, F(x) == f(x) then this will establish that if we run F on a sequence of more and more precise machines, the output of F(x) will converge to f(x). To put it another way, we can obtain any degree of precision in F(x) by computing F(x) on a sufficiently precise machine.

Ada® Test and Verification System: (ATVS)

Tom Strelich

General Research Corporation
5383 Hollister Ave.
P.O.Box 6770
Santa Barbara, CA 93111

1 Introduction

The Ada Test and Verification System (ATVS)[1] is an integrated set of software tools for testing, maintaining, and documenting Ada programs. The objectives of the ATVS are to improve the reliability and maintainability of Ada programs. GRC performed the research and analysis leading to the specification of ATVS requirements and its high-level design[2].

1.1 Background and Overview

Software testing, verification, validation, and certification are critical software development problems facing NASA. To overcome these problems, NASA has invested large amounts of time and money to correct and certify systems only to find that, when deployed, they often behave erratically or produce incorrect results. Spending more time and money on exhaustive testing won't solve the problem either since most software programs found in mission critical systems (such as the Space Station) are of such size and complexity that no amount of testing can guarantee completely correct, error-free performance. The objective then is to make the testing process as effective as possible by providing computer-aided assistance to the software engineer to help them discover the greatest number of errors for every hour spent testing.

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

[1] This work was performed under Rome Air Development Center Contract F30602-84-C-0118

[2] Ada Test and Verification System (ATVS): Final Report, General Research Corporation, CR-6-1301, September 1985.

A proven approach to software testing is the use of Automated Verification Systems (AVS). This technology was pioneered both by NASA and Rome Air Development Center, and GRC has participated actively in these efforts. For NASA, GRC developed an AVS for the AED language. For RADC, GRC developed AVS's for FORTRAN, COBOL, and JOVIAL J73 (FAVS, CAVS, and J73AVS). The ATVS represents the logical evolution of AVS technology in support of the Ada programming language.

Ada provides a high-level programming language with advanced capabilities addressing reliability issues (e.g., strong data typing, exception handlers, information hiding, etc.). However, the Ada language alone represents only a partial solution to software development problem confronting NASA: the full benefit of Ada to Space Station Software development will be realized through the synergistic interaction of the Ada language, the Software Development Environment, and supporting software tools (e.g., ATVS).

1.2 Operational Concept

Figure 1.1 illustrates the ATVS high-level operational concept:

1. Ada source code is submitted to the ATVS for Static Analysis (e.g., package dependencies, program call tree, global symbol information, data flow anomalies and errors, unreachable code, potential task deadlocks, etc.). In response to the Static analysis reports and displays, the user makes whatever corrective actions are required and repeats the process until there are no statically detectable errors in the source code.

2. The user's Ada source code is then Instrumented with run-time data collection probes which capture execution information (such as execution coverage, performance timing, and task state activity) for subsequent analysis and reporting.

3. The instrumented Ada source code is then compiled, linked, executed (with user supplied test data) with the ATVS instrumentation probes collecting run-time execution information. Assertion violations are reported to the user who may then make corrective actions and repeat the process.

4. The run-time execution data collected by the ATVS instrumentation probes is analyzed producing execution coverage, timing, and task state reports. Based on these reports the user takes corrective actions such as modifying the test data

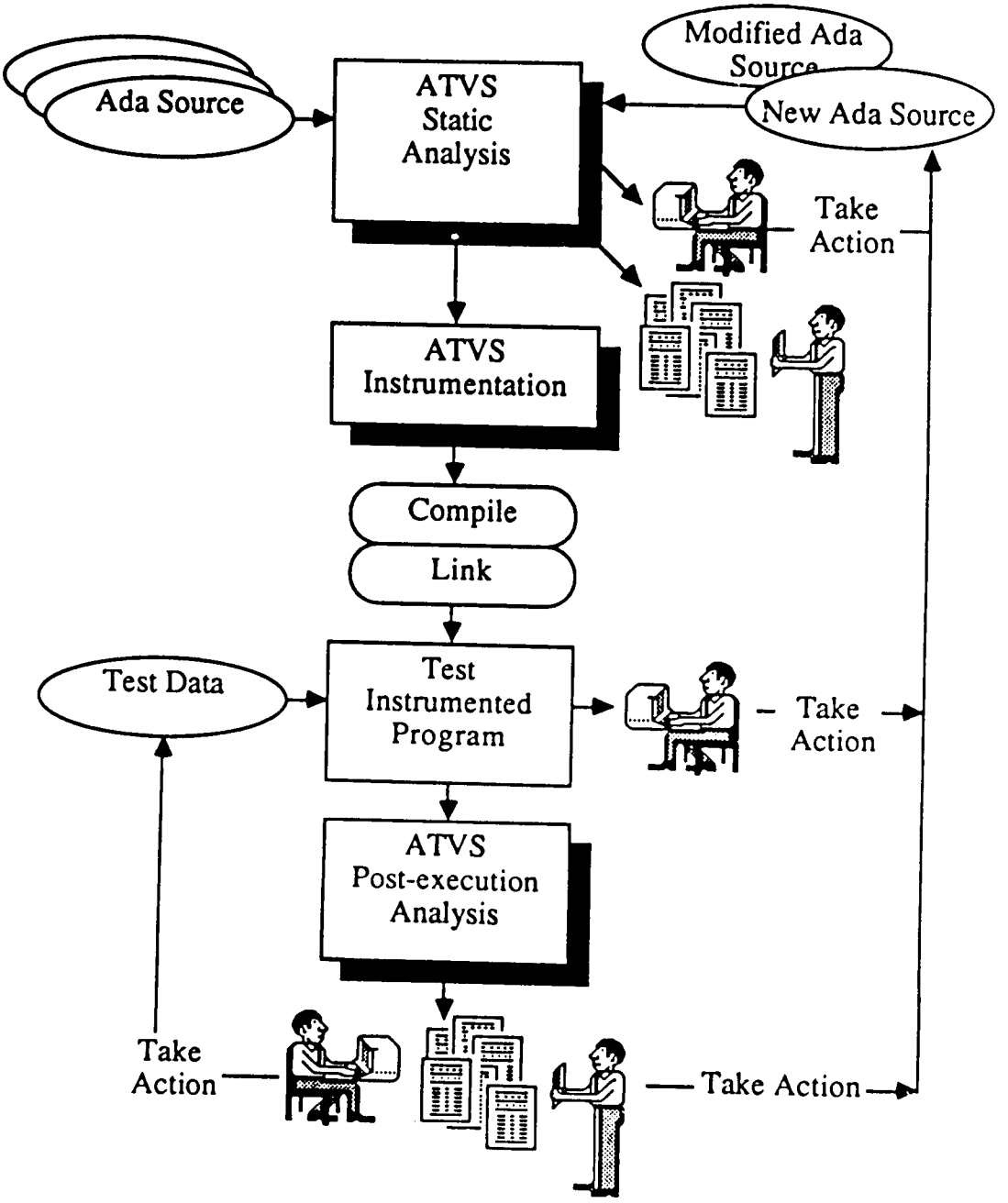


Figure 1.1. ATVS Operational Concept.

to effect execution coverage or modifying the source code to improve performance, eliminate unanticipated task interactions, and correct logic or design errors.

As suggested by the previous scenario, application of the ATVS is focused on the coding and testing phases. Figure 1.2 illustrates the role of the ATVS in the DOD-STD-2167 software development cycle: namely, Coding and Unit Testing, CSC Integration and Testing, CSCI Testing, and Maintenance Phases (while the Maintenance phase is not explicitly described in DOD-STD-2167, we have included it since the ATVS is expected to be used quite heavily for software maintenance).

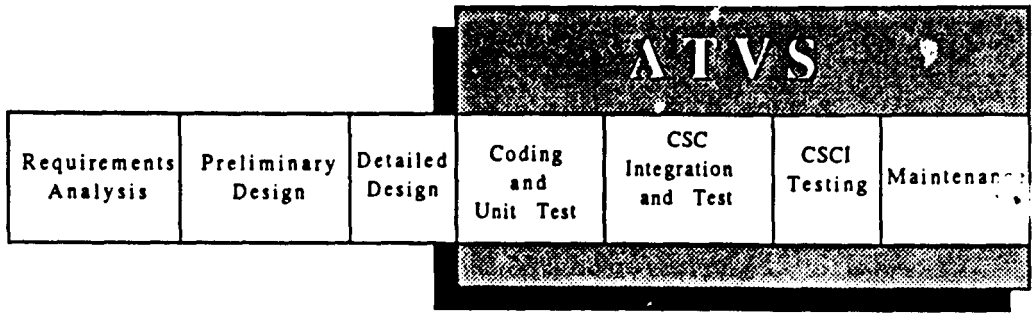


Figure 1.2. Role of the ATVS in the Software Life Cycle.

1.3 Objectives

The objective of the ATVS is to provide a set of computer-based tools which improve the reliability and maintainability of Ada software systems. The specification and design of the ATVS concentrated on the environmental context: that is, its effective integration within an advanced software development environment (such as NASA's SDE) and its contribution to that environment (e.g., support for project management, change and configuration management, test and integration, documentation, requirements traceability, etc.). The ATVS will provide detailed program information for software engineers and programmers and summary information for software project managers. The ATVS can provide management visibility by serving as a window into the software development process.

2 Capabilities

The ATVS will provide both Static and Dynamic Analysis of user programs. The requirements and design of the ATVS concentrated on providing support for the unique features of the Ada language, host-target testing issues, distributed environments, and advanced user interface capabilities.

ATVS capabilities fall into four functional groups: Static Analysis, Dynamic Analysis, Report Generation, and User Interface capabilities. Table 2.1 summarizes ATVS Functional Capabilities by group. Specific capabilities of the ATVS are described in the following paragraphs.

Table 2.1. ATVS Functional Capabilities by Group

Static Analysis	Dynamic Analysis	Report Generation	User Interface
• Source Processing	• Instrumentation -- Coverage -- Timing -- Tasking	• Automated Reports	• Batch and Interactive User Interface
• Static/Structural Analysis	• Executable Assertions	• DOD-STD-2167 Documentation	• Interactive Walkthrough
• Static Task Analysis	• Post-execution Analysis	• Prologue Insertion & Extraction	
• Programming Standards	• Unit Testing	• Software Quality Metric Data	

2.1 Static Analysis Capabilities

Ada Source Processing. The ATVS will process the Ada language and perform lexical, syntax, and semantic analysis necessary for subsequent static and dynamic analysis. It will produce a DIANA intermediate representation of the users program which will be used to build the ATVS database. The ATVS database is the central repository of program information and serves as the primary means of communication between ATVS tool components.

Static and Structural Analysis. The ATVS will provide extensive static and structural analyses concentrating on analyses unique to the Ada language. The analyses include:

- Package Dependencies -- describes "with" and "use" context clause dependencies and is valuable for change impact analysis
- Compilation/recompilation Order Dependencies -- Provided by most compilers, it is useful for maintaining system consistency subsequent to program modification
- Data Flow Errors/Anomalies -- identifies variables declared but not used, uninitialized variables, actual output parameter not set, etc.
- Global Symbol Use -- Identifiers, Types, Overloadings, Generics, Exceptions, Interrupts

Static Task Analysis. This capability identifies the set of all possible sequences of concurrency in a given program. This sequence set is then used to identify features of the program's synchronization structure such as: all possible task rendezvous, all potential areas of concurrent execution, and areas of potential task blockage (i.e., deadlock). This capability will utilize the Temporal Semantic Analysis approach described by Buhr, et al[3] .

Programming Standards Checking. This capability provides for user source code auditing against a set of modifiable programming standards. For example, "the maximum # of statements in a procedure is 25". The ATVS has defined a set of 46 programming standards.

2.2 Dynamic Analysis Capabilities

Instrumentation and Executable Assertions. Instrumentation consists of the insertion of software probes into the user source code. These instrumentation probes collect run-time program information for subsequent analysis and reporting. The types of instrumentation include: program execution coverage, program timing, and tasking activity. An executable Assertion is a statement placed in the source code by the programmer to indicate that the specific condition should exist. For example:

---

[3] Buhr, R., et al. "Experiments with PROLOG Design Descriptions and Tools in CAEDE: An Iconic Design Environment for Multitasking, Embedded Systems," Proceedings of the 7th Int'l Conf. on Software Engineering, IEEE Computer Society, 1985.

--. assert ((velocity - v_naught) > epsilon)

When violated (i.e., the statement evaluates to false) during program execution, the Assert statement can either display an assertion violation message to the user, or take some alternative action defined by the user.

Post-execution Analysis (Coverage, Timing, and Tasking). This capability processes the program execution data collected at run-time by the instrumentation probes embedded in the user's source code. Analyses include: (1) execution coverage for programs at the subprogram, branch, and statement level; (2) execution timing at the subprogram, named block, or statement level; (3) task state transitions, basically a trace of the program's tasking activity. The tasking analysis information can be used in cooperation with the static task analysis information to determine the extent of task sequence set coverage (task synchronization set coverage represents the functional analog of execution coverage in sequential programs).

The ATVS will provide data collection for both single and multiple program executions. This capability allows post-execution analyses to reflect incremental and cumulative execution coverage, timing, and tasking information. This type of historical information is an essential part of software documentation.

Unit Testing. This capability provides for automatic (with user direction) construction of Ada drivers and stubs. It will identify the undeveloped portions of a program and will construct Ada driver and stub "skeletons" which can be customized to a user's particular testing requirements. This capability supports both top-down and bottom-up development methods.

ATVS Dynamic Analyses will be supported for both host-resident and target-resident Ada programs (assuming an upload/download capability between the host and target).

2.3 Report Generation Capabilities

It is important to note that the ATVS design has separated the process of static and dynamic analysis from the process of report generation. The effect of decoupling these two activities (which communicate through the common database) is that it allows definition and incorporation of new analyses and reports to proceed independently of one another. This approach provides the flexibility necessary for the incorporation of new capabilities into the ATVS allowing it to evolve over time in response the the

environment, the user community, and advances in software engineering. Table 2.2 summarizes ATVS automated reports.

Table 2.2. Summary of ATVS Automated Reports.

Static Analysis Reports	Dynamic Analysis Reports
Summary Information Report Compilation Unit Overview Report Compilation Order Report Subprogram/Task Dependency Report Subprogram Cross Reference Task Cross Reference Package With/Use Dependency Report Package Element Set/Use Cross Reference Data Dictionary Report Global Entities Cross Reference I/O Statements Report Type Information Report Type Cross Refrence Report Object Cross Reference Report Type Derivation Report Generic Instantiation Report Exception Handling Report Interrupt Handling Report Overloading Information Report Statement Profile Report Software Metrics Report Target Code Cross Reference Data Flow Anomaly Report Programming Standards Report Source Re-analysis Report	Testcase Report Execution Coverage Summary Report Branch Coverage Summary Report Detailed Coverage Report Branch Report Reaching Set Report Execution Timing Report I Execution Timing Report II Task State Report
	DOD-STD-2167 Reports
	Calling Tree Report Functional Allocation Report Global Data Definition Report Input Data Report Local Data Definition Report Output Data Report Element Utilization Report File Description Report Record Description Report

Automated Static and Dynamic Analysis Reports. All static and dynamic analyses performed by the ATVS will be available to the user in both interactive display and hardcopy forms. The ATVS will provide 25 Static Analysis Reports and 9 Dynamic Analysis Reports.

DOD-STD-2167 Compatible Reports. The ATVS will provide nine automated reports consistent with DOD-STD-2167. These reports are variants of the ATVS automated reports and are generated from database-resident information provided by ATVS static and dynamic analyses. The separation of analysis and report generation described above allows for the definition of several reports based on the same analysis. This will allow

definition of new reports (both informal and DOD-STD) without requiring development of new analyses.

Prologue Insertion/Extraction. The ATVS supports the insertion of selected automated report information (e.g., package, subprogram, and task dependencies, global symbol use, etc.) into a prologue (i.e., a descriptive preface to a program unit). Prologues are embedded in the user's source code as Ada comments and can be augmented with user provided information. Automatic insertion of prologue information ensures current and consistent program documentation. Prologues can be automatically extracted from the source code to generate formal documentation.

Raw Software Metric Data. The ATVS will provide raw software quality metrics for analysis by other environment tools. These metrics (37 individual metrics supporting 18 software quality criteria) are consistent with the STARS Data Collection Forms, Software Evaluation Report and Software Characteristics Report[4].

2.4 User Interface Capabilities

Batch and Interactive User Interface. The ATVS will provide both a batch and interactive user interface. The batch interface will utilize a batch command language to direct ATVS processing. The full complement of ATVS capabilities (except for exclusively interactive activities such as Interactive Walkthrough) will be accessible through the batch command language.

The Interactive User Interface will be based on a hierarchical menu structure providing users controlled access to ATVS functions. There will be an extensive on-line help facility providing both reference and tutorial information. The Interactive User Interface will take advantage of advanced terminal/workstation bit-mapped graphics capabilities such as multiple windows, pull-down menus or palettes, and alternate input devices such as mice.

Interactive Walkthrough. Interactive Walkthrough replaces the manual process of "digging" through large source listings, cross reference reports, and other forms of documentation. It provides users with controlled, interactive access to the source code comprising a large software system. The user can browse the source code based on the program's call

---

[4] Interim Software Data Collection Forms Development -- Software Evaluation Report, Software Technology for Adaptable, Reliable Systems (STARS), RADC/COEE Griffiss AFB, NY, June 1985.

tree or as directed by the user, and the multiple window capabilities of the interactive user interface allow simultaneous access to various ATVS static and dynamic reports.

3 Database and Workstation Issues

3.1 ATVS Database

The ATVS database was designed as an "Entity-Relation-Attribute" (ERA) Database composed of 13 database entities and 17 associated relationships. The ERA model was selected for its expressiveness and flexibilty: The ATVS database contains a great deal of semantic program information that is best represented in the ER model.

3.2 ATVS Functional Distribution to Workstations

The ATVS was designed to operate in whole or in part on either a host machine (such as a VAX) or a microcomputer workstation (such as a SUN or VAXStation II). This flexibility allows program managers to relegate certain ATVS functions (e.g., source processing, instrumentation, etc.) to the host machine, and other functions (e.g., static analysis, post-execution analysis, interactive walkthrough, etc.) to the workstation. Microcomputer workstations often provide advanced capabilities (such as multitasking, bit-mapped graphics, multiple windows, etc.) that the host cannot easily (if at all) provide without serious degradation in system response. An additional benefit target system testing since microcomputer workstations are often used as embedded system development environments.

4 Current Status and Conclusion

The ATVS functional description and high-level design[5] are complete and are summarized in this paper. The ATVS will provide a comprehensive set of test and verification capabilities specifically addressing the unique features of the Ada language, support for embedded system development, distributed environments, and advanced user interface capabilities. Its design emphasis was on effective software development environment integration and flexibility to ensure its long-term use in the Ada software development community.

---

5 Ada Test and Verification System (ATVS): Functional Description, General Research Corporation CR-2-1301, September 1985.

The Testability of Ada Programs

David Auty, SofTech, Inc.
Norman Cohen, SofTech, Inc.

Software development for NASA's space station poses a significant challenge; considered the most difficult challenge by some. The difficulty is the magnitude and complexity of the required software. With the requirements for remote control and communications, software will lie at the heart of many essential and complex systems within the station. The combined requirements for highly-reliable systems exceed any software development effort yet attempted.

NASA's previous experience with software development centers on the assembly code and the code in the high-level language HAL/S, developed for the space shuttle. Within the development of that software there was heavy reliance on careful testing and thorough multi-level checkout. Within the HAL/S development environment, the checkout procedures could depend on the stable characteristics of and limitations on program behavior inherent in the language. This paper addresses the concerns raised by consideration of the requirements for testing and checkout procedures for the space station software. In particular it addresses the use of Ada in the development of widely distributed yet closely coordinated processing.

This analysis is done in two contexts. First, an evaluation of the language is presented, discussing how the rules and features of the Ada language effect the testability of software written in it. Second, some general techniques in software development which can augment testing in the development of reliable software and some specific recommendations for tools and appropriate compilation are presented.

This paper is a summary of a full report prepared at the conclusion of an extended study effort on this topic. It therefore does not go into detail in elaborating each point of interest. An attempt has been made to cover the breadth of the report and present its key findings.

Evaluation of Ada

We begin by discussing how a programming language can be evaluated for testability. For our purposes, testability is the ability to determine, by test execution of software, whether the software will function correctly in operational use. Testability measures the extent to which it is possible to construct tests such that the behavior of the software on those tests reflects

the behavior of system when deployed. Among the issues related to testability are the ease of generating comprehensive test cases, the predictability of resource utilization under all circumstances and the deterministic repeatability of processing sequences.

This definition applies principally to the developed program, but it can be extended to apply to the language used to express that program. A programming language supports testability to the extent that it facilitates the writing of testable software. We have identified the following attributes of a programming language which facilitate testability:

- support for modular decomposition (i.e., supporting the testing of units independently of their use in the system),
- existence of interface specifications constructs which are clear and comprehensive
- complete type and program unit specifications allowing comprehensive consistency checking during program compilation,
- well-defined run-time error handling,
- predictable resource allocation and utilization,
- support for the writing of test drivers and hardware stimuli simulation, and
- support for the creation of high-level abstractions.

With these evaluation criteria, we considered the following aspects of the Ada language:

- Data Types and Subtypes,
- Separate Compilation and Packages,
- Subprogram Definition,
- Generic Units,
- Exceptions,
- Concurrent Processing and
- Storage Management.

Each aspect was considered from the viewpoints of conformance with evaluation criteria, risks to testability and recommendations for reducing those risks.

Fig. 1 shows an evaluation criteria versus features matrix showing the extent of support of the Ada language for testability. The matrix shows where aspects of the language support the evaluation criteria, independent of the possible risks within the same feature area. In general, the strong typing rules of the language and the concept of separate specification and program unit bodies provide excellent support for testability.

	Data Types and Subtypes	Separate Compilation and Packages	Subprograms	Generic Units	Exceptions	Concurrent Processing	Storage Management
Modular Decomposition	●	●	●	●		●	
Clear & Comprehensive interface specifications	●	●	●	◐	◐	●	●
Compile time consistency checking	●	●	●	●			
Well-defined run-time error handling	●				●	●	●
Predictable resource use and allocation	●	●	●			●	●
Support for test and test driver programs		●	●	●	●	●	
Support for creation of high level abstractions	●	●	●	●	●	●	●

Fig. 1, Evaluation Criteria vs. Feature Matrix

Two areas of particular interest are represented as only half-filled circles in the evaluation matrix. These represent qualified support for the evaluation criteria. In the case of exception management, the rules for the raising of exceptions, including user specified raise statements, and for exception propagation, allow for a very concise treatment of exception processing. Thus, when properly documented, exception processing as defined in the language is an important part of a module's interface, supporting the requirement for clear and comprehensive interface specifications. Because it is dependent on optionally included comments, however, this can be considered only qualified support for the evaluation criteria.

The second half-filled circle is under generic units. This is a similar situation as for exceptions. The rules for formal generic parameter specification and for generic instantiations allow for a clear and concise specification of the units interface. However, as will be discussed under risks, there are secondary aspects of actual parameters (which we term second order properties) which are not documented, such as functional requirements on actual procedure parameters. Because these secondary aspects can be critical, yet possibly undocumented, support in this area is also qualified.

Testability Risks

In the evaluation of the Ada language features, several risks to testability as well as the above benefits were identified. These risks fall into two broad categories of inefficiency and hidden interfaces, plus one additional concern without such convenient categorization.

The concern over efficiency is based on a simple assumption that features which fail to provide adequate efficiency will not be used in many applications. The resulting program which may be more or less convoluted in its avoidance of this feature will certainly not have benefited in its testability. Although processing capabilities and memory sizes are increasing dramatically, the requirements to surpass the increased capabilities are already being considered. Concerns over efficiency in Ada fall into three areas:

- excessively expensive run-time checks,
- inappropriate or undirected instantiation of generic units, and
- excessively expensive tasking architecture.

These can be collected under the general concern of inefficiency in support of high-level abstractions.

The second broad concern is that of hidden interfaces. Despite the strong support in the language for detailing important interface information, several possibilities for hidden interfaces exist. Hidden interfaces exist wherever interactions or dependencies exist which are not part of the specification or declarations of the unit. These can be classified as being due to:

- global variables (side effects of procedure and function calls, contention over access between separate tasks),
- the raising and propagation of exceptions,
- dynamic storage utilization,
- dynamically determined timing behavior, and
- second order properties (e.g. functional requirements on actual procedure parameters) for generic instantiations.

An example of second order properties would be the case of a generic sorting procedure. A typical implementation will have the type of the objects as a generic parameter, requiring a second parameter to be a function which can compare values of that type and return a boolean value on the basis of the condition "less than". The second order property of the actual function used during instantiation is that it must return a proper ordering of all values of the type. In fact, it is conceivable that the sorting routine may never reach

an exit point if the function does not have this property. Yet this property is not required in any way by the language during instantiation.

One last risk for testability is the general non-determinism of tasking interactions. While not so much a fault of the language, as asynchronous concurrent processing is inherently non-deterministic, the presence of tasking in an Ada program can complicate the testing of that program.

Recommendations to Reduce Risk

In response to the identification of these risks, several recommendations for reducing the risk were made. These fall under the general headings of:

- requirements for appropriate development practices and training,
- requirements for appropriate tools, and
- requirements for appropriate compilation.

The principle behind the requirements for appropriate development practices and tools is based on the recognition that their use can help assure reliable software where testing is difficult. Testing practices can be augmented by the use during development of proof techniques, static program analysis and runtime monitoring. Throughout the development process, verification techniques can be used to insure principles identified and verified early in the development are held true through implementation.

For appropriate programming guidelines and training, the following suggestions were made:

- For numeric processing, training should include a discussion of digital computation algorithms and their interaction with underlying numeric precision in determining the accuracy of the computed value. This is necessary to put the rules for numeric precision of the language in proper context.

- Programming Guidelines should be established for:
  - the judicious use of suppress and inline pragmas to provide efficiency as necessary,
  - the avoidance of global variables and hidden side effects,
  - the hiding of persistent variables in package bodies (and therefore private to the package), and
  - the use of out parameters from procedures over unconstrained composite results from functions (allowing better storage utilization).

- Training should emphasize:
  - concurrent programming concepts and practices
  - the concept and significance of second order properties of generic parameters

- Standards (with enforcement) should be established for:
  - the documentation and use of exceptions
  - storage utilization practices

A more reliable approach to improving testability is through the use of appropriate tools to aid in the development process. The following are some tools to specifically address the risks for testability identified:

- Proof systems for verifying 2nd order assertions in generic instantiations and assertions about task interactions, task state systems and other program properties.
- Runtime monitors for deadlock and other deadness errors, storage utilization parameters, and other runtime properties.
- Static program analysis for tasking interactions, storage utilization and other program properties including adherence to the programming guidelines listed above.
- Expert system support such as a "real-time assistant" for cyclic-based system generation.

Having identified program efficiency as a risk to testability, in that good features of the language will not be used if they are not sufficiently efficient, several suggestions for appropriate compilation should be considered. In general, a highly optimizing compiler, with efficient, deterministic runtime support is a necessary goal. Particular attention should be given to the following features:

- optimization of subtype range constraint checking,
- reduction of uncertainty in the raising of predefined exceptions,
- space efficient compilation with pragmas and representation clauses for user control of storage utilization,
- optimization of tasking interactions with special support for tasking paradigms through pragmas or pattern recognition, and
- efficient size and speed of generic instantiations with pragmas for user specification of instantiation criteria.

Summary

In summary, it was found that the language offered the potential to greatly improve the testability of software, provided that certain guidelines were followed. The language introduces features to deal with higher level abstractions and the complexities of concurrent processing and dynamic storage utilization. These features are considered necessary to deal with the complexities of the space station software requirements, but can decrease the testability of that software. These risks to testability can be dealt with through a combination of appropriate development practices and training, appropriate tool support and appropriate compilation.

FORMAL VERIFICATION AND TESTING: AN INTEGRATED APPROACH TO VALIDATING ADA PROGRAMS

Norman H. Cohen

SofTech, Inc.
One Sentry Parkway, Suite 6000
Blue Bell, Pennsylvania 19422-2310

NCohen@Ada20

Formal verification is the use of mathematical proof to confirm that a program will behave as specified when it is executed. Formal verification can produce a much higher level of confidence in a program than testing. Nonetheless, formal verification requires large amounts of skill, human time, and computer time, so it would be impractical to verify formally an entire Ada program for a typical embedded computer application.

We propose an integrated set of tools called a validation environment to support the validation of Ada programs by a combination of methods. The validation environment exploits the Ada distinction between module interfaces and module implementations to validate large Ada programs module by module. The proposed validation environment is called the Modular Ada Validation Environment, or MAVEN. MAVEN does not yet exist, nor have efforts begun to construct it. Rather, MAVEN is our vision of the context in which Ada formal verification should be applied. A more complete discussion of MAVEN can be found in [1].

Our vision of MAVEN is based on several requirements that we have identified for the validation of Ada programs. These requirements are based on the recognition that Ada programs for mission-critical applications are large, that skilled software engineers are in short supply, that the construction of a verifier is an expensive undertaking, and that the use of a verifier may be time consuming. Our requirements are as follows:

1. Formal proofs should not be based on the behavior of a particular implementation.

2. It should be possible to validate a large program module by module.

3. For typical mission-critical applications, verification will have to be integrated with other forms of validation.

4. It should be easy to request the proof of certain critical properties which, while they do not imply correctness of a module, significantly raise our confidence in its reliability.

See [2] for a more complete discussion of these requirements.

When software engineers use the term "validation and verification," they usually do not have formal verification in mind. To avoid confusion, this paper uses the terms validation and verification in two distinct and precise senses:

> Verification is the use of formal proof, checked by machine, to establish properties of a program's run-time behavior.
>
> Validation is the process of increasing one's confidence in the reliability of a program. Formal proof is one of many methods for validating software.

Confusion may also arise from our use of the term environment. Ada Programming Support Environments (APSE's) already exist, and have functions that overlap those we propose for a validation environment. We do not envision MAVEN as a full APSE or as a tool set independent of an APSE. Rather, we view MAVEN as an integrated tool set embedded within an APSE. It can be thought of as a "subenvironment." Many APSE tools, including an Ada compiler, may be used both for validation and for other purposes.

1 Integration of Multiple Validation Methods

One reason for validating programs module-by-module is so that different modules can be validated in different ways. There are many software unit validation methods, all of which have been used successfully in the past. These include:

- formal proof generated with machine assistance and checked by machine
- informal proof carried out by hand
- code walkthroughs
- unit testing
- acceptance of a software component as trustworthy, based on experience using the same component in a previous system

It is not necessary for a project to choose one of these validation methods for use throughout a program. Given the right framework, different methods can be combined in an effective symbiotic relationship to ensure the quality of a system.

While formal verification is the most effective means of ensuring consistency between a program and its specifications, it has limitations. These include the problem of validating that the specifications themselves specify what the customer wants; and the cost -- in both machine time and the time of skilled personnel -- of developing and checking the proof. The manufacture of software, like any manufacturing process, entails a tradeoff

between cost and level of quality assurance. In some programs there are modules for which any form of validation less powerful than formal proof would be socially irresponsible. Sometimes the same program also contains many modules for which formal proof would be a wasteful misallocation of resources.

Furthermore, there may be some modules that cannot be verified because they use features of the language for which there are no proof rules. Features may be excluded from the "verifiable subset" of Ada even if there are occasional legitimate uses for such features. Such legitimate uses can be isolated in modules that are validated by some means other than formal proof. In particular, low-level features of the Ada language are inherently machine dependent and thus not characterized by proof rules. Low-level features can be isolated in interface modules, allowing the rest of a system to be validated by formal proof.

Many factors combine to determine the most appropriate form of validation for a module. The cost of formal proof must be compared with the possible impact of an error in the module. Low-level, target-dependent interface modules might best be validated by informal proof. For certain hard-to-specify modules, for e ample a graphics display builder whose desired output is specified pictorially, testing might be not only the cheapest, but also the most reliable form of validation. For modules that are not particularly critical, and for which test drivers would be difficult to write, code walkthroughs might be most appropriate. Software might simply be trusted (until integration testing) if it has been extracted from a working system in which it has functioned reliably.

To ensure complete coverage, different forms of validation cannot be combined haphazardly. There must be a unifying discipline. One of the functions envisioned for MAVEN is to provide such a discipline.

2 Validation Libraries

The Ada language was designed to facilitate the construction of huge programs. A pervasive theme in the design of the language is the division of a program into units that can be understood individually yet checked for consistency with each other. If this theme is extended from unit compilation to unit validation, one unit of a program can be changed and revalidated without revalidating the rest of the program. This is especially important during program maintenance.

Module-by-module validation of a large program can be achieved in the same way as module-by-module compilation. Compilation of an Ada program unit consists not only of code generation, but also consistency checking. A unit's syntactic specification is compiled before either the unit's body or any external uses of the unit. This compilation puts information about the syntactic specification into a program library. Later, when either the unit's body or an external use of the unit is compiled, this information is

retrieved from the program library and used for compile-time consistency checks.

The consistency checks that occur during compilation are limited to the information found in a unit's syntactic specification, such as the number, types, and modes of subprogram parameters. Except for this limitation, however, they are analogous to the checks that occur during unit validation. Just as a unit has a syntactic specification that is checked during compilation, it has a semantic specification that is checked during validation. Just as syntactic specifications are recorded in a program library, semantic specifications are recorded in a MAVEN validation library.

Semantic specifications are textually embedded in syntactic specifications in the form of structured comments like those found in Anna [3]. This unifies the notions of syntactic and semantic specifications. When MAVEN is directed to compile a specification, it invokes the Ada compiler to place the syntactic specification in the program library. If no compile-time errors are found, the semantic specification is then extracted from the structured comments and added to the validation library.

2.1 Validation Order

To facilitate compile-time consistency checks, the Ada language restricts the order in which units may be compiled. MAVEN imposes analogous restrictions on the order of validation. Specifically, a module's semantic specification must be entered into the validation library before the implementation or any use of the module is validated. Then the implementation and each use of the module may be validated in any order. Validation of the implementation establishes that the body fulfills the semantic specification. Validation of a use of the module involves assuming, while validating the using module, that the semantic specification is correctly implemented. This assumption is permitted as soon as the semantic specification is entered into the validation library, even before the body has been demonstrated to fulfill the semantic specification. (This is analogous to the compilation of a subprogram call after the subprogram declaration has been compiled but before the subprogram body has been compiled.) It implies that validation of one unit can proceed considering only the specifications of the units it invokes, without considering their bodies. This is the essence of module-by-module validation.

Some program units may be validated by fiat. That is, after a code walkthrough or simply on the basis of trust, a unit may simply be decreed to be "validated." This still must be done explicitly, by a request to MAVEN, and the usual validation order rules must be obeyed. In particular, a unit may not be decreed to be validated before the specifications it is meant to fulfill have been entered into the program library.

2.2 Revalidation Order

Just as the Ada language restricts compilation order, it imposes recompilation requirements to ensure that consistency checks have always been performed on the latest version of a program. If a syntactic specification is recompiled, all consistency checks based on the old syntactic specification are rendered invalid. The corresponding body and all uses of the unit must then be recompiled so that the consistency checks may be repeated with respect to the new syntactic specification.

MAVEN imposes analogous revalidation requirements. If a module's semantic specification is changed, both the implementation and all uses of the module must be revalidated if they have already been validated. This is relevant during program development and program maintenance.

In program development, failure to validate a body may mean either that the body does not correctly implement the corresponding logical specification or that the logical specification itself is incomplete. In the first case, the body can be corrected and validated. In the second case, the logical specification must be modified and all other units using that logical specification must be revalidated. This may require still further modifications and revalidations.

In program maintenance, revalidation requirements indicate which parts of a large program are potentially affected by a change. This can reduce or eliminate the "ripple effect" typically resulting from a change to a working program. All possible implications of the change will be flushed out by the ensuing round of revalidations, assuming the revalidation is sufficiently thorough. (If the revalidation is by unit testing, this process amounts to regression testing. Rather than blindly repeating all tests, however, we use validation dependency relationships to identify the tests that might possibly have been affected by the change.)

A unit validated by fiat is subject to the same revalidation requirements as any other unit, even if revalidation consists only of reissuing the decree by which the unit was originally validated. This encourages software engineers to consider whether the original decree is still valid given the new specifications. For example, it may be discovered that an off-the-shelf package originally thought to be applicable to the current application is inappropriate given the revised specifications.

2.3 Other Information in the Validation Library

A validation library contains information besides the semantic specifications of program units. A validation plan can be entered into the library in advance, stipulating how a unit will be validated once it is written. The validation library also records which units have been validated, and according to which validation plans.

Each module may have its own validation plan. The plan specifies the validation method applied to the unit (testing or formal proof, for example) and the details of the validation criteria (which files contain the test driver or test data, algorithms for evaluating test results, or which properties are to be proven, for example). A validation plan may specify several rounds of validation, all of which must succeed for the unit to be considered validated. For example, a plan may call for testing to find and eliminate obvious errors, followed by formal proof to ensure the absence of more subtle errors. No one round of validation need provide complete coverage of the unit's semantic specification. Some parts of a unit's semantic specification may be proven valid, some validated by testing, and some simply assumed to be valid, for example.

Besides allowing MAVEN to enforce validation and revalidation order dependencies, the data kept in the validation library allows MAVEN tools to generate reports on the progress of system validation to date. The reports indicate which units have been validated and how rigorously. During development, validation of units can be tracked and compared with schedules. When an error arises, information about the validation methods applied to each unit and the properties validated for each unit can help pinpoint suspect modules. The revalidation implications of a proposed change can quickly be estimated.

3 Other Components of a Validation Environment

A verifier is only one of the tools that a validation environment should provide. We have already mentioned the need for a validation library. This implies the need for library management tools, including the report-generation tools discussed above. Other tools can assist in the writing of specifications, the retrieval of reusable software from a large catalogue, and the execution and analysis of tests.

Formal specifications are at the heart of MAVEN, but they are difficult for the typical software engineer to write. Therefore MAVEN must supply tools to help the software engineer express his intent. These tools are collectively called the specification-writer's assistant. One component of the specification-writer's assistant is a knowledge-based tool that will construct formal specifications based on a dialogue with the user. The specification-writer's assistant also includes an interpreter for a logic programming language, similar to PROLOG but providing the higher level of data abstraction found in the Ada language. This tool can be used for rapid prototyping, to test specifications as they are written.

The Ada language is meant to encourage the reuse of general-purpose software components. This approach can only have a significant impact on software development costs if there is a large corpus of general-purpose software available for reuse; but such a large corpus presents an awesome information-retrieval problem. While software retrieval is not usually thought of as a validation problem, Platek [4] has noted that formal

specifications and verification can form the basis of a retrieval tool. In addition to a validation library, MAVEN might include a catalogue of general-purpose, reusable software components, all of which have been formally specified. Given the semantic specification of a module required in the design, a MAVEN tool would search the catalogue for reusable components that can be proven to have compatible specifications.

Because testing is the most frequently used validation method, MAVEN contains tools specifically supporting testing. These include tools to generate subprogram stubs, tools to generate test drivers, tools to generate test data, and tools to analyze test results. All of these tools can base their outputs at least in part on the semantic specifications found in the validation library. For embedded applications, there should be software simulation tools and tools providing interfaces with hardware mockups. A related tool would administer tests automatically, based on the validation plans found in the validation library. Such a tool could also revalidate those units validated entirely by testing, whenever revalidation is required. In essence, this automates regression testing.

4 MAVEN and the Software Life Cycle

MAVEN tools are primarily concerned with unit validation. This can lead to the impression that the benefits of MAVEN are primarily reaped during the unit validation stage of the life cycle. In fact, the use of MAVEN imposes a discipline on software development and provides benefits throughout the software life cycle. This section walks through a typical waterfall model of the life cycle and describes the impact of MAVEN on each stage.

4.1 Requirements Analysis

The specification-writer's assistant supports the formal expression of requirements. Requirements can be entered into a new MAVEN validation library as the semantic specifications of the main program and of tasks declared in library packages. These formally stated requirements can be checked for consistency using a verifier. They may later become the basis for design verification and code verification. An integration-testing plan may be derived from the formal requirements and stored in the validation library until software integration time.

4.2 Design

During high-level design, the modular decomposition of a system is determined and the specifications of each module are written. Algorithms for top-level modules may also be written. MAVEN can play four roles at this stage -- design documentation, recording of unit validation plans, software-component retrieval, and design verification.

Design documentation consists of entering the semantic specifications for each design module into the validation library. The specification-writer's assistant again comes in handy here. The semantic specifications entered at this stage become the basis for later verification of module bodies. The appropriate time to formulate unit validation plans is just after unit semantic specifications have been identified. One of the responsibilities of an Ada designer is to look for existing software that can be incorporated in a design. As noted earlier, formal specifications might provide the basis for software automated software retrieval. The top-level algorithms of a high-level design can expressed in executable Ada code verifiable in the same way as lower level modules. Using only the specifications of the main system modules (the main program and tasks declared in library packages), it can be proven that the top-level algorithms correctly implement the system specifications.

4.3 Unit Development

There is not a clear dividing line between design validation and unit validation. The same techniques applied to the top-level modules during design validation are applied to lower-level modules during unit validation. The unit validation plan placed in the validation library during system design is retrieved and applied. A round of validation is repeated until it is successful, and then the next round specified in the validation plan is begun. The validation plan is restarted from the first round any time a change is made to the unit, its semantic specification, or the semantic specifications of the modules that the unit invokes.

Validation can uncover implicit assumptions that underlie the correct functioning of a module, especially when validation is by formal verification. Such assumptions must be added to a module's semantic specifications if the module is to be verified. Thus the validation process contributes to the development of complete and up-to-date specifications.

4.4 Integration Testing

The main impact of MAVEN on integration testing will be a drastic reduction in integration problems. The Ada compiler will already have checked all units for syntactic consistency with each other. MAVEN will already have checked all units for consistency with their own semantic specifications and the semantic specifications of the modules they invoke. The few integration problems that remain will arise from incomplete module specifications (for example, specifications that address functional requirements but not performance requirements) and insufficiently rigorous unit validation (for example, use of code walkthroughs as the sole means of validation or the use of tests that do not provide adequate coverage).

4.5 Maintenance

MAVEN will reduce the costs and risks of program maintenance. Both the data MAVEN collects during program development and the discipline MAVEN imposes on program modification will help confine the "ripple effect" of a change. MAVEN will also keep documentation up to date after changes have been made.

The most frequent problem associated with program maintenance is a change that violates an implicit assumption upon which a different part of the program depends. This problem is less likely to arise when using MAVEN for two reasons. First, the validation process applied during program development has served to make implicit assumptions explicit. The documentation will warn the maintenance programmer right from the start that certain changes must be disallowed unless further changes are made in other modules. Second, if the semantic specification of a module is changed, MAVEN will enforce the revalidation of all modules that may be affected by the change. The revalidation dependencies alone clarify the potential impact of a contemplated change. The actual revalidation, which may follow the original unit validation plan created during the initial design, leads the maintenance programmer to discover which potential impacts are truly significant, to revise the affected modules, and to validate the revisions. If the revised modules can themselves affect other modules, revalidation of these other modules will also be required. If sufficiently rigorous, revalidation anticipates and averts all possible ripple effects.

MAVEN keeps documentation current during program maintenance in the same way that it does so during initial development. Every time a unit's semantic specification changes, MAVEN records the fact. This makes the next round of maintenance easier.

5 Conclusions

We have described our vision of a Modular Ada Validation Environment, MAVEN, to propose a context in which formal verification can fit into the industrial development of Ada software. While proof of correctness is unquestionably the most rigorous and effective form of validation, there are contexts in which it is inappropriate. Nonetheless, formal proof can be effectively combined with other validation methods to raise confidence in a program's reliability.

MAVEN offers software engineers a continuum of more and less rigorous validation methods. This continuum makes a wider variety of validation methods available to a larger group and applicable to a greater number of modules. MAVEN provides a unifying framework in which different validation methods may be applied to the same program. By exposing software engineers to more rigorous methods than those they may be familiar with, MAVEN

encourages learning and promotes wider use of formal methods in the situations where they are appropriate.

MAVEN includes components that are at and beyond the state of the art. We do not propose that construction of MAVEN in its entirety should start today. Rather, MAVEN can serve as framework for the specification, design, and construction of individual tools, including a verifier. If such tools are viewed as eventual MAVEN components and if the MAVEN philosophy is kept in mind when the tools are specified, then MAVEN can be assembled over a number of years from independently developed components.

REFERENCES

---

1. Cohen, Norman H. MAVEN: The modular Ada verification environment. Proceedings, 3rd IDA Workshop on Ada Verification, Research Triangle Park, North Carolina, May 1986

2. Cohen, Norman H. The SofTech Ada Verification Project. AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, Long Beach, California, October 1985, 399-407

3. Luckham, David C., von Henke, Friedrich W., Krieg-Brueckner, Bernd, and Owe, Olaf. Anna, A Language for Annotating Ada Programs: Preliminary Reference Manual. Technical Report 84-261, Stanford Computer Systems Laboratory, July 1984

4. Platek, Richard. Formal specification. Proceedings of the First IDA Workshop on Formal Specification and Verification of Ada, Alexandria, Virginia, March 1985, paper C

Programming Support Environment Issues in the Byron Programming Environment

Matthew J. Larsen
Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

abstract: This paper discusses issues which programming support environments need to address in order to successfully support software engineering. These concerns are divided into two categories. The first category, issues of how software development is supported by an environment, includes support of the full life cycle, methodology flexibility and support of software reusability. The second category contains issues of how environments should operate, such as tool reusability and integration, user friendliness, networking and use of a central data base. This discussion is followed by an examination of Byron, an Ada based programming support environment developed at Intermetrics, focusing on the solutions Byron offers to these problems, including the support provided for software reusability and the test and maintenance phases of the life cycle. The use of Byron in project development is described briefly, and the paper concludes with some suggestions for future Byron tools and user written tools.

1. Introduction

Over the past two decades, producers and consumers alike of software products have become increasingly concerned with what has become known as the "software crisis". As computer hardware has evolved to enable the processing of more and more data at faster rates, the range of practically solvable problems has grown. Yet our ability to manage the growing capabilities of computer hardware, as Djikstra [1] has stated, has lagged. In order to combat the software crisis such weapons as design methodologies and software support tools have come into existence. Collections of these tools have become known as programming support environments, and there has been a gradual realization that such environments can be valuable. Ivie [2] identifies several benefits of such systems, including commonality of documentation, development of standards and enhanced programmer mobility and retrainability.

* Byron is a trademark of Intermetrics, Inc.

There is much disagreement concerning exactly which tasks a programming environment should support. The DoD has issued Stoneman [3], a document specifying the requirements an Ada* programming environment must meet, but Stoneman focuses primarily on how the tools are to work in general, not on the needs to be fulfilled by the tools. In this paper we shall first examine issues which programming environments, particularly Ada environments, must address. This will be followed by an examination of Byron, an Ada based programming environment developed at Intermetrics, and how Byron deals with these issues. We shall then examine how Byron might be applied to a project.

2. Programming Support Environment Issues

There are two sets of issues relating to programming environments. The first set focuses on how the environment supports software engineering. Included here are full life cycle support, support of software reusability and methodology flexibility. The second set is concerned with how the environment operates internally, including issues of environment integration, flexibility and user friendliness.

2.1 Software Engineering Issues

The purpose of a programming environment is to support software engineering. There are four concerns which must be addressed in order to do this effectively. First, the full software life cycle must be supported. Second, the user must be able to move freely from one life cycle phase to another. Third, the environment must not restrict the choice of methodologies available to the user, and finally, the environment must actively support the reuse of software.

2.1.1 Full Life Cycle Support

Frequently, the software life cycle is modeled as a discrete, linear process [4]. Initially requirements are drawn up, then a software system is specified, designed, implemented, tested and finally maintained. Each phase is treated separately, and is completed before the next phase begins. If revisions must be made, the process loops. For example, implementation might halt while the design is reworked, and then the implementation would be modified. The result of each phase is a document describing the results of that phase (in implementation this is the actual code). Note that these documents are often of vital importance to the following phases. For example, it is impossible to test a software system without knowing what it is required to do. Similarly, a design document may give a valuable overview of a system to the maintenence team.

* Ada is a trademark of the Department of Defense

In the past, automated support existed only for the implementation phase. Even now, research on programming environments is mostly directed toward the code-compile-debug cycle [5]. However, errors are cheaper and easier to fix if they are discovered earlier in the lifecycle. Also, if the computer is only usable for implementation, programmers will tend to concentrate on that phase. So the need for good tools which assist with earlier life cycle phases is paramount. As Gutz et al. [6] report, an environment must provide support throughout the life cycle.

2.1.2 Mobility Between Life Cycle Phases

Although the view of the life cycle as a discreet process is useful, it is not wholly adequate. Often an error is discovered which requires adjustments in an earlier phase. Because of deadline pressures, the corrections are usually made only in the current phase, which then bears some relation to the previous phases, but is not a direct descendant. Thus, the resulting implementation is based on an underlying design which evolved separately from the design document. The differences are likely to be subtle and difficult to understand, but are almost certainly important. If, however, there is a simple way to update the results of a previous phase (in this case the design document), the results of the phases are more likely to remain consistent with each other.

The essential problem, therefore, is to keep the documentation for the earlier phases consistent with the current phase. Naturally the previous phases are reflected in the current phase, although the information may be implicit rather than explicit. For instance, in Ada code some portions of the design are readily visible in the specifications of packages and the decisions concerning the grouping of subprograms into packages. Since the packages also contain information unimportant to the design, what is needed is a tool to distill the design out of the code. But in order to do this, the entire design must be explicitly stated, as must any other information we might want to use in creating reports. One way of providing easy mobility between life cycle phases is to introduce a programming language which permits explicit statement of information concerning all phases.

2.1.3 Methodology Flexibility

There are many different software engineering methodologies, and new ones appear with frequency. Even such basic concepts as the life cycle are called into question [7] and revised regularly. It has become clear [8] that environments must be flexible enough to permit a variety of methodologies and the evolution of new methodologies, since different problems require different methods of solution. In order to provide this flexibility, environments must permit the expression of many different kinds of information and also the categorization of this information in many different ways. The environment must then provide access to this information, as we will see below. The importance of this flexibility cannot be overstated.

2.1.4 Software Reusability

The software industry has recognized the need to avoid continuously rewriting various pieces of software. One of the major goals of Ada is the proliferation of large libraries of reusable packages, in order to address this need. However, there is a very real danger, even in small environments, that one programmer might not know what other programmers have already done. Even if code is known to exist, it may be difficult to determine whether the package actually does what is necessary (and whether it has side effects), or whether it can be easily modified. If the only way to identify the functionality and effects of a package is to read the code, much of the advantage of reusing the code may be lost.

The suitability of a given package for a given task is best evaluated by examining the design of the package, if that information is accurate. Therefore we see that the design information should be explicitly stated, and extractable from the code. Furthermore, this information must be in a concise and standard format, so users will be able to quickly sift through the available packages to find what they need. It is important for the environment to support the act of finding software which could be reused.

2.2 Environment Operation Issues

Although the support of software engineering is the primary goal of programming environments, issues concerning the operation of the environment are also important. If the tools are too clumsy to use, the environment will not be useful. Osterweil [9] identifies five characteristics essential to programming environments: breadth of scope and applicability, user friendliness, reusability of components, integration, and use of a central data base. The first of these includes the issues we identified above as methodology flexibility and life cycle coverage. The rest we shall consider below.

User friendliness is a broad term, including many fairly obvious points. User interfaces should be consistent; help should be on-line and easily accessible; tools should perform obvious functions and be free from contradictory and confusing options. A less obvious aspect of this issue is that tools should not overlap in function, which will tend to confuse the users in choosing which tool is best suited to a specific task. Also, a user who needs to perform a specific task should be able to find the tool which does that task without intimately knowing all the tools.

In order to provide a flexible methodology as discussed above, the component toolset must itself be flexible. This toolset can then be the basis for new tools tailored to fit the project. Bergland and Gordon [10] comment that "if the tools come first, too often the design and development methods end up accommodating the tools instead of vice versa." This implies, among other things, that a facility for combining tools must exist. The power of this approach is well known from experiences with the Unix* programming

environment. It is, however, not well understood which tools should comprise the toolset. Presumably, the next few years of research will begin to identify the essential tools.

It is also important that the tools be well integrated, that is, they should work together to provide an abstraction which assists the user in working within a particular development methodology, and shields the user from the details of the environment. Thus the interaction of the tools should be controlled in order to avoid hidden side effects, yet reuse operations where possible. Since we have already acknowledged the fact that the user is expected to augment the environment with additional tools, this goal can only be partially achieved. However, the set of reusable elementary tools should certainly abide by these rules.

The idea of a central data base which contains all the information relevant to a project is one of the most widely accepted concepts concerning programming environments [11]. We identified a need earlier for a language which can be used to express all the information concerning a project. It is even more important that all this information be stored in one place. This can then be used to maintain various versions of a project, structure and retrieve information in manageable pieces and most importantly, maintain a single set of documents which describe the state of the project at any given moment.

3. The Byron Programming Support Environment

We will now review the Byron Programming Support Environment, and examine how it addresses the issues identified in the previous section. The three important aspects of the environment are how the data enters the environment, how it is stored, and what tools are available to access the stored information. The primary means of expressing information to be entered into the Byron environment is the Byron program development language (PDL). The PDL text is analyzed and stored in a structured data base (the program library). Once stored, the information is available to the various tools which comprise the Byron programming support environment.

3.1 The Byron PDL

The Byron programming support environment is centered around an Ada-based program development language (Byron/Ada PDL). Byron is compatible to Ada since any legal Ada program is also legal Byron, and vice versa. Byron provides a consistent way of entering information into the environment throughout the software life cycle, and thus smooths the transition from one phase to another.

* Unix is a trademark of Bell Laboratories

Byron constructs are included with the Ada code in the form of annotated comments. (see [12] and [13] for more detail). The Byron PDL is designed to augment Ada's design language abilities by formally and efficiently expressing information produced in the course of engineering a large software system which cannot be expressed in Ada. Ada has many features which assist and improve design; however, it has been recognized that there is information which is not required by or even expressible in modern programming languages, including Ada, but which is nevertheless important and valuable [14], [15], [16]. This information is mostly semantic in nature, concerning the use or purpose of data items or subprograms. Consider the following Ada subprogram specification

function CopyLinkedList (List : in ListPtr) returns ListPtr;

This is sufficient for compilation; however, in order to use the function there are details one needs to know, such as whether a physical copy of each list element is made, or merely a copy of the pointer to the list. Byron permits the methodical inclusion and retrieval of such information.

Information is expressed in Byron either as Ada code or as Byron annotations. Annotations are formed with the prefix "--|" followed by text. In general, the text of an annotation is associated with the Ada construct that precedes the annotation. An annotation may also contain a keyword which categorizes the information. This permits the user to tailor the Byron PDL to suit many different tasks. For example, the effect of the CopyLinkedList subprogram might be described with the effects keyword, e.g.

```
--|Effects: Creates an exact copy of the list passed in. A copy
--|of each element is made, so the copied list shares no elements
--|with the original.
```

The user may specify what keywords may be used and what Ada context they are to be expected in. This permits the user to define a specific development methodology, giving the user the methodology flexibility discussed above. For instance, a methodology might require that every use clause that is placed in code be followed by a Byron annotation justifying the presence of the use clause. A Byron keyword "Use_Justification" could be used to enforce this requirement.

One problem with methodologies is that it is sometimes difficult to get programmers to adhere to them. Byron attempts to alleviate this problem through the mechanism of "phase checking." The user specifies what development phase a given keyword should be used at (keywords may also be optional). Program source is then categorized within the program library according to what phase has been reached based on what keyword annotations are present. The analyzer will warn a user who indicates that code has reached a phase which it has not; tools may also be written to report what phase any portion of code is currently in. Thus, the "Use Justification" keyword described above could be required at implementation phase. Warning messages would

indicate the absence of Use_Justification annotations when code was analyzed with a phase of implementation.

3.2 The Byron Data Base

The Byron system provides a database called the Ada program library, which provides a central repository for all the information concerning a project. This information is primarily stored in the intermediate form known as DIANA, including both the regular Ada syntactic and semantic information, and the Byron PDL information. Tools may be written which access either kind of information, and may be independent of life cycle phase or not. The PDL code enters the program library through the Byron analyzer. This program is the front end of an Ada compiler, and provides full syntactic and semantic checking, as well as the checking specified for Byron annotations.

The program library permits Ada programs to be broken down into any number of separate catalogs containing compilation units, which are linked together to form the program library which comprises a program. Catalogs may be either read-only resource catalogs, which contain a specific release of a set of compilation units, or modifiable primary catalogs which generaly represent a new revision under construction. Configuration management is assisted by the use of different revisions of a resource catalog. Thus, two projects might be using different revisions of the same resource catalog, so that the project using the older revision could avoid recompilations or regressions in the newer revision.

3.3 The Byron Tools

Tools are an essential part of a programming support environment but it is the selection of tools and the relationship between them that characterizes the working details of a truly integrated system. As we saw in section two, there are many factors to be considered when examining a programming environment. The Byron tools have been designed with an eye toward flexibility and smooth dissemination of the information concerning systems under development.

As we noted before, a programming support environment must include tools to support each phase of the software life cycle, and must smooth the transition between phases. The Byron tools fall into two broad categories: first, tools which assist with methodology and the life cycle phases, and second, tools which assist with programming tasks without regard to a specific discipline or life cycle phase. Methodology and life cycle tools include an Ada based PDL, described earlier, configuration manager for source and documentation, design requirements traceability package, data dictionary system and more. Of the second type of tool, Byron provides a variety of technical programming tools for static analysis. These include an Ada compiler, linker, recompilation manager, global cross-referencer, source formater, program lister and others. Another way of categorizing Byron tools is by the form of data they operate on. Many of the tools, including the global cross-referencer, the data dictionary

generator and the generalized document generator, operate on the data available in the program library. Other tools, such as the pretty printer, statement profiler and the compile order generator, operate directly on Ada source code.

In an earlier section of this paper, we pointed out that an environment must permit user constructed tools. We have seen that the Byron PDL permits the user to store arbitrary information in the program library. Tools are also provided which the user may use to extract that information from the program library, as well as Ada syntactic and semantic information. The first of these tools is a generalized document generator, which creates documents based on user written specifications. These specifications are written using an interpreted language, BDOC, which permits the extraction of information from the program library and the output of that information in a formated form. The second tool is the program library access package (PLAP), a set of Ada subprograms which provide a window into the library. The user can extract information about his program, as it evolves, without being concerned with the internal structure of the library. Ada programs can be written which utilize the PLAP to query the program library and output individually tailored reports. Using this package, it is possible to construct complex tools such as a hierarchy chart drawer or a program interconnectivity matrix. These two tools provide the elementary tools spoken of in 2.2.

Also pinpointed earlier was the need to support reusability. It is not unusual for a programmer to duplicate the work of an associate simply because no one knows that the work has been done before. This problem is especially pronounced in a distributed computing environment. Even if a piece of code is available which does a similar task, it may be nearly as difficult to modify as to write from scratch, since the programmer must first understand how the existing code works. The userman tool provided with Byron can assist in relieving this problem. The document created by userman is a description of the purpose and use of package or subprogram, and is intended to be a document of the design of a package or subprogram following the design methodology suggested by Liskov [14]. Other documents supporting other design methodologies could be produced. One can then envision a design library stored on a computer to which programmers could refer when looking for a package to do a specific job. Another way to encourage reusability would be to create a user defined keyword "keywords." This keyword would be permitted on all library units, and the text following it would be a list of keywords describing the functionality the unit provides. A simple program could be written using the Plap which would extract the keyword list from each unit in the program library. Each element in the list would be compared to a string which the user of the program would supply, and if they matched, an overview of the unit would be printed. This would assist users in sifting through large librarys of software to find appropriate tools.

Documents of this nature also help support the software life cycle, by helping to show the design as it currently exists, rather than as it is intended to exist or as it used to exist. Byron also offers a tool to support the tracking of requirements, to ensure that the final result of the project does in fact fulfill the needs it was intended to. The userman and requirements tracking tools help Byron to support transitions from one phase to another, as does the fact that many of the tools are useful in multiple phases.

4. Project Use of the Byron Programming Environment

A comprehensive development system, such as Byron, is difficult to visualize at work. An operational view is necessary to appreciate the ability of such a large number of tools to function together usefully. The following scenario is presented as a brief illustration of how a hypothetical project might evolve using this system.

First of all, Byron provides methods and tools to assist management with organization, planning, tracking and reviewing of this project throughout its entire life cycle. Since the user is permitted to decide what information is to be stored in Byron annotations, valuable project information such as names of implementors and/or designers, project progress information, project statistics may be easily stored and accessed. Tools for computing the Halstead and McCabe complexity metrics are included, which assist software management in several ways, including estimating the number of outstanding bugs and the time needed to complete pieces of software.

The requirements phase of this project defines the problem to be solved, defines a system design to solve the problem, and allocates the requirements of that design to hardware and software. The design requirements traceability tool provides the facility to relate requirements to design elements and modules. This is especially useful in later phases where the impact of change may be quickly traced.

During specification and design work shifts from functionality to decomposition. Program architecture and data structures may be developed using Byron PDL. The result of this process will be a set of heavily annotated Ada specifications, from which the document generator can create design documents which may be reviewed. The template driven document generator can produce documents as complicated as MIL standard C5 specifications, as well as other specialized documents a user might need. Sufficient flexibility is available to support many different design methodologies, by the careful selection of Byron annotations.

The transition from design to implementation is a smooth one since an Ada based PDL was used as the program design language. It is a small step from Ada specifications to subprogram bodies which contain nothing more than a few lines of comments describing what that routine is to do, in fact, a tool could be written using the program library access package to automaticly generate simple

bodies on the basis of annotated specifications. As implementation begins in earnest, the user can take full advantage of the PDL aspects of Ada, and where Ada is deemed inappropriate, Byron annotations may be defined to fill the gap. A user defined Byron annotation "TBD" or "to-be-determined" might be used as a general purpose annotation to mark these comments, and permit their extraction and inclusion in documents. Two such documents might be a report on which modules are not yet fully implemented and what work needs to be done on them, or perhaps a design document of a more detailed nature than that produced by the userman tool. Coding and debugging are assisted by frequent reports such as cross-references and compilation listings. When implementation is complete enough, the Ada compiler will generate object code (also stored in the program library) which may be linked for testing. Implementation and testing are further assisted by a symbolic debugger and performance analysis tools.

Once the system begins to work, it must be carefully tested. Software which has not been adequately tested cannot be considered reliable. The Byron tools assist testing in several ways. First, the design requirements tracer shows which modules implement which requirements, helping to focus testing efforts. Second, when a module is designed it's purpose is well understood, and the test which should be applied are often more obvious than after implementation. The functionality which a module is intended to provide should be tested, not a specific implementation. Byron annotations provide a means for expressing this information at whatever point in the life cycle it can best be specified.

Software systems spend the majority of their life cycle in maintenance phase. The cost in terms of both time and money of repairs and enhancements can be greatly reduced by the availability of accurate documents which describe a system at various levels, from requirements down to detailed design. If an engineer must read code to understand a system, it may be a considerable amount of time before changes can be made to the system. Comments, when they exist, tend to be vague and incomplete. Byron provides a mechanism for specifying what structures should be commented and what type of information the comments should include. One major purpose of Byron is to provide a series of documents which describe the system, providing insight at several levels of complexity.

5. Conclusion

We have identified a number of concerns which programming environments must address, and seen how Byron addresses them. Of these issues, perhaps the most important are the full coverage of the life cycle, including transitions between phases, and flexibility, both with respect to the methodology supported and the tools available. It is expected that successful environments will support a wide coverage of the life cycle and permit great flexibility.

Byron provides a great deal of flexibility, both in what information is to be stored and what tools can access that information. Although there are limits to

how this information can be expressed, such as the Ada framework surrounding the information, these limitations serve to focus the user's attention on the purpose of the environment: to assist the creation of Ada programs. Thus, the environment supports primarily the act of producing programs, not the act of using that product. This is accomplished by helping the user organize the information which would otherwise be in some possibly out of date design document, or as a series of random comments, or perhaps not at all. The user may then use this organization to extract only as much of the information as is necessary for a specific tool to do its work, or to answer specific questions concerning the software system.

Acknowledgements

This paper was reviewed and commented on by Michael Gordon, David Ortmeyer and Haynes Turkle. Their suggestions and support are greatly appreciated.

References

[1] Dijkstra, E.W. "The Humble Programmer" (Turing Award Lecture). *Communications of the ACM*. vol. 15, No. 10 (October 1972) : 859-866.

[2] Ivie, E.L. "The Programmer's Workbench - A Machine for Software Development." *Communications of the ACM*. Vol. 20 No. 10 (October, 1977) : 746-753.

[3] Department of Defence. *Requirements for Ada Programming Support Environments, 'STONEMAN'*. (Washington : US Department of Defence, 1980).

[4] Yourdon, E. and Constantien, L.L. *Structured Design*. (New York : Yourdon, Inc., 1978) : 3-9.

[5] Henderson, P., ed. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. New York: The Association for Computing Machinery, Inc., 1984.

[6] Gutz, S., Wasserman, A.I., and Spier, M.J. "Personal Development Systems for the Professional Programmer." *IEEE Computer*. Vol. 14 No. 4, (April 1981) : 45-53.

[7] McCracken, D.D., and Jackson, M.A. "Life-Cycle Concept Considered Harmful." *Software Engineering Notes*. Vol. 7 No. 2, (April 1982) : 29-32.

[8] Hoffnagle, G.F., and Beregi, W.E. "Automating the Software Development

Process." *IBM Systems Journal.* (vol. 24, No. 2, 1985) : 102-120.

[9] Osterweil, L. "Software Development Environment Research: Directions for the Next Five Years." *IEEE Computer.* Vol. 14 No. 4, (April 1981) : 35-43.

[10] Bergland, G.D. and Gordon, R.D. "Software Development Environments." *Tutorial - Software Design Stratagies.* 2nd ed. New York : IEEE, 1981 : 347-353.

[11] Wasserman, A.I. "Automated Development Environments." *IEEE Computer.* Vol. 14 No. 4, (April 1981) : 7-10.

[12] Larsen, M.J., Ortmeyer, D.O., Turkle, H., and Gordon, M. "The Byron1100 Program Support Environment." *Proceedings of Use, Inc. Fall Conference, vol. 1.* Anaheim, CA, (November, 1985) : 119-134.

[13] *Byron Program Development Language and Document Generator.* Cambridge : Intermetrics Inc., 1985.

[14] Liskov, B. *Modular Program Construction Using Abstartions.* MIT Computation Structures Group Memo 184. September 1979.

[15] von Henke, F.W., Luckham, D., Krieg-Brueckner, B., and Owe, O. "Semantic Specification of Ada Packages." *Ada in Use: Proceedings of the Ada International Conference.* Cambridge: Cambridge University Press, 1985 : 185-196.

[16] Luckham, D.C., von Henke, F.W., Krieg-Brueckner, B., Owe, O. *Anna: A Language for Annotating Ada Programs.* Computer Systems Laboratory Technical Report 84-261. Stanford University. July, 1984.

An Ada Programming Support Environment

Al Tyrrill
A. David Chan

North American Aircraft Operations
Rockwell International
Lakewood, California

ABSTRACT

This paper describes the toolset of an Ada Programming Support Environment (APSE) being developed at North American Aircraft Operations (NAAO) of Rockwell International. The APSE is resident on three different hosts and must support development for the hosts and for embedded targets. Tools and developed software must be freely portable between the hosts.

The toolset includes the usual editors, compilers, linkers, debuggers, configuration managers and documentation tools. Generally, these are being supplied by the host computer vendors. Other tools, for example, pretty printer, cross referencer, compilation order tool and management tools have been obtained from public-domain sources, are implemented in Ada and are being ported to our hosts.

Several tools being implemented in-house are of interest, these include an Ada Design Language processor based on compilable Ada. A Standalone Test Environment Generator facilitates test tool construction and partially automates unit level testing. A Code Auditor/Static Analyser permits Ada programs to be evaluated against measures of quality. An Ada Comment Box Generator partially automates generation of header comment boxes.

1 INTRODUCTION

Rockwell International North American Aircraft Operations (NAAO) is constructing a facility for the development of Ada software. The facility will support an avionics integration laboratory where both simulation and embedded avionics software are to be developed. Ada software development will occur on three different hosts.

1. A supermini widely used in the aerospace and scientific communities.

2. Another supermini noted for high "number crunching" horsepower. This processor model will support the simulations and simulation development.

3. A processor designed specifically for Ada software development, on which all system software has been implemented in Ada.

Each of the development hosts will interface to a user maintenance console that supports several of the embedded avionics processors. The maintenance console can pass data between the target processor memories and the hosts and control execution of the targets.

The avionics processors are connected to each other, various actual aircraft hardware and the simulation host by means of several high speed data busses. Software in the avionics processors can be tested with actual hardware online or with hardware simulated by models in the simulation host.

The hosts are to be networked with an Ethernet line so that software, associated products and development tools can be easily transported.

Rockwell is constructing an Ada Programming Support Environment (APSE) for the development facility. The APSE consists of a set of tools whose objective is to support the production of a well-organized, structured and maintainable software product, in a cost effective manner. The APSE itself must be constructed in a cost effective manner.

The cost requirement on the APSE dictates that available tools be used as much as possible. This reduces the potential level of tool integration, as tools implemented in isolation from each other generally will not share common interfaces. The interface that is shared by most of the tools is the Ada language, however, and its rigid standardization makes assembly of a toolset from disparate sources feasible.

2 APSE COMPONENTS

This section summarizes the components of the NAAO APSE and indicates the sources from which the tools will be obtained. Section 3.0, Locally Developed APSE Components, describes in more detail some of the components that are to be implemented at NAAO.

2.1 Development Tools

These tools support the design and coding phases of the software development process. They are an Ada Design Language, text and program editors, compilers and assemblers, a library of primitives and common packages, and link editors.

2.1.1 Ada Design Language

The objective of the NAAO Ada Design Language (DL) is to provide a means of expression for both control flow and data structure and relationships. The Ada language itself provides an excellent means for expressing data structure, but some other means of describing control flow is necessary prior to actually committing a design to Ada code.

Accordingly, the Ada DL uses compilable Ada to represent data structure and a traditional Program Design Language (PDL) to represent control flow. The PDL statements are embedded as comments within the Ada specifications so that the entire Ada DL description is compilable. Several tools are available to support construction of Ada DL designs. These include a "TBD" package, the Ada DL preprocessor, the processor for the traditional PDL and an Ada body part generator.

The Ada DL is described, along with its use in object oriented design, in more detail in section 3.1, Ada Design Language.

2.1.2 Editing Tools

Several tools support the editing of Ada DL, Ada code and documentation files.

2.1.2.1 Text Editors - Text editors are provided for editing of documentation and other non-Ada files. These were obtained with the system software on each of the hosts.

2.1.2.2 Ada Syntax Sensitive Editors - A syntax sensitive Editor is one that contains the syntax equations of the target language in its database. Templates are expanded to their syntactic substructure. The means exists to traverse between templates and delete templates for optional constructs.

Two of the three hosts have Ada syntax sensitive editors available from the system vendor. In one of these, initial entry of a file begins with the template [compilation], which by repeated expansion and replacement of templates with text, is converted to the desired code. The templates have the same names as the syntax equations from the Ada LRM. When adding to an existing file, it is necessary to enter the starting template e.g. [later_declarative_item], [statement] manually (and one must know what they are called).

On the Ada based host, a construct is prompted by entering an initial keyword, e.g. "procedure", "if", and requesting the editor format the file. It identifies the construct and expands it into its components.

2.1.2.3 Source Formatter (Pretty Printer) - The source formatter reformats existing Ada source into a consistent form. The level of statement indentation is made proportional to the nesting depth. Spaces and line breaks are added to improve readability. Declarations and line comments are aligned where appropriate. The source formatter was obtained from a public domain source, is written in Ada and will be modified to improve functionality. On the Ada based host, the source formatter is integral with the editor.

2.1.3 Compilers and Assemblers

Program development will occur in different environments in the development facility. Native mode code will be generated for initial program testing and for tool implementation. Code will be generated for the simulation host on that host. Ada written for the simulation host must interface with existing FORTRAN code. Ada code will be written for the embedded processors. This code must interface with existing JOVIAL code.

2.1.3.1 Ada Native Mode Compilers - Each of the development hosts has a validated Ada compiler available from the system vendor. Each has an associated library manager for creating and maintaining Ada program libraries.

2.1.3.2 Ada Embedded Processor Cross Compiler - Cross compilers for the embedded processor are available or will be available for all our development hosts, although none have been validated. For two of the hosts, the system vendor will be supplying the cross compiler. for the other, one of several possible third party vendors will be selected.

The different vendors products are currently being evaluated. The selected product will hold a validation certificate or otherwise have been demonstrated to correctly compile those features required by the avionics software.

2.1.3.3 JOVIAL Avionics Processor Cross Compiler - This compiler will translate JOVIAL to the object code of the avionics processor. The object file format will be compatible with that generated by the Ada compilers for the avionics processors. It will be possible for JOVIAL to call Ada and vice versa without the use of interface routines when the parameter types have analogues in both languages. The JOVIAL cross compiler will be obtained from the Ada cross compiler vendor.

2.1.3.4 Avionics Processor Cross Assembler - These assemblers will run on the hosts and generate avionics processor object code. The object file formats will be compatible with that generated by the Ada compilers for the avionics processors. The Ada cross compiler vendors each have compatble cross assemblers available.

2.1.3.5 Simulation Host FORTRAN Compiler - The native mode FORTRAN compiler on the simulation host will generate object files compatible with those of that host's Ada compiler. Such a compiler is available from the system vendor.

2.1.4 Library of Primitives and Common Packages

The library of primitives and common packages will be a collection of commonly used functions in the areas of navigation, weapons delivery and math functions. Initially, a set of primitives will be identified for inclusion in the library and implemented when they are first needed. Additional primitives will be developed as the need for them is identified.

Some type of "browser" utility that will enable the potential user to intelligently search the library is being planned.

2.1.5 Link Editors

The linkers in the APSE shall have the means to determine that all modules dependent on a module that has been recompiled have also been recompiled, or that otherwise the full set of object modules involved in the link edit is in a consistent state.

2.1.5.1 Host Link Editors - These linkers will link object files produced by the hosts' native mode Ada compilers to produce an image executable on the host. Each host system vendor has augmented its link editor to provide the

required consistency checking.

2.1.5.2 Avionics Processor Cross Linker - These linkers will generate executable avionics processor images from object files produced by the Ada cross compiler, the JOVIAL cross compiler and the avionics processor cross assembler. The avionics processor images can be executed interpretively by simulators on the host or downloaded to an avionics processor.

2.2 Testing Tools

The Ada environments on the hosts will be integrated with the symbolic debuggers provided with the hosts' operating systems. Symbolic debuggers will be procured for the avionics processors which will support standard debugging operations without incurring additional overhead in the target. A tool will exist to create an environment in which to test Ada compilation units in a standalone mode.

A data bus monitor will support the capture and display of selected bus data and the simulation of bus transmissions to facilitate integration testing.

The development host will have the simulation and support tools necessary to execute the avionics software in an integrated mode with the actual or simulated aircraft hardware or in a software environment only. This includes a host simulator designed to execute flight software in native code supported by environment programs and I/O simulated in software. The hosts will have target processor simulation including input/output and interrupt simulation.

2.2.1 Host Symbolic Debuggers

These tools, used for debugging native mode programs on the hosts, supports examination and deposit, setting of breakpoints and watchpoints, stepwise execution and trace, all referenced to Ada source statements or declarations. The debuggers are part of the host system vendors' software support packages, but each has been modified to support tasking and other unique features of the Ada language.

2.2.2 Avionics Processor Interface and Debugger

These tools supports downline load of executable images to the avionics processors, execution control of the avionics processors and transmittal of status information back to the hosts. Symbolic debugging is supported from the hosts. Symbol table information is maintained in the hosts and not downloaded to the avionics processors. Target debugger support is provided by all the Ada cross compiler vendors, but additional interfacing to support NAAO's particular test environment will be required.

2.2.3 Standalone Test Environment Generator

This tool determines the inputs, outputs and external entry points of a set of Ada programs under test. The tool prompts the user for inputs, executes one of the specified programs and displays the outputs. Pre-canned functions can be specified for the inputs and the program executed repeatedly with variation of an independent variable, such as time. Outputs can be plotted against inputs

or the independent variable.

The Standalone Test Environment Generator will be implemented in-house at NAAO, and is described in more detail in a subsequent section.

2.2.4 Data Bus Monitor

The bus monitor will interface with the various data busses in the avionics integration laboratory and perform the following functions. The bus monitor will be implemented by augmenting existing capabilities.

Generate real time displays of selected bus data.
Generate profiles of bus data by message type and subtype.
Generate simulated bus data for test stimulation.

2.2.5 Host Avionics Processor Simulator

Simulators for the avionics processors will be available to support the testing of avionics processor images that would otherwise require the actual hardware. A conventional simulator will interpret executable images down to the instruction field level. A faster simulator in which the Ada code is compiled into procedure calls on the host that duplicate the computations of the avionics processors without actually interpreting at the bit level is also being acquired. Both of these are available from the Ada cross compiler vendors.

A simulator is being implemented in Ada inhouse that will be capable of concurrently simulating several avionics processors, with interprocessor communications implemented as transfers through common memory buffers.

2.2.6 Documentation Support Tools

The documentation generators will construct data dictionaries from sets of Ada programs. They will construct trees of calls and context references (WITH's). A header comment box generator will summarize that information in the program headers that can be extracted automatically from the program source. These processors will accept a list of files, or scan a link editor command file and process the sources for all the input modules for the linking of the executable image. A report formatter/word processor will be available for general document preparation.

2.2.6.1 Ada Data Dictionary Generator - This tool scans a set of Ada program source files and records the full context of declarations, recognizing Ada scope and visibility rules. It generates a data dictionary with locations of declarations, set references and use references in a format compatible with required documentation. This will be implemented by augmenting public domain software, implemented in Ada.

2.2.6.2 Ada Called-by/Withed-by Generator - This utility does a scan of a set of Ada source programs. For subprograms it constructs trees of calls and called-by references. For packages, it constructs trees of context clause (WITH statement) references. The generated reports are in a format compatible with required documentation. This tool will also be obtained by augmenting an

existing public domain program.

2.2.6.3 Header Comment Box Generator - This tool scans the source of an Ada compilation unit for that information which is required to be in the header comment box, such as inputs and outputs, subprograms called and called-by, imported data structures and routines, and other resources used. It creates a new header comment box or updates the existing one. This tool is being implemented in-house in Ada and is described in more detail in a subsequent section.

2.2.6.4 Report Formatter - These utilities process a file of text with interspersed formatting commands. They perform word processing functions such as right margin alignment, indentation, assignment of heading numbers, table of contents generation and others. Several of these are already installed in our facility, from various sources. They are the most widely used support tools in the laboratory.

2.2.7 Configuration Management Support

These tools support the adherence to software standards and the controlled maintenance of source and documentation files.

2.2.7.1 Code Auditor/Static Analyser - The code auditor scans the source for an Ada compilation unit and generates a report of areas of non-conformance to software standards, as specified in the Ada Style Guide that was developed jointly by several Rockwell divisions.

2.2.7.2 Configuration Control System - The configuration control system will create and maintain libraries of controlled files, which can be Ada DL source, program source, documentation or any other textual material. It will track changes by associating them with retrieval and replacement of library elements. It will monitor access and be able to generate a historical record of the accesses to each element in a library. Each host has such software available from the system vendor.

3 LOCALLY DEVELOPED APSE COMPONENTS

The following sections describe in more detail some of the tools that are being implemented in-house at NAAO. Of particular interest are the following.

1. Ada Design Language processor, that will permit embedding a traditional PDL within compilable Ada specifications.

2. Standalone Test Environment Generator, that will determine the inputs and outputs of a program under test, then generate input values, execute the program and capture and display the outputs.

3. Code Auditor/Static Analyser, which will permit Ada programs to be checked for conformity with software standards, and be evaluated against various measures of quality.

4. Ada Header Comment Box Generator, which will automate collection of some of the information required to be in the header comment box of program units.

3.1 Ada Design Language

Traditional PDLs, like those widely used in the computing community over the past decade are good at describing control flow, but poor at describing structure, hierarchy, data relationships and interfaces.

Ada specifications are good at describing these things, but do not describe control flow. Use of compilable Ada to describe control flow is awkward, at best, because it does not permit specification of detail to be deferred.

The idea of using compilable Ada as a design language is gaining acceptance because it specifies at design time what the software product will look like. I.e. the Ada specs are a form of "contract" for the software that is to be implemented.

Traditional PDLs are coming to be regarded as a decade old technology that is little more than an improvement on flowcharts.

The NAAO Ada Design Language combines compilable Ada with Reconfigurable Design Language (RDL), a traditional PDL with an Ada-like syntax, to obtain the benefits of each. RDL was implemented at another Rockwell division in Ada and can be installed on any host with a validated Ada compiler. Aside from the syntax change to make it more Ada-like, it is similar in appearance and capabilities to a commercially available PDL widely used in the computing community for over a decade.

3.1.1 Use of the Ada DL

Use of this design language consists of the following steps.

1. Description of the structure, operation and interfaces of a design using Ada specifications.

2. Construction of the Ada bodies, starting with the specifications, then with further development.

3. Description of the control flow within units, using RDL statements in specially marked comments.

3.1.1.1 Development of Ada Specifications - The design language user first identifies the objects to be implemented. These suggest the top level package structure of the design. Then, the actions to be performed on these objects are identified, these suggest the procedures and functions these packages will support.

Externally visible data structures are identified, then Ada types and objects are defined to represent these. Parallel event streams suggest creation of tasks to support them. Textual comments are added to further explain the purpose of the constructs so defined.

3.1.1.2 Development of Ada Bodies - Then, using an Ada body part builder, body parts for the specifications are created. Data structures not to be visible externally are defined within the bodies of the packages and subprograms. Use of the available "TBD" package permits the user to defer assigning specific types to Ada objects.

3.1.1.3 Development of RDL Descriptions - The control flow within the subprogram bodies is now designed and specified with RDL procedures. The RDL statements are enclosed in specially marked Ada comments to keep the entire design description compilable. Reference to data defined in the pure Ada part can be made by the RDL. Use of RDL permits the existing RDL processor to be used to generate data dictionaries and calling trees.

Large designs may require several iterations of this process before the design is complete. The completed design consists of Ada specs with embedded textual comments and Ada bodies with embedded RDL procedures and comments.

3.1.2 Ada Design Language Utilities

Several tools and utilities are available to assist in the generation of Ada DL descriptions.

3.1.2.1 TBD Package - This TBD package, which is public domain software, provides types, objects, functions and a procedure which can be referenced in a design when the actual type of the object or subprogram parameter is not known. TBD values for the quantities in package SYSTEM, such as maximum integer, smallest fixed point delta, etc. are also defined. As a design is evolved, the TBD quantities are replaced with the actual objects. All names in the package contain the substring "TBD" so they can be located with an editor search.

3.1.2.2 Body Part Generator - This tool generates a body part from an Ada specification. It is available as a primitive on the Ada based development host, and also from a public domain source for any processor with a validated Ada compiler.

3.1.2.3 Ada DL Preprocessor - This utility, which will be implemented in-house, scans the Ada Design Language descriptions and records all the type, object, subprogram and task specifications. It extracts the RDL procedures from the Ada bodies and generates RDL declarations for the objects declared in the Ada and referenced in the RDL. It formats the RDL into a form acceptable to the RDL processor and submits it for generation of an RDL report.

3.1.2.4 RDL Processor - The RDL processor, currently installed on two of our hosts, generates a formatted report from an RDL description. It also produces a data dictionary and calling trees for the segments (subprograms).

3.2 Standalone Test Environment Generator

Traditionally, unit level testing is done by implementing special purpose data generators and data monitors, linking everything together, running the program under test, then analysing the data. The next routine requires new data generators and monitors.

The Standalone Test Environment Generator (STEG) being developed at NAAO will act as data generator and monitor for a large class of subprograms and will partially automate the unit test process.

A unit to be tested includes a subprogram and its dependent units. They are first compiled cleanly.

The STEG will scan the unit under test, identify the calling parameters, then determine those objects declared at a higher scope that are used but not set (inputs) and set but not used (outputs). It will detect those that are both set and used, as these could be inputs, outputs, both or neither.

The STEG will prompt the tester for the names of inputs and outputs not identified in the scan. It will then generate an Ada shell that supplies the program's inputs and captures its outputs. This will be compiled and linked with the program under test. Stubs will be provided for subprogram that are not provided. An OUT parameter from a stub is regarded as an input.

The STEG will then prompt for the values of the identified inputs and pass them to the target program. It will execute the program under test, then display the values for the identified outputs. Exceptions returned from the target program will be identified. Facility to generate an exception from a stub will also be supported.

A command language will be provided for repeatedly executing the program under test while varying the values for selected inputs. The command language will be a subset of Ada. Plotting and data reduction features are to be provided.

3.3 Code Auditor/Static Analyser

The purpose of this tool is to support the enforcement of software standards and good programming practices. It will gather statistics that may be indicative of the use or non-use of these standards and practices and prepare a report that might serve as the starting point for a code review or structured walkthrough. The code auditor will gather the following types of statistics.

1. The amount of commentary relative to the amount of code will be determined. Textual comments will be distinguished from delimiting comments (blank lines and lines of dashes, etc.). Of course, it will be unable to distinguish a useful comment from something like "-- Mary had a little lamb".

2. Measures of program complexity will be developed, such as number of nodes in a program's directed graph, then statistics will be developed from our experience with implementing and maintaining these programs relative to their measured complexity.

   It is generally regarded that overly complex program units cause maintenance problems. However, simpler programs mean more program units are necessary, and this complicates the integration process.

The number of subtypes and derived types defined and their frequency of reference versus the frequency of reference of the predefined types. Use of subtypes and derived types makes better use of the strong typing features of Ada.

4. For subprograms, the number of parameters passed versus the direct references to objects declared at a higher scope (global variables). Use of global references is regarded by some as producing harder to read code.

5. Statistics on identifier length will be gathered. Average length, distribution of lengths and frequency of reference of various lengths will be recorded. These statistics will be gathered for various classes of identifiers, e.g. scalars, record components, FOR loop indeces, subprogram formal parameters, etc. Use of overly short identifier names is regarded as a poor practice, but it is not clear that longer is always better.

6. Use of PRAGMAs, particulary PRAGMA SUPPRESS, will be recorded and summarized.

7. Placement of more than one type or object declaration on a line, or more than one executable statement on a line will be flagged. Code so written is likely to be harder to read.

8. Types and objects declared but not referenced, objects declared at a higher scope than necessary and uninitialized objects will be flagged.

9. The number of declarations and executable statements for each subprogram will be recorded. These values will be provided both including and excluding nested subprograms. The maximum nesting depth for control structures, subprograms and tasks will also be determined for each program unit.

10. The number of GOTOs and jump target labels (<<LABEL>>, not LABEL:) will be counted, and a measure of the "branching complexity" of a routine will be determined.

11. Unlabelled blocks and loops will be flagged. Use of these labels often provides a valuable form of commentary.

12. Declaration of typed constants vs. universal numbers will be flagged when appropriate. Use of a DELTA other than a power of 2 for fixed point types will be detected. Use of a radix other than 2, 8, 10 or 16 will also be flagged.

3.4 Header Comment Box Generator

Software standards at NAAO specify that each compilation unit be headed by a comment box that contains detailed information about the unit.

Among other things it is required that the comment box list all sets and references to global variables (objects declared at a higher scope), all subprograms and tasks called, task entries, exceptions generated (other than the usual Ada exceptions) and exceptions handled, and all packages, tasks and

subprograms defined internally.

The header comment box generator will detect the presence of these features and create the part of the header comment box that describes them. If already present, the existing comment box will be revised.

Gathering this information for the comment box is a tedious task which implementers do without enthusiasm, and thus without attention to correctness and detail. Frequently the information is ignored when it is needed (say, by a tiger team called in to fix a high-priority problem) because it is assumed to be incorrect and out of date. Automating its generation should greatly increase its reliability and usefulness.

4 CONCLUSIONS

The Ada development environment described here meets most of the needs of our current and near future development requirements. The objectives of a cost effective APSE implementation and a versatile development environment are expected to be satisfied.

SOFTWARE ENGINEERING ENVIRONMENT

TOOL SET INTEGRATION

William P. Selfridge

Rockwell International Corporation

12214 Lakewood Boulevard

Downey, California 90241

Telephone: 213 922-2935

ABSTRACT

Space Transportation System Division (STSD) Engineering has a program to promote excellence within the engineering function. This program resulted in a capital funded facility based on a VAX cluster called the Rockwell Operational Software Engineering System (ROSES). This paper concentrates on the second phase of a three phase plan to establish an integrated software engineering environment for ROSES. It discusses briefly phase one which establishes the basic capability for a modern software development environment to include a tool set, training and standards.

Phase two is tool set integration. The tool set is primarily off-the-shelf tools acquired through vendors or government agencies (public domain). These tools were placed into categories of software development. These categories are: 1) requirements, design and construction support, 2) verification and validation support (i.e. quality evaluation), 3) software management support. The integration of the tools set is being performed through concept prototyping and development of tools specifically designed to support the life cycle and provide transition from one phase to the next.

Tools that integrate category 1 tools are: 1) the Documentation Utility Package - supports the development of software development library products that meet DoD-STD-2167; 2) the Software Development File Manager - supports the tracking and reporting of incremental development of the software development library products; 3) the Life Cycle Traceability Matrix Manager - supports the automatic extraction of originating requirements and traceability of propagated products through a relational data base.

Tools that integrate category 2 tools are: 1) Evaluation Translators - supports the static analysis of software development library products; 2) Automatic Software Testing and Reporting (White/Black) - supports the automatic testing of software component's logic (white box) and CSCI requirements verification/validation (black box).

Tools that integrate category 3 tools are: 1) the Software Management Utility - supports the control of baselined products and development configuration items; 2) the Integrated Software Change Control Management System - supports the tracking and reporting of changes to configuration items from change inception through release of product; 3) the Information Management System - supports user identification and acquisition of reference material and reusable software components and their documentation; 4) the Project Performance Measurement System - supports matrix management based on the earned value technique of schedule/cost tracking and reporting.

Phase three of the plan is briefly discussed and it applies advance technology to software development through the application of AI expert system concepts.

PROCEDURES AND TOOLS FOR BUILDING LARGE Ada SYSTEMS

Ben Hyde
Intermetrics Inc.
Cambridge, Massachusetts

This paper address some of the problems unique to building a very large Ada system. We do this through some examples taken from our own experiences. In the winter of 1985-86 Intermetrics bootstapped the Ada compiler we have been building over the last few years. This systems consists of about one half million lines of full Ada.

Over the last few years we have adopted a number of procedure and tools for managing the life cycle of each of the many parts of an Ada system. Many of these procedures are well known to most system builders: release management, QA testing, source file revision control, etc. Others are unique to working in an Ada language environment; i.e. recompilation mangement, Ada program library management, Ada program library management, and managing multiple implementations.

First we look at how a large Ada system is broken down into pieces. The Ada definition leaves unspecified a number of issues that the system builder must address: versions, subsystems, multiple implementations, synchronization of branched development paths, etc.

Having introduced how our Ada systems are decomposed, we then look, via a series of examples, at how the life cycles of those parts is managed. We lood at the procedures and tools we use to manage the evolution of the system. It is our hope that other Ada ststem's builders can build upon our experiences of the last few years.

Rational's Experience Using Ada[1] for Very Large Systems

James E. Archer, Jr.
Michael T. Devlin

1. Introduction

The development of very large software systems challenges human intellect and creativity. It has been described as one of the most complex activities undertaken by Man. The risks associated with such an effort are increased by its size and by the involvement of participants from different organizations, locations, and, possibly, countries. By any measure, the software for Space Station ranks among the most ambitious projects ever undertaken.

Considerable research effort has been devoted to solving the problems involved in the construction of such large systems. Unfortunately, while much of the resulting technology is available in the literature, it is not widely used [16]. Reducing theory to practice is always difficult; the rate at which this has been accomplished for software seems particularly discouraging. These difficulties prompted the Department of Defense to start the STARS program [7] and to establish the Software Engineering Institute [3] to focus on improving the state of practice.

2. Motivation

In 1981, Rational[2] set out to produce an interactive environment that would improve productivity for the development of large software systems. The mission was to create an environment that supported high-level language development of systems developed using modern software engineering principles. This mission was built on the belief in recent advances in programming languages, methods, and environments.

In designing the Rational Environment[2], object-oriented design [4], abstraction [9], information hiding [5], and reusability [10] were important both in terms of use in our design and as methods to be supported. Prototyping [11] was of particular importance because it gave access to the advantages of the environment and its component technologies, at the earliest possible time.

The language to be supported was Ada. This was an easy choice. Ada appeared to be the latest and best engineered language for building large systems [1]. In particular, the separation of specification from bodies appeared to offer a real advantage: it allowed the language to be used during design, as well as implementation, and it supported a realistic opportunity for reusability [8].

Experience with research programming environments had shown that access to a set of integrated facilities could greatly leverage the ability of individuals to produce systems. The most widely used of these environments supported interpretive or dynamically typed languages, most notably Lisp [13]. Research efforts to support more appropriate, strongly typed languages were interesting, but they centered mostly on interpretive implementations for student subsets [2, 14]. Even so, the benefits of these systems

---

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

[2] Rational and Rational Environment are trademarks of Rational.

suggested the feasibility of building a compilation-based environment for team development of large systems.

From the outset, it was clear that the Environment itself was an example of the kind of system whose development it was intended to facilitate. Although it would not be possible to use the Environment early in its construction, the other central technology themes, language and methods, were still available. To support these before the environment was functional, a set of tools was constructed to support Ada development in a conventional, batch-oriented manner. We don't think of the resulting tool set as an "environment"; however, it does constitute an APSE in the Stoneman sense [12], it includes a validated compiler, and it is comparable to other commercially available Ada compilation systems. The development of this tool set involved more than 300,000 lines of Ada code; building it helped to improve our understanding of the problems and opportunities associated with the evolution of the Rational Environment.

3. Environment Characteristics

The Rational Environment is the operating software for the Rational R1000, a time-shared computer implementing a proprietary, Ada-oriented architecture. It is written entirely in Ada, with considerably less than 1% of its statements being machine code insertions.

The Environment is the system interface; all users of the system use the same facilities. Although general-purpose computing is well supported, the system is designed to be used by project-related personnel with some interest in and facility with Ada and programming language concepts.

3.1. Ada Framework

The Environment directory structure is hierarchically organized. Names in this structure are Ada simple identifiers separated by periods, as with Ada qualified names. This structure contains a variety of objects of a variety of system and user-defined types. One common object type is *file*; another is *Ada*. Files resemble files on any other system. Ada objects are more interesting.

An Ada unit under development is an Ada object. The name of the object is the name of the unit that it represents. Ada objects corresponding to library units have two parts: visible part and body. Separate subunits of Ada units are children of their parent Ada unit and are named as such. As a result, the same name is used to refer to the unit when it is edited, compiled, and executed. All of the units residing in the same directory substructure constitute an Ada library, and there are provisions for creating libraries, hierarchical or otherwise, from multiple simple libraries.

The treatment of Ada units as typed objects is central to the design of the Rational Environment. In addition to supporting an object-oriented view of the unit throughout the compilation process, the storage of the program object as an attributed DIANA tree [6] provides access to the program structure in a way that makes a variety of interesting facilities available. These include Ada-specific editing operations, incremental compilation, compilation ordering and interconnection facilities, and direct execution of Ada statements.

3.2. Compilation

The traditional compilation model involves reading files of program source into a series of tools that produce various processed forms of the original program. During this process, new objects with new names are created and the user is forced to track the correspondence between the current program text and the current executable version. In contrast, the Environment compilation model centers on Ada units as definable objects that are transformed by editing and compilation between three principal states: source, installed, and coded.[3]

A source unit has been parsed, but has yet to be compiled. It isn't just another form of file: it's a DIANA tree sufficient to support interactive syntax checking and to perform operations that depend on the structure of the unit. Maintaining this structure makes it convenient to keep units syntactically consistent at all times, greatly reducing the time lost trying to compile units with syntax errors. Ada was explicitly designed to have a declarative structure that facilitates the expression of complex system interaction in a way that can be statically checked. Installed units have passed the semantic checks necessary to assure that they are consistent, both internally and with units that they reference. Getting units semantically consistent and keeping them consistent is one of the major programming activities in Ada development. Once a unit is installed, coding is just a matter of time and computation required to get into execution; there is no intellectual effort involved. Coded units are ready for execution. Programs are intended to be executed, so this is the final state in the compilation process, if not the most interesting one. The existence of separate installed and coded states reduces the compilation effort, increasing interactivity during one of the challenging parts of the programming process.

The Environment supports a spectrum of compilation paradigms:

- Batch installation and coding with fully automatic ordering
- Editor-based installation and coding of individual units
- Incremental, statement/declaration-level changes to installed units

All these paradigms make use of the system's knowledge of the structure of the units being processed to determine correctness and compilation ordering. Incremental changes to compiled units has an immediate intuitive appeal regardless of the language involved. Making small changes and only recompiling what has actually changed reduces both the total compilation effort and the time between a change and getting the benefit from that change. This is particularly important for Ada: getting immediate feedback on the legality of a change makes it possible for the developer to use the declarative structure more effectively in evolving the program. Early detection of problems minimizes wasted effort.

Another benefit to be derived from incremental operations is the ability to add new functionality to a specification with minimal compilation effort. The goal is to add declarations to the visible part of a package without allowing illegal changes or requiring clients of the package to be recompiled -- all of the benefits of strong typing without the consequences. Providing this facility was one of the more interesting technical challenges in building the Environment [15], but it was certainly worth the effort.

Immediate semantic information about programs under development is not limited to the compilation process. Part of providing incremental semantics was building a database of

---

[3] Compilation for targets other than the R1000 may involve more than these three states.

declaration-level dependency information sufficient, in conjunction with the DIANA trees, to determine the legality and impact of incremental changes. This information turns out to be generally useful. For installed Ada units, the relationships between declarations and their uses is a matter of great interest. Given the rich structure of Ada naming (use clauses, renaming declarations, overloading), it isn't possible, much less desirable, to keep track of these relationships on the basis of the program text. Using the compilation dependency information, these relationships can be checked interactively.

Definition is the name of an operation to show the declarations of an object that is referenced somewhere in an Ada unit. As typically used, the user points to the reference of interest[4] and presses the key to provide its definition. The declaration of the object referenced is brought onto the screen in the context of its Ada unit. It is also possible to find the implementation of the declaration. Definition is very useful in refreshing familiar code in the user's mind; it is invaluable in understanding unfamiliar code. A generalization of this mechanism to all system objects is the basic command for visiting objects of any type, traversing the directory structure, and changing context.

Show_Usage is the name of the operation that goes in the opposite direction: it provides the set of references shared by a declaration, a form of interactive cross-reference. If only one unit references the declaration, the referencing unit is brought onto the screen with each of the references underlined in a way that it is easy to traverse from one marked reference to the next. Where multiple units reference the declaration, a menu of units is present and the definition operation applied to any of the menu entries brings up a marked image of the indicated unit. Show_Usage runs in time proportional to the number of first-level references, typically a second or two. It is an invaluable aid in determining the impact of an anticipated change.

3.3. Ada Command Language

Conventional systems typically provide some sort of command shell that executes *programs*, specially prepared and loaded collections of units that can be executed from the command shell. These procedures must either live in a simplified world without parameter passing or understand how to read arguments from the command line. Then, if a normal procedure wants to call one of these programs, it is necessary to understand how to invoke a shell and construct the parameters as if they were being passed in from the shell.

In the Rational Environment, any coded visible subprogram can be executed simply by calling it, provided that the closure of units required by Ada rules is also coded. This has a profound effect on the accessibility of code for execution and testing. By unifying the shell/program interface to use the normal Ada parameter mechanisms, the interface is made both simpler and more powerful.

One salient improvement is the ability to use the richness of Ada semantics. This ability to reference the declarations in Ada units isn't limited to procedures and functions; it extends to all Ada declarations: types, objects, constants, generics. The advantages that Ada has for the expression of application designs are available for the specification of the system interface or user-written utilities. This generality has far-reaching implications on the appearance, usage, and implementation of the system. References to procedures with complicated parameter profiles can be expressed using the name notation, parameter defaulting, and overloading. Interfaces to predefined packages, e.g.,

---

[4] It is also possible to type its name if there is no immediate occurrence to point to.

Text_IO, are just as easily invoked as commands to create files, using the same interface.

The full power of Ada is important to making the command interface work. An Ada-like interface that is limited to normal command-style entries might seem an attractive tradeoff between generality and implementation effort, but closer inspection reveals the limitations of this strategy. Cutting isolated features from any language is a treacherous undertaking. As a simple example, private types are useful for providing abstract interfaces to system functionality, and having private types in the command interface turns out to be quite useful. This usefulness, in turn, depends on the ability to provide function results as parameters and, in many cases, to make them default reasonably.

The command interface is an Ada declare block into which the user typically enters a single procedure call that is executed. In the general case, it is possible to write complete Ada programs using tasks, generics, or any other Ada construct in this block. The block is then compiled, and code is generated and executed.[5] The completeness of the facility is often exploited in learning Ada and determining "what would happen if ...". Since the interface is Ada, it is strongly typed; it benefits from detection of errors during compilation rather than during execution; syntactic and semantic completion are provided.

3.4. Editor-Based User Interface

All interactions with the Rational Environment are through a general, object-oriented, multi-window editor. At one level, the editor provides familiar "what you see is what you get" on the images corresponding to the objects being edited or viewed. The text of the images can be modified directly using character, word, and line operations; portions of images can be copied or moved to other locations in the same or different images; there is a general search/replace interface. All of these capabilities allow the user to view and modify objects in a human-readable form: text.

Many of the various types of objects in the system, most notably Ada, are stored in more interesting data structures than text. To support the transition from text editing to object representation, the editor supports an incremental change, parse, pretty print cycle. Changes to the text are saved for processing by type-specific editors that understand the syntax of the particular object. The changes are processed by the incremental parser to create consistent object structures. As necessary, the revised data structures are reflected onto the image with any corrections or embellishments that are deemed appropriate by the editor for the type. A type-specific editor, called an *object editor*, is available for each of the main object types. All of these implement similar editing cycles, but the operations, grammar, and semantics for Ada, discussed below, are the most interesting.

The actual operations provided for editing an object are logically separated into three classes: image operations, common object operations, and type-specific operations. Image operations are the outer-level, character-oriented operations; these are the same for all object types. Common operations are those that are expected to be available for all object types, but depend on the characteristics of the type; these include edit, structural selection, detail control, and various state transformations. Type-specific operations are provided by some types of objects where the characteristics of the type require additional functionality. Creating an object is type-specific.

---

[5] There is a special fast path provided for a common subset of known procedures for which no code is generated. This covers about 80% of the command executions. Users are typically unaware of which path a particular command takes.

A simple, but commonly used, object editor is the one provided for the subclass of files corresponding to text. Its use with files, whether created by the editor or written by programs with Text_IO, is fairly conventional but benefits from the ability to select text from other object types for inclusion in documents, mail, or bug reports. The text object editor is also responsible for dealing with Standard_Input and Standard_Output for interactively executed programs. In this mode, the user has full access to the features of the editor in providing input to programs and scanning their output, either while they are running or long after they have completed.

One of the features of the editor interface is that it doesn't impose any particular interaction sequence on its users. As a result, it is possible to freely switch between objects being edited and executing programs. The input required by an executing program can be provided by copying the text from another object or from a previous run of the same program. To support multiple concurrent activities, all visible windows are kept current with the values of the underlying objects, including (optionally) scrolling windows into which program output is being generated. This makes it convenient, for instance, to maintain a window on a long-running command to monitor its progress while continuing to get work done on something else.

3.5. Ada Editing

Ada was designed to allow the specification and construction of complex systems that could be read, understood, and maintained. A person has to write the programs, preferably using the expressive capabilities that will serve well throughout the life of the code. The purpose of the Ada object editor is to make the writing as easy as possible.

By understanding the syntax of Ada, the editor is able to provide interactive syntax checking and completion. Syntactic completion is based on the notion that many tokens in the syntax are redundant; providing the additional tokens is only marginally harder than detecting their absence. For instance, most of the structures of Ada syntax are signaled by keywords or punctuation that bracket constructs; e.g., the existence of the keyword *if* implies the future existence of *end if* and at least one statement in between. The editor uses this information to provide the keyword structure and, if required, prompts for the expression and statement portions of the statement. The result is logically very similar to operations provided by syntax-directed editors, but is stylistically similar to normal text editing and only enforces syntactic correctness at user-specified points in the editing process. Used frequently, the program can be kept syntactically correct; when necessary, wholesale editing can take place without incurring checking overhead until the changes are believed to be complete. Prompts are presented in a special font and obligingly disappear when typed over, providing convenient reminders of code still to be written. Any attempt to execute a prompt raises an exception.

A less frequently used, but powerful form of syntactic completion is provided to construct skeletal bodies for the visible operations of a unit. Completion saves typing the same procedure headers in both the visible part and the body. A related operation creates a private part with prompted completions for each of the private types in the package.

The logical extension of syntactic completion is *semantic completion*. Semantic completion fills out the contents of expressions, most commonly subprogram calls or aggregates, in a manner analogous to the way syntactic completion fills in the structural parts of the language. When making an incremental change in an installed or coded unit, it is possible to enter part of an expression, typically a procedure or function call, and request that the system fill in the parameter profile with prompts for parameters

without defaults. In doing so, the system will provide the full name-notation presentation of the call, supporting good stylistic use of the language without requiring the user to do the additional typing.

3.6. Debugging

The Rational Environment supports debugging in the same spirit as the other parts of the programming process. Debugging a program is just like running it without the debugger, except that a different "execute" key is used. No special preparations are required to set programs up to be debugged. Debugging is not intrusive: two people can be debugging the same program at the same time without getting in each other's way. Interaction with the debugger is at the source level. Program locations are displayed by bringing up the Ada image of the statement and highlighting it. Variables and parameters can be displayed by selecting them and pressing the "Put" key or by entering a command with the name of the desired variable. The value displayed is presented as it would appear in program source: record values are printed as aggregates with field names; enumeration values are printed as the appropriate enumeration literal.

3.7. Host-Target Support

Although the R1000 provides an attractive environment for the execution of Ada programs, the system was designed to support the development of programs that would run on other targets, not to be a target itself. With the exception of the execution interface, the system provides all of the facilities described for target development.

Editing and compilation appear the same for other targets as for the R1000. Indeed, the target being compiled for is a declarative property of the library and affects the content, but not the form, of the basic operations. Since we don't expect that Rational will supply code generators for every possible target, there is a general compilation interface that captures target dependencies in installation and coding, without user intervention.

Execution and debugging are less easily specified, but the debugger architecture includes support for the same set of operations on targets connected by communication lines as for native R1000 programs. There is also provision for target-specific debugging operations in a manner analogous to that used by the editor to provide type-specific operations. A variant of this host-target strategy was used successfully in debugging the Environment in its early stages.

3.8. Configuration Management and Version Control

Supporting an object-oriented view of Ada units implies support for configuration management and version control within the same integrated context. Previous experience with research environments suggested that programs need not be files, but all of these efforts focused on lone developers on prototype systems, not teams producing a product. Conventional systems solve the problem by separating program storage and version control from compilation; this separation is impractical without compromising compilation, completion, and other facilities.

A separate, but related, problem that arises in a large system is control over the configuration to be compiled and executed. Early experience showed that the connectivity of a large Ada system (the environment itself) makes it attractive to break the system up into subsystems to allow changes in one part of the system to be tested before being used by another. Simply executed, this strategy provides some relief, but it still strains compilation resources at integration points. This strain was especially bothersome, since integration took place during a prototyping stage when long delays in reintegration were undesirable.

The solution to this configuration problem was to structure subsystems to have the equivalent of visible parts and bodies. Subsystem interfaces, a subset of the visible units of the subsystem, provide the correspondent of visible parts. The complete set of units corresponds to the body of the subsystem. As with Ada units, the contract made by the visible part must be fulfilled by the body, but the implementation of the body can be changed without recompilation of clients of the visible part. An extension to the notion of incremental change of visible parts within a subsystem is that of *upward-compatible changes*. Upward-compatible changes are additional declarations that can be added to the subsystem interfaces such that references compiled against a version of the interface without the new declarations will continue to work, but new code can start to reference them.

One very effective addition to the subsystem technology was the ability to hide the private parts of packages.[6] Private parts are instrumental in providing abstract interfaces whose underlying implementation can be changed without rewriting referencing code. This extension makes it possible to change the representation without recompiling, just as if the completion of the type were in the body. For our code, this capability was particularly useful. It is common to have a package that exports private types whose completions are types exported from instantiation(s) of generics that are only referenced for this purpose. Closing the private part makes it unnecessary for the interface to appear to *with* the package exporting either the generics or the types involved. Reducing the *with* closure reduces the size of the interface while reinforcing the spirit of abstract interfaces.

This ability to compose a system of compatible subsystems that have not been directly compiled together greatly facilitates integration, especially since the assurance of semantic integrity is not lost. It does not directly address the version control problem, but leads to a version control policy based on a series of *views* -- configurations of the entire subsystem library, each spawned from the previous version of the view.

Experience with these mechanisms and experience with the compilation system have led to the construction of a more sophisticated form of view that combines the advantages of subsystems, reservation-model "source" management, and differential storage of changes to provide a facility that effectively combines the best of conventional version control with the advantages of subsystems for forming configurations. By managing views for the user, it is possible to provide support for these various forms of multiplicity in such a way that there seems to be more than one version only when differentiating configurations is part of the work at hand.

3.9. Life-Cycle Support and Extensibility

The goal of the Rational Environment is to support all of the life-cycle activities involved in software development. The initial implementation effort has focused on support for detailed design through maintenance and on building an environment that is conducive to extending these facilities into other parts of the life cycle. Our experience has been that Ada, by itself, provides a useful basis for program design, especially where it is possible to compile the designs and trace through the dependencies.

Many of the facilities that make the Environment attractive for programming also make it attractive for tool development and use. The access to DIANA and semantic information holds out the promise of building tools to analyze programs and and their development. The ability to construct interactive, editor-based interfaces has proved

[6] The R1000 architecture provides efficient support for this form of truly private type.

attractive and has helped in the process of providing useful interfaces for interesting funetionality.

4. Experience

The Rational Environment itself consists of about 800,000 lines of Ada. Development of the Environment also required building about 700,000 lines of Ada to provide cross-development tools (compilers, debuggers) and hardware/microcode development tools (simulators, translators, analysis programs). The product was first shipped to customers in February of 1985. Several significant upgrades (involving greatly improved performance, increased functionality, and improved robustness) have been delivered since then.

This development has provided considerable experience in the use of Ada with modern software engineering practices. This experience can be summarized by the following statements:

1. Adoption of Ada and the software engineering practices referenced earlier has been somewhat more difficult than anticipated. Significant investment in tools, training, and experience has been required.

2. The benefits are very real. Improvements in productivity and quality have been evident in all phases of development: design, implementation, integration, test, and maintenance.

4.1. Early Ada Experience

In 1981 and early 1982, a series of programs were constructed: development and simulation tools and prototypes of high-risk components of the Environment. These typically consisted of 50-100K lines of Ada.

Ada proved to be an excellent language for applying the concepts of information hiding, data abstraction, and hierarchical decomposition based on levels of abstraction. The basic package mechanism, separation of specification and implementation, and private types allowed rapid construction and modification of large, modular programs.

Ada cannot force good design, but it does capture and clarify the decomposition and connectivity of programs, allowing early detection and correction of architectural flaws in the design. Ada became our primary design tool, particularly for detail design. With experience, we were able to produce high-quality designs quite rapidly.

The interaction between semantic checking and modularity produced significant improvements in productivity. Using modularity and type structure to capture design information increased somewhat the time required to first execute the program, but it also greatly increased the chances that the first execution would be productive. New arrivals frequently complain that they aren't ever going to get the program to compile, only to come back later amazed that it worked the first time. When problems did arise at runtime, constraint checking allowed the errors to be detected early in execution. A common, effective debugging strategy is to run the program until an unexpected exception occurs; the problem is often evident with no additional information. Even when this is not the case, the modularity of most programs reduces uncertainty about interactions and allows much more rapid isolation of errors. It is also much easier to reason about the structure of programs and predict the consequences of a change.

Early experience also showed that all these wonderful benefits were not free. Ada semantic analysis is very expensive, increasing compilation times significantly relative to other languages. The early detection of interface and type-safety errors was handicapped by the use of batch compilation technology to report these errors. This confirmed our belief that an interactive environment for Ada with support for incremental compilation would greatly improve productivity.

4.2. Large-Scale Development and Integration

In 1983 and 1984, the development focus at Rational shifted from developing programs consisting of 10-100 packages to incrementally constructing and integrating a complete system made up of 30-40 subsystems, where each subsystem was the size of one of the earlier programs.

The system was decomposed hierarchically into five major layers, with each layer consisting of 5-8 subsystems. Although there were significant structural and interface changes over the life of the project, the basic architecture was surprisingly stable. This architecture allowed considerable parallelism in the overall development process and was instrumental in the evolution of our understanding of the configuration management and version control issues in developing large Ada systems.

At a very early point, the components of the system (or skeletons of the components) were integrated into a complete system. This initial system had very limited functionality, but allowed the basic architecture to be "debugged" before the entire system was constructed. This integration allowed system design issues such as storage management, concurrency, and error handling to be addressed very early in the development process. Early integration also served to stabilize major interfaces.

Development of the individual subsystems proceeded in parallel, with periodic integration to provide a new baseline for further development. The use of hierarchical decomposition allowed enough independence for development to proceed in parallel, while providing tight interface control to minimize integration problems. It was this integration process that led to the evolution of the subsystem concepts and supporting tools described in section 3.8.

The combination of the Ada language with object-oriented design techniques, tool support for integrating configuration management and compilation management, and an incremental integration strategy proved very effective for this particular project.

4.3. Maintenance

The Rational Environment has been in field use for about 16 months in multiple releases. Supporting it has provided some limited insight into the maintenance phase of a large Ada system. At Rational, maintenance is the responsibility of the original development team; it was crucial that new development proceed in parallel with maintenance without significant increase in development staffing.

Our experience has indicated that Ada's greatest value may be in maintenance. In this particular case, *maintenance* included bug fixes and minor enhancements, addition of major new functionality, redesign and reimplementation of several subsystems to improve performance, and reorganization of parts of the user interface. Since initial product introduction, not only has it been possible to provide desired new functionality, but reliability and robustness have improved and overall system performance has been increased by at least a factor of 3.

Efforts to improve performance are interesting examples of both the power and the

associated dangers of modularity and abstraction. Bringing up a large system in minimum time was greatly facilitated by abstract interfaces and the ability to reuse code (Ada generics). There were several cases where performance-critical sections of code were operating through generics in multiple layers of the system, where a much faster implementation was possible. Ironically, the same modularity and abstraction that introduced the problems contributed to the solution of the problems: these sections were completely redone and integrated into the system without major disruption. Abstraction is not an end in itself, but used carefully, it can help produce reliable, maintainable software to meet performance constraints.

4.4. Experience Using the Rational Environment

Our experience using the Rational Environment has confirmed those advantages we foresaw when we started the project. Interactive syntactic and semantic information makes a tremendous difference in the ease of constructing programs and making changes to them. The ability to follow semantic references makes it easier to understand existing programs and the impact of changes. The integrated debugger makes it much easier to find bugs and test fixes quickly. Taken together, these facilities have helped greatly in reducing the impact of ongoing maintenance on our ability to produce new code. We anticipate similar improvements as we achieve the same level of integration and interactivity for configuration management and version control.

The Environment has also proved useful in introducing new personnel to the project and existing personnel to new parts of the system. New personnel benefit from the assistance with syntax and semantics; everyone benefits from the ability to traverse and understand the structure of unfamiliar software. It is often possible for someone completely unfamiliar with a body of code to use these facilities to understand it well enough to successfully diagnose and fix bugs in a matter of minutes.

Acknowledgments

The design and implementation of the Rational Environment was a group effort involving too many people to mention individually. Each member of the team contributed invaluable ideas, effort, and experience. It has been a challenging, rewarding, and enjoyable process.

References

1. *Reference Manual for the Ada Programming Language.* Washington, D.C., 1983. United States Department of Defense.

2. J.E. Archer, Jr. *The Design and Implementation of a Cooperative Program Development Environment.* Ph.D. Th., Cornell University, August 1981.

3. M. Barbacci, A. Habermann, and M. Shaw. "The Software Engineering Institute: Bridging Practice and Potential". *IEEE Software 2*, 6 (November 1985), 4-21.

4. O-J. Dahl and K. Nygaard. "SIMULA - An Algol-Based Simulation Language". *Comm. ACM 9*, 9 (September 1966).

5. D.L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". *Comm. ACM 15*, 3 (December 1972).

6. A. Evans, K. Butler, G. Goos, W. Wulf. DIANA Reference Manual. TL 83-4, Tartan Laboratories, Pittsburgh, Pa., 1983.

7. L. Druffel, S. Redwine, and W. Riddle. "The STARS Program: Overview and Rationale". *IEEE Computer 16*, 11 (November 1983), 21-29.

8. J. Ichbiah. "Rationale for the Design of the Ada Programming Language". *SIGPLAN Notices 14*, 6 (June 1979). Part B.

9. B. Liskov, and S. Zilles. "Specification Techniques for Data Abstractions". *IEEE Trans. on Software Eng. SE-1* (March 1975).

10. M. McIlroy. Mass-Produced Software Components. In *Software Engineering Concepts and Techniques*, NATO Conference on Software Engineering, 1969.

11. T. Standish and T. Taylor. Initial Thoughts on Rapid Programming Techniques. Proceedings of the Rapid Prototyping Conference, Columbia, MD, April, 1982.

12. *Requirements for Ada Programming Support Environments (Stoneman).* Washington, D.C., 1980. United States Department of Defense.

13. W. Teitelman. A Display Oriented Programmer's Assistant. CSL77-3, Xerox PARC, 1977.

14. R.T. Teitlebaum and R. Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. 79-370, Cornell University, Department of Computer Science, 1979.

15. T. Wilcox and H. Larsen. The Interactive and Incremental Compilation of Ada Using DIANA. Rational, Mountain View, CA.

16. R. Yeh. "Survey of Software Practices in Industry". *IEEE Computer 17*, 6 (June 1984).

USING ADA (R) ON A WORKSTATION FOR LARGE PROJECTS (Abstract)

Arra S. Avakian
Benjamin M. Brosgol
Mitchell Gart

Alsys, Inc.
1432 Main Street
Waltham, Mass. 02154

Alsys has implemented validated Ada compilers that are hosted and targeted on a variety of microprocessor-based workstations, including the IBM PC/AT (*). The availability of Ada compilers for these kinds of inexpensive, widely available machines considerably enhances the development options for large efforts such as the NASA Space Station, and we address this from both an implementation and a user perspective. First we discuss the issue of large program development on a workstation: how the compiler must handle this, and how an inherently decentralized approach can be managed. Next, we focus on code efficiency and describe the compiler and run-time design decisions that help meet this goal. We conclude with a presentation of benchmarks that are quite encouraging with respect to the run-time efficiency of Ada code compared with other languages.

Developing Large Programs

One of the principal design goals of Alsys' compilers is the ability to handle large programs. The technical approach combines a host interface package that inplements a virtual memory mechanism for compile-time data, a dynamic loader (for the 68000-based systems), and a user option for protected mode that allows programs as large as the full memory capacity of the workstation (for the 80286-based systems).

As users of our own tools, we have "bootstrapped" the compiler and its supporting environment, comprising over 300,000 lines of Ada, on the PC/AT. Getting the compiler to compile itself has given us considerable experience as users of the PC/AT compiler for a large software project. Our development system was originally hosted on 2 Vax minicomputers, but we switched to AT's as soon as the compiler was sufficiently robust. Our project is now being completed on a network of 2 Vaxes and about 10 ATs, (each engineer has an AT on his desk) tied together with an Ethernet.

Our experiences as "pioneer users" of the PC/AT compiler in a networked workstation environment are encouraging. The main problem in such an environment is keeping versions of source code and object programs organized, and we will discuss in the full paper the solutions that we have developed. The main advantage has been an increase in productivity. The subjective perception among the engineers who have switched from a time-sharing to a personal computing environment has been overwhelmingly positive.

Run-Time Efficiency

A critical compiler objective is the attainment of high quality code. To achieve this, the compiler design includes two intermediate languages -- one at a high level comparable to DIANA, and the other at a low level -- and two optimization passes. Additionally, the code generator performs machine-specific optimizations and the run time design emphasizes efficiency of subprogram linkages and object references.

A set of benchmarks, originally coded in Pascal by an independent organization, were rewritten in C and Ada and run on a variety of commmercially available compilers on 68000-based workstations and also on the IBM PC/AT. The conclusion from these tests is that it is possible to get efficiency with Alsys' Ada compilers at least as good as from compilers for C and Pascal. (A more complete description of the benchmark tests will be given in the full paper.)

Conclusions

The trend in hardware is toward decentralization, with an increase in cheap computing power and low cost memory. The problem has been a scarcity of software tools to take advantage of this increased power and capacity. With the emergence of Ada compilers in the workstation environment, such as Alsys' for the PC/AT, and advances in techniques for managing and integrating separately developed components, there is an opportunity to have the best of both worlds: the benefits of Ada, and low-cost software and hardware development environments.

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

(*) Note: The PC/AT compiler has been internally prevalidated; it will be formally validated by the time of the conference.

A Distributed Programming Environment for Ada*

Peter Brennan, Tom McDonnell,
Gregory McFarland, Lawrence J. Timmins and John D. Litke
Grumman Data Systems
1000 Woodbury Road
Woodbury, New York 11797

Abstract

Despite considerable commercial exploitation of 'fault tolerant' systems, significant and difficult research problems remain in such areas as fault detection and correction. This paper describes a research project to construct a distributed computing test bed for loosely coupled computers. The project is constructing a tool kit to support research into distributed control algorithms, including a distributed Ada compiler, distributed debugger, test harnesses, and environment monitors. The Ada compiler is being written in Ada and will implement distributed computing at the subsystem level. The design goal is to provide a variety of control mechanisms for distributed programming while retaining total transparency at the code level.

Introduction

Many new system designs specify a distributed architecture to attain incremental growth or increased computational power. These systems typically have homogeneous processors linked either by shared memory or by a message passing system. Concomitant with easy expandability and large computational power, one gains some resiliance against hardware faults. That is, if one processor fails, only the work executing on that processor is lost, not the entire work load. If one adds the capability to detect processor failure and to move the work from that failed processor to other working processors, then some tolerance for both hardware and software faults is attained that cannot be achieved with single processor systems.

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

Further, if one can move work from a failed processor to an active processor, rather easy extensions allow work to move from one active processor to another, achieving a load balancing effect for maximum output from the processor resource.

This ability to function despite hardware failure has made distributed, loosely coupled architectures a favored architecture for ultra reliable systems. To make this architecture effective, we must partition a problem into several parts so that each part can execute relatively independently on separate computers. The partitioning process introduces requirements to coordinate the execution of the several parts and to verify that each part is operating properly so that, if a failure occurs, corrective action can be taken. Such methods for problem coordination and control in a distributed environment are the principal focus of this research. We wish to assess whether, given the proper tools, one can construct loosely coupled, distributed applications that are cost effective, reliable, and efficient.

When a problem solution is developed for execution on a single processor computer, the usual method is to design several modules that jointly solve the problem. Coordination of the solution process requires a communication channel between modules, usually via shared variables or messages. Further, any shared data must be specified and storage allocated. This design results in intimate coupling between the several modules, with a significant chance for error. Ada provides extensive checking of the interfaces between modules and the operations allowed on each data element, greatly reducing the severity of module interface errors.

When a problem is partitioned for execution on a distributed processing host, one designs several programs (instead of modules) that jointly solve the problem. Interface errors may still occur, but since a compiler can process only one program at a time, there is no compiler support for checking and controlling the inter-program interfaces. Hence one would like to extend the power of Ada to distributed programs. In such an extended language, a problem is still decomposed into separately executing programs (Ada tasks), but data sharing and module synchronization are implemented and checked by the compiler. While such an extension itself presents implementation difficulties, two additional problems are present in a loosely coupled environment: assuring liveness and serializability. Thus, we require a test environment to evaluate candidate implementation methods and to develop efficient new algorithms. The Ada language was designed to provide support for a distributed programming paradigm. Its' visibility and synchronization rules provide a model for data sharing, while the task and rendezvous constructs provide a control model. Ada provides primitive mechanisms for assuring liveness and serializability, but the attainment of these goals is left to the programmer. To assess the viability of Ada's

model, we require two things: a distributed host with a validated Ada compiler, and a tool kit for developing, debugging and measuring the performance of distributed programs. However, because Ada provides a model but not an implementation of distributed programming constructs, we must expect to try a variety of implementations before settling on one with acceptable performance. Hence we require a compiler that we can modify to try various implementations of Ada.

These considerations have led to the establishment of a Fault Tolerant Computing project at Grumman Data Systems with the following goals:

1. To construct an Ada compiler for a distributed architecture host so that the implementation of Ada's model can be varied significantly.

2. To implement several distributed programming models and assess their viability for serious problem solving in realistic environments.

3. To develop models and methods for solving the liveness and serializability problems, and to test these ideas on the distributed programming environment provided by the first two goals.

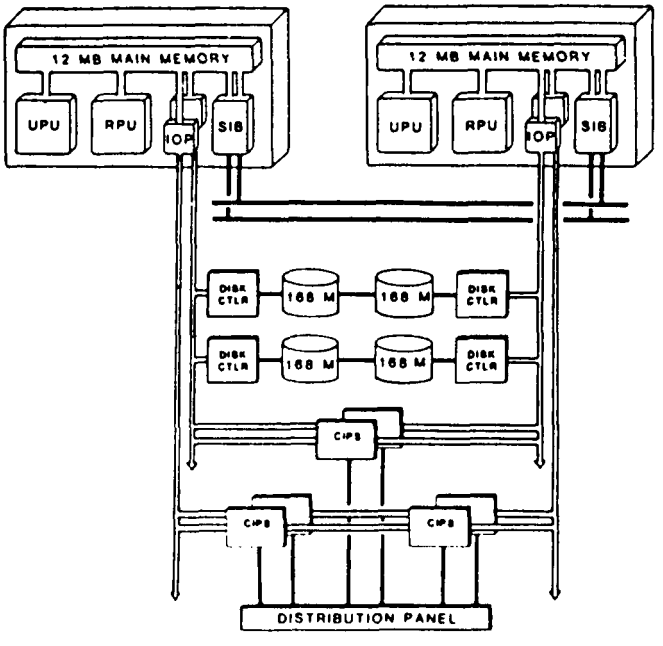
The project began in July of 1985 with a goal of constructing the foundation Ada compiler by summer 1986 and providing the first implementation of a distributed programming model by early 1987. The remainder of this paper describes the design and development of the foundation compiler and its supporting environment, and concludes with an outline of a distributed programming implementation for Ada.

Hardware Technology

The hardware base is the Eternity E-5000 computer system from Tolerant Systems, San Jose, CA. This system contains loosely coupled processors built with the National Semiconductor 32000 series VLSI processors. The operating system is TX, a superset of Unix BSD 4.2 and System V with extensions for transaction processing, distributed file systems, and built- in fault detection and recovery. The hardware is targeted for the commercial on-line transaction processing market, and so features a particularly robust and flexible communications capability. The fault tolerant capability is achieved with fail fast processors, dual redundant communication paths, and fault detection and reconfiguration software. Further, operating system services themselves are distributed in such a way as to support process migration, either to avoid faults or to provide load balancing. This support for distributed programming algorithms is an important advantage; it minimizes the infrastructure we must build.

Each processing element is actually a tightly coupled set of 32000 processors. (See Fig. 1). One processor (UPU) is dedicated to user applications, one (RPU) to the operating system, and one (CIP) to I/O and communications protocols. The UPU provides a UNIX compatible executing environment, while the CIP provides a real-time environment. Both processors have a common system language (C) and machine language. Although operating system services differ somewhat on each processor, one compiler can produce code that will execute on either processor. This permits us to develop an Ada compiler that will produce code for both a time sharing and a real time environment.

The file system is UNIX compatible at its interface, but highly modified in its implementation to provide a global name space and a robust foundation for system operation. In addition to traditional services, the file system provides an efficient, guaranteed message delivery system and plexed files with automatic restoration after failure. This is an essential system service for effective implementation of Ada's distributed programming model.



E-5000 Configuration Example
Figure 1

Compiler Technology

The Ada compiler must be constructed in such a way that the run time library can be modified. Since Ada's model for distributed computing is centered on the task construct, the

inter-task rendezvous and task scheduling algorithms must also be modifiable. We have chosen the retargetable compiler technology from TeleSoft, San Diego CA as the base on which to build. This system provides the syntactic and semantic analysis for Ada, manages an Ada library, and provides output in a tree structured form at approximately assembly language level. Our task is to build a suitable code generator for the E-5000 hardware. A key feature is that sufficient information on the Ada task implementation is available so that we can modify the Telesoft implementation model if required.

One of our themes when implementing this compiler is program execution efficiency. Execution efficiency not only requires an efficient algorithm, one of the primary foci of this research, but also an efficient implementation of those algorithms by the compiler. Thus code optimization becomes a theme of the first part of the project. Because of our implementation strategy, the potentially arduous construction of optimization algorithms splits naturally into three parts. We will depend on the TeleSoft front end for optimization flow of control, common sub-expression elimination, etc. The output of the compiler is National Semiconductor assembly code for the 32000 processor. The assembler on the E-5000 implements extensive optimizations that are effective for a C language compiler, such as code hoisting and instruction selection. Thus our code generator will concentrate on optimizations such as register allocation, minimization of bounds checks, efficient exception propagation, and the like.

Since the compiler will produce code for a real time environment, we must ensure that efficient programs are possible. Further, a highly modular language like Ada could invoke a large number of subroutine calls, making efficient call/return mechanisms a requirement. We focus on our implementation decisions surrounding the call/return mechanism as an example of tradeoffs involved during the compiler construction process.

The call/return mechanism has several basic requirements. It must:

1. Allow passing of data into and out of a subroutine.

2. Allow saving and restoring of temporary registers.

3. Allow access to out of scope variables.

4. Allow exception propagation out of the local scope.

5. Allow task switching and hence logical reentrancy.

The E-5000 uses a stack mechanism, growing down from high memory locations, for temporary variables including frame pointers. Thus the basic call/return paradigm is a classic one:

Call: **Put variables on the stack**
Put return address on the stack
Branch to the subroutine

(in called routine)
Save old stack base and old frame pointer on stack
Set new stack base and new frame pointer

Return: **Restore old stack base and old frame pointer**
Branch back to return address on stack

(in calling routine)
Remove return variables from stack

To extend this model for Ada, we must decide how to allocate stack space considering the multi-tasking Ada model and how to propagate information to/from the called routine with a minimum of overhead. Ada requires extra information be passed across this interface to allow out of scope variable references and to propagate exceptions. It was a particular goal to minimize the overhead of these latter requirements.

For the stacks, we have adapted the results from [GUPT85], namely to use a static storage area for module instances and a dynamic heap for temporary variables, satisfying requirements 1,2,5. (This scheme is often called a Berry-heap after [BERR78]). When allocating stacks for independent tasks, one must account for the possibility of collision of these stacks with each other [YEH86]. There are only two solutions, impose a static limit on the size of the stacks, or dynamically create room when required. In either case, the stack-full detection mechanism provided by the hardware is no longer useful for multiple stacks. We must implement the checks efficiently in software.

We allocate an initial stack with the intent to dynamically allocate more stack space if and when required. This approach makes effective use of available memory even for very large numbers of tasks, and imposes very little overhead [YEH86]. However, we now must check for stack overflow before every stack usage, an unacceptable overhead. Our first solution was to check, before every call, that parameters would fit on the stack, and then check at every entry that local variables (the frame) would fit on the stack. This is a two call overhead for every original call, an unacceptable result. The final design depends on the observation that stack requirements for local data and parameter passing are known at compile time, so that we can substitute one call on entry to each routine to check for sufficient stack space. Further, since routines that invoke no other routines typically have very small stack requirements, by requiring a small buffer space be present on all stacks we can remove all stack checking overhead for such calls. We accepted such minimal overhead for the benefits of a highly dynamic stack allocation mechanism.

The remaining two requirements, to implement out of scope references and to permit exception propagation to cross the call/return interface, each require separate treatments. Out of scope references in a multi-task environment are often implemented by copying a 'display' onto the currently active stack before every call. This display contains the storage offset pointer for each visible module, including the calling module. Each out of scope reference is implemented as an indirect reference relative to the proper pointer plus an offset. The difficulty with this solution is the requirement to set up the stack before each call. Although optimization algorithms could avoid setting up unnecessary displays, we would prefer to avoid the overhead altogether.

Our solution requires a static display area, one per task. Each module has a statically determinable lexical level that serves as an index into this table. When calling any module, we save the current value in the table at our lexical level, and overwrite the proper frame pointer in the table. On return from the routine, we merely copy back the original contents of the display. This requires an overhead of one load and two stores per call, optimizable to no action at all if we can determine that the routine being called does not reference any variables at our lexical level or higher and calls no other routine.

An efficient solution to exception propagation requirements is more complex. For locally raised exceptions, we can clearly use a direct transfer to the exception handling code. However, if an exception must be propagated to an outer scope, we must 'unravel' the stack frames as we search for the handler. In addition, we require that the cause and location of the exception raising be determinable in case a handler is not found. For real time programming, we would like such a mechanism to be swift. Further, if the exception could not be handled at any level, for debugging purposes we should not unravel the entire stack before we determine that the exception is unhandlable. Otherwise, essential debugging information is lost.

Our solution requires no overhead at call time and uses a binary search to identify the relevant exception handler before unraveling the stack. At compile time, each exception is uniquely identified as to raise location and reason, and every exception handler is uniquely identified as to the exceptions it handles, permitting identification of exceptions in a user friendly way should a handler not be found. The identification information, together with the addresses of the scope of each exception handler, is stored in a table in static memory. An initialization routine sorts this table before the program runs. If an exception must be propagated, the propagation code follows the stack pointer chain backwards, searching the common exception table for exception handlers that apply to the address given by each instance of the stack pointer chain until a handler is found. The table can be searched quickly for

rapid exception propagation. When a handler is found, the stack is quickly unraveled in one operation to the proper point and the handler invoked.

While not an exhaustive list, these items illustrate some directions we are taking in the development of an efficient Ada compiler. Many of our efficiency oriented algorithms are heavily parameterized so that we can vary their effect and study the resulting program behavior. This approach will allow us to tune the compiler for best effect under realistic conditions. Results of these efforts will be reported at a future conference.

Distributed Programming Model

When implementing an Ada compiler for a distributed programming host computer, there are three levels of capability to be considered, namely:

1. Minimum capability that satisfies the Ada Language Reference Manual [ANSI83].

2. Permit advice from the programmer to influence the implementation or execution of the model.

3. Enhanced functionality within the requirements of the Ada Language Reference Manual.

The remainder of this section addresses some issues pertinent only to the minimal capability implementation.

The execution of parallel, distributed processes under one computational model introduces such complexities that few practical systems today are entirely transparent to the user. The mark of a successful implementation is correctness, general applicability, and the capability to simplify the task of programming parallel execution paths. In contrast, Ada seeks to achieve two different goals: a simple inter-process communication paradigm and the efficacy of a complete semantic check of the entire collection of processes, viewed as a whole. Whether these goals are necessary or sufficient for a successful implementation is to be determined.

Ada defines a task model that provides a set of primitive communication mechanisms (accepts/entry calls) to implement parallel tasks. Although use of these requires explicit programmer cognizance, the programmer's task is simplified somewhat. The price for this simplification is that the compiler writer must implement correct interpretations for three shared elements: data, control via exceptions, and program state. Each of these elements is considered separately below.

To provide a background for this discussion, some fundamental design decisions must be noted. The first version of the distributed compiler will produce an executable image that executes on each distributed host unaltered. In other words, the instantiation of any module will execute on only one host, though its code image is present on all hosts. This decision means that the code on each host is larger than the minimal required, but that addresses not on the stack and not dynamically allocated are universally correct from host to host. Further, our hardware is a segmented, virtual memory machine, so that physical memory is not significantly wasted by this decision.

A second design decision is to use the operating system message passing facility for all intertask communication. Since we have compiled the program as a whole, targeted for one uniform processor, this communication need not incur the overhead of formatting/unformatting data, and so it can be used for co-located tasks as well as distributed tasks.

A third decision is that only tasks will be considered for distribution during the first implementation. (While this is not strictly true as we shall see, this provides the primary focus when designing the implementation model.) Further, to ease initial implementation efforts, no access variables can be referenced across a distributed interface. Now let us return to a discussion of how we intend to share data, control via exceptions, and program state information.

Data sharing between two asynchronous tasks takes several forms. The first is via data that is visible to two different tasks by operation of the scope rules of Ada. The Ada Language Reference Manual specifies that two tasks can 'see' the effects of updating shared variables only at synchronization points such as those associated with a rendezvous or by pragma 'SHARED', allowing every access of a variable to be a synchronization point. However, the Ada Language Reference Manual does not require that the compiler detect erroneous programs that violate these rules. A second, indirect way to share data is by the common invocation of library routines that reference static data. For example, a terminal I/O routine in a library package might reference static data to define the current line number on the screen; every call to this routine may alter the data.

Motivated by these two concerns, we have decided that the pragma 'SHARED' will not be allowed for two tasks that are not co-located within one host process. To address the indirect sharing of variables via library packages, we define three classes of objects (functions and procedures): idempotent, serially reusable, and re-entrant. The first class will execute correctly without any historical information. Any routine that does not access static data or any 'state of the world' is in this class. The second class indicates routines that access some static data, but that can accept successive

calls once the first call is complete. Most library routines are in this class. The third class, while they may depend on static data, may also be called by another routine before the first request is complete. These routines, such as I/O drivers, usually depend on a separate temporary data store (stack) per task to achieve their re- entrancy.

If a routine is declared idempotent, then we may execute any available local copy of the relevant code, taking no care to share static data among distributed tasks. This is the default nature for procedures and functions. If the routine is declared serially reusable, then we will execute the call on the one host that contains the instantiated version of the routine, and all calls will be queued in a FIFO manner. If the routine is declared re-entrant, then we will execute the call on the local host and broadcast any updated static data at the completion of the call. It is the programmer's responsibility to ensure that the specification of the proper behavior model matches his or her intent.

Another information sharing between two concurrent tasks is via the exception propagation structure. Since the colocation of a task and any exception handlers that it may invoke are not guaranteed, we must provide both a fast means to determine the location of the exception handler and a means to propagate the exception to that handler. Our decision to use a common program image allows the exception propagation logic to execute as a idempotent routine, determining the location of the handler before invoking any communication overhead. The communication required to pass control to a remote site is reduced to the identification of the raised exception.

Global state information is shared among distributed processes by the Ada requirement for task termination. When a task has an open terminate alternative, it must consider the state of all dependent tasks, sibling tasks, and the state of the parent task before entering the terminated state. In turn, this means that one must achieve a globally consistent picture of the state of all such tasks so that a correct decision can be made. There are only two solutions to this requirement. One solution elects or appoints a master controller to determine the state of the world, while the other solution requires periodic broadcasting of all states to achieve a consensus on a consistent state. The latter approach is often called a consistent checkpoint method, and often entails significant overhead waiting for all tasks to achieve a stable state. For this reason, we have elected to use the first method, by electing a 'controller' task as that task that dominates the immediate termination decision. By polling means, outlined in [JAH85], this one task (actually the local run time system attached to that task) will calculate the termination condition for all subordinate tasks.

Our general direction for implementation of the Ada distributed programming model has been decided. Our next step is to consider means to debug distributed processes and to measure the effectiveness of our initial implementation. This effort will result in a test suite of distributed programs, designed especially to test distributed control algorithms rather than just the computational advantage of parallel computation. The suite will be then used to evaluate the effectiveness of various distributed programming models.

References

[ANSI83] ANSI/MIL-STD 1815A, Reference Manual for the Ada Programming Language; January 1983

[BERR78] D. Berry, L. Chirica, J. Johnston, D. Martin, and A Sorkin, "Time required for reference count management in retention block- structured languages, part 1,", Int. J. Comput. Inform. Sci., 7(1), pp.91-119 (1978)

[GUPT85] Rajiv Gupta and Mary Lou Soffa, "The efficiency of storage management schemes for Ada programs", Ada Letters, Vol 5, 2, pp.164- 172, (1985)

[JAH85] Rakesh Jha and Dennis Kafura, "Implementation of Ada Synchronization in Embedded, Distributed Systems", Virginia Tech report TR-85-23, 1985.

[YEH86] D. Yun Yeh and Toshinori Munakata, "Dynamic Initial Allocation and Local Reallocation Procedures for Multiple Stacks", Comm. ACM, Vol 29, 2, pp.134-141, February 1986

Distributed Ada[1]: Methodology, Notation, and Tools

Greg Eisenhauer
Rakesh Jha
J. Micheal Kamrad, II

Honeywell Systems and Research Center

Abstract

The task of creating software to run on a distributed system brings with it many problems not encountered in a uni-processor environment. The designer, in addition to creating a solution to meet the functional requirements of the application, must determine how to distribute that functionality in order to meet the non-functional requirements such as performance and fault tolerance. In the traditional approach to building distributed software systems, decisions of how to partition the software must be made early in the design process so that a separate program can be written for each of the processors in the system. This design paradigm is extremely vulnerable to changes in the target hardware environment, as well as being sensitive to poor initial guesses about what distribution of functionality will satisfy the non-functional requirements. The paradigm is also weak in that no compiler has a complete view of the system. Many of the advantages of using a powerful language system are lost in a one-program-per-processor environment. This paper will present another approach to the development of distributed software systems, Honeywell's Distributed Ada program.

Our Approach

The goal of Distributed Ada is to develop methodology and tools which will significantly reduce the software design complexity for reliable distributed systems. We believe that the functional specification of a system (what it will do) can and should be separated from its non-functional specification (how it will be mapped onto the underlying system). The functional specification can be developed and expressed in Ada. To this is added the specification of the non-functional attributes of the system. Separating the problem space into two smaller problems means that the designer can concentrate on solving each of them in turn rather than attacking them together. It also allows software development to proceed before hardware final design is complete and enhances the portability of the functional specification.

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

The software development paradigm we advance is described by the following scenario. The designer develops a functional solution to the problem in Ada using uni-processor development tools. With a functional solution in hand, she then creates a specification of the non-functional characteristics of the solution (more details on the nature of this specification will be given later). Using the tools being developed under our program, these two specifications can be used to create the distributed solution incorporating the non-functional attributes. At this point, the distributed solution can be tested for acceptability according to non-functional criteria and modified if necessary to meet non-functional requirements.

The advantages of this and similar approaches over the traditional approach of up-front distribution decisions are self-evident. When non-functional specification is separated from functional specification, software development can proceed with limited knowledge of final hardware configuration and will be little impacted by changes in the underlying system. We believe, however, that the granularity of distribution and the mechanism of specification employed in our approach separate our work from that done by other researchers.

As opposed to other projects which limit the unit of distribution to the Ada library unit and limit remote access to tasks and subprograms in the visible part of remote units [Inv 85, Sch 81, Sof 84, Vol 85], we believe that an effective and extensible non-functional specification should allow distribution of all subprograms, packages, tasks and objects in the Ada specification. A narrower stand on the objects of distribution requires the designer to be more conscious of the non-functional requirements while searching for the functional solution. While it can be argued that a designer who is aware of all the requirements of an application will produce a more efficient solution, we believe that the tools he uses to produce the distributed solution should impose as few constraints as is possible. Constraints imposed at this level directly impact portability and robustness of a given functional solution in the face of a changing hardware environment.

Many researchers argue that the PRAGMA construct in Ada should be used for non-functional specifications such as distribution of entities [Inv 85, Vol 85]. We have chosen another approach for several reasons. One concern is that an approach involving PRAGMAs will not be extensible to specification of non-functional attributes such as dynamic relocation of objects or fault tolerance strategies. Pragma-based schemes for specifying distribution are complicated already, attempting to extend these schemes to additional domains might prove unwieldy. We also consider it a disadvantage that the pragmas would be embedded in the source and scattered throughout the Ada specification. This

makes sharing of library units between applications difficult or impossible. It also impedes manipulation of the specification of distribution. If this specification were concentrated in one location rather than dispersed throughout the code, it would be easier to form a global picture of system distribution. We also observed that the function to be performed by these notations was to establish a structuring hierarchy distinct from that of Ada. This led us to create a separate specification notation, the Ada Program Partitioning Language (APPL) [Cor 84, Hon 85, Jha 86].

Ada Program Partitioning Language (APPL)

The goal of the APPL design process was to produce a compact, convenient notation for specifying the non-functional attributes of a program. APPL addresses issues of distribution of Ada entities, and dynamic relocation and replication of those entities. Extensions to APPL to cover fault tolerance specification are under consideration. For brevity, this discussion will consider only APPL in general and static distribution in specific. The reader is referred to the APPL Reference Manual for a more detailed and formal description.

It is useful at this point to introduce some terms.

> A FRAGMENT is a user-specified collection of entities, such as packages, subprograms, tasks and objects, from the Ada source program. Every entity belongs to one and only one fragment. Membership in a fragment is attained either implicitly, as a result of default rules, or explicitly, as a result of inclusion in an APPL fragment declaration.
>
> A STATION designates a computational resource in the underlying system. Typically, this is a node in a distributed system.
>
> MAPPING a fragment to a station causes all entities in that fragment to reside on that station at runtime.
>
> A PROGRAM CONFIGURATION refers to a specific partitioning of a program into a collection of fragments, and the specific mapping of the resulting fragments onto stations.

An APPL specification completes a Program Configuration and consists of two parts. The first of these, the configuration specification, specifies the fragmentation of the Ada program, while the latter, the configuration body, specifies the mapping of fragments to stations.

The configuration specification provides a mechanism for specifying Ada entities to be bundled together as a fragment. With a few exceptions, such as within unnamed blocks, these entities can be selected from within any declarative region in the program. As a convenience, APPL semantics implicitly declare a fragment for every library unit which make up a program. It also provides a mechanism for further bundling fragments into

fragment groups. Fragment groups, like fragments, are mutually exclusive and are treated like fragments in mapping.

The configuration body is a simple section specifying a correspondence between fragments and stations.

As an example to illustrate the use of APPL, consider the following Ada text.

```
with TEXT_IO, REAL_OPERATIONS; use REAL_OPERATIONS;
package EQUATION_SOLVER is
   procedure QUADRATIC_EQUATION;
   procedure LINEAR_EQUATION;
end;

package body EQUATION_SOLVER is
      .
      .
      .
end EQUATION_SOLVER;

with EQUATION_SOLVER;
procedure MAIN is
begin
      .
      .
end MAIN;
```

Also consider the following configuration specification.

```
with MAIN, EQUATION_SOLVER, REAL_OPERATIONS;
configuration PROTOTYPE is
   fragment QUAD_EQUATION is
      use EQUATION_SOLVER;
      procedure QUADRATIC_EQUATION;
   end QUAD_EQUATION;
end PROTYPE;
```

Recall that APPL implicitly declares a fragment for each library unit involved. Thus the implicitly declared fragments are: MAIN, EQUATION_SOLVER, TEXT_IO, and REAL_OPERATIONS. QUAD_EQUATION is an explicitly declared fragment containing the procedure QUADRATIC_EQUATION from the library unit EQUATION_SOLVER. An example configuration body is shown below.

```
configuration body PROTOTYPE is
   map EQUATION_SOLVER, MAIN, TEXT_IO onto STATION_1;
   map QUAD_EQUATION onto STATION_2;
   map REAL_OPERATIONS onto STATION_3;
end PROTOTYPE;
```

An APPL specification, together with the Ada source, constitute a description of a distributed software system. It is the function of the tools we are developing to actually produce this system.

Distributed Ada Tools

In order to avoid spending a large amount of development time on issues not strictly related to distributed systems, we have chosen the approach of modifying an existing Ada language system rather than creating one from scratch. Two major tools in any Ada language system are the compiler, which maintains the Ada program library and produces object code for strings of compilation units, and the linker, which must determine and generate code for library unit elaboration and actually assemble the final executable image. In the compilation environment, these tools are the most drastically affected by retargeting to a distributed environment.

Modifications to the compiler are perhaps the most dramatic. Obviously, the compiler must be made aware of the fragmentation and mapping specified by APPL. Therefore, the first phase of distributed compilation consists of modifying the intermediate representation (DIANA, for our purposes) of the Ada library units and their secondary units by adding a "fragment" attribute to the DIANA nodes. This allows the compiler to determine the station of residence for that entity.

From the modified DIANA representation of a compilation unit, a linearizer generates intermediate language (IL) code for the compilation unit. This linearizer, in particular, must be significantly more complex than it is required to be in a uni-processor compiler. It must now produce an IL code module for each of the stations to which fragments of the compilation unit have been mapped. Of even more significance, is the fact that it must generate proper code to reference entities on remote stations. This task is simplified somewhat, because the problems associated with distributed Ada tasking will be dealt with by the runtime environment (discussed in the next section) and will be invisible to the compiler. However, most every other aspect of IL generation is affected. Fetches and stores to remote variables, for example, will require calls to special runtime primitives for remote data access.

As another example of the issues involved in this stage, consider the problem of parameters to a remote subroutine call. In a uni-processor system, it is efficient to pass large parameters by reference rather than by value. A pass-by-reference mechanism could be employed in remote subroutine calls by adding a station address to the parameter address. But this would mean that every reference to the parameters of a procedure that could be called remotely would involve the remote data access mechanisms. Since parameters are likely to be heavily utilized in computation, this appears to be an undesirable situation. Our solution to this problem involves the generation of local 'stubs' whose purpose is to package the parameter values and transmit them to the remote system. The runtime environment on the remote station will disassemble the package and call the procedure in question. Since this call looks just like any purely local call, the code generated for the called procedure is unchanged. (Note: This mechanism cannot be applied to access types. They must be handled by a reference mechanism similar to that mentioned above.)

Once these multiple intermediate code modules have been produced, object code generation on each of them should continue in a fairly normal manner and the final object files can be passed on to the linker. In the Distributed Ada environment, our scenario involves the production of multiple executable images, one for each station in the system. This will require modifications to the linker, which will have to resolve symbols between multiple executable images, something which no uni-processor linker would ever have to do. Fortunately, these linker modifications are not conceptually difficult and represent only an engineering problem.

Runtime Environment

The execution environment considered consists of a network of stations and a copy of the runtime system on every station. The runtime system makes the underlying hardware appear to the distributed application as an Ada virtual machine.

There is a minimum set of facilities that must be provided by the distributed runtime system, independent of the granularity with which an Ada program is partitioned. It must provide reliable inter-station communication and synchronization, a consistent view of distributed state information at each station, a globally consistent view of time, and means to deal with partial failures in the underlying system.

The overall complexity of the distributed runtime system depends on the support it provides for binding the application fragments together dynamically, for making the application fault-tolerant by masking station and network failures from it, and for representation conversion between heterogeneous stations.

There is a spectrum of possible binding times. If binding is done statically before execution time, it is not possible to reconfigure an application during execution by remapping one or more of its fragments. Dynamic binding is the most flexible. The mechanism can be used effectively by the runtime system to reconfigure an application as a means of providing fault-tolerance, or of changing the configuration as resource requirements change during execution. If the underlying system is heterogeneous, the binder must also insert representation conversion filters for values that are passed between the remote fragments that it binds together.

The complexity of the runtime system is only marginally affected by the choice of the set of Ada entities that can be distributed. The apparent similarity between concurrency in Ada tasks and concurrency of execution on a network of processors may initially suggest that tasks be made the unit of distribution. However, a close examination will quickly show that this restriction does not really simplify the runtime support needed.

The allowed granularity of partitioning has a greater impact. For the sake of an example, consider the case where Ada library units are the unit of distribution. The runtime system must support calling of remote subprograms, reading and writing remote data, and tasking operations on remotely located tasks. Since Ada task dependencies do not cross library unit boundaries, the semantics for task termination can be implemented in a manner that gainfully uses the knowledge that the task dependencies cannot cross station boundaries. This simplification is not available if a finer granularity of partitioning is allowed. In application areas where the size and efficiency of the runtime system are critical, we think that the specific requirements of the application domain should be taken into consideration when deciding the granularity of program partitioning.

Project Status and Plans

Honeywell's Distributed Ada project was started in 1982. A preliminary version of APPL was defined in 1983 [Cor 84]. A prototype implementation based on source-to-source transformation and an unaltered uni-processor compiler was built during the following year. In 1985, the structure of APPL was changed and the language revised and formalized [Hon 85, Jha 86]. Current development focuses on creating the specialized tools and runtime environment described above. In order to manage the implementation, we have divided its development into several stages. Phase 1, which we are currently working under, calls for a fully functional system, limited to homogeneous systems and static distribution of objects declared in the visible portion of library units (and the units themselves). We hope to complete this phase of development by the end of 1986. We are

using the VERDIX Ada Development System (VADS[2]) as the baseline compiler from which to create the Distributed Ada system and are operating in a simulated network environment using processes under Unix[3]. Future development phases call for support for heterogeneous systems, dynamic reconfiguration, object replication and fault tolerance.

References

[Cor 84] D. Cornhill, "Partitioning Ada Programs for Execution on Distributed Systems," **IEEE 1984 Proceedings of the International Conference on Data Engineering.**

[Hon 85] "Honeywell Distributed Ada Project," 1985 report.

[Inv 85] P. Inverardi, F. Mazzanti, and C. Montangero, "The use of Ada in the design of distributed systems," **Ada in Use Proceedings of the Ada International Conference**, Paris, 14-16 May, 1985.

[Jha 86] R. Jha, J.M. Kamrad, D. Cornhill, "Ada Program Partitioning Language: A Notation for Distributing Ada Programs," submitted for publication.

[LRM 83] "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A, U.S. Department of Defense, 1983.

[Sch 81] S. Schuman, E.M. Clarke, and C. Nikolau, "Programming distributed applications in Ada: A first approach," **Proceedings of the 1981 International Conference on Parallel Processing.**

[Sof 84] Softech, "Programming distributed applications in Ada," December 1984.

[Vol 85] R.A. Volz, T.N. Mudge, A.W. Naylor, and J.H. Mayer, "Some problems in distributing real-time Ada programs across machines," **Ada in Use Proceedings of the Ada International Conference**, Paris, 14-16 May, 1985.

[2]VADS and VERDIX are registered trademarks of the VERDIX Corporation.

[3]Unix is a registered trademark of AT&T Bell Labs.

An Ada[1] Implementation of the Network Manager for the Advanced Information Processing System

Gail A. Nagle
Technical Staff
The Charles Stark Draper Laboratory
555 Technology Square
Cambridge, Massachusetts 02139
(617) 258-2238

Introduction

The Advanced Information Processing System (AIPS) is a data processing architecture designed to meet the reliability requirements of space vehicle applications. The Charles Stark Draper Laboratory is presently building an AIPS proof-of-concept prototype[2]. Ada was selected as the programming language in which major system services would be implemented. One part of the AIPS architecture is a fault tolerant input/output network which is under the control of a software module called the Network Manager. Ada provides a user with a significant number of options for implementing a given aspect of a design. During the development of the prototype Network Manager, some language constructs were found to be particularly well suited for certain types of situations. In one case the language did not provide a desired feature. Experience with Ada as a programming language for this application will be described here.

Background

Using Ada

Training in Ada was accomplished by a combination of viewing a subset of video taped tutorials presented by Jean Ichbiah, Robert Firth, and John Barnes, participation in an in-house course in Ada using the Grady Booch text Software Engineering in Ada, and a lot of "learning by doing". Initially the work in the in-house course and the "learning by doing" were somewhat impeded by the absence of a reliable in-house compiler which supported full Ada. This problem was greatly reduced by the timely arrival of the Digital Equipment Corporation's (DEC) Ada compiler and development system for the VAX[3].

The microprocessor used in the prototype system is the Motorola 68010. Since a compiler which handled full Ada was not available for this machine, it was decided that initial design and development of programs would be done on the VAX using the DEC Ada compiler. This strategy was based in large part on the portability of Ada code and the fact that Ada compilers which target the 68000 microprocessor were expected to be available well within the development time of the

---

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

2 This work is supported by NASA under JSC contract NAS9-17560.

3 VAX is a registered trademark of the Digital Equipment Corporation.

prototype system. Thus progress in designing and programming the various modules could continue unimpeded by artificial constraints in the language.

The AIPS System

The AIPS architecture is highly modular. The needs of a specific application can be met by selecting components from a set of hardware building blocks and software system services.

One such building block is a fault and damage tolerant input/output network which allows a data processing element (typically a Fault Tolerant Processor or FTP) to communicate serially with I/O devices. The network consists of a number of full duplex links that are connected by circuit switched nodes to form a conventional multiplex bus. In steady state, the network configuration is static and the circuit switched nodes pass information without the delays associated with packet switched networks. Since not all pathways are enabled, the network has a set of spare links which allow it to be reconfigured in response to a failure. A network may serve only one processing element or it may be shared by several processing elements which contend for access to the network. In the case of a network dedicated to one processing element, a unique network configuration is possible. Such a network may be divided into subnetworks which allow an application to conduct simultaneous I/O operations with redundant, parallel devices from each subnetwork. Network organization and operation is completely transparent to an application running on the system.

The system service which is responsible for the reliable operation of an I/O network is the I/O Network Manager. The Network Manager can be run in any processing element connected to the physical network to be managed. It performs network initialization, fault detection and isolation, reconfiguration to a fault free state, testing for latent faults and status reporting.

High level design objectives of the network manager software for the prototype include transparency to network users, adaptability to dynamically changing system configurations, portability within the system, and modularity. Ada language constructs have been found which support these design goals. A full Ada version of the design has been compiled and run on a VAX 8600 using DEC's Ada compiler. To facilitate testing on the VAX, an Ada simulation of the network has also been developed. Installation of the full Ada version on the AIPS Fault Tolerant Processor must await the release of a compiler which targets the Motorola 68000. However, a modified version of the network manager has been compiled on the VAX using the Telesoft 1.5 cross compiler and is awaiting system test and integration.

Implementing the Network Manager in Ada

Overview

The number of Network Managers which a system needs depends on the number of physical networks in use. This number can vary from system to system and within a system over time. However, the number of networks which can be managed from a given processing site is bounded by the number of physical I/O interfaces it has. For the prototype system this upper limit is six. Furthermore, when a network is partitioned into subnetworks, each partition requires its own I/O interface. Thus a given processing element could manage at most six networks and/or subnetworks. From the point of view of the Network Manager, there was no functional distinction between the control of a network and the control of a partition .

The Network Manager is a system service which would be provided on demand of the System Manager. The System Manager is another software module which coordinates all other

System Services. The software for an active Network Manager process would consist of two major parts: a data store describing the topology of the network to be managed and the coded algorithms to provide the functions described above. Specific information about the network topology (e.g. the number of nodes and links in the network) would not be available until run time. Thus two factors motivating the design were the need to be able to start and stop the process on demand, and the ability to manage a network topology which is to be determined at run time.

The fact that several networks could be managed in parallel from a given processor required a non-reentrant module to coordinate the starting and stopping of the various manager processes. However, each manager was itself an atomic unit, requiring only information about the topology to be managed for it to be off and running on its own. Thus the Ada package was used to implement the system service of network management on a particular processor. The Ada task type was chosen to conduct the logic of managing a particular network. Other Ada packages were used to coordinate access to information about the various network topologies in the system and to encapsulate the data format required for communication with the prototype network nodes. Finally, the need for keeping the System Manager apprised of the status of network components was met by another task type which provided mutually exclusive read/write operations to a protected object containing current status information. The relationship among these various components is graphically depicted in Figure 1.

Major Features of the Implementation

package IO_NETWORK_MANAGER

This package provides the capability to manage the fault tolerant network defined by the AIPS architecture. A user, in this case the System Manager, can then start or stop management of any network in the system. The software for this module would need to be resident in each processing site which could in fact manage a network.

The visible interface to this package is composed of two procedure calls, *START* and *STOP*. The calling process first designates the definition (i.e. the topology) of the network to be managed through its interface to the data base package. It then calls the *START* procedure. When this call completes, network management is underway and network status is available. The call to *STOP* is also preceded by a call to the database to designate the network to be stopped. When the call to *STOP* completes, management of the indicated network is terminated and all resources allocated to that process are restored to the system. Thus network status is no longer available for that network.

task type NETWORK_MANAGER

A task object is created in the body of *IO_NETWORK_MANAGER* for each I/O network to be managed from a particular FTP. If a network is partitioned into a number of subnetworks, each subnetwork will be allocated its own manager task.

The concept of a partitioned network was devised to allow applications to conduct I/O operations with redundant, parallel devices resident in separate partitions. Within each subnetwork are a certain number of spare links which allow failures to be repaired intrapartition. While such a repair is taking place, communications on the other subnetworks can operate normally. To support this feature, management of the I/O networks is not conducted synchronously. Each partition is under the control of its own task object which performs its functions independently of the other subnetworks.

Since the number of possible networks which a given processor can manage is known in advance and is a relatively small number (currently six), a table of access types to these task objects is declared within the package body. The *START* and *STOP* procedures described above have access to this table. The task object has three entry calls. Not surprisingly they are

start, stop and *start_status*. During the *start* rendezvous, the task object makes a local copy of its network definition. During the *start_status* rendezvous, the network manager task initializes the protected status object. This rendezvous is also used to synchronize the two processes which can access this shared status object; i.e. the status reader will not be able to read until the status writer has written at least once. The task proceeds to "grow" a network. It then enters a loop whereby it will either accept an entry call to *stop* or will periodically monitor the network for faults. If faults are detected during monitoring, fault isolation and reconfiguration logic is activated. An alternate approach to the monitor-maintain cycle currently under consideration would provide this activity on demand when communication errors are detected in communications conducted on the network for application functions. The call to *stop* causes the process to exit its loop and come to its natural end at which time its resources are explicitly deallocated.

package IO_DATA_BASE, package NODE_MESSAGE_FORMATTING and other data structure considerations

The numbers of various network elements , i.e. nodes, links, I/O devices,etc., can vary from network to network, but within a given network topology, they are static. The first approach to the data abstraction process focussed on defining types to contain network topology information. The basic connecting unit of a network is a node. The AIPS prototype node has five ports. Each port may be connected to another node, a processor interface unit or an I/O device interface unit. Hence information about the element adjacent to a given port could be contained in a discriminated record where the information stored would depend on the type of that element. Five such records grouped as an array could make up one field of a larger record containing other information about the given node. Finally, a collection of these node records would define a topology for a given network. This collection was also housed in a discriminated record where the discriminant was the number of nodes (which was given a default value) and the other field was an array containing that number of node records. This structure has the additional feature that objects of this type could be declared within the network manager task type and would upon allocation of the task object have the default value number of nodes. Later this object could be updated to reflect the actual number of nodes in the network to be managed. A major strength of this approach was that of run time reliability. The compiler generated checks will ensure the correct usage of this structure, i.e. the user cannot access a portion of the structure where values are meaningless. A simple array that is large enough to hold data for any case could be misused in this way. However, the major drawback to this design was that each object so declared was allocated enough memory to hold as many members as the maximal value of the type of the discriminant.

A second design solved this problem of wasted memory space while retaining the ability to dynamically create array objects with the correct number of cells. This design used an access type to an unconstrained array type. A variable of this type is declared in the body of the task type. The number of nodes and a pointer to an array of node records are passed as rendezvous parameters to the activated task. During the rendezvous, the object accessed by the local pointer is allocated with as many cells as there are nodes in the network. These cells are assigned values by applying the 'all' construct to the local access variable and the rendezvous parameter. The only feature that is lost with this solution is the ability to later change the number of cells in the object. Since this network topology is constant for the lifetime of the task, this feature is not necessary here.

The discriminated record array structure did prove useful in another application. Since the network is a shared resource, the various processing elements using the network must contend for access. To reduce the overhead of the contention processing, a set of messages are grouped together in what is called a "chain". Messages are sent to nodes in chains. However, the number of messages to be sent to the nodes will vary with the reason for the communication. Thus the number of messages in a given chain will vary. For example, when monitoring the network, all

the nodes are sent messages. When growing the network, only one or two nodes are sent messages. When reconfiguring the network or testing spare links, it may be necessary to send messages to several nodes in one chain so as not to leave the network in an inconsistent state for

other network users before completion of the reconfiguration or test. Thus objects containing node messages will vary in length during the life of the task. Rather than create an object for each possible length chain, an object of the discriminated record array type was used. In this situation,the cost in extra memory is relatively small since each node message is only six bytes long and the prototype network may contain at most thirty-two nodes; however, the extra flexibility facilitates processing.

An operation provided by the data base package allowed a significant reduction in the memory needed to store topology data as well as the need to ensure that multiple copies of data remain consistent. Any FTP connected to a network can manage that network. The definition of the network used by a manager is the same regardless of the processing site except for the particular nodes (called root nodes) which connect the site to the network. Given the array of node records described above and the identity of the FTP, it is possible to derive the root node information. Thus network definitions can be stored centrally without regard for local variations which are derivable on demand.

A final Ada feature which proved useful in the data abstraction process was the representation clause. The prototype node expects to receive a message containing six bytes of data. Each byte in turn contains one or two bit wide fields which the node decodes to obtain its control information. Rather than having to remember that bits zero and one of byte three control whether or not a node is permanently reconfigured or only reconfigured for the next transmission, the representation clause allowed a type called *CONFIGURATION_LIFETIME* to be given two values, *ONCE_ONLY* and *PERMANENT*, with specific base two representations. The representation clause further allowed the node message type to be assigned to a specific two bit wide field for the lifetime information. Other fields in this record were named and positioned in a similar fashion. The programmer need not be concerned with masking and shifting to set up a node message. Code using these messages could be written more quickly and more reliably. Furthermore, the code becomes self-documenting and therefore easier to test. When the message needs to be stored in a general area of memory, unchecked conversion would allow the safe transfer of the byte organized information. This is the case when the message is written to a dual ported memory just prior to transmission on the network. Finally, this node dependent information was packaged as a unit which would shield the rest of the software from any necessary design changes in node hardware or protocol.

A Perceived Shortcoming in the Language

Ada currently does not allow a function to accept 'in out' parameters. While this makes sense in the context of a mathematical function, in the context of a computer program, a broader definition of 'function' can be supported. In this context, a function is a language construct that does something and returns a value as part of its call. In the network manager such a language feature would have been a great asset in conjunction with the short circuit 'and then' construct. During growth of a network, a node is subjected to a series of tests before it is formally added to the network. These tests are sequential in nature. If a node fails a test in the sequence, the remaining tests are doomed to fail and therefore need not be performed. A very elegant way of coding this testing sequence was:

```
if PASS_TEST_1
   and then PASS_TEST_2
   and then PASS_TEST_3
   and then PASS_TEST_4
 then ACTION;
 else OTHER_ACTION;
end if;
```

where the PASS_TEST_Ns are boolean functions. This code is easy to read and understand; it is also self-documenting.

The problem arose because each test needed to log error detection information as it was discovered. Since a global object was not desired here, other designs were examined. These included procedure calls for the tests within nested if then else statements and the calling of these procedures from functions declared locally within each subprogram performing the tests. However, none of these designs were so simple, straightforward or self-documenting as the original. It is hoped that this example will provide some additional motivation for a change in this restriction. Perhaps another type of subprogram would be the most acceptable solution.

Conclusions

From an implementation standpoint, the Ada language provided many features which facilitated the data and procedure abstraction process. The language supported a design which was dynamically flexible (despite strong typing), modular, and self-documenting. Adequate training of programmers requires access to an efficient compiler which supports full Ada. When the performance issues for real time processing are finally addressed by more stringent requirements for tasking features and the development of efficient run-time environments for embedded systems, the full power of the language will be realized.

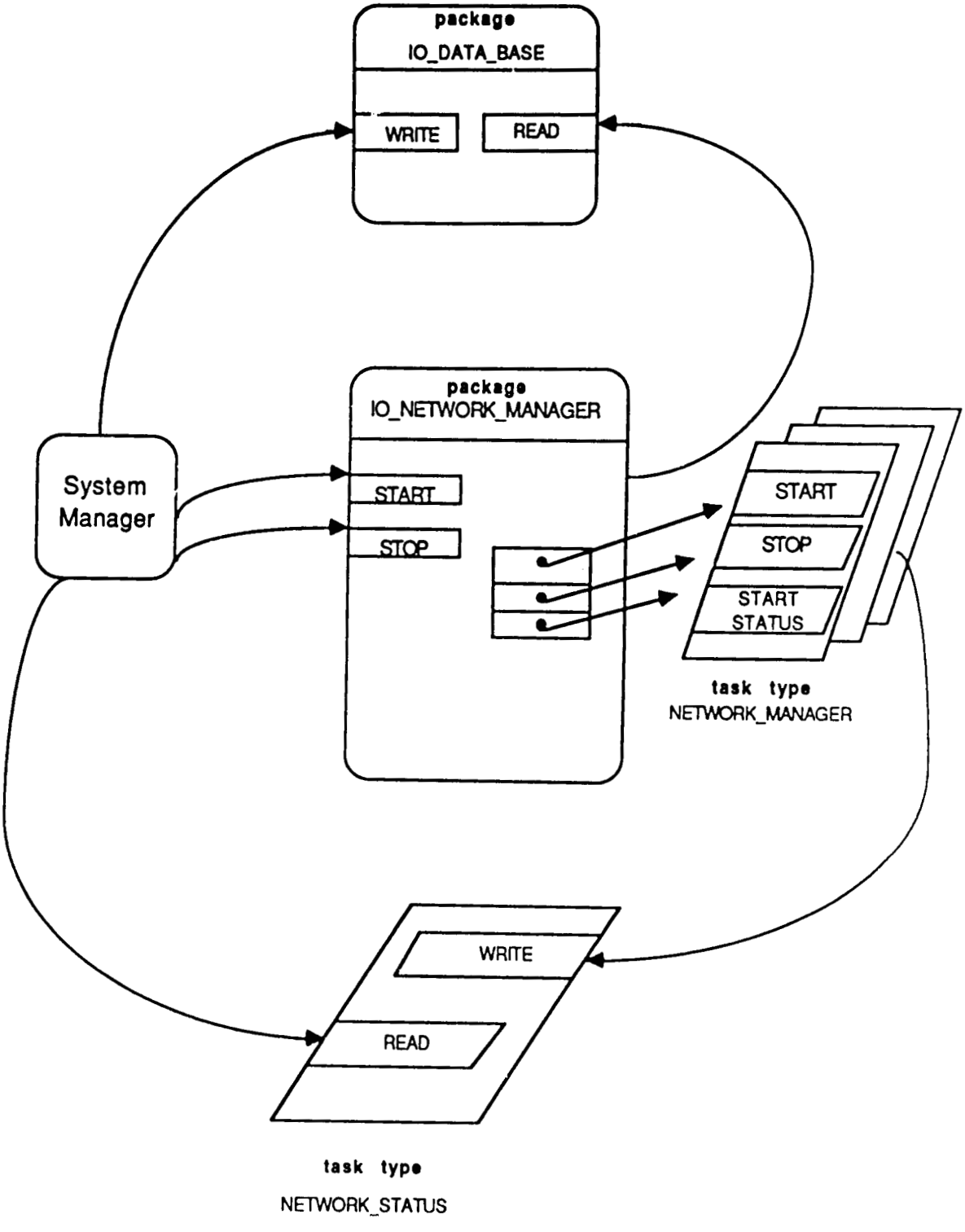


FIGURE 1: SOFTWARE COMPONENTS FOR MANAGING NETWORKS

DISTRIBUTING PROGRAM ENTITIES IN Ada[1]

Patrick Rogers
Charles W. McKay
High Technologies Laboratory
University of Houston
at Clear Lake

Introduction

In any discussion of distributing programs and entities of programs written in a high order language (HOL), certain issues need to be included because they are generally independent of the particular language involved and have a direct impact on the feasibility of distribution. Of special interest is the distribution of Ada program entities, but many of the issues involved are not specific to Ada and would require resolution whether written in Pascal, PL/1, Concurrent Pascal, HAL/S, or any language which provides similar functionality. The following sections will enumerate some of these issues, and will show in what ways they relate to Ada. Also, some (but by no means all) of the issues involved in the distribution of Ada programs and program entities will be discussed.

Justification

Before introducing such a subject, it is perhaps reasonable to provide a rationale for distributing any named resource of a HOL program in the first place. The reasons are straight-forward.

First, and probably most important, is the issue of reliability. Computers are increasingly used in applications which require high reliability, because they impact life and property (sometimes literally). Embedded applications which provide life support, control guidance and navigation, or manage weapons are examples. A failure of such an application can be disastrous. By decentralizing the software (and of course, the hardware), we can provide systems that not only do not have single points of failure, but that are fault-tolerant. Such systems can recover from

---

[1] Ada is a registered trademark of the U.S. Government (AJPO)

failures once they are detected. (This approach should not be confused with fault-avoidance, which attempts to prevent failures from impacting the system in the first place.)

The second reason is that of the decreasing cost of hardware, especially with respect to the ever-increasing cost of software. In order to make the most, economically, of the power of software, utilization of multiple processing resources is desirable. Parallel processing is an example.

The third reason is extensibility, in the domains of performance and functionality. When the software system is designed with distribution as a design criteria, the resulting modularity provides a design that does not necessarily have to be radically changed for increases in processing power (for performance) or for the addition of new modules (for additional functionality). In a system intended to have a long, evolving life cycle, this is a major issue.

Fourth, given limited resources of operational costs, hardware, communications, and information, when those resources are themselves distributed (as in Space Station), resource sharing implies that only those elements that require direct access and are to be held accountable for the integrity of the resource should be located in proximity to that resource. In this case, distribution of the software allows only that part which interacts with the resource to be present (with potential benefits of reduced communications costs and localization of accountability).

The fifth reason is the issue of the fidelity of modelling solutions to real world problems that are distributed in nature. Such problems are complex enough without adding additional complexity by distorting the solution model to fit a non-distributed HOL with no support for cooperating, parallel activities, or for recognizing both exceptions to normal processing and the context in which the exceptions occur (so that appropriate fault tolerance and fail-soft activities can be supported). For example, the Space Station Program will eventually involve ground support stations, free-flying platforms, the Station, orbital transfer vehicles, and other components. These components are intended to interact in an integrated, end-to-end information environment. (Put simply, any authorized user at any component of the environment who desires to access entities should be given timely access to such entities without regard for the location, replication, number of processors supporting the access, or means of providing fault tolerance.) Obviously, a model of the solution to these challenges involves a high degree of distributed parallel processing activities which must evolve

in a cost-effective, adaptable, and safe fashion.

Finally, the issue of performance should be addressed. It, too, is straight-forward. When the application demands the advantages and benefits of distribution, the price of decreased efficiency must be paid. It should be understood, however, that distribution will not automatically mean poor performance. In fact, distribution will in some cases improve performance by decreasing communication costs, taking advantage of remote hardware resources, and so on.

The above reasons should be sufficient for illustrating the need for distributed software. The general issues involved in distribution will follow.

Visibility

One of the primary underlying concepts in distributing a HOL program is that of "visibility". In this context, visibility means "the set of objects which may be potentially referenced at any particular point in a program". These objects include both data and code modules, such as variables and subroutines. Depending on the distribution scheme, these objects may or may not be locally available. In those instances where the object is remote, the Run Time Support Environment (RTSE) will be required to help fulfill the semantic requirements of a given reference. For example, the program may have some of its variables distributed across remote sites. A reference to such a remote object will require cooperation among the two RTSEs. The calling RTSE will have to contact the RTSE of the processing site at which the variable is located, with a request for the current value of the variable. The remote (called) RTSE must locate the variable, get its value, and send back a message containing that value. (The recovery of a failure of one of these messages is non-trivial.)

As can be seen, the visibility of objects plays a considerable part in determining the complexity of the RTSEs involved.

Distribution Scheme

A distribution scheme may often be described in terms of the visibility rules of the implementation language. Traditional block-structured languages, such as ALGOL and Pascal, use nesting to control visibility of locally declared data and subroutines. The visibility rules of these languages are such that the inner declarations of subroutines and data are visible to further nested units in

the same declarative region, but not to outer units at the same nesting level. A global section of data is directly visible, and of course outer-level subroutines are visible to successively declared subroutines at the same level, in a linear manner.

As previously shown, the visibility rules directly impact the complexity of the required RTSE by determining the set of entities that may be referenced at a particular point. This complexity represents a major factor in determining the feasibility of a distribution scheme itself. Those schemes which reflect visibility rules that restrict the size of the name space are easier to implement.

The distribution schemes form a spectrum based on the visibility rules and the constructs of the source language involved. For example, if the distribution is to be at the individual statement level, (representing one extreme), then any object referenced may be remote, including components of complex expressions. (The resulting RTSE requirements would be extensive. The instance discussed under "Visibility" above is an example.) If distribution is to be at the compilation-unit level, (the other extreme), then the set of all entities that may be referenced is reduced to globally visible entities, such as subroutines and their parameters. In effect, the distribution scheme controls the size of the distributable name space, and therefore the complexity of the RTSE.

Time

Another important concept is that of time, either expressed in the program directly, or in the underlying RTSE. The basic problem is that in order to provide correct semantic execution, distributed program units require the same effects as a consistent, unified version of time that would be provided in a non-distributed environment.

As an example of directly expressed timing, if one module requests a service of another remote module, with a specified amount of time allowed for the request to be fulfilled, the two modules must have a common view of time for the request to have any meaning. Note that this does not mean that the two modules' clocks are necessarily synchronized, only that they be mutually consistent while the request is being served.

In the underlying RTSE, certain operations and actions often need to be synchronized with respect to each other for correct operation and support of a source program. This will

also be required in a cooperative manner among the RTSEs supporting distributed programs.

Semantic Integrity

A critical concept is that of semantic integrity, which means that the meaning of constructs and program units must be maintained without regard for distribution. For instance, a call to a subroutine must have the same semantic effect, or meaning, regardless of the routine's actual location with respect to the caller. Note that this does not mean that the behavior is the same, especially with respect to temporal performance. (In other words, it has to work the same, but not necessarily with the same timing and space profile.)

A specific aspect of semantic integrity is that the semantics of a given construct are to be invariant over failures of the processors executing the corresponding object code. For example, the semantics of a subroutine call are such that, once the called routine is completed, execution resumes in the calling module. In a distributed context, in which the called routine is remote from the caller, if the called module's processor fails, the calling module will be suspended indefinitely. The semantics would thus be (incorrectly) different in the distributed environment. Semantic integrity, in this case, means that the caller must not be allowed to permanently suspend, since the semantics of a call do not include that situation. (Obviously, if the called routine is designed to never complete, due for example to an infinite loop, then the caller will never resume. However, that is not a result of the semantics of a subroutine call.) Similarly, if the processor(s) executing outer-level units in a nested structure fail, the inner-level units must not be allowed to proceed normally since they depend on the outer-level scopes for their execution context. This is, again, an issue that may be partially addressed by the distribution scheme, by constraining the units that may be distributed to those at the outer-level.

Resource Management

A more obvious issue than those above is the management of resources. These resources include storage, processors, and information (among others, such as devices). Specifically, storage management involves dynamic, static and temporary data, as well as the management of code (which may also be dynamic).

Processor management involves dispatching potentially remote processors to processes, as well as scheduling, which determines the units that are to be able to execute at a given moment. Both are, of course, requirements of the RTSE.

Information management involves the maintenance of consistent, current status information regarding individual modules' contexts, processing status and workloads, the global program state for each executing program, descriptive information about data and code, and so on.

Different languages have varying degrees of resource management requirements, as well as varying degrees of programmer-level control over them. Thus the amount of RTSE support required varies. For instance, languages which allow the allocation and deallocation of dynamic objects from a heap will require different RTSE support from those languages which have no such capabilities (often intentionally, such as in HAL/S). Some languages have only static data, and thus require different storage management techniques that those which are stack-oriented. In a distributed context, where heaps may be effectively distributed and/or shared, the management of dynamic objects will require specialized RTSE capabilites.

ISA Homogeneity

The Instruction Set Architectures (ISA) of the processors that comprise the target environment are also an issue. If these processors are potentially heterogeneous, target dependencies become a problem. One such dependency is of course implicit in the object code itself, since the machine code was generated for a particular ISA. Also, the source code may contain explicit target dependencies. These could include references to absolute addresses and specific devices, as well as specific data representation requests, and so on.

Furthermore, the default representation of data may vary among ISA's with different capabilities. This difference in representation will be a problem when objects are visible to (two or more) remote modules on non-homogeneous ISAs, as well as when objects are passed as parameters between such modules.

Changes In Situ

In systems which are intended to have a very long, evolving life-span, such as Space Station, changes to the software are inevitable. These changes will occur as a

result of upgrades in technology, and as a result of changing requirements in functionality. The design of the software must, in its initial form, provide for such changes. (Alterations to the design after-the-fact present a much more difficult situation.) Currently accepted complexity-control methods of modularity and information hiding, along with the requirement for changing a system without first halting that system, dictate that separate programs be employed in the construction of the software. Each program is to be distributed as necessary, or not at all. This approach is in contrast to one in which a single, monolithic program is distributed across the network(s).

Issues in Distributing Ada Programs & Program Entities

Justification for Selecting Ada

Provably Correct Constructs

Older HOLs were designed in an era of single monolithic processors that were typically expected to execute programs that were small (by current standards), and that were developed by one programmer. The three oldest high order languages, FORTRAN, LISP, and COBOL, were developed (in 1957, 1958, and 1959, respectively) before the development and wide recognition of the concepts of building "structured" software from a small set of provably correct constructs. Thus it is understandable that natural reenforcement for consistent use of such constructs is lacking. In fact, those who use early languages in building solution models for many of today's complex problems often find themselves penalized for such use. In contrast, the Ada language provides direct support for developing solutions to large, complex problems that are demonstrably correct, maintainable and adaptable.

Support for Parallel Activities with Fault Tolerance

These early languages are called sequential because they have no support for modelling concurrent or parallel actions. Additionally, they provide support for normal processing only, with no means for expressing the response to run-time errors. Again, The Ada language provides direct support for such activities. To distort the solution model with such a language as FORTRAN or Pascal would require extensive programming in assembly language and use of operating system calls in order to compensate for the inadequacies of the language. The resulting software system

would be too expensive to build, much more difficult to maintain and operate, and far more difficult to adapt to changing requirements. Similarly, to distort the solution model by failing to support distributed program entities, as well as distributed programs (when appropriate), would be to add rather that to reduce complexity, since the resulting model would be far less representative of the problem.

Distribution Scheme

The central theme in the following discussion is that of the distribution scheme. As demonstrated, its control over visibility has a considerable impact on the complexity of the underlying RTSE, and thus the feasibility of distribution. In Ada, the spectrum of distribution begins with constants and variables, continues to nested program units (blocks, subprograms, packages and tasks), and ends at the other extreme of compilation units. (It should be noted that Ada provides greater control over the name space via packages.) Compilations units in this case would be Ada's "library units": specifically, subprograms and packages. At this level, the only visible entities are these library units, parameters for these units when they are subprograms, and declarations in the visible parts of library unit packages. Distribution at this level is the easiest to support. Distribution at the nested program unit would limit some visibility, (i.e., the declarations local to nested units), but not globally visible data and routines. Thus it would not result in less RTSE complexity. Obviously, the simpler the requirements for the RTSE the better, since the implementation of distribution support is simpler.

However, other factors besides RTSE complexity must be considered in the choice of distribution level support. Specifically, the amount of fault-tolerance required must be seriously considered. If little fault-tolerance is required, the system may be allowed to deal with it transparently (in very deterministic ways), such that the programmer is not directly involved with the response to failures. As such, the programmer has no need to express aspects of distribution dynamically in the source language. However, in some applications only the programmer can know what is to be done in response to failures. The appropriate response may be a specific reconfiguration of the program units involved. Since the only dynamic program unit is the task, the distribution scheme may have to support distribution of tasks in order for the programmer to specify the reconfiguration.

Time

The concept of time in Ada may be expressed explicitly in several ways, based on the delay statement. An example of the need for consistency across remote units is, of course the timed entry call, which requests a service to be provided to the caller in a specific amount of time. If the server is to respond meaningfully, it must perform the request for rendezvous in the amount of time indicated by the call. However, since the clocks of the two processors will not be synchronized, and there will be an indeterminable communication lag, difficulties will exist. Specifically, the server may respond too late, such that the caller will have timed-out and continued on as if the service was never provided. If not handled by the RTSE, the program would then be in a logically inconsistent state.

An example of timing issues in the underlying RTSE is the activation of remote tasks. The parent task must not begin execution until all tasks declared in its declarative region are successfully activated. If one or more of these activations fail, then Tasking_Error must be raised in the parent.[2]

Another example is the elaboration of the library units named in the context clauses of a main (sub)program. These must be elaborated in an order that is consistent with the transitive dependencies. As a result, distributed library units cannot simply be elaborated when the remote host site is ready. Rather, there must be communication and cooperation among the sites.

Semantic Integrity

Ada subprogram calls will exhibit the behavior described under the general section on "Semantic Integrity" with respect to failure of the called unit (i.e., they too will not return). Furthermore, an entry call will exhibit those same characteristics when the processor supporting the called entry fails. Conditional and timed entry calls can protect the caller from permanent suspension prior to the start of the rendezvous. However, these calls do not protect the caller once the rendezvous has begun.

---

[2] Note that in a distributed context, the activation status messages may be lost. The resulting indefinite suspension of the parent would be an example of failed semantic integrity.

It should be noted that in a distributed execution environment, the conditional entry call is not the same as a timed entry call with a zero delay. The reason is as follows. In the Language Reference Manual (LRM)[3], the phrase "immediately possible" in the discussion of the conditional entry call refers to the readiness of the called task to accept the call, (not to an amount of time). The conditional caller is dependent upon the called task to indicate whether or not it can accept the call. If not, the caller will resume under the "else" part of the call. If the called task indicated that it could perform the rendezvous (resulting in the caller being suspended), and then failed, the caller would be indefinitely suspended (unless fault tolerant programming techniques are applied). This is not the case with a timed entry call. Under a timed call, the caller is not dependent on the called task. (The caller does the timing.) If the call is not performed in the specified delay, then the caller continues on, without regard for the status of the called task. Thus, the semantics are not the same.

Resource Management

Distributed Ada will require all the resource management activities outlined in the general section on resource management, and specifically those for a stack-oriented language. One aspect that has received attention is the subject of dynamic data, supported in Ada by the "access type". Some implementations of distributed Ada restrict parameters such that values of access types are not passed between remote program units.[4] This is an expedient approach, but not an absolutely necessary one. In Ada, dynamic objects are referenced as abstractions, which is why they are called "access" types rather than "pointer" types. The value gives "access" to the dynamically allocated object. This is of course typically implemented (on uniprocessors) as an actual address. The common reaction to distributing access types is then that such distribution is not possible. However, in keeping with the abstraction concept, in passing an access value to a remote site, rather than passing an address which will be meaningless to the remote site, a "token" should be passed which uniquely identifies the dynamic object. The identifier will have to

---

[3] Ada Language Reference Manual, ANSI Mil-Std-1815A, Section 9.7.2

[4] A Feasibility Study to Determine the Applicability of Ada and APSE in a Multi-microprocessor Distributed Environment (Final Report, March, 1983) TXT, CISE, SPL

be unique over the entire target environment, and may be passed at will among distributed units.

ISA Homogeneity

Ada programs will have the same problems of data representation that any HOL program would, when the processors comprising the target environment are heterogeneous. These problems will be exhibited when global objects are referenced by two or more remote program units on different ISAs, and when parameters are passed between such program units via subprogram and entry calls. The specific incarnation of the problem is package Standard, which logically encloses the units comprising a program. (Package System is also a problem to a lesser extent.) Package Standard defines type Integer, Float, Character and so on, for an entire program. The question then is how different ISAs can efficiently represent those common types.

One approach is to resort, in all cases, to representing passed data at the level of the common denominator: type String. This is considered too extreme, since not all communicating program units will be on heterogeneous processors. However, the concept of a common format, a "canonical data format", may be the most expedient approach. A promising alternative is the concept of "self-defining data structures", in which the passed data includes a description of its representation.

Changes In Situ

As stated in the general section, changes to the software in a system with an long, evolving life cycle will be required. It may often be the case on Space Station that the subsystem being changed is critical and cannot be stopped in order for the changes to be installed. Also, good design, maintenance aspects, and the sheer volume of software involved mandates that multiple Ada programs be utilized in the construction of the software system. This is not in conflict with the LRM, although a casual reading might imply that the LRM requires only one program to be "in existence" at a time. Nothing in the LRM has been found to require such a restriction.[5]

Each program would be distributed if the requirements dictated that approach. Each would be only as distributed as

---

[5] The issue of multiprogramming is (appropriately) not addressed in the language reference manual.

necessary, to reduce the costs of distribution support. Furthermore, if the RTSE is constructed in a layered, modular fashion, those programs not requiring distribution support would not pay an overhead penalty since the RTSE would be configured to the minimum support necessary. A non-distributed program would then be supported by a traditional configuration of runtime support services.

Although the details of supporting the integration of a new subsystem without first stopping that subsystem are not clear, it is felt that such an activity is impossible if separate programs are not employed.

Conclusion

As shown, many of the issues in distributing Ada programs are common to distributing any high-order language. The distribution scheme, because of its impact on the underlying RTSE complexity, should be carefully chosen when implementing distribution of the language. In making the choice, special consideration must be given to the amount of fault-tolerance required, and the level of programmer response. In Space Station, such issues will be critical.

Bibliography

A Feasibility Study to Determine the Applicability of Ada and APSE in a Multi-microprocessor Distributed Environment (Final Report, March, 1983) TXT, CISE, SPL

American National Standards Institute
Reference Manual for the Ada Programming Language
ANSI/MIL-STD-1815A-1983

Cornhill, Dennis
A Survivable Distributed Computing System for Embedded Application Programs Written in Ada
Ada LETTERS, vol. 3, no. 3, pp. 79-87

Cornhill, Dennis
Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets
Proceedings of the IEEE Computer Science Conference on Ada Applications and Environments, St. Paul, MN (Oct. 15-18, 1984) pp. 153-162

DeWolf, Barton, Nancy Sodano, Roy Whittredge
Using Ada for a Distributed Fault-Tolerant System
Draper Labs Report No. CSDL-P-1942 (dated Sept. 1984)

Dapra, A., S. Gatti, S. Crespi-Reghizzi, et al
Using Ada and APSE to Support Distributed Multimicro-processor Targets
Ada LETTERS, vol. 3, no. 6, pp. 57-65

Gehani, N. H.
Concurrent Programming in the Ada Language: the Polling Bias
Software Practice and Experience, vol. 14, no. 5 pp. 413 - 427

Grover, Vinod, and Reuben Jones
Programming Distributed Applications in Ada
SofTech, Inc. Report No. 9076-3 (Dec. 1984)

Knight, John C. , and John I. A. Urquhart
On the Implementation and Use of Ada on Fault-Tolerant, Distributed Systems
Ada LETTERS, vol. 4, no. 3, pp. 53-64

Rossi, G. F., and Zicari, R.
Programming a Distributed System in Ada
Journal of Pascal and Ada, Sept/Oct 1983

A Distributable APSE

S. Tucker Taft
Intermetrics, Inc.
733 Concord Ave.
Cambridge, MA 02138

for: The First International Symposium on Ada for the NASA
Space Station, June 2-6 1986
Nassau Bay Hilton Hotel
Houston, TX

1. Introduction

A distributed Ada(r) Program Support Environment (APSE) is one in which programmers, managers, customers, testers, etc., may work on separate computers, linked by a high-speed network. It also may imply that program development proceeds in a series of relatively independent subsystems, which are then combined into larger Ada programs as part of final integration. (This reminds one of the frequent similarity between the structure of programs and the structure of the organizations that build them.)

This paper will discuss an approach to the implementation of a distributed APSE which provides for parallel development on separate computers while sharing "catalogs" of compiled units, but avoiding global locking or naming bottlenecks.

2. The Ada Program Library

Ada as a language is somewhat unusual in that a "program library" must be maintained across separate compilations, holding compiler- produced information necessary not only for later linking, but also for later compilations. To support a distributed APSE, it is essential that the Ada program library may itself be "distributed," because it is too expensive in disk space and/or compile-time to maintain on each computer a copy of the entire program library.

Even on a single computer, there are reasons to "distribute" the Ada program library. As defined in the Ada Reference Manual (ARM 10.4) the program library holds the "universe" of compilation units available for "WITH" references at compile time, and for eventual linking into an Ada program. Conceptually at least, the library includes all the language-defined packages, such as TEXT_IO, CALENDAR, etc. These by themselves represent a major investment in compile-time and disk space, and most Ada compilation systems have devised some way to share such compiled packages across program libraries.

2.1 Program Library as Network of Catalogs

As a generalization of sharing language-defined compiled packages, we have defined a conceptual Ada program library as a set of interconnected "catalogs," some of which may be connected into other program libraries as well. Each catalog holds a set of (compiled) Ada compilation units represented in a DIANA form, as well as a more conventional object-module form. A conceptual library is constructed from a read/write "primary" catalog plus links to a set of read only "resource" catalogs.

Every program library must provide the language-defined packages, which in our case are

gathered together to form the "RTS" (run-time system) resource catalog. A typical large program might have a series of other resource catalogs for utilities, like a DBMS catalog, MATH catalog, a DEBUG catalog, etc., plus one catalog for each major subsystem.

Each resource catalog is actually part of a set of revisions. Two revisions may share some of their compiled units, and differ in others. We therefore provide for both sharing of compiled units across different program libraries, as well as across revisions of the "same" conceptual program library.

3. The HIF

To support this distributed program library structure in a host independent way, we have defined a standard Host Interface (HIF) to a (distributed) database system. The Hif database is organized as a set of "nodes", partitioned by "Hif user" (where a Hif user maps to a user or sub-project on the Host system). There is a "top-level node" associed with each Hif user, analogous to the "home directory" of a conventional file system.

Hif nodes have string-valued attributes, and relationships from one node to another. The relationships are uni-directional, meaning that they can be viewed as directed arcs in a graph of nodes. A subset of the relationships, called the "primary" relationships, form a strict tree reaching every (non top-level) node by exactly one path. The "secondary" relationships form an arbitrary graph.

3.1 HIF Node Kinds and Partitions

Two kinds of HIF nodes exist: structural and file. File nodes have a host file associated with them (typically containing the DIANA or OBJMOD representation of an Ada compilation unit), while structural nodes serve only as connectors between other nodes, and as carriers of attributes.

The subtree of nodes beneath the top-level node associated with each HIF user, plus all of the host files associated with these nodes form a partition of the HIF database. The information necessary to represent a user's partition is gathered into a single host directory. The node-structure database is represented by 3 files: a B-tree of nodes, a hash-table of relation/key/attribute identifiers, and a heap of attribute values. The file-node host files are assigned HIF-generated names in the host directory.

3.2 Program Library Implementation via the Hif

The program library is implemented using Hif nodes, taking advantage of the partitioning by Hif user. The set of revisions of a resource catalog, plus all of the compiled units included in one or more of the revisions, are combined into a single Hif partition.

In addition, some number of primary catalogs may coexist in the same Hif partition. In particular, the primary catalog used to create the next revision of the resource catalog must be in this same partition.

It is possible to put more than one resource catalog revision set in a single Hif partition. However, maximum flexibility of distribution results from defining a separate Hif user for each resource. Separate partitions for testing help further, by keeping the resource partitions free of test stubs and drivers.

4. Unique Identifiers

Given an Ada program library distributed among primary and resource catalogs, and a Hif database distributed among partitions, a number of interesting technical problems arise in the area of unique naming.

Unique identifiers are needed for compilation unit revisions to correctly determine when a compilation unit goes out-of-date. The compiler must record the unique identifier of all compilation unit revisions referenced while compiling the unit (e.g. the "WITH"ed specs), and then when these are replaced in the (conceptual) program library, the unit must appear out-of-date.

Unique identifiers are also needed for subprograms, so that references at calls may be resolved to the appropriate body. Overloading means a simple string will not suffice.

Finally, unique identifiers are needed for each Ada type, so that strong type checking and overload analysis may be performed correctly. Long identifiers and potentially deep nesting make the full Ada name an inappropriate choice.

In each case it is desirable that the unique identifier be relatively short (e.g. 32 or 64 bits) since there are a very large number of references, and yet be distinguishable from all other identifiers in the distributed program library. This is made more difficult when compiling is proceeding independently on separate computers, presuming there is no centralized assigner of globally unique identifiers.

4.1 Context-dependent Unique Identifiers

We have solved each of these unique identifier problems by using the concept of context-dependent identifiers, with context-dependent translation performed as part of moving the identifier from one context to the next.

4.2 Node Ids, Partition Ids, and Partition Maps

To uniquely identify compilation unit revisions in the distributed Ada program library, we rely on the general Hif node identifier, which consists of two integers, a "partition" id, and a node id. The partition id is simply an index into a "partition map," selecting an entry which identifies the location of the host files representing the partition within the host file system, as well as which partition map (if different from this one) to use for interpreting partition-ids appearing within that partition. The node-id is used as a key into the B-tree (host) file which represents the partition, and is assigned sequentially within the partition as nodes are created.

Each computer can maintain its own partition map relatively independently, assigning its own partition ids. When a reference is created to a partition on another computer that is not yet in the partition map, a partition-id is assigned for use from the referencing computer. The entry in the partition map indicates the location of the partition, as well as the location of the partition map to be used to interpret its partition references. When a node reference (partition-id, node-id pair) is copied from a partition on one computer to a partition on the other computer, the partition-id is translated according to the correspondence between the partition maps on the two computers.

4.2.1 Exporting Partitions and Partition Maps The partition map mechanism makes for a convenient method for exporting a set of partitions on tape, by simply including the partition map on the tape. Then, when the partitions are read back in off the tape, so is the partition map. The partitions are entered into the "master" partition map on the receiving computer, and their entry in the partition map indicates that when interpreting partition references within them, to use the partition map also copied from tape.

For convenience, a partition does not embed its own partition id in self-references, but rather uses the special partition-id zero. This way, if the partition is totally self-contained, there is no need to ship the partition map when shipping the partition all by itself.

4.3 Unique Ada-Entity Identifiers

A second kind of unique identifier, an Ada-entity identifier, must specify a particular Diana node, which represents the entity, among all of the Diana nodes in all of the compilation units in the (distributed) program library. Nevertheless, since there are many thousands of such references in a large program, the node identifiers ("locators") must be kept as small as possible (e.g. 32 bits). This apparently conflicting set of requirements was resolved by making each Diana file its own context for interpreting the locators.

4.3.1 Diana Node Locators; Segment + Offset Node locators are broken up into two halves, 16-bits of segment index, and 16-bits of segment offset. When the segment index is positive, it is an intra-file reference, and the segment index simply selects in which 64K segment of the file the Diana node appears. The segment offset always gives the byte offset within segment.

When the segment index is negative, it as an inter-file reference, and the absolute value of the segment index selects the element in the Diana file's "external segment definition table" which identifies (with a Hif relationship) the compilation unit being referenced, and the segment within it.

This mechanism allows each compilation unit to refer to 32K other compilation unit segments, each of which is up to 64K bytes in length. However, it means that a locator must always be interpreted relative to the file in which it resides. To simplify the manipulation of locators by the compiler, a "master" segment definition table is defined, and all locators are translated to "master" locators as they are retrieved from a Diana file. By design, the master segment definition table becomes the external segment definition table for the Diana file being created at that time, meaning that no additional locator translation need be done on storing into the file being created.

5. Summary and Experience

A distributed Ada program library is a key element in a distributed APSE. To implement this successfully, the program library "universe" as defined by the Ada Reference Manual must be broken up into independently manageable pieces. This in turn requires the support of a distributed database system, as well as a mechanism for uniquely identifying compilation units, linkable subprograms, and Ada types in a decentralized way, to av falling victim to the bottlenecks of a global database and/or global unique-identifier manager.

We have found the ability to decentralize Ada program library activity a major advantage in the management of large Ada programs (in particular, the multi-targeted/multi-hosted Ada compiler itself). We currently have 18 resource-catalog revision sets, each in its own Hif partition, plus 18 partitions for testing each of these, plus 11 partitions for the top-level

compiler/linker/program-library-manager components. Compiling and other development work can proceed in parallel in each of these partitions, without suffering the performance bottlenecks of global locks or global unique-identifier generation.

Implementation of Ada Protocols on MIL-STD-1553 B Data Bus

by
Smil Ruhman and Flavia Rosemberg
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot, Israel

Standardization activity of data communication in avionic systems started in 1968 for the purpose of total system integration and the elimination of heavy wire bundles carrying signals between various sub-assemblies. First issued in 1973, MIL-STD-1553 (USAF) replaced point-to-point wiring with a digital time-multiplexed common-bus for serial data transmission. Reissued in 1975 as a tri-service standard (version A) and again revised in 1978 (version B), it came into wide use and is supported by integrated hardware. However a major development effort must still be invested in every real-time system for interprocessor synchronization and scheduling of information transfer in the absence of a high-level language possessing communication constructs.

The growing complexity of avionic systems is straining the capabilities of MIL-STD-1553 B, but a much greater challenge to it is posed by Ada, the standard language adopted by the US Department of Defense for real-time, computer-embedded-systems. The stochastic, distributed nature of Ada with its rendez vous protocol for interprocess synchronization is not matched well by the deterministic central control of communication in MIL-STD-1553 B. Accordingly, the authors have proposed hardware implementation of Ada communication protocols in a contention/token bus or token ring network {1}.

However, during the transition period when the current command/response multiplex data bus is still flourishing and the development environment for distributed multi-computer Ada systems is as yet lacking, a temporary accommodation of the standard language with the standard bus could be very useful and even highly desirable. By concentrating all status information and decisions at the Bus Controller, it was found possible to construct an elegant and efficient hardware implementation of the Ada protocols at the bus interface, and this solution is the subject of our paper. No compromises are taken with the bus standard, and no changes imposed on Remote Terminals. Implementation hardware is restricted to the Bus Controller and its alternate, the Bus Monitor.

The idea is based on polling of the Remote Terminals by the Controller for entry calls, accept statements, or results (output parameters). The Controller interface maintains all the entry call queues and the list of ready accept statements, searches for a match, and issues the appropriate commands for transfer or execution depending on the presence of input and/or output parameters. In addition, the Controller interface times the delays of selective waits and of timed entry calls, and controls the execution of delay alternatives and of "else" clauses. Most of these operations are clearly of a match-making or associative nature. To avoid long Controller response

times due to conventional searching of extensive files, all queues and lists are stored in a common associative memory which is microsequenced from a control store. The paper presents the algorithms employed, defines the command and data formats, and outlines the hardware organization. The resulting bus traffic and speed of operation are discussed. It is interesting to note that while our algorithms take advantage of mode commands to reduce traffic, no such use was found for broadcast commands.

The proposed approach renders distributed intertask synchronization transparent to the designer and implements it in hardware at the bus interface. In addition, data buffering becomes unnecessary, since transfer is delayed until both parties are ready. Many important advantages result, chief among them being: facilitation of the development environment; major savings in specific development effort; conservation of system resources such as host processing and line transmission capacity; and faster system response.

Reference:

{1} Rosemberg, F. and S. Ruhman, "Hierarchical partitions in cyclic closed systems: a hardware oriented approach", Proceedings of Computers in Aerospace V Conference, Longbeach, CA, October 1985, pp. 148-155.

SOFTWARE ENGINEERING AND ADA* IN DESIGN

Don O'Neill

IBM FSD
March, 1986
WADAS

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

About the Author

Don O'Neill has been with IBM's Federal System Division (FSD) for the past twenty-six years. He is presently the Ada Technical Assistant to the FSD Vice President for Technology. As Manager of Software Engineering for FSD (1977-1979), Mr. O'Neill was responsible for the origination of FSD software strategies and the preparation of the FSD software Engineering Practices. He received an IBM Outstanding Achievement Award for his contribution to this effort. Mr. O'Neill has been applying modern software engineering on production software development projects. He has recently been leading the activity to prepare FSD for Ada use on projects.

Mr. O'Neill is a member of the Executive Board of the IEEE Technical Committee on Software Engineering. In addition, he has been a Distinguished Visitor of the IEEE Society. Mr. O'Neill also serves as a member of the AIAA Software Systems Technical Committee. He received his BS degree in mathematics from Dickinson College in Carlisle, Pennsylvania.

PREFACE

Modern software engineering promises significant reductions in software costs and improvements in software quality. The Ada language is the focus for these software methodology and tool improvements. The community may have underestimated the preparation for Ada, including compiler development and education. More must be done. On the other hand, the community may have underestimated the benefits of Ada productivity and quality. Perhaps expectations should be raised.

Software Engineering and Ada for Design overviews the IBM FSD Software Factory approach, including the software engineering practices that guide the systematic design and development of software products and the management of the software process. The revised Ada design language adaptation is revealed. This four level design methodology is detailed -- including the purpose of each level, the management strategy that integrates the software design activity with program milestones, and the technical strategy that maps the Ada constructs to each level of design. A complete description of each design level is provided along with specific design language recording guidelines for each level.

Finally, some testimony is offered on education, tools, architecture, and metrics resulting from project use of the four level Ada design language adaptation.

Section 1

INTRODUCTION

Software may be throttling the industrial development of the United States. As the information society takes hold, the demands for software are increasing. Furthermore, public expectations are increasing too; people want software that provides the right answers on time, everytime, and does so in a user-friendly manner. Software is intended to provide for the harmonious cooperation among people and machines. People possess an infinite variety and machines do only what is instructed, notwithstanding the promise of artificial intelligence. As a result, the burden on software is substantial indeed and is increasing.

Recently, software engineering has provided for the systematic design and development of software products and the management of the software process. The result should be quality software products obtained through design, sustained through development, and monitored through technical reviews. We have always known that good projects are ones with few errors at the end. We now know that good projects are also ones with few errors at the beginning. What may be needed now is a refinement of these methods, especially in the requirements and specification areas, their broad application, and preparation of adequate tools that re-enforce and enforce their use while assisting in productivity gains.

Section 2

SOFTWARE ENGINEERING FACTORY

Software engineering provides for the systematic design and development of software products and the management of the software process. Software

engineering may be viewed in the form of a state machine composed of inputs, transitions, outputs, and retained data (Figure 2-1).

The inputs to the process include the qualified people, labor saving tools, and practical technology needed to apply modern design, development, and management practices in the production of usable and reusable software products of sufficiently high quality to ensure life cycle benefits and confident customer ownership. People today are required to be highly qualified and equipped with specialized training in both software technology and applications. Testimony from early Ada users indicates that the training needs may be substantial. Harlan Mills observed that as we shape our tools, our tools may later shape us. Tools represent an institutionalized expert system, knowledge base of software methodology and style. Tool investments often lag behind their need. Practical technology requires the application of basic principles from advanced technologies repackaged into intuitive approaches and simplified for use by industry practitioners and acceptance by customers. Technology must employ an understandable conceptual model to assist the transition from user need to usable product.

The transitions of the software engineering state machine are governed by the software engineering practices for design, development, and management, applying across the full life cycle. Software design includes methods for producing and verifying modular designs and structured programs. Designs are recorded using a design language based on Ada, including both procedural designs and data designs. Advanced design ensures semantic correspondence of specifications through data dictionaries. Systematic design provides for functional allocation and decomposition of procedures and data. Systematic programming includes the elaboration of program designs using stepwise refinement, program design language, and correctness techniques. Taxonomy includes a proper program with a single entry and single exit, a prime program composed of zero or one predicate constructs, inner syntax of data refinement and operations and tests. The Ada based design language used to record designs assists the reasoning of the designer and his communication with others in getting the design right, knowing it, and convincing others. Software development includes the methodology for the early implementation and integration of detailed designs into product increments represented as source code libraries, configuration controlled through library hierarchies. In addition to incremental releases, the concepts of rapid prototyping, software first, and component reuse are being refined for routine use on projects in the future. Software management assures the effective application of qualified people within a predictable process to originate a quality product that satisfies performance requirements on schedule within cost. The use of Software Development Plans and technical reviews ensure an accurate view of status.

THE SOFTWARE FACTORY

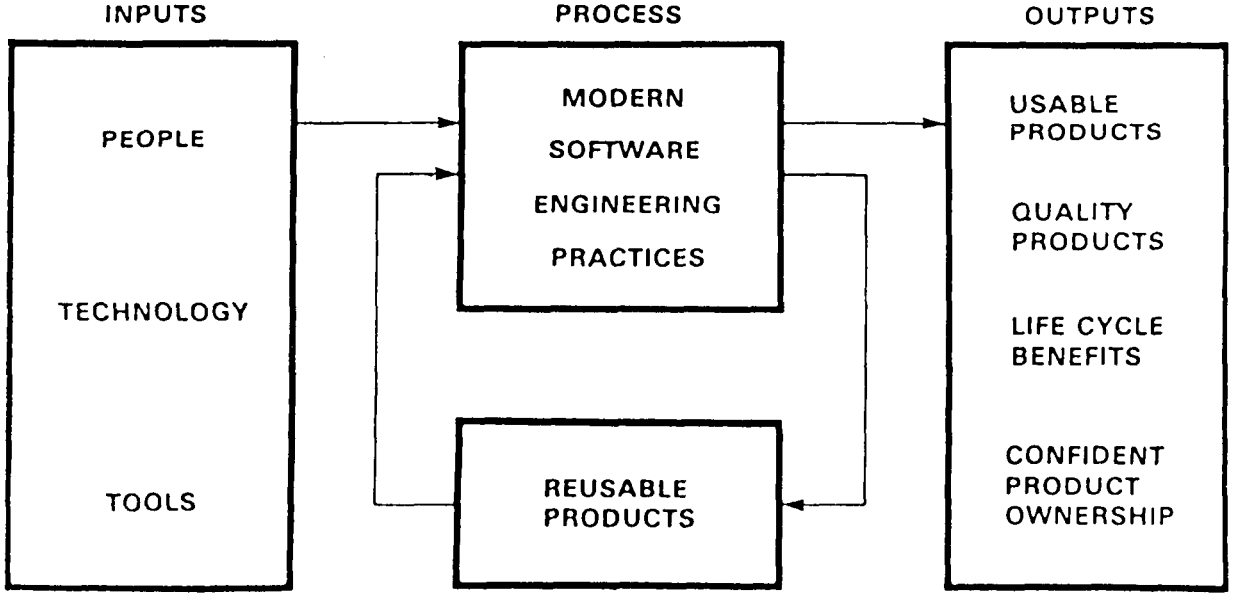


Figure 2-1. Software Factory

The outputs of the process include usable products of high quality that may be reusable capable of assuring confident customer ownership. Usable products are those that operate harmoniously within the user organization. They are adaptable to new requirements and feature userfriendly interfaces. The reward for this may be friendly users. Quality products are those that have few errors at the end. These are the same ones that have few errors at the beginning. A reusable product is one that continues to meet changing requirements through product enhancements. Futhermore, reusable products are transportable to other systems for similar uses. The Ada language promises to provide for software reusability. Using artificial intelligence, a components library of specifications can be interrogated for software components needed for new applications. As the industry becomes skillful and expert at matching existing products with new needs, the software factory may become a reality.

Section 3

SYSTEMATIC USE OF ADA AS A DESIGN LANGUAGE

Flirting with Ada? Careful. She is more than a programming language but less than a compiler for many. In Megatrends, John Naisbitt points out that trends are like horses. If you want to ride them, it pays to go in the same direction the horse is already traveling. He also points out that fads originate at the top, tend to peak, and then fade out. On the other hand, trends are bottom up, possess broader support, and persist.

The use of Ada as a programming language may correspond to Naisbitt's characterization of a fad, top down, perhaps explaining its sluggish beginning. For Ada the programming language, this is the awkward period between promise and delivery. On other hand, the use of Ada as a design language may be a trend, arising from the bottom as a popular choice. It is happening today.

A design language may be used for a number of reasons. It provides the facility to record design decisions. Once recorded, these design decisions can be shared with others forming the communication baseline among system engineers, software engineers, and integration and test engineers. It provides the basis for the designer to be more convincing in the defense of his design. It provides others with a clear reference point to focus their criticisms. The result is a better design. The use of Ada as a design language encourages good software engineering while at the same time permitting the design to obtain rigor in syntax and semantics through the use of Ada compiler product tools. Ada as a design language provides a platform for systematically accomplishing rapid prototyping through use of the emerging software design and product itself. In ways yet to unfold, Ada design language may also be a useful basis for assisting the access of reusable components. To be able to support these various uses systematically, Ada as a design language needs to be integrated into a software engineering methodology.

3.1 Four Level Design

An Ada based software design methodology has been adapted from the software engineering practices discussed in advanced design, systematic design, and systematic programming. This adaptation features four levels of design supported by a management strategy and a technical strategy. The management strategy maps the first two levels of design to the specification process and its review and the last two levels of design to the detailed design process and its review. The technical strategy purposefully and rigorously utilizes the expressive power of Ada at each level of design by mapping particular Ada constructs for use at each level.

The purpose of each level of design (Figure 3-1) considers the expected audience hierarchy within a project, ranging from readers to writers and including programmers, engineers, and managers. Early design levels must be intuitively understandable by all members of the audience and cannot depend on everyone being fully Ada literate. To support this need, Level 1 design is intended as the user contract. The user should be thought of as other software products that might utilize or interface with the software being designed as opposed to the end user of the system. Level 2 design portrays the design parts and their relationships both data interfacing and tasking. Level 3 design elaborates a detailed functional design that is independent of the target operating system and instruction set architecture. Finally, Level 4 designs are detailed designs that are fully targeted to the operating system and instruction set architecture, ready for implementation considering efficiency and capacity constraints.

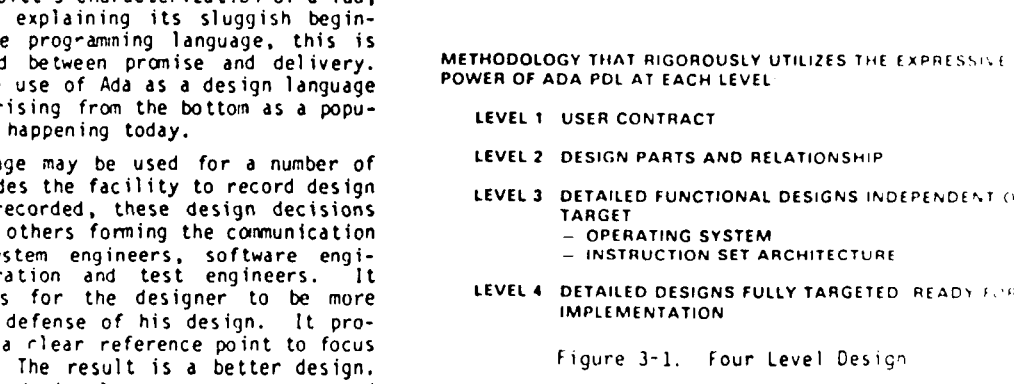


Figure 3-1. Four Level Design

3.2 Management Strategy

The management strategy for the four level design approach (Figure 3-2) maps levels 1 and 2 to the specification review milestone and levels 3 and 4 to the design review milestone. The specification review milestone equates to the Preliminary Design Review (PDR), the design review milestone equates to the Critical Design Review (CDR). In the MILSTD 2167 process level 1 and 2 designs are included in the Software Top Level Design Document; level 3 and 4 designs are included in the Software Detailed Design Document.

Beginning with Level 1, the specification is input to the software design process. A Level 1 design is produced and recorded in the form of an Ada Package Specification. The Level 1 design and

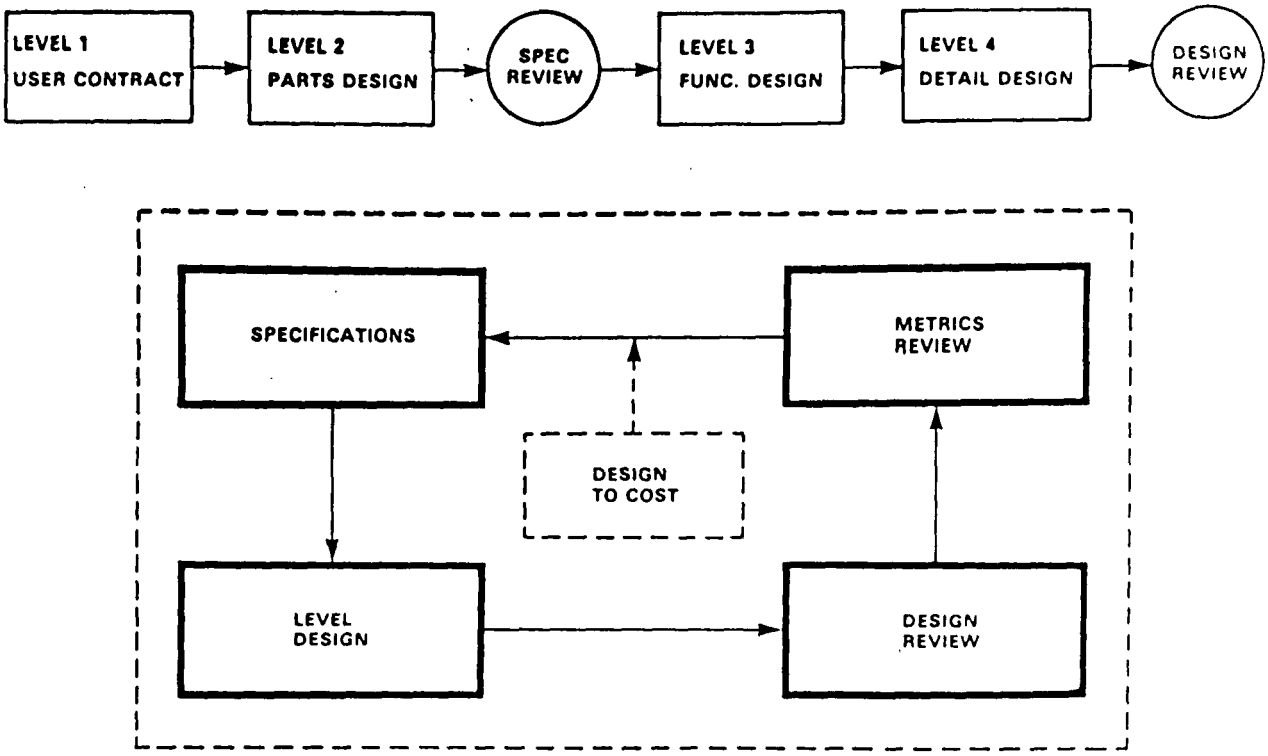


Figure 3-2. Management Strategy

any reuse candidates are subjected to a design review. The design review may be conducted electronically, or it may be conducted through a meeting of team members. Participants are highly trained experts committed to reviewing the design for completeness, correctness, usability, performance, and overall user satisfaction. Each reviewer must be personally satisfied with every aspect before the design review is concluded. The application of modern software engineering practices and their enforcement through a unanimous concensus of these highly trained experts is expected to provide a powerful impetus to dramatically improved product quality.

Once the design is satisfactory, the metrics associated with the software engineering process and the software product are reviewed and expectations revised. For the software engineering process the metrics include productivity and quality expectations. For the software product these metrics include complexity measures, reliability, and computer resource loading. These metrics are reviewed for compliance with budgets, perhaps necessitating adjustments in the design in an effort to achieve compliance. The specification itself may need to be reassessed and partitioned into essential requirements and desirable features. Certain desirable features may need to be eliminated or reduced in order to comply with management budgets.

At Level 2, the design for each component part identified in level 1 is recorded as an Ada package specification and its body. The Level 2 Ada package specification and body are evaluated for reuse candidates, continuing the systematic exploitation of reusability. The design review is conducted, as in Level 1. Metrics data is acquired and analyzed for Level 2 with the design to cost procedure followed if necessary. The Software Top Level Design Document is then subjected to the Preliminary Design Review (PDR). Throughout this process, systems engineers and software engineers work in a dependable relationship in shaping and fitting designs to meet user needs.

At levels 3 and 4, the design for each identified subunit is recorded as an Ada procedure with its accompanying intended function commentary. Procedure CALL semantics and intended function commentary are evaluated for reuse candidates, again continuing the systematic exploitation of

reusability. Design reviews are conducted. Metrics data is analyzed. The design to cost procedure continues to operate but with diminished flexibility since the specification has been baselined at PDR.

3.3 Technical Strategy

The Technical Strategy (Figure 3-3) governs the mapping of Ada constructs to each level. This mapping is intended to follow the architectural line of the language. Futhermore, the construct mapping by level provides a natural partitioning suitable for educating both readers and writers a little at a time.

Ada constructs are mapped to each level for outer and inner syntax. Outer syntax includes organizing units and control structures, both sequential and asynchronous. The inner syntax provides the format for expressing data and the operations and tests on the data. The constructs are assigned to each level with the objective of satisfying the purpose of that level. Once assigned to a level, a construct is permitted to be used in subsequent levels.

The Ada product form for Level 1 is the package specification used to express the user contract. This calls for an organizing outer syntax along with inner syntax commentary. Level 1 is limited to a few self evident constructs needed to accomplish its purpose. Those constructs are listed in Figure 3-4. They can be conveniently organized into a package specification template used to govern the style of the design recording. Other design recording guidelines may be set forth, including naming convention, commentary for intended functions, and key words in structured commentary useful in encouraging the use of the state machine model.

The product form for Level 2 is the package specification and package body used to express the design parts and their relationship. This calls for an outer syntax of structuring and tasking constructs. The inner syntax may be expressed as Ada abstractions, including abstract data types. Procedural elaborations are not carried out in Level 2 but instead are permitted to appear as procedure calls. The additional Level 2 constructs are shown in Figure 3-4. Those too can be conveniently organized into package specification and package body templates used to govern the style of the design recording. Design recording guidelines of Level 1 may be expanded to include the abstract data types permissible.

The product form for Level 3 is the procedure elaboration used to express a detailed functional design. This calls for a full complement of outer syntax function expressions and inner syntax data refinement, including predefined data types and Ada primitives. The additional Level 3 constructs are shown in Figure 3-4. Here too, procedures and task templates are used to guide the style of the design recording. Additional recording guidelines may be stated. Furthermore, to control the quantity of the Ada PDL being produced, Level 3 may be limited to the elaboration of only those procedures present in Level 2 as procedure calls.

The product form for Level 4 is the procedure elaboration, as well as function elaborations used to express a fully targeted, detailed design. Full MIL-STD 1815A is available at Level 4 (see Figure 3-4).

Section 4

CONCLUSION

Software Engineering and Ada in Design is but an early milestone report on the systematic use of Ada as a design language. From this experience, it is clear that the preparation for the use of Ada has been underestimated in several areas, including Ada compiler acquisition, tool integration, and education.

The Ada compiler acquisition difficulties in industry are well known. The need for Ada products during the design activity has received less attention. It is nice to have an Ada front-end product for semantic and syntax analysis during levels 1

TECHNICAL STRATEGY

	PURPOSE	OUTER SYNTAX	INNER SYNTAX: FUNCTION	INNER SYNTAX: DATA
LEVEL 1	USER CONTRACT	ORGANIZING	COMMENTARY	COMMENTARY
LEVEL 2	DESIGN PARTS AND RELATIONSHIPS	STRUCTURING, TASKING	PROCEDURE CALLS	ABSTRACT DATA TYPES
LEVEL 3	DETAILED FUNCTIONAL DESIGNS INDEPENDENT OF TARGET	—	EXPRESSIONS	PREDEFINED DATA TYPES, ADA PRIMITIVES
LEVEL 4	DETAILED DESIGNS FULLY TARGETED	—	REFINEMENT	REFINEMENT

Figure 3-3. Technical Strategy

	PURPOSE	OUTER SYNTAX	INNER-SYNTAX FUNCTION	INNER-SYNTAX DATA
LEVEL 1	USER CONTRACT	PACKAGE SPECIFICATION PROCEDURE SPEC TASK SPEC WITH, USE	COMMENTARY	COMMENTARY ABSTRACT DATA TYPES

LEVEL 2	DESIGN PARTS AND RELATIONSHIPS	PACKAGE BODY
		IS SEPARATE
		BEGIN
		IF THEN
		CASE
		LOOP (WHILE FOR, EXIT WHEN)
		FUNCTION
		ACCEPT, DO
		SELECT
			PROCEDURAL CALLS
			TASK ENTRY CALLS
				GENERIC INSTANTIATIONS OF ABSTRACT DATA STRUCTURES
				PRIVATE DATA TYPES
				DERIVED DATA TYPES
				TASK TYPES

LEVEL 3	DETAILED FUNCTIONAL DESIGNS INDEPENDENT OF TARGET OPERATING SYSTEM AND INSTRUCTION SET ARCHITECTURE	ELSIF WHEN	:=, + , -, *, **, / REM, MOD OR, AND, XOR, NOT RANGE, ABS =, , =, , =, /= TERMINATE DELAY EXCEPTION, RAISE	ADA DATA TYPES RECORD ARRAY RANGE ACCESS TYPE CONSTANT SUBTYPE
LEVEL 4	DETAILED DESIGNS FULLY TARGETED READY FOR IMPLEMENTATION		PRAGMA ABORT	DELTA, DIGITS FOR, USE, AT

ADA CONSTRUCTS

Figure 3-4. Ada Constructs

and 2. It is a necessity to have this tool available and ready for use during levels 3 and 4. Without it, the reenforcement of Ada education through the design activity is lost. Furthermore, the error discovery opportunity is postponed to downstream. The design inspection accompanying each design level needs the output of the Ada front-end. Where rapid prototyping is intended, the Ada compiler itself is needed to permit code generation and execution.

For early Ada projects, the education of the project team may need to be integrated with the design activity. One approach to this is to train people in one design level at a time, followed by the performance of the design activity and its review. In this way, the training schedule can be distributed throughout the performance period, the training for each level can be refined based on the results of the preceding design review, and project people new to Ada can progress through the experience sharing problems and obtaining assistance within the team. In the four level design approach, Level 1 represents only four constructs, all contained in a template. As a result, there is an early success for the new Ada PDL designer. Level 2 adds more constructs and is again guided by templates, assisting success. Level 3, however, represents the first time the Ada PDL designer must operate substantially on his own with a large number of Ada constructs. At Level 3, design reviews may result in a substantial reworking of the design. By Level 4, the Ada experience begins to pay off, and Ada PDL designers are completing their designs with confidence.

In formulating architectures for Ada software designs, new thinking may be needed. Important benefits are possible in Ada through modern software engineering. To obtain these benefits, software designs must make the transition from designs that simulate data flow to designs that encapsulate data in ways natural to the application providing only as much visibility as necessary and as much information hiding as possible. Furthermore, to obtain these benefits, the Ada tasking model needs to be exploited at appropriate levels in the design. The interface with commercial software products needs to be accommodated in a way that retains the cost benefits of these products, but does not dominate the software architecture. More work is needed in uniform design morphologies for Ada to provide useful Ada architecture models for early users, as well as the framework for exploiting reusable components by all users.

Very little is known about Ada metrics. As a result, there are many questions about the size of Ada programs and designs, Ada productivity, Ada quality, and Ada performance. The early experience with Ada PDL seems to show that a low ratio may exist between Ada source lines and Ada design lines. It may be 2:1 or 3:1. Where Ada is both the target language and the design language, the Ada PDL is part of the product. In this case, insight about the ratio may assist the allocation of effort and schedule between the design and code activities. The recent experience showed that the combined Level 1 and 2 ratio was about 25:1, Level 1-3 about 10:1, and Level 1-4 less than 5:1. Not enough is known to use these results as management budgets.

The revised IBM FSD four level Ada PDL methodology has demonstrated some important benefits in recent use (Figure 4-1). Expanding the audience of

BENEFITS: FOUR LEVEL ADA PDL METHODOLOGY

• AUDIENCE	— BOTH TECHNICAL AND NON-TECHNICAL
• PRODUCTIVITY	— TEMPLATES AT LEVEL AND CONSTRUCT
• QUALITY	— MINIMUM CYCLOMATIC COMPLEXITY
• PERFORMANCE	— FOCUS ON TASKING AT LEVEL 2
• PORTABILITY	— FULLY TARGET INDEPENDENT LEVEL 3
• REUSABILITY	— LEVEL FORMAT PERMITS EFFECTIVE ACCESS FROM COMPONENTS LIBRARY
• ADA TRAINING	— LEARNING AND USING ADA, A LITTLE AT A TIME, IS AN EFFECTIVE APPROACH TO ADA TRAINING, ALONG ARCHITECTURE LINE
• MAINTAINABILITY	— FOUR LEVELS PROVIDE A STAGED, LAYERED INTRODUCTION TO DESIGN AND IMPLEMENTATION DETAILS
• PREDICTABILITY	— MEETING COST AND SCHEDULE AS ASSISTED BY DESIGN TO COST FEATURE OF MANAGEMENT APPROACH

Figure 4-1. Benefits: Four Level Ada PDL Methodology

design reviewers from technical to non-technical permits useful and needed user input to the completion of the specification and to early design decisions. This is made possible by a training program, patterned after the four levels, that teaches Ada a little at a time along the architectural line of the language. Furthermore, the templates that govern the product style at each level provide a crutch for the early Ada user both reader and writer, a boost to productivity, and the assurance of uniformity in design style. Productivity may be given a more substantial boost when reuse of existing Ada components can be obtained. The Level 1 template format may assist this component reusability by providing the semantics needed to access a components library. Managing and meeting cost and schedule budgets is assisted by the systematic use of the design to cost feature embedded in each design level. Once completed, the four levels of Ada PDL provide the layered introduction to design details needed by the maintainer to learn design details as needed and to originate any required product adaptations with confidence. Designs produced with the four level Ada PDL methodology tend to be the simplified designs that result from modern software engineering. At the same time these designs consider performance requirements and meeting real time deadlines through the tasking focus at Level 2 and through the metrics at every level. Finally the methodology supports portability through the Level 3 target independence of operating system and instruction set architecture.

Although true that the community has underestimated the preparation for Ada, this preparation has been started and is underway. It may also be true that the community has underestimated the benefits of Ada which are substantial and are still being discovered.

BIBLIOGRAPHY

1. "Reference Manual for the Ada Programming Language (MIL-STD-1815A)," Department of Defense, 17 February 1983.
2. "Methodman II," Institute for Defense Analysis (IDA), Memorandum Report M-11, November 1984.
3. "Survey of Ada™ - Based PDL's," Naval Avionics Center Technical Publication, TP-598, January 1985.
4. Naisbitt, J., "Megatrends: Ten New Directions Transforming Our Lives," Warner Books, Inc., 1982.
5. O'Neill, D., "Software Engineering Program," IBM Systems Journal, December 1980, Vol. 19, No. 4.
6. O'Neill, D., "An Integration Engineering Perspective," The Journal of Systems and Software, 3, 77-83 (1983).
7. O'Neill, D., "At IBM - A Strategy for Software Management," Information Systems News, February 1981.
8. "Ada as a Design Language," IEEE Computer Society Working Group (P. 990), Draft 1985.
9. O'Neill, D., "An Overview of Gloval Positioning System Software Design," Software Engineering Exchange, October 1980, Vol. 3, No. 1.

Key Words

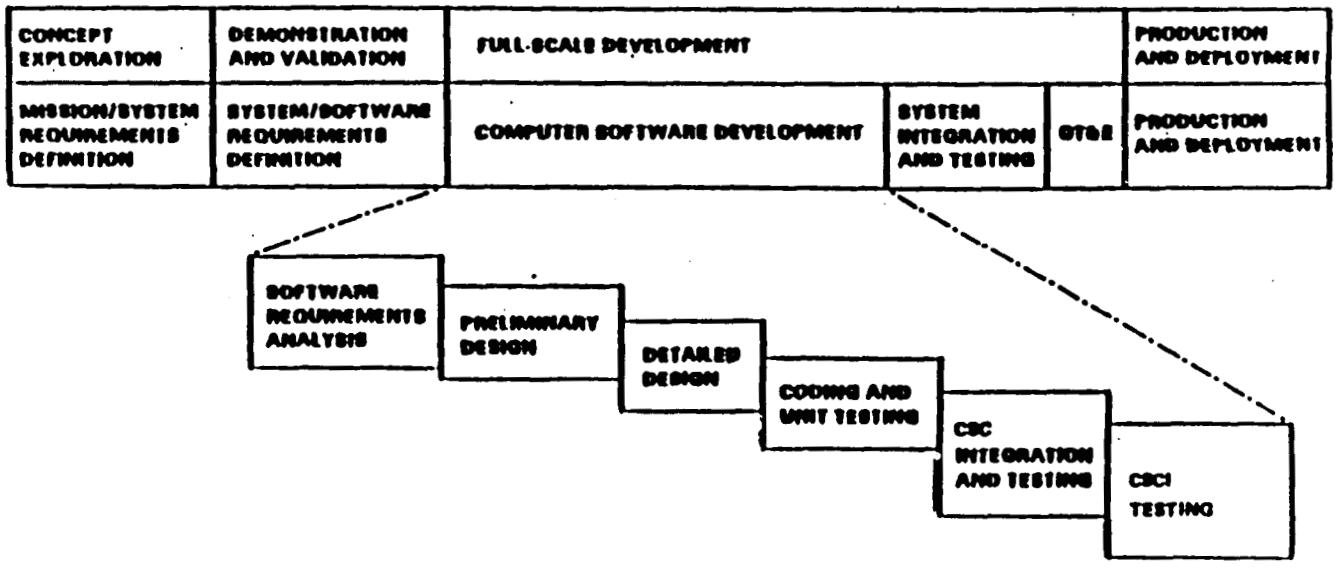
Software Factory
Four Level Design
Ada based design
Management Strategy
Technical Strategy
Ada Constructs

Analysis and Specification Tools in Relation to the APSE

John W. Hendricks

Systems Technology, Inc.

Ada and the Ada Programming Support Environment (APSE) specifically address the phases of the system/software lifecycle which follow after the user's problem has been translated into system and software development specifications. The "waterfall" model of the lifecycle identifies the analysis and requirements definition phases (now known as the concept exploration and the demonstration & validation phases in the lifecycle as described in the new DOD-STD-2167) as preceeding program design and coding.



Since Ada is a programming language and the APSE is a programming support environment, they are primarily targeted to support program (code) development,

testing, maintenance, etc. The use of Ada based or Ada related specification languages (SLs) and program design languages (PDLs) can extend the use of Ada back into the software design phases of the life cycle (for example, see Goldsack). However, there seems to be some agreement that Ada is not appropriate as a language for dealing with the "problem space" and the earliest phases of the lifecycle (Brodie, Mylopoulos, and Schmidt, p.410; Booch,p. 359).

The Ada Programming Support Environment (APSE), and indeed the Ada language itself, was defined as a response to the "software crisis" in DOD embedded systems. Booch (p.7-8) lists a number of symptoms of this situation, including:

- o Responsiveness. Computer-based systems often do not meet user needs.

- o Modifiability. Software maintenance is complex, costly, and error prone.

In particular, software maintenance is identified as being responsible for between 40% and 70% of the total hardware and software expenditures for these systems. We can expect that many of the systems for the NASA space station will share important characteristics with the DOD embedded systems (e.g., complexity, long-lifetime, changing requirements, real-time interfaces), and they should be subject to many of these same problems.

The world's best programming effort can not produce a system which is responsive to the user's needs if the requirements upon which it depends do not describe an appropriate solution to the user's problem or if the requirements are in a form which

we have great difficulty translating into an implementable design. Also, if this problem exists with the original requirements for a system, it can be repeated every time there is a change in the problem. We do not have data which characterize the distribution of software maintenance costs between "bug fixes" and changes in requirements, but it would not be surprising if a large part of the "maintenance" costs are caused by evolution of the requirements, especially for systems which are in service for a number of years. Therefore, both the responsiveness problems and a large part of the maintainability problems which characterize the software crisis may be beyond the reach of Ada and the APSE, unless facilities to deal with the processes of concept exploration and demonstration & validation can smoothly be linked into the APSE.

There are a number of developments which demonstrate the feasibility and desirability of formalizing specifications or architecture designs at higher levels of abstraction than that provided by a programming language (e.g., Balzer; Zave). These efforts share an objective of reaching out toward the "problem space" with a representation which is much easier to use than a programming language for describing the requirements, but is still capable of being translated or transformed into compilable code with limited manual intervention (automatic programming). They also share a commitment to extensive use of computer based tools to support the processes of analysis, specification and design. To the degree that these approaches succeed, they can address the problems of responsiveness to initial user needs and maintenance of responsiveness as these needs change over the lifetime of the system.

It is unlikely that any of these efforts will eliminate the need for substantial amounts of human programming in the development of the large and complex systems for

which Ada and the APSE are designed. If these new techniques are to be exploited for major projects such as the NASA space station, they must be capable of being used in conjunction with program design and development under the APSE.

One of the most promising of these new systems is Process Architecture Design Technology (PADtech). Systems Technology is working with the developers of this system, Associative Design Technology, Ltd (USA), to introduce and support this new technology for aerospace and military applications. An overview of PADtech and some of the issues raised by its use with the APSE should suggest both the promise of these new systems and some of the issues to be considered in "integrating" these new tools into major projects which will be using the APSE.

PADtech includes both a methodology and a set of computer based tools to support the use of the methodology in creating an architecture design for a complex system. The methodology provides a representation to formally describe:

- the structure of processes which we expect the system to implement, the events which will cause each process to be executed, and the events which each process can cause to occur; and

- the conceptual structure of the entities involved in the processes in terms of the role relation hips between the concepts, object types and objects.

This representation (Process Architecture Design specification Language or PADL) describes processes which may be implemented by hardware, or by persons following procedures, as well as by software. However, PADL has a precise semantics which enables it to be transformed into executable forms, and this inevitability makes its

application a more demanding process. By way of contrast, Goldsack (p.11) noted that "...the ease of use of PSL, SADT and many others, is partially due to the absence of a precise...semantics."

The computer based tools for the application of PADtech include the following:

- o A design workbench which provides a high performance, color, icon driven, interactive graphics interface for the creation and manipulation of the graphical form of the Process Architecture Design specification Language. The design workbench supports the system architect in the evolutionary process of analysis, specification and design. It also provides support for interactions with problem area experts and with program designers and programmers.

- o Modules which translate between the graphical form and the textual form of the Process Architecture Design specification Language;

- o A data manager which provides bookkeeping support for the evolving process architecture design;

- o A facility for building up a customized set of icons, process models, etc. which are appropriate for specific problem areas.

- o An interpreter for simulated execution of the process architecture for an early prototyping, iterative design cycle.

- o A "monitor" which collects the results of the interpretive execution.

- o A "debug" environment for controlling and examining the results of interpretive execution.

- o Code generation facilities for transforming Process Architecture Design specification Language descriptions for process and conceptual structures into the implementation languages, Ada and SQL.

PADtech is designed to be applicable to the analysis, specification and design process at several different levels. First, it can be used at the strategic planning level. For example, one can build a process architecture to represent an entire organization or a major project, and use this "enterprise model" to identify and specify automated information and communication systems to support operation of the entire enterprise. Second, PADtech can be used at the system or integration architecture level for a specific system. It can be used to design the architecture which defines the overall structure for a complete system, or to design and implement a database and communication "substrate" to integrate many separately developed modules, including man- and hardware-in-the-loop elements. Third, PADtech can be used to specify, design and implement (by code generation) systems which can readily be characterized by "object processing" processes, i.e., processes which create and change the state of both abstract and "real" objects.

PADtech will be most beneficial when applied to systems with some of the following characteristics:

- Requirements which are complex, not completely understood, and are expected to evolve over the life of the system,
- Requirements for very high speed execution involving parallel and/or distributed execution,
- Requirements for real-time responsiveness,
- A requirement for high speed management of complex, interactive data bases and communication structures,
- Integration of a large number of processes while maintaining protection against catastrophic failures.

We expect that there will be a number of space station systems with these characteristics, and that PADtech and other innovative tools for analysis, specification and design will be required to make these systems responsive to the requirements and maintainable over a long lifetime.

What are some of the issues raised by the use of these tools with Ada and the APSE? First, tools which are geared to creating a problem space oriented, executable specification or design specification tend to cut across the phases of the lifecycle as defined in the waterfall model. These tools gain much of their utility from an iterative cycle of analysis, execution and evaluation of the specification as a "prototype," re-analysis, etc. They emphasize direct involvement of the users or problem area experts in evaluating the implications of a design specification as they are revealed by repeated prototyping. The analysis and prototyping processes are supported by an interactive environment which is heavily dependent on "prototype execution" and graphics for presentation and manipulation. Also, these new techniques push formalization back toward the problem specification and use (partially) automated transformation to generate code modules. This allows maintenance which is occasioned by changes in the requirements, to be performed on the specification/design rather than on the code. Then, the revised specification is transformed into updated code modules. Use of these new techniques will be made more difficult if a rigid segmentation into the phases of a waterfall lifecycle model is imposed by procurement processes or by implementations of the APSE.

Second, there are several reasons why specification and design tools should be linked into the APSE. Most importantly, if design specifications such as those in PADL are to be used for maintenance and are to become a part of the permanent documentation

of a system, it is important to have control over their versions as one does for code modules. Also, in spite of everyone's tendency to claim that his system is complete and universal, none are. All of the analysis, specification and design tools would benefit from being able to interface with other systems which could complement their own capabilities (for example, see Ripken). An "open" APSE could coordinate between several "outside" tools, as well as between these tools and code development under the APSE.

Third, the amount of effort being put into the development of Ada and the APSE *creates a certain momentum towards making them all inclusive. If Ada is the* programming language, why not use it as the basis for a design language, a specification language, a conceptual design language, etc., and mandate their use? If the APSE is to control the programming process, why not mandate that only tools which are fully integrated into the APSE can be used from concept exploration onwards? The potential benefits of such a coherent, start-to-finish development environment need to be balanced against the potential costs of using much less than optimal tools in the pre-programming phases of the lifecycle.

A detailed examination of these issues would be a major project and is not contemplated here. However, we will suggest that in applying Ada, the APSE and standards such as 2167, we should be careful not to let their application expand to a point where they stifle innovation. The continuing revolution in microelectronics is providing an opportunity to create systems to solve increasingly complex problems; new techniques for specification and design will also be needed to exploit this opportunity. Many of these new techniques, which will be needed to build the increasingly complex systems we require, will not be developed exclusively for use by

one industry or one language. Retaining the option to select different methodologies for problems which have differing characteristics may be the only effective approach at this time.

Recall that the standardization of the APSE as a programming support environment is only now happening after many years of evolutionary experience with diverse sets of programming support tools. Restricting consideration to one, or even a few chosen specification and design tools, could be a real mistake for an organization or a major project such as the space station, which will need to deal with an increasingly complex level of system problems. To require that everything be Ada-like, be implemented in Ada, run directly under the APSE, and fit into a rigid waterfall model of the lifecycle would turn a promising support environment into a straight jacket for progress.

References

Balzer, R., "A 15 Year Perspective on Automatic Programming," IEEE Trans. Software Eng., vol. SE-11, pp. 1257-1268, November 1985.

Booch, G., Software Engineering with Ada, Menlo Park, CA: Benjamin/Cummings, 1983.

Brodie, M., J. Mylopoulos and J. Schmidt (eds.), On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases and Programming Languages, New York: Springer-Verlag, 1984.

Goldsack, S. (ed.), Ada for Specification: Possibilities and Limitations, Cambridge, England: Cambridge University Press, 1985.

Pepper, P. (ed.), Program Transformation and Programming Environments, Berlin: Springer-Verlag, 1984.

Zave, P., "The Operational Versus the Conventional Approach to Software Development," Commun. ACM, vol. 27, pp.104-118, Feb. 1984.

SOME DESIGN CONSTRAINTS REQUIRED FOR THE USE OF GENERIC SOFTWARE IN EMBEDDED SYSTEMS: PACKAGES WHICH MANAGE ABSTRACT DYNAMIC STRUCTURES WITHOUT THE NEED FOR GARBAGE COLLECTION

Charles S. Johnson

ABSTRACT

The embedded systems running real-time applications, for which Ada was designed, require their own mechanisms for the management of dynamically allocated storage. There is a need for packages which manage their own internal structures to control their deallocation as well, due to the performance implications of garbage collection by the KAPSE. This places a new requirement upon the design of generic packages which manage generically structured private types built-up from application-defined input types. These kinds of generic packages should figure greatly in the development of lower-level software such as operating systems, schedulers, controllers and device drivers; and will manage structures such as queues, stacks, link-lists, files, and binary/multary (hierarchical) trees. Generic structures like these will have to be carefully controlled to prevent inadvertent de-designation of dynamic elements, which is implicit in the assignment operation. A study is made of the use of the limited private type, in solving the problems of controlling the accumulation of anonymous, detached objects in running systems. The use of deallocator procedures for run-down of application-defined input types during deallocation operations is also discussed.

INTRODUCTION

Reusability is crucial to programs developed for Integration and Test (I & T) applications. The Ada language was specifically developed for use on embedded systems where most of the real-time applications work is performed. The creation of a software support environment for real-time work must first deal with the selection of a design approach which maximizes the reusability of Ada software components. The issue of Ada reusability does not just address problems of portability across machines and between projects, but also reusability within one project, and for one machine. One property of generic abstraction is the containment of a solution for a system- and application-dependent problem. Once having been solved generically, that solution is available for reliable reuse by all the applications of the system.

BRIEF BACKGROUND

Kennedy Space Center/ Engineering Development/ Digital Electronics Engineering Division is in the process of prototyping distributed systems supporting I & T applications, particularly the Space Station Operations Language (SSOL)

System, which is the I & T subset of the User Interface Language (UIL) for the Space Station. The discussions in this paper were developed from the results of systems designed and developed in Ada to demonstrate the feasibility of developing reusable software specifically targeted for real-time embedded applications. The Ada environment used was that of VAX Ada under VAX/VMS.

USE OF ADA IN EMBEDDED SYSTEMS

The implementation of the Ada KAPSE for a computer system can be performed in one of two ways. The KAPSE can be layered over an existing operating system, using it's services and saddled with it's limitations. The KAPSE can also be directly layered onto the computer hardware, and act as a limited operating system. Ancillary operating system services will then need to be supplied by Ada applications. For most embedded systems the latter alternative will hold, for both developmental and performance reasons. Developmentally, it is harder to re-host both the operating system and the KAPSE to new computer hardware, than it is to re-host the KAPSE alone. Also, for applications developed on a layered KAPSE, performance will suffer as requests for system services have to be processed at two levels. The organization and system approach for the two levels of support, since they were not designed specifically to be integrated, will almost certainly be mismatched in many ways.

For systems with a native KAPSE, the optional features of the Ada language (some pragmas, services) will be slow in appearing, or may be seen to be negative in effect. The system garbage collection feature in the KAPSE will be one of those features that won't appear initially. When it does appear, in many implementations, it's use will be precluded in real-time systems. [1]

The garbage collection feature of the KAPSE tracks, and deallocates anonymous objects in the Ada system, thereby freeing the system resources that they use.

Anonymous objects are previously-designated objects of a type associated with an access type (pointer type). A designated object is created by an allocator, which associates it with an access object (pointer object), which then, of course, designates it. Designated objects are implicitly declared by that allocation as objects of the designated subtype (subtype of object pointed to) of the access type, and are compatible with all objects declared of the designated base type (original type referenced in the access type definition).

Designated objects become anonymous objects by three means, all have to do with assignment:

1. The access object designating the object is assigned to the value of another access object of the same type.

2. The access object designating the object is assigned to a new value by an allocator.

3. The access object designating the object is assigned to the value "null".

Unless the previously-designated object was designated by more than one access object, after access object reassignment it becomes an anonymous object.

The use of access types is necessary if a system is to be flexible, and capable of creating objects in response to needs that cannot be specified until the need arises. Release of the system resources used by objects of designated subtypes, is essential in that flexibility (or static types rather than dynamic types could have been initially specified).

In layered systems built on general-purpose operating systems, the tracking down and subsequent deallocation of the resources consumed by these anonymous objects (the garbage-collection process) will be a built-in feature. In VAX-VMS the KAPSE performs this service. In AT&T Ada for the AT&T UNIX System V (Release 3), this service is implicit in the system, because all Ada objects are created on the system heap, which is managed by the system. In both cases, there is an ever-present background process, performing rundown of dynamic objects declared in the system. The performance detriment due to this background process is unpredictable, both for when it occurs (it is concurrent and unsynchronized with the applications) and for the system resources it consumes.

It is noted here that access types can be both data and task types. The problem of garbage-collection exists for both task and data types. In this paper, only the data type problem will be discussed.

There is no requirement in the Ada Reference Manual (ARM) [2] for the garbage collection feature to be implemented in the KAPSE. For many embedded systems running real-time applications, it will be required that the garbage-collection feature, if present in the KAPSE, retain the capability of being turned off. The presence of unpredictable resource consumption is contradictory to the principals of real-time computing, in particular, the response to external interrupts in a timely and reliable manner.

This poses a new problem. Without garbage-collection, the only time that anonymous objects are collected by the system (deallocated), is upon the expiration of the scope of the application which contains the definition of the access type. For anything other than restrictive use of the access type, this will usually be a package specified at the highest scope in the program. This scope, by not expiring, implies that normal collection will never occur (without garbage-collection).

For programs running on embedded systems, this means dynamic objects will continuously be converted into anonymous objects, consuming more and more system resources, until the program aborts when the system resources are exhausted. This self-destructive behavior may not be noticed during verification

or validation, if the process of creating anonymous objects is sufficiently slow. Indeed, well-written processes that are conservative in their exhaustion of system resources may live long before the limitations are breached.

These programs must, then, control their own storage allocation and deallocation. A pragma for declaring the storage management for an object as being controlled by the application (pragma CONTROLLED), and a generic package for deallocating controlled objects (UNCHECKED_DEALLOCATION) will be available for embedded systems development. The problem is that the implementation of these features must be standardized in the development of the application system, for there to be any assurance that anonymous objects will not collect.

A design philosophy encouraging abstraction would tend to drive the Ada source code using these features into the hidden scope of a package. This would create, in the system, an assortment of packages which define, declare and manage private access types, while retaining complete control of the allocation and deallocation of objects designated by those types. The control of the storage allocation in these packages would need to be implemented in an efficient way, such that the use of the package types would be flexible and easy (to encourage package use). A requirement of these packages, stemming from real-time considerations, would be that the behavior of systems using these packages should differ from that of systems using garbage-collection. The overhead incurred by the deallocation of storage should occur in predictable amounts, and in synchrony with, or under the control of the operation that incurs the overhead.

A design philosophy encouraging maximum reusability of software for the system, would tend to drive those packages, where possible, into a smaller family of generic packages using generic formal parameters which determine the differences between instantiations. Maximum reusability of these generic packages could be accomplished by the use of generic formal parameters matching the widest variety of input types, and by declaring internally controlled dynamic types which match the widest variety of applications (flexibility of use).

GENERICALLY STRUCTURED ABSTRACT TYPES

At some point in most Ada textbooks, a generic package is described that maintains a generically structured abstract type. The type is declared inside the package, and contains a component type within it which is defined from a generic formal type parameter (an application defined type contained within a generic structure). The example given is typically for a generic stack, list or queue, and the generically structured object may be hidden within the package, or declared as private type, or just as a type.

The important point of these textbock examples is the demonstration that the procedures for managing even very complex structures such as lists, queues, binary trees, multary (hierarchical) trees and files can be made general and separated from the procedures for managing the objects that they

contain. And, of course, that Ada supports the separation of these management procedures in a slick and easy-to-use manner.

If the design constraints on the system (storage control) can be embedded into the packages managing generic structures composed of application-defined types, many possibilities open up. The creation of what could be very complex systems such as operating systems containing schedulers, controllers and drivers becomes much simpler. These kinds of programs can be based on the use of just a few simple types of structure.

In an example, if a generic structure such as an index were managed in a storage controlled way, many system structures and much system processing could be based upon it. An index is a list of elements of one type (can be composite), ordered by elements of a second type, the index key. Many sample applications are possible. Logons could be controlled by a list of user names versus passwords, ID's, priorities, etc. Batch printing could be performed using a priority ordered list of print files. A disk directory could be held as a list of files ordered by name, or lists of lists. Batch scheduling of tasks could be ordered by priority or timestamp. More pertinent to I & T applications, a list of logical designators for the control of hardware on a Test System could order the blocks which contain their logical-to-physical access information. In this case a hierarchically ordered list of designators versus access blocks would probably be more useful.

The focal point of the impact of this technology is on the reuse of software components within a project. The system-dependent functioning buried in the body of packages, will not be nearly as portable between machines and areas of application as it is reusable within a project. Some external software will be incorporated, of course, like it is today: DBMS, graphics support, user interface packages, communications support. These kinds of packages will be available where there are broad areas of commonality of function, and where system-dependent features can be profitably developed in packages by vendors.

Standardization by the use of generically managed structures makes possible the idea of technology insertion directly into the applications of a system. If a system- or application-dependent problem is solved one time, in a flexible and reusable manner, the developer can beat that solution to death, reusing it over and over.

Maintenance of reusable software enhances the system effectiveness. That reusable solution can be tuned at a minimum number of locations in the system, and re-inserted into the applications. If a better hashing function is found for the key of our index example, for instance, a widespread increase in performance will result.

DESIGN GOALS AND CONSTRAINTS

The design of packages managing generically structured abstract objects must begin with the establishment of goals and

constraints. The goals and some of the constraints are independent of the problem of embedded systems. [3]:

1. Package-managed generic objects that are declared in the application software should, where possible, be defined as abstract types, that is, made private.

2. Maximize the generality of the package. This comes from the use of formal generic parameters, particularly for types, that match the widest variety of application input types (type private instead of digits <>, for example).

3. Maximize the usability of the application interface to the package. Extend, as far as possible into the application domain, access to the structures managed in the package, without violating the integrity of the internals, or the independence of the application from the generic software component (generality).

4. Maximize the completeness of the application interface to the package. Give the application developer all the operations required to access and manipulate the internal structures, in a package-controlled manner.

5. Support, if possible, multiple objects with the same package. This limits the need to re-instantiate the package several times within the same scope, for processing of multiple objects.

6. Design for flexibility: a single tool, suited to a wide range of applications, is more likely to be remembered, and used by developers.

7. Cover the infrequent failure modes. Most failures of algorithms and processing logic in programs occur at the extremes of their domain of applicability. Testing should cover the ends of ranges and the infrequent states of the application. If the software component is reusable, it will be used in a wider range of applications, and the infrequent failure modes will occur more frequently.

Some of the constraints on the design of packages managing generically-structured abstract objects stem from requirements generated by the use of Ada on embedded systems, and are therefore application-dependent:

8. The package operations must control and deallocate any internally allocated dynamic storage.

9. The package must, by it's implementation, disallow any inadvertent de-designation of package managed dynamic

structures or elements. The application must be prevented from creating anonymous objects.

10. The overhead involved in the processing of package operations must be predictable and controllable by the application (in contrast to the garbage collection of anonymous objects by the KAPSE).

SELECTION OF DESIGN APPROACH

The index package, which was described above as a list of elements ordered by another set of associated index key elements, will be used as an example for the selection of design approach. The index structure itself should be some kind of private type. Functions for index lookup by key item, element add/delete, and for stepping through the index sequentially should provide a useful set of operations for index manipulation. The INDEX type itself should be defined in the package specification, not hidden, so that it can be declared as an object in the package scope.

The importance of having the index object in the scope of the application is in the flexibility of use of the object at the application level. The developer should be capable of passing the object as a parameter to subprograms developed at the higher level. If the object of type INDEX is hidden, this flexibility is not there.

This generates a conflict with the application-specific constraint about allowing the application to inadvertently generate anonymous objects. If the object of type INDEX is declared in the user scope, any kind of assignment operation to it will create an anonymous INDEX object.

USE OF THE LIMITED PRIVATE TYPE

The definition of the INDEX type as limited private prevents reassignment of it's value in any operation. It cannot be reassigned in the deepest level of any procedure (Ada), or generic software component that knows of it's typing. This allows the access object to be declared in the user scope, and used as a parameter, without any chance of creating anonymous objects from reassignment (unless the package itself does).

The removal of needed functionality by the definition of the type as limited private, creates a need for the definition of analogous functions: assignability, comparability, nullability.

The assignment function which has been removed cannot be replaced exactly. If the application is given the ability to assign the same value to INDEX objects, even controlling the creation of anonymous objects during reassignment won't help. Having two INDEX objects of the same value implies that the package cannot explicitly deallocate either INDEX designated object, without creating an erroneous circumstance (an INDEX object designating a deallocated object). This cannot be allowed. Therefore assignment (call it ASSIGN the ":="

operator cannot be overloaded) will first clear the access object value by deallocating the current designated object, and then copy the object designated for assignment, element for element, until two copies exist.

If the need for mutual designation by the same INDEX object was a requirement, creation of anonymous objects could be controlled by the installation in the structure of the INDEX designated object of a semaphore-type variable, which would provide concurrent access to the structure along with the protection by mutual exclusion. This would allow the package to keep a count of the number of INDEX objects accessing the structure of the index, with the capability to deallocate the INDEX designated object upon the reassignment of the last INDEX object designating it.

The compare function, "=", can be overloaded for limited private types, and should be defined to compare the elements designated by the two objects of type INDEX, one for one, to establish equality. It should be noted here, that the application itself could define "=", if the capability of stepping through the INDEX elements one by one, and retrieval functions for each element are provided.

The re-initialization of the INDEX object ("null" assignment) is replaced by a DELETE function which deallocates the designated object (the entire structure).

APPLICATION DEFINED DEALLOCATOR PROCEDURES

There is one last potential for the inadvertent creation of anonymous objects by the package itself. The package allocates a node when it adds an element to the INDEX designated object, and it deallocates a node when a delete of an element occurs. However, if the type that was passed as the formal generic parameter for the key type or the element type is itself an access type, deallocation of the node will create anonymous objects that were previously designated by access objects of the application-defined input types.

The solution for this problem depends upon the developer. For every application-defined component type which is passed into the generic package as a generic formal parameter to be incorporated into a generically-structured storage-managed type, there must be an accompanying generic formal parameter indicating a procedure which deallocates any objects designated by an object of the application defined component type. This allows the generic package to invoke that procedure for the components of the structure, so that the subsequent component deallocation will not create any anonymous objects.

For application-defined types that are not or do not contain access objects, the deallocator procedure passed would simply provide a null return, and do nothing.

To repeat this rather complicated rule in other words, there is a need for every generic formal parameter of an application-defined type for a structural component, to have an accompanying deallocator procedure, not for the type itself, but for designated objects of that type, and designated objects of

those designated objects, and so on. If the developer wishes to incorporate structures within structures, the price of this complexity must be paid.

INCREMENTAL DELETE FEATURE

It is not reasonable to assume that the size of the structure being managed by the generic is known before the application is coded, or else the developer might have chosen a static rather than a dynamic type. The processing overhead incurred from the deletion of an entire structure or one part of a structure is then also not predictable. This can put the real-time performance of the package operations back to square one.

If a real-time application performs a delete operation, the return from the subprogram must be made within application defined time-constraints for the package to be useful. In an example indicating the problem, a real-time application, while in between accepting interrupt entries from a hardware device (a time-critical operation, for hardware interrupts are not queued), attempts to initialize the access object designating a structure, during the time window that is known to exist between interrupts. During initialization of the structure it is necessary, of course, to run down the entire structure, deallocating each component of the current structure exhaustively, until the access object can be initialized. Unfortunately, during the time that the subprogram took control away from the real-time application, several interrupts were overwritten, and critical data was lost.

The solution to this problem is to supply an incremental delete function. The overhead incurred from the delete and subsequent deallocation of a single element is knowable. An incremental delete operation can then be defined, such that upon input of the logical parameter indicating how much of the structure to remove, and a physical parameter indicating the number of elements to remove for each successive invocation, the structure will be whittled away incrementally. The order of deletion/deallocation should be such that a reference always exists to the remaining increments of the section of the structure that are to be removed (for example, delete a tree from the leaves in toward the root).

CONCLUSION

It is concluded, by our studies, that it is feasible to create families of highly reusable generic software components, specifically tailored to support kinds of applications. These generic packages can maximize the reusability of software developed within and for a particular project. At the same time they can address the performance requirements of software developed for embedded systems running real-time applications. These requirements stipulate that such software be responsive and controllable in terms of direct processing overhead, and incur little or no background processing overhead of an

unpredictable nature (in contrast to the garbage collection of anonymous objects by the KAPSE).

ACKNOWLEDGEMENT

I gratefully acknowledge the support given by the Kennedy Space Center/ Engineering Development/ Systems Integration Branch in supplying the computer facilities for the feasibility studies that provided the basis of this work. I also thank my wife, Bronwen Chandler, for her support.

REFERENCES

1. Burns, A. 1985. Concurrent Programming In Ada. Cambridge, Great Britain: Cambridge University Press.

2. United States Department of Defense. February 17, 1983. Reference Manual for the ADA Programming Language. ANSI/MIL-STD-1815A-1983. New York, New York: Springer-Verlag.

3. Johnson, C., 1986. "Some Design Constraints Required for the Assembly of Software Components: The Incorporation of Atomic Abstract Types into Generically Structured Abstract Types", Proceedings of the First International Conference On Ada* Programming Language Applications For The NASA Space Station, E.1.1.

A Computer-Based Specification Methodology

Robert G. Munck

The MITRE Corporation
Bedford, MA 01730
Munck@MITRE-Bedford.ARPA

ABSTRACT

It is becoming clear that our standard way of writing specifications -- requirements, design, test, and other types -- is inadequate for large, complex, and long-lived systems. The process by which they are created is unstructured and often cursory, and the resulting paper documents are bulky, vague, inconsistent, and difficult to publish, distribute, and update. We are especially bad at writing understandable, consistent, and sufficiently-detailed requirements specifications.

Part of the problem comes from the shortcomings of written English and the essentially *linear* sentence/paragraph/chapter structure of specifications; it has been aggravated by widespread use of word processors that support nothing but text. Text will always be a part of specifications, but there are other forms of expression that can be more suitable for other parts: data-flow, SADT[ROSS75], and Buhr[BUHR84] diagrams, non-linear text or "hypertext,"[VAN DAM70] spreadsheet-supported tables, charts, and graphs, animation of algorithms and procedures, geometric modeling, and voice and video processing. All of these become viable possibilities in the high-capacity, display-oriented workstations of the proposed Space Station Data Management System.

The use of exotic, "high-tech" presentation media in specifications will not automatically make them easy to produce and understand; it is more important that there be a methodology for creating them that emphasizes correctness and clarity of presentation, and which supports cooperative work over a network. The most complete and mature such methodology is SofTech's SADT™. SADT is unique in the amount of attention it pays to the way people work together and as individuals and in its facilities for specifying requirements independent of any particular implementation.

SADT's most widely-used component is the hierarchical box-and-arrow diagram notation. In the full methodology, that notation is supported by an "infrastructure" of procedures, formats, protocols, and "ways of thinking" that make it possible for many people to *work together* on a large project. For example, the Reader/Author Cycle is a peer review procedure that emphasizes constructive criticism and a disciplined exchange of ideas. Reader Kits and their associated Kit Files provide a mechanism for working on part of a specification without losing sight of its relationship to the whole AND for tracking the evolution of the specification over time.

The paper proposes a network-based system for writing, reviewing, and publishing multi-media specifications with tools and procedures based on SADT methodology. It envisions people at universities, companies, and NASA sites all over the world working together to prepare a requirements or design specification and discusses the possibility of semi-automatic conversion of such a specification to Ada[1] code, currently under investigation at MITRE. The computer-based SADT tools developed in that project will be described.

1 Everybody's Talking

Natural language has evolved over the millennia as our most powerful tool, that which truly separates us from animals. However, it is becoming apparent that "written English" using traditional forms and media (chapters and paragraphs, paper and ink) is insufficient to communicate the very large, complexly-interrelated concepts of a modern computer-based electronic system. To put it another way, our

---

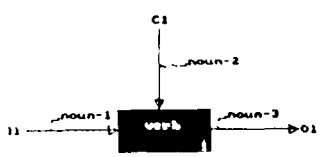
1. Ada is a registered trademark of the United States Government (Ada Joint Program Office)

specification documents vary from "unsatisfactory" to "horrible" in doing what they are supposed to do.

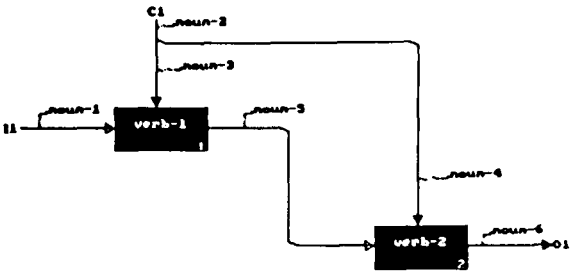
1.1 Words in a Row

Written English (or the language of your choice) is essentially the spoken form translated from sound to ink patterns on a word-by-word basis and laid out in linear form. As such, it makes many of the same assumptions as speech; for example, pronouns are based on the assumption that the listener has a high-speed short-term memory that can match a pronoun to the last object named, thus reducing repetition of names. Unfortunately, in technical writing the use of pronouns is overwhelmed by the great number of objects needing to be named. Names of things are therefore repeated over and over, and are often long proper noun phrases containing several capitalized adjectives and adverbs. The resulting text is much like a road containing frequent potholes and boulders.

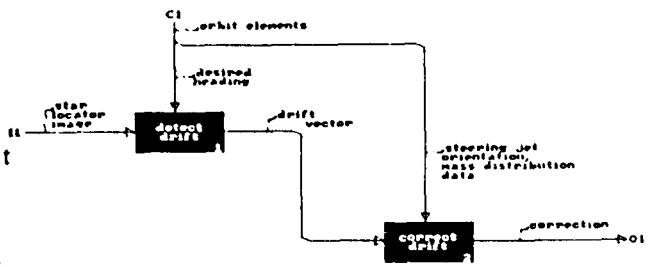
Graphical languages such as SADT boxes-and-arrows and Buhr diagrams reduce the need to repeat names by using two dimensions instead of the linear one-dimensional representation of written text. That is, an object can be associated with things above, below, right, and left of it, not just to the left. SADT also generalizes the *noun - verb - object* sentence structure of English into two dimensions, thus better approximating the way people think. In a simple example, the SADT fragment



is the equivalent of the English "*noun-1* is *verb*ed to make *noun-3* as controlled by *noun-2*". A more complex example, which demonstrates the generalization, is



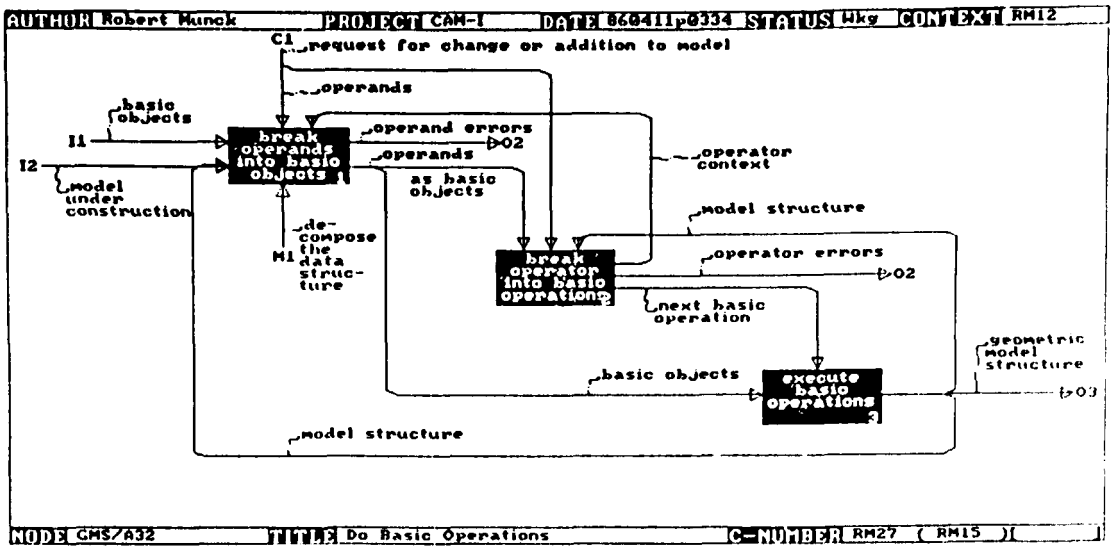
which says "*noun-1* is *verb-1*'ed to make *noun-5* as controlled by the *noun-3* aspect of *noun-2*; *noun-5* is *verb-2*'ed to make *noun-6* as controlled by the *noun-4* aspect of *noun-2*". A somewhat more real example:



Put into English, it says (approximately):

> The star locator image is used to detect drift by using the desired heading from the orbit elements to calculate a drift vector. The drift vector is then used to compute a correction from it and the steering jet and mass distribution data from the orbit elements data.

A complete SADT diagram, such as this:



Might translate into a page or more of normal text. That text would be as readable or unreadable as text specifications normally are.

The diagrams shown above all have activities, actions, or verbs in their boxes and data, things, or objects as their arrows. There are in fact four kinds of diagrams, of which these are only one, called the **Activity Diagram**. There are also **Data Diagrams, State Diagrams,** and **Transition Diagrams**. Data diagrams, in which the arrows are activities, are similar to those drawn in Data Base design and Object-Oriented Design. State and Transition diagrams are useful in real-time systems.

1.2 What It's All About

Human communication is based on the fact that we have a vocabulary in common with each other. Unfortunately, that commonality is only approximate; we can never be entirely certain what someone else means by "red" or "big" or "Multi-modal Phased Radar Array Serial Interface." Technical specifications can be characterized as a semi-ordered set of terms and the definitions of those terms -- in the ultimate the entire specification is a definition of its title. The problems that arise include multiple and conflicting definitions, the definition of a term being "far away" from its usage and difficult to find, and multiple terms having the same definition.

Definitions of terms often themselves contain terms that need definition. SADT uses the hierarchy resulting from this as its organizational backbone. Each box on a diagram contains a word or term that has some meaning to the author of the diagram; it may have a different meaning to a reader of it. If the author feels that readers might have a different meaning for a box than his intent or not know what it means, he creates a new (child) diagram that "explains" or "defines" the box in greater detail. Boxes on the child diagram that need further explanation are themselves expanded into diagrams, until all terms in all unexpanded boxes are in common parlance and unambiguous.

One of the strengths of natural language is that words can have different meanings in different contexts. This reduces by several orders of magnitude the number of different words we need. However, specification writers often attempt to give certain important words rigid definitions for all contexts, placing those definitions in a glossary. Those reading the specification must, in effect, memorize the

entire glossary for the duration of their reading; otherwise they will have to flip back and forth to it continuously, with no way to know if they need to look up a particular word. In SADT, there is an indication on each box if it is expanded. Boxes on different diagrams containing the same word or phrase may have the same or different expansions. This is the SADT equivalent of the common notion that a word or phrase may have different meanings when used in different contexts.

1.3 SADT Media and the Message

SADT was originally designed for use with no computer support; a team having standard office supplies and a copier could create very large, very high-quality specifications. In fact, users tended to resist having their diagrams even typed or typeset; a diagram produced with a good pen, a straight-edge or flowchart template (for the curved corners), and legible handwriting seemed more "comfortable." An early attempt to computerize the production of diagrams using a time-shared mainframe[SMITH81] was unsatisfactory due to slow response time.

Despite the power of the SADT filing and archive system (discussed later), large and long-term projects found the maintenance of a large set of diagrams (thousands) to be burdensome. Fortunately, the personal computer has now become powerful enough to support SADT, and in fact is proving to be an extremely valuable addition. There are at least four announced or planned SADT systems on the market.[SAIB85]

The SAda[MUNCK85] system, implemented by the author as an IR&D project at MITRE, runs on an IBM PC or equivalent. A satisfactory system with the necessary graphics and telecommunications can be bought for $2500 hardware costs; a "super" system with a big color display and laser printer might cost $10,000.

2 The Way We Work

The above discussion has shown a few of the many ways that SADT makes it possible to have a readable, understandable technical specification. In general, it does so by relaxing or generalizing English grammar, sentence and paragraph structure, and the division into sections, appendices, glossaries, annexes, and volumes of normal specifications. With SADT, the most complex systems that we are capable of building can be specified understandably. Among the most complex system specified in SADT to date is the financial system of the Department of Energy. The complete specification took more than 25 analyst-years to write and, printed double-sided, was over two feet thick. Because it was done on paper before computer support was available, the document is quite intimidating by its sheer mass, but still vastly preferable to a text equivalent.

Of course, there is no free lunch. Specifying a complex system well with SADT takes a great deal of hard work by trained, experienced, smart people. That work is made as productive as possible by other features of SADT that deal with the way people work together and individually. These features might be called the "management" or "sociological" aspects of SADT.

2.1 All Together Now

The creation of specifications is usually a quite chaotic process in most organizations. A common feature is the "brainstorm session" at which a number of people present ideas, argue, and fill blackboards with scribbling. At the end, several participants are charged with "writing up the results." However, they will capture only the last set of ideas proposed and not rejected; other good ideas disappear forever the next time the blackboard is erased or never appear because their conceiver is absent or doesn't communicate well in noisy meetings. The basic idea of brainstorming is good: communicating "half-baked" ideas quickly to others who can grab the good ones and add their own improvements. We need a better process and medium than the noisy meeting and blackboard.

SADT includes the Reader/Author Cycle to replace this aspect of writing specifications. It works as follows:

1. One analyst, called the *Author*, creates a small number of diagrams. His SADT training tells him to limit the diagrams to one major thought or amount of information, approximately a half-day's work on his part to create and an hour's work to read. This amount is typically one parent diagram and three to five child diagrams. The SAda drawing tool helps him create the diagrams using a mouse and keyboard such that his thinking is about the subject matter, not the mechanics of drawing; there is no need to sketch diagrams on paper and enter them into the computer as a separate step.

2. The Author assembles these diagrams into a *Reader Kit* and sends the kit to a small number (1-4) of his colleagues, called *Readers*. These diagrams are transmitted by electronic mail and appear in a "to-be-read" directory in the Readers' machines.

3. Each Reader reads the kit within one working day. He writes comments on the diagrams with arrows and circles indicating where they apply, using the mouse and keyboard. SADT Readers are trained at great length to make their comments constructive and non-threatening; in effect, there is a "code of courtesy" for writing comments. Note that the Author does not have a large "psychic investment" in the diagrams; he has spent a relatively short amount of time creating them. This contrasts to the difficulty of criticizing something that someone has spent weeks or months producing. Readers who are also trained to be Authors comment on the format and understandability of the diagrams as well as their technical content.

4. The Reader transmits his comments back to the Author.

5. The Author reads the comments from each Reader within one working day and writes a reply to each one. Here again, the Author is trained to write replies that are constructive and helpful, not argumentative. While doing this, he also makes notes on the diagrams indicating changes to be made that the comments have inspired. Comments, replies, and notes are overlays or windows that can be added and removed from the diagram on the display; on a color display, they appear in color.

6. The Author transmits each Reader's comments back to him.

7. The Reader reads the replies and adds additional notes of his own. The diagrams, comments, replies, and notes are added to his files.

8. If necessary, the Author revises his diagrams and sends them out again, starting another cycle. This time, however, the Readers have the previous revision with the comments and replies. They can therefore check that problems they noticed have been fixed.

The Cycle is "kept going" in the manual system by the Librarian, a clerk trained in SADT procedures. He does the mechanical tasks such as copying and filing, and makes sure that the participants do their jobs in the time allowed. In the computer-based system, no Librarian is needed, and the participants may be far apart physically on a loosely-coupled network.

The Reader/Author Cycle has been shown to be an extremely powerful organizing influence on technical work of all kinds. Many organizations that were exposed to it through SADT training now use it for most or all of their work, even when other aspects of SADT are not involved. It appears to be a good match to the needs and organization of NASA.

When used with the "code of courtesy" and other aspects of SADT, the Cycle brings out the best, most creative thoughts of the participants, reduces conflict, and captures the processes by which decisions are made, not just their results. People who have worked on successful SADT projects tend

to urge others to use it with the fervor of religious converts.

2.2 In Organization There is Strength

We have mentioned the two-dimensional aspect of diagrams and later a third dimension, that of expansion of boxes into diagrams. There is also a dimension of time, in which each diagram has a pointer to the diagram that it replaced and to the one that replaced it, and notes by the author explaining why it was replaced. The result is a fairly complex data structure, but one that proves easy to navigate with the right computer support.

A single set of diagrams related hierarchically, starting from a single "top level" diagram, is called a model. A model is a top-down exposition of a single aspect or part of the system as seen from a single, stated viewpoint. For example, we might have models of a single instrument from viewpoints such as a user, a maintenance technician, a programmer, a telemetry system, and a power system. Each of these models will emphasize the parts that are important from its given viewpoint and will have pointers to other models for other parts and to models of other aspects of the system to which it is related.

3 Make it Run

As done on paper, an SADT specification can be an extremely readable document, leading to much better implementation. In the computer-based system, there are even more possibilities:

- A model of the activities of a project, with estimated time and manpower attached to each box, can be analyzed by the machine to determine a schedule and indicate which activities are on the critical path. This project model can be maintained by the program office as the master project schedule, with pointers from each box to the current status report for that activity. One would be able to review progress informally and conveniently by browsing through the model.

- A model of a piece of software can have execution time and resource use estimates attached to each box. It can then be "executed" as a simulation to predict performance.[BUCHERT81] The simulator could "animate" the model on a graphic display as it executes. Small meters or bar charts could be attached to boxes and arrows on the display to show current values such as processing rate, queue length, frequency, and values of variables.

- A detailed model of a piece of software can be converted into skeleton Ada[Ada83] code defining the task structure. Each lowest-level box can then be coded by an Ada programmer (or the appropriate function found in a library) and combined with the skeleton to make a running system. The SAda project at MITRE is beginning to explore this possibility.

- A model could be connected to its implementation, hardware or software, by diagnostic or metering probes. It could then "run" in the same way that the simulator animation discussed above did. A person monitoring the system could move up and down between levels of detail.

- A detailed model might be able to be converted mechanically into a custom integrated circuit or piece of wafer-scale integration. This model might also have run as a simulation or been converted into running Ada code.

Most of the above suggestions have been tried in one way or another, and all showed promise. The time is ripe to begin work on an *infrastructure* or *support environment* on which the tools for writing, reading, and "exercising" computer-based specifications can be integrated. It is clear that such spec-writing support environments would have a great deal in

common with programming support environments, to the point of both being part of a single larger system.

4 Conclusion

Standard practices for creating and using system specifications are inadequate for large, advanced-technology systems. We need to break away from paper documents in favor of documents that are stored in computers and which are read and otherwise used with the help of computers. An SADT-based system, running on the proposed Space Station data management network, could be a powerful tool for doing much of the required technical work of the Station, including creating and operating the network itself.

References

[Ada83] U.S. Department of Defense **Reference Manual for the Ada Programming Language**, ANSI/MIL-STD-1815A-1983.

[BUCHERT81] Buchert, R.F., K. H. Evers, and P. R. Santucci, "SADT/SAINT Simulation Technique," *National Aerospace and Electronics Conf. Proc.*, 1981.

[BUHR84] Buhr, R.J.A, **System Design With Ada**, Prentice Hall, Englewood Cliffs, NJ, 1984

[COMBELIC78] Combelic,D., "User Experience with New Software Methods (SADT)," *Proc. NCC*, Vol. 47, 1978, pp. 631-633.

[MUNCK85] Munck, R, "Toward Large Software Systems that Work," *AIAA/ACM/NASA/IEEE Computers in Aerospace V Proc.*, Oct. 21-24, 1985.

[ROSS75] Ross, D., et al, **SADT Structured Analysis and Design Technique Author Guide**, SofTech, Inc. 6490-1, October, 1975, Waltham, MA.

[SAIB85] Saib, S., "A Life-Cycle Environment," *AIAA/ACM/NASA/IEEE Computers in Aerospace V Proc.*, Oct. 21-24, 1985.

[SMITH81] Smith, D.G., "Integrated Computer-Aided Manufacturing (ICAM) Architecture Part II -- Automated IDEF-0 Development," NTIS B062454-B052459, August, 1981.

[VAN DAM70] van Dam, A, and D.E. Rice, "Computers and Publishing: Writing, Editing, and Printing," **Advances in Computers**, Academic Press, New York, 1970.

Biography

Robert Munck received an AB in computer science from Brown University. In twenty years in the field, he has taught at Brown and worked at SofTech, Prime Computer, the Naval Research Lab, and MITRE, where he is presently writing a CAIS operating system in Ada for the Intel 80386.

TOWARDS A DOCUMENT STRUCTURE EDITOR FOR SOFTWARE REQUIREMENTS ANALYSIS

Vincent J. Kowalski and
Dr. Anthony A. Lekkos, University of Houston
Clear Lake

1. Introduction

Of the six or seven phases of the software engineering life cycle, requirements analysis tends to be the least understood and the least formalized. Correspondingly, a scarcity of useful software tools exist which aid in the development of user and system requirements.

In this paper we propose that requirements analysis should culminate in a set of documents similar to those that usually accompany a delivered software product. We present the design of a software tool, the Document Structure Editor, which facilitates the development of such documentation.

The requirements analysis phase of the software engineering life cycle may be defined as the phase of software development in which the requirements of the user of a proposed software package are identified in a precise, complete and logically coherent manner [6,7]. System constraints that result from the target hardware as well as non-functional constraints such as budget, time, and human resources must also be a part of a complete requirements analysis.

Two approaches to the problem of representing software requirements that appear frequently in the literature are:

- natural (textual) language approach [10, 12]
- formal representation approach [3, 5, 9, 12, 19]

The first of these attempts to specify requirements in a manner that is easily developed and understood by humans. It has the disadvantage that it may give rise to logically incorrect sets of requirements. The second approach, though it prevents logical inconsistencies, has as its main drawback the fact that a formal language must be used. This is not necessarily a desirable situation since user requirements are best provided by users, not programmers.

Several software packages are spoken of as aids to the requirements analysis phase of the software engineering life cycle. A list of some of the more well-known of these packages is the following:

- PSL/PSA [21]
- SREM [17, 18]
- SADT [15, 16]
- SSA [8]
- HOS [2, 13]
- Gist [1]

A close examination of the above tools has revealed that they are more suitably classified as program design and structure tools. Though the design of code is an essential phase in the software engineering life cycle, it is most appropriately thought of as largely independent of requirements analysis.

Finally, the relative importance of good requirements analysis is the motivation for this work. Several studies have shown that the further a software project is along in the software engineering life cycle, the more difficult and costly it is to fix bugs, make changes, and add new requirements [4, 11]. As we have found, requirements are a difficult part of software development because of the lack of automated tools that specifically aid requirements generation and maintenance

2. Document Structure Editor

2.1 Purpose and Goals

The complete set of documentation that in general accompanies a delivered software package provides a very complete set of requirements for that software package. Such documentation is, however, usually developed after the code for the package has been designed, implemented and tested. Examples of such documentation include:

- General Information Manual
- User Manual
  - Language (or Command) Reference
  - Guide
  - Tutorial
- System Requirements Document

The general goal of the Document Structure Editor is to provide an automated software tool for the development and subsequent management of documents such as those listed above. The most important feature of the DSE is that once the general structure of such a document is determined it may be stored as a Template for use in the generation of other similarly structured documents.

2.2 System Overview

The Document Stucture Editor system is depicted in Figure 1. At the highest level of the system are the users. Next, the users' interface to the system consists of a set of commands supplied by DSE. This interface may be taylored to a user's particular needs and in essence each user has his or her own interface to the DSE. Commands are interpreted at the next lower level in the system. These commands invoke any combination of the lowest-level components of the system. These lowest-level components are:

- Structure file
- DBMS
- Panel Primitives
- Text / Graphics Editor

The Structure File is the internal data structure that reflects the structure of a given document. In most cases, this structure will be hierarchical. The DBMS is used for archival of document Templates and the data associated with particular documents. Panel Primitives are the software packages in the DSE which perform the necessary mappings between the Structure File and a particular CRT or workstation. Finally, the Text / Graphics Editor is the means by which a user enters data (text and digitized graphical objects) into the DSE.

2.3 Basic Terminology

Below are listed the definitions of terms defined in the Document Stucture Editor. Several of the terms defined below are illustrated by Figure 2. Figure 2 is an example of a document or Template stored in the DSE. It should be noted that the document is divided into parts, which are further divided into chapters, which in turn are divided into sections. This structure is typical of most technical writing and is easily developed and stored by the DSE.

Topic The atomic unit of a document. A Topic consists of a Heading and a Body. In actual documents, a Topic may be thought of as a generic term for parts, chapters, sections, and subsections.

Heading This is a line of text that comes at the top of a Topic. A Topic must have a Heading. In a real document, a Heading may be a title, the name of a chapter or the like.

Body The Body is the content portion of a Topic. A Topic does not need to have a Body (although the DSE reserves space for a Body in every Topic).

Level The Level of a Template is how far up or down in a document's hierarchical structure you are. For example, the title of a textbook is its 0th Level, the parts are its 1st Level, the chapters are its 2nd Level, the sections are its 3rd Level, the subsections are its 4th Level and so on.

Depth Given a Level, how many Levels are contained within it. If we talk about a document (or Template) that has a title, chapters, and sections, the Depth of the Level that corresponds to the title is 3 (you include the Level you are looking at).

Breadth The Breadth of a Level is how may Topics are contained within that Level. In other words, if Template has five chapters and the Level being considered is that which corresponds to chapters, that Level has a Breadth of 5.

Template This is the sum total of all the Topics and their assigned Levels-the total document under development.

Menu A Menu is a special Command available to users of the DSE. Menu is used to build selection menus and may be invoked by a Profile or a User-Defined Command.

Profile Profiles are user-written files that consist of DSE commands and system-related commands. A profile is executed when the DSE system is entered at (or enters) some particular point. For instance, when a user logs on the sytem, the User Profile is immediately executed.

Command -Line The typing-in of DSE or User Commands is performed on a space on the terminal screen called the Command Line. Commands entered in this fashion are referred to as Line Commands.

the Level you are looking at).

Breadth The Breadth of a Level is how may Topics are contained within that Level. In other words, if Template

Breadth The Breadth of a Level is how may Topics are contained within that Level. In other words, if Template has five chapters and the Level being considered is that which corresponds to chapters, that Level has a Breadth of 5.

Template This is the sum total of all the Topics and their assigned Levels-the total document under development.

Menu — A Menu is a special Command available to users of the DSE. Menu is used to build selection menus and may be invoked by a Profile or a User-Defined Command.

Profile — Profiles are user-written files that consist of DSE commands and system-related commands. A profile is executed when the DSE system is entered at (or enters) some particular point. For instance, when a user logs on the sytem, the User Profile is immediately executed.

Command -Line — The typing-in of DSE or User Commands is performed on a space on the terminal screen called the Command Line. Commands entered in this fashion are referred to as Line Commands.

Command -User — A User Defined Command is similar to a Profile, except that a User Defined Command may be invoked anywhere in the system a Command Line is available. The User Command consists of DSE and host system commands and is assigned a name by the user who writes it.

Command -General — Commands are the means by which a user tells the DSE what to do. Commands are the basis for the interface between the user and the DSE system.

Scroll — Scrolling a Template is a feature of the DSE that allows a user to view a Template as one continuous piece of text.

DSE — Software that converts DSE or User Commands into instructions that the host computer understands.

Structure File — A file that contains the Level information and hence the structure of a Template. It is defined here for the purpose of completeness.

Currency — The DSE "knows" what template or topic or whatever you might be referring to by keeping Current values for such items. The currency is usually set using some Select command.

Command -Key — A Key Command is an association (or mapping) between a short key sequence and a DSE or User Command. These associations are defined in a Profile and the last Profile executed takes precedence over any previous Profiles with regards to these Key Command definitions.

3. Related Efforts

In many respects, storing the associated structure of a given document is the logical next step for word processing software packages. Several commercial packages have structure editing capabilities. These packages generally fall into one of two categories:

- Automatic Indexers
- Outliners

Automatic indexing software usually is available as an option to many popular word processors. Outliners, on the other hand, have outlining of documents as their primary purpose with limited word processing capabilities. Such packages run on microcomputers exclusively. In addition, the integration of the components of these packages is questionable [20].

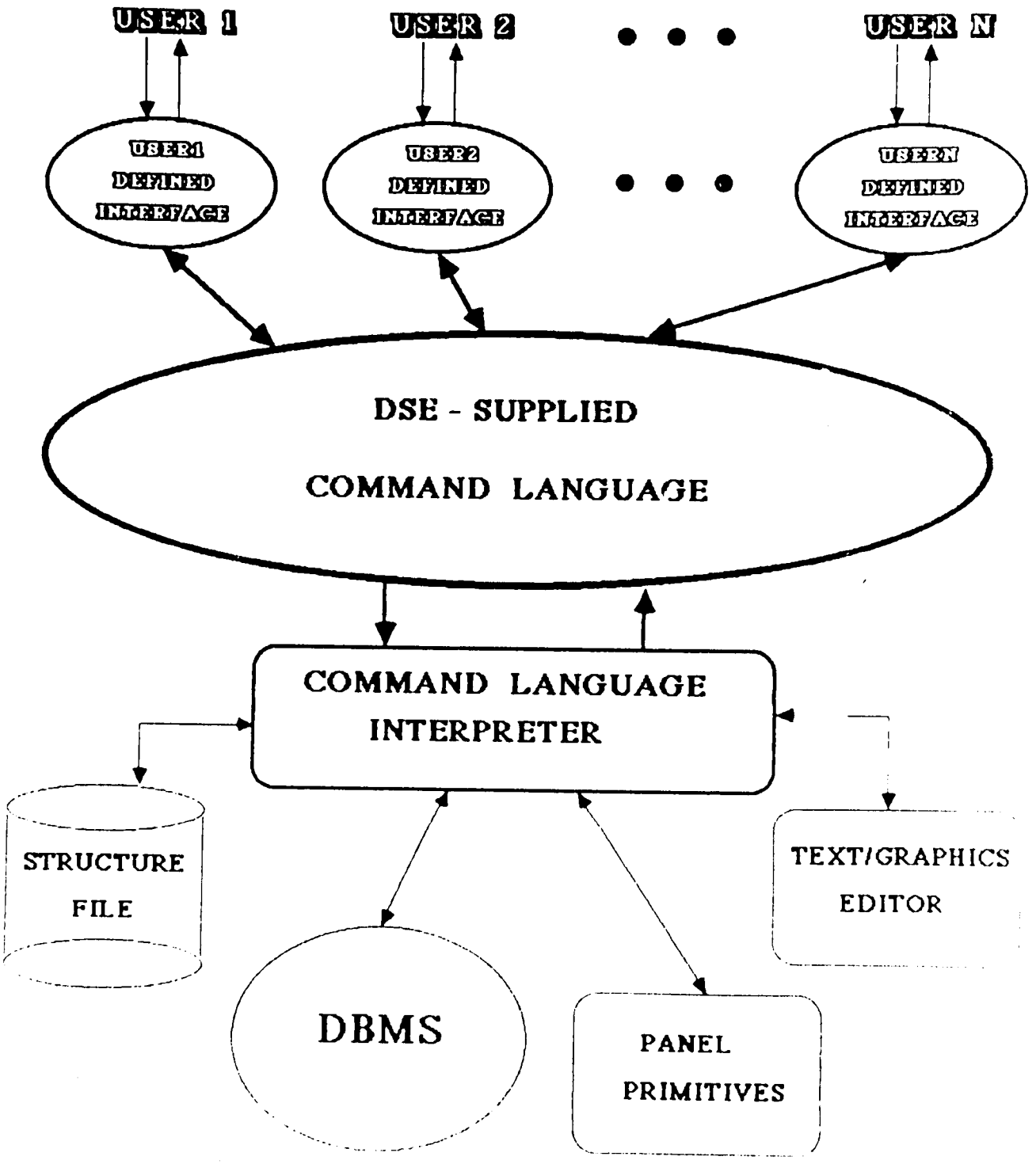

USER 1
USER 2
USER N
USER1 DEFINED INTERFACE
USER2 DEFINED INTERFACE
USERN DEFINED INTERFACE
DSE - SUPPLIED
COMMAND LANGUAGE
COMMAND LANGUAGE INTERPRETER
STRUCTURE FILE
TEXT/GRAPHICS EDITOR
DBMS
PANEL PRIMITIVES

Figure 1.

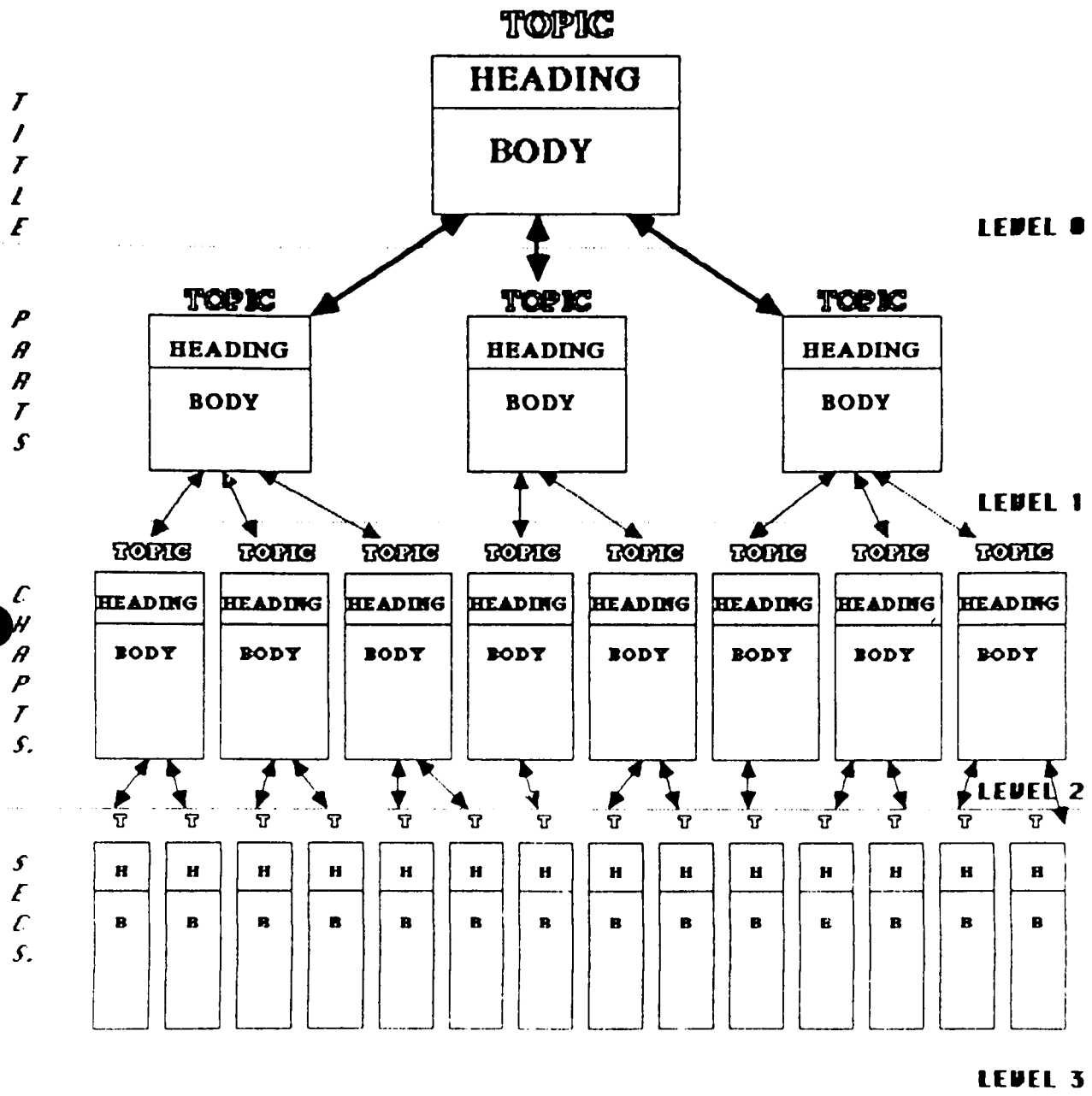

TOPIC
HEADING
BODY
TITLE
LEVEL 0
TOPIC
HEADING
BODY
TOPIC
HEADING
BODY
TOPIC
HEADING
BODY
PARTS
LEVEL 1
TOPIC
HEADING
BODY
CHAPTS.
LEVEL 2
H
B
SECS.
LEVEL 3

Figure 2.

References

[1] Balzer, R., Gist Final Report, Information Sciences Institute, University of Souther California, Feb. 1981.

[2] Barlas, J., "Higher-order computing generates high order business for Cambridge Company," Mass High Tech, Jul. 23, 1984.

[3] Bell, T., et al.: "An Extendable Approach to Computer-Aided Software Requirements Engineering,"Trans. Software Eng., Jan. 1977.

[4] Booch, G., Software Engineering With Ada, Benjamin/Cummings Publishing Co., Menlo Park, California, 1983.

[5] Borgida, A., and Greenspan, S., "Knowledge Representation as the Basis for Requirements Specifications," IEEE Computer, Apr. 1985.

[6] Fairley, R., Software Engineering Concepts, McGraw-Hill Publishing Co., New York, 1985.

[7] Freeman, P., "Requirements Analysis and Specification: The First Step," Advances in Computer Technology, Aug. 1980.

[8] Gane C., and Sarson, T., Structured Systems Analysis: Tools and Techniques, Prentice-Hall, Englewood Cliffs, N.J., 1979.

[9] Greenspan, S., et al., "Capturing More World Knowledge in the Requirements Specification," IEEE Proceedings of the Sixth International Conference on Software Engineering, 1982.

[10] Heninger, K, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," IEEE Transactions on Software Engineering, Jan. 1986.

[11] Jones, C., Programmer Productivity, McGraw-Hill, New York, 1986.

[12] Levene, A., and Mullery, G., "An Investigation of Requirements Specification Languages: Theory and Practice," IEEE Computer, May 1982.

[13] Martin, J., System Design From Provably Correct Constructs, McGraw-Hill, New York, 1985.

[14] Robinson, L., et al., "A Formal Methodology for the Design of Operating System Software," Current Trends in Programming Methodology, vol I, Prentice-Hall, Englewood Cliffs, N.J., 1977.

[15] Ross, D., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering, Jan. 1977.

[16] Ross, D. "Applications and Extensions of SADT," IEEE Computer, Apr., 1985.

[17] Rzepka, W., et al., "Requirements Engineering Environments: Software Tools for Modeling User Needs," IEEE Computer, Apr 1985

[18] Scheffer, P., et al., "A Case Study of SREM," IEEE Computer, Apr 1985

[19] Shaw, A., "Software Specification Languages Based on Regular Expressions," in Software Development Tools, Springer-Verlag, Berlin, 1980.

[20] Spezzano, C., "Unconventional Outliners", PC World, Mar. 1986

[21] Teichrow, D., et al., "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," Trans. Software Engineering, Jan. 1977

DEC Ada* Interface to Screen Management Guidelines(SMG)

by

Somsak Laomanachareon
Dr. Anthony A. Lekkos [1]

University of Houston, Clear Lake

INTRODUCTION

DEC's Screen Management Guidelines are the Run-Time Library procedures that perform terminal-independent screen management functions on a VT100-class terminal. These procedures assist users in designing, composing, and keeping track of complex images on a video screen.

There are three fundamental elements in the screen management model: the pasteboard, the virtual display, and the virtual keyboard. The pasteboard is like a two-dimensional area on which a user places and manipulates screen displays. The virtual display is a rectangular part of the terminal screen to which a program writes data with procedure calls. The virtual keyboard is a logical structure for input operation associated with a physical keyboard. Other features included in SMG are input and output operations, control of asynchronous actions, optimizing performance, and many more.

SMG can be called by all major VAX languages. Through Ada, we use predefined language Pragmas to interface with SMG. They are Pragma Interface and Pragma Import_Valued_Procedure. In association with these Pragmas, we also used the three other predefined packages: System, Condition_Handling, and Starlet. With these predefined Pragmas and packages, we can put together another package that contains all the procedure calls to SMG which allow Ada application programs to access the SMG.

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

[1] Supported by NASA/JSC-UHCL Ada Beta-test site contract

The Screen Management procedures provide terminal independence by allowing user to perform all screen functions without concern for the type of terminal being used and if the terminal being used does not support the requested function in hardware, the Screen Management procedures perform the requested function by emulating it in software. The important aspect of the Screen Management Facility is the separation of user programs from the physical device. For example, the user program writes to the virtual display instead of the physical screen. The separation of virtual operations from physical operation allows the terminal-independent aspect to be realized.

Working with the SMG involves three fundamental elements in the screen management model. First, a pasteboard is always associated with a physical device. A pasteboard can be either smaller or larger than the physical screen, but each output device can have only one pasteboard associcated with it. A pasteboard can be thought of as a logical coordinate system where position(1,1) corresponds to the upper left-hand corner of the screen. With this coordinate system, the virtual display, described later, can be placed anywhere and it may be partly visible on the physical screen.

Second, a virtual display is a rectangular part of the terminal screen to which a program writes data and text. When a virtual display is associated with a pasteboard, it is said to be pasted. When the display is removed from a pasteboard, it is said to be unpasted. To make a virtual display visible, you have to paste to a pasteboard. Your program can create and maintain several virtual displays and each display can be pasted to more than one pasteboard at the time.

Third, a virtual keyboard is a logical structure for input operation associated with a physical keyboard or it maybe associated with any file accessible through Record Management Services(RMS). The advantage of using virtual keyboards is device independence. The Screen Management procedures maps the different of code sequences into a uniform set of function codes.

All the attributes associated with pasteboards, virtual displays, and virtual keyboards that your program created can be modified and maintained at all times. A virtual display can be pasted, unpasted, and moved around a pasteboard. Input and output of each virtual display is independent of each other.

Text can be added, inserted, and deleted from a virtual display. Their video attributes can also be altered. The cursor position on a virtual display can be requested or set to any position on the virtual display.

The cursor position on a virtual display should not be confused with the physical cursor position on the screen. Although each virtual display has an associated virtual cursor position, only the cursor position on the most recent modified virtual display corresponds to a physical cursor. Line drawing capabilities and control of asynchronous events are also provided as well as support of Non-DIGITAL terminals.

SMG can be called by all major VAX languages. In Ada, predefined language Pragmas are used to interface with SMG. Pragma Interface which allows Ada program to call subprogram written in another language. A Pragma Interface has the following form

Pragma Interface (language-name, subprogram-name);

Together with Pragma Interface, the Pragma Import_Valued_Procedure is specially designed for calling system routines. System routines return status values using the same parameter-passing as Ada uses for returning function results. Some system routines also cause side effects on its parameters. Ada treats a routine that returns a result as an Ada function, but a function with IN OUT or OUT parameters is not legal in Ada. Pragma Import_Valued_Procedure allows such a routine to be interpreted as a procedure in an Ada program, and as a function in the external environment. Note that the first parameter of the imported procedure must be an OUT parameter passed value. The value is returned as function value. The other parameters can be specified with the mode IN, IN OUT, or OUT, according to the service routine parameters. For example:

```
with System, Condition_Handling;

package SMG is

  procedure Create_Pasteboard
    (Status : out Condition_Handling.Cond_Value_Type;
     Pasteboard_Id : out Integer;
     Output_Device : String  := String'Null_Parameter;
     Rows, Columns : Integer := Integer'Null_Parameter;
     Screen_Flag   : Boolean := Boolean'Null_Parameter);
```

```
  pragma Interface (SMG, Create_Pasteboard);
  pragma Import_Valued_Procedure
    (Create_Pasteboard, "SMG$CREATE_PASTEBOARD");

procedure Create_Virtual_Display
    (Status : out Condition_Handling.Cond_Value_Type;
     Rows, Columns : Integer;
     Display_Id    : out Integer;
     Display_Attribute,
     Video_Attribute,
     Char_set      : System.Unsigned_Longword
           := System.Unsigned_Longword'Null_Parameter);
  pragma Interface (SMG, Create_Virtual_Display);
  pragma Import_Valued_Procedure
    (Create_Virtual_Display, "SMG$CREATE_VIRTUAL_DISPLAY");

             ...       ...        ...        ...
             ...     Other procedures        ...
             ...       ...        ...        ...

end SMG;
```

From the example above, the package System provides types and operations for manipulating system-related variables and parameters. The package Condition_Handling provides VAX Ada types for VAX/VMS condition values as in the above status parameter which is returned by a system routine. Another package, not shown, is Starlet which provides VAX Ada type, VAX Ada constants for symbol definitions, and VAX Ada operations for calling system and RMS services. The package Starlet is specially useful in the application program which calls procedures in the SMG package that use symbol definition, for example:

```
with SMG, System, Condition_Handling, Starlet;

procedure Screen is
   Status   : Condition_Handling.Cond_Value_Type;
   Screen_1 : Integer;
    ...         ...          ...
    ...         ...          ...
begin
    ...         ...          ...
    ...         ...          ...
   SMG.Create_Virtual_Display
       (Status,
        Rows            => 7,
        Columns         => 70,
        Display_Id      => Screen_1,
        Video_Attribute => Starlet.SMG_M_REVERSE);
```

```
    ...          ...          ...
    ...          ...          ...
end Screen;
```

As shown in the example, all output in the virtual display named Screen_1 will be in the reverse video.

With these packages and pragmas, we can put together a package which contains all the Screen Management procedures that we need. Then Ada application programs can use this Screen management package to create and manage application screens.

A PROPOSED CLASSIFICATION SCHEME FOR ADA-BASED SOFTWARE PRODUCTS

Gary J. Cernosek
McDonnell Douglas Astronautics Co. - Houston
16055 Space Center Blvd.
Houston, Texas 77062
(713) 280-1500

1.0 INTRODUCTION

As the requirements for producing software in the Ada* language become a reality for projects such as the Space Station, a great amount of Ada-based program code will begin to emerge. Although this software will exist in Ada source code form, it will display varying degrees of quality based on the manner in which it was developed. In spite of the fact that Ada supports the most modern and effective concepts of programming available, poorly written programs can be created in Ada just as they have been in previous languages.

Consequently, the term "written in Ada" could have many connotations. The mere fact that a program exists in Ada source code form does not imply to any degree that there is any more quality in that product than would be if it were written in FORTRAN or C. If the modern features of the Ada language are not utilized to support the principles of software engineering, then the entire motivation and justification for moving to the Ada language will be defeated.

Recognizing this potential for varying levels of quality to result in Ada programs, what is needed is a classification scheme that describes the quality of a software product whose source code exists in Ada form. This classification assessment would be based on the overall process in which the software was developed, as well as the characteristics and attributes associated with the resulting source code produced. This provides an "after the fact" evaluation, and thus will not directly support proper development. However, the knowledge of the classification scheme may help in deterring bad development approaches and indirectly increase the overall quality consciousness of Ada-based software development.

This paper proposes a 5-level classification scheme that attempts to decompose this potentially broad spectrum of quality of which Ada programs may possess. The number of classes and their corresponding names are not as important as the mere fact that there needs to be some set of criteria from which to evaluate programs existing in Ada. An exact criteria for each class is not presented in the paper, nor are any detailed suggestions on how to effectively implement this quality assessment. The paper is merely intended to introduce the idea of Ada-based software classification and to suggest a set of requirements from which to base further research and development.

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

2.0 THE NEED FOR A CLASSIFICATION SCHEME

The purpose of the Ada language can be viewed from two perspectives. Technically, Ada was designed to strongly support the goals and principles of software engineering. However, the main influence driving the definition of Ada was economical. The "software crisis" was recognized in the early 1970's and the major cost factors were identified in software maintenance activities. Therefore, Ada was designed to give the potential for reducing software costs. Cost reductions start by providing a common language that consequently requires less compiler development and less programmer re-training. And as the amount of Ada code developed increases, the re-use of verified software components can further decrease development expenses.

Since the discipline of software engineering focuses on both technical and economic issues, the Ada language must be used as a software engineering tool and not merely as another programming language. Ada will not automatically meet its purpose and goals - it has to be used as it was designed to be used.

Therefore, it is unrealistic to expect that all software projects developed in Ada will realize the many benefits that the language has to offer. This is true not because the language is deficient, but rather because there are many different approaches to using any language. Several reasons why Ada may not be properly used on initial projects are outlined below:

Technical - The education and training required to learn how to effectively use Ada may be significant, especially for individuals without previous exposure to higher-level languages. Ada quality may suffer by having improperly trained personnel pre-maturely work on Ada development efforts.

Economical - The initial costs involved in moving to any new language are high. This characteristic may drive decision makers to short-term solutions, such as code translation approaches.

Political - Many organizations feel they are "locked" into a particular programming language, and often the machines that run their software. Even when Ada is shown to be technically superior and actually cost-effective, political influences can stifle attempts to upgrade an outdated software development environment.

Inertial - It is only natural for organizations to be reluctant to change. Ada, as well as other advances in computer engineering such as distributed processing, may intimidate people who feel more comfortable with their present environment. This natural state of inertia should be accepted and effectively dealt with rather than be a front line for personal battles.

With these issues and many more to contend with, it is obvious that most organizations will have to transition into an Ada environment. As this transition is taking place (and possibly thereafter), a varying degree of quality must be expected to result among different development efforts. One way to measure the progress of transition is to classify the quality of the Ada software resulting from these efforts. The goal must be set to produce only the highest level of quality in Ada software. However, the reality must be recognized that it will be difficult to meet this goal in initial projects.

The suggested approach is to get started with Ada and do the best job possible under whatever circumstances may exist. The previously described road blocks should not prevent the exploration of Ada. However, the learning curve must be steep and be based on good sources of Ada training and education. Poor development habits must be broken and good ones must be created and enhanced. And most importantly, engineers and managers have to encourage the training and use of Ada. Without both peer-level and management support, effective transition to Ada will be difficult.

The most important theme to understand and constantly keep in mind is that the basis for "good" and "bad" rest in the goals and principles of software engineering. Software engineering represents the stable point of professional programming that can separate quality standards from personal style and allows concentration on issues above the language level.

Therefore, in order to measure the progress of transitioning to Ada, a software engineering-based classification method is needed. This is also in accordance with the DOD-STD-2167 Software Documentation Standard, which has changed the emphasis on Quality Assurance to Quality Evaluation. A proposed classification scheme for evaluating Ada software quality is presented in the next section.

3.0 CLASSIFICATION METHOD AND CRITERIA

Each of the classifications below are described with the following format:

- **Classification level number: 5 (lowest) to 1 (highest)**
- **Development Process Statement - phrase that references the approach taken in development:**
  - **Level 5 - "Translated To Ada"**
  - **Level 4 - "Coded In Ada"**
  - **Level 3 - "Programmed In Ada"**
  - **Level 2 - "Designed Into Ada"**
  - **Level 1 - "Engineered With Ada"**
- Description of the process in which the program source code was created
- Characteristics and attributes indicative of the particular level of quality

Level 5 - "Translated To Ada"

This lowest class of Ada software implies nothing more than the fact that the program code exists in Ada form. The Ada code is created by some type of code translation, either through a manual and direct mapping performed by a human coding specialist, or by an automated code translator. Level 5 classification is intended for programs that have been previously developed in another language and have been converted to Ada merely to meet a requirement for the software to exist in Ada. However, programs that have been properly re-structured or re-designed into Ada have potential for a higher quality assessment.

The characteristics of Level 5 software include significant maintenance problems due to lack of readable and understandable code. None of the aesthetic qualities of the Ada language are evident due to the absence of human engineering. Additionally, the overall program structure is characteristic of the original language's form and represents the most inappropriate and ineffective use of the Ada language. A possible exception to this evaluation is when an organization wants to escape the previous language environment and allow 100% of its future development and maintenance in Ada.

Level 4 - "Coded In Ada"

Although Level 4 programs are humanly written in Ada, they lack the basic quality characteristics possible in good Ada programs. The development process is generally based on program development personnel that are not properly trained in utilizing the Ada language and its support environment properly and effectively.

The approach to development is ad hoc with no basis on formal software requirements definition and no documented design process. Level 4 developers incorporate coding semantics of other languages into their Ada programs that are inappropriate to Ada.

Corresponding characteristics include abbreviated identifiers, unstructured control features, and lack of effective problem modeling and abstraction due to the absence of appropriate data structures. Overall program design lacks modularity, utilizes excessive amounts of global data structures, and fails to control visibility of objects with the information hiding techniques of package structuring. The characteristics of Level 4 software defeat the purpose of requiring the Ada programming language for program development. A possible exception here is to allow developers to get started with Ada for hands-on training. However, in this case, developers must learn proper Ada structure very quickly.

Level 3 - "Programmed In Ada"

Level 3 represents the lowest acceptable criteria for justifying the existence of software in Ada form. The developers are properly trained in the basic principles of the language and know how to effectively utilize its features for developing readable and maintainable software. The software requirements are known and understood with a significant amount of pre-implementation thought going into the design of the program structure.

Level 3 programs have meaningful identifier names, use only structured programming constructs, and accurately model real-world objects with appropriate data structures. Program structure is highly modularized with inter-module coupling minimized and internal module structure strongly cohesive. Packages are properly used to support principles of information hiding, object encapsulation, and abstract data types. Visibility of objects is strongly controlled, data is strongly typed, and use of global objects is strictly limited.

Level 2 - "Designed Into Ada"

This level of quality concentrates on issues above the programming language level. A software design approach is adopted to properly define the structure of the modules of the software system independently of the implementation details of the target programming language. One or more design methodologies may be used to create consistency and reliability in the program structure. Since Ada directly supports the principles of good software design, an Ada-based Program Design Language (PDL) is very appropriate. However, the main idea is that the software system is specified and verified to a large degree prior to the implementaton phase, at which point problems and errors are much more costly to correct.

The main characteristic of Level 2 software is that the overall software system design displays a very understandable structure that allows reliable modifications and enhancements. Software design documents are produced as deliverable products prior to program source code development. The design methodologies may be supported by automated tools that help verify interface consistency and requirements completeness. The actual source code programs resulting from the software design display all of the quality attributes associated with Level 3 software. Consequently, Level 2 software is more reliable, understandable, and more easily adapted to new applications.

Level 1 - "Engineered With Ada"

This classification corresponds to the highest degree of quality possible in Ada-based software. The software is created with a comprehensive software life-cycle approach by developers who are well trained and knowledgeable in the goals and principles of software engineering. The main emphasis in the process is in the distinction between the problem domain and the solution domain of the computer-based solution. The requirements analysis phase of development is utilized to fully understand the problem space and to determine exactly what the software is to do in the first place. A variety of methodologies and technologies may be used to ensure that valid requirements are specified up front and that the associated costs and risks are reduced. The analysis phase may include utilization of techniques such as rapid prototying and higher-level applications generators for defining and refining user interface and system requirements, and for generating feedback from the user community. The remaining phases of design, implementation, testing, and debugging are all in the solution space of the development process and are concerned with how to meet the requirements specification.

Software that is engineered with Ada strongly supports the goals and principles of software engineering. Analysis is the main key to understanding which components of the software design actually need to be developed from scratch and which ones can be satisfied by existing reusable components. A very coherent and useable set of documentation is produced in the engineering process relating to the various phases of the life cycle, as well as documentation applicable to all phases of development. The concept of a project data or object base is realized and implemented for accurate control and accountability of personnel, products, and organizational information. Automated support tools are effectively utilized throughout all forms of development to increase productivity, support proper and disciplined development, and to reduce the manual effort required from software developers. And finally, an intense concern for maintainability is prevalent throughout all decision-making and phases of development.

4.0 USE OF HIGHER-LEVEL SPECIFICATION LANGUAGES

It is difficult to assess the quality of Ada code that is automatically generated from a higher level of specification. The concept of an executable requirements or design language does support software engineering principles. However, the issue of quality rests in the question of what level of specification will the software be maintained at. If it is strictly at the higher level of requirements or design specification, then the actual source code generated will not be visible to the human progammer, and thus its structure will not be of great significance. However, if the resulting Ada code will be subject in any way to human analysis and subsequent modification, then the level of quality will be directly related to the same factors associated with well-engineered and manually-written Ada programs.

Therefore, in this latter case, the attractive process of generating Ada source code from a higher level of specification must be designed such that the corresponding characteristics and attributes associated with the resulting code coincide with those indicative of well-written Ada software developed directly by a human programmer. The degree of quality associated with the higher-level specification will consequently be based on the degree to which the automatically generated code displays the good human engineering principles needed for understandable and maintainable software.

5.0 CONCLUSIONS

The usefulness of the preceding classification scheme for Ada-based software is highly dependent on a more precise and tangible definition of criteria for each class. Although this level of detail was not given, the taxonomy proposes a starting point from which to base futher analysis. The main idea of the paper is to create an awareness of the potential problems to expect when transitioning to a new programming language such as Ada. The Ada language alone cannot solve the problems currently prevalent in large organizations such as NASA in which software costs are a significant portion of the budget. Ada, and its corresponding support environment, merely provide the best available set of tools which support and encourage the adherence to the proven and solid principles of software engineering.

The mandate for the Space Station Program to move into the "Ada culture" will be totally ineffective if engineering principles and corresponding methodologies are not properly utilized. Obviously, education and training will be essential for developing a smooth transition into the software engineering discipline. The spectrum of potential Ada software quality classes presented here can help create and maintain the awareness and importance of viewing software engineering as a true engineering discipline. This recognition will be essential for the success of the up-coming proliferation of Ada-based software projects in the Space Station Program.

Omit

COMMISSION
OF THE
EUROPEAN COMMUNITIES

INFORMATION TECHNOLOGIES
AND TELECOMMUNICATIONS
TASK FORCE

The Status of Ada in Europe

Dr Mike W Rogers
at Information Technologies and Telecomms Task Force
Commission of the European Communities
A25 5/15
200 Rue de la LOI
B1049 Brussels
Belgium

31.3.86

There are currently no better candidates for a coordinated, low risk synergetic approach to software development than the Ada programming language and the associated environment work. Developed in Paris in the mid 70's, Europe has developed centres of excellence on the aspects of Ada technology, and the indigenous industry is now experimenting with applications. Some 2M lines of Ada code exist already in use.

The aim of the presentation would be to build on a paper prepared for the May 1985 Paris Ada conference based on an extensive survey of the penetration achived by Ada. Furthermore there would be a summary of three major activities in Europe inthe month of May 1986 -
- the 2nd Ada Users Congress
- the 4th Ada Europe/SIGAda JT Conference in Edinbrugh
- A survey on Tools published about then.

The nature of applicatiosn suggest that more details will be available if only abstracts are published; as domains often lie in sensitive areas of an organisations activities.

Space is of particular interest to the EEC, who support civil applications; and some Ada research and development. This area is ideal for test bedding Ada,as Ada can bridge different approaches to problem solving by use of its portability.

Finally; the "social" infrastructure of Ada R and D in Europe will be summarised.

M W Rogers

Arpanet : mrogers at USC-ISIF
Adakom : m w r c
Eurokom : Mike W R

ADA(R) ASSESSMENT : AN IMPORTANT ISSUE

WITHIN THE EUROPEAN COLOMBUS SUPPORT TECHNOLOGY PROGRAMME

P. VIELCANET

INFORMATIQUE INTERNATIONALE
2, rue Jules Védrines
31400 TOULOUSE FRANCE

Tél. (33) 61.34.01.92

1. INTRODUCTION

Software will be more important and more critical for COLUMBUS than for any ESA previous project. As a simple comparison, overall software size has been in the range of 100 K source statements for EXOSAT, 500 K for SPACELAB with IPS, and will presumably reach several millions lines of code for COLUMBUS (all elements together).

Based on past experience, the total development cost of software (facilities, simulation, test items, on-board software...) can account for about 10 to 15 % of the total space project development cost. For COLUMBUS, this share will grow over the entire space system life cycle as maintenance and evolution will be vital within its very long operationnal phase. Considerable savings will be possible by properly managing software and by exploiting fields of commonality.

The Ada technology may support the strong software engineering principles needed for COLUMBUS, provided that technology is sufficiently mature and industry plans are meeting the COLUMBUS project schedule.

Over the past three years, Informatique Internationale has conducted a coherent programme based on Ada technology assessment studies and experiments, for ESA and CNES as indicaded in figure 1.

This specific research and development programme benefits from Informatique Internationale fifteen years experience in the field of space software development and is supported by the overall software engineering expertise of the compagny (e.g deep involvement in the european ESPRIT and MAP programmes).

(R) ADA is a registered trademark of the US Department of Defense

2. ADA TECHNOLOGY ASSESSMENT PROGRAMME

The logical construction of the space station oriented Ada technology assessment programme appears in figure 1. Four main layers may be distinguished :

a) Ada development environments procurement policy (Rolm ADE and Verdix VADS), set up of convenient methods and development of new tools :

GET, a tool for automatic production of interactive test environments for Ada packages.

SOPHIA, an advanced syntax-directed editor for Ada designed to operate on advanced work stations and providing features for adding new functionalities (e.g. static or dynamic analysis of programs).

b) Ada space specific experiments for CNES and ESA aiming at a rather broad investigation (e.g. ground and space segments) :

ADEXII, a two years experiment and assessment project undertaken for CNES (100 man-months budget over 83-85) with following main tasks based on careful monitoring of the activity :

. Assessment of the Ada language with respect to training, effective use and degree of applicability

. Assessment of the Ada environment and resulting Ada products

. Production of guidelines for an efficient transition to Ada.

ESA/ADA, one year experiment conducted for ESA in 84-85, aiming at the Ada development of a complete simulation of the GIOTTO spacecraft Attitude and Orbit Control System from an existing Fortran program. The organization of the project based on partial and parallel development by INFORMATIQUE INTERNATIONALE, CESELSA (sub-contractor) and ESA itself successfully demonstrated unique features and suitability of the Ada language for large space projects (significiant guidelines on an Ada development methodology have been established).

CCSDS, six months project conducted for CNES in 85 demonstrating the successful use of Ada as a data description and data handling language for the GALILEO spacecraft telemetry (modelling and processing according to the international CCSDS standards).

c) On-board Data Management System (COLUMBUS class) feasibility studies

- ESA/OBCA, comparative study on distributed microprocessor based computer system architectures

- ESA/HOL, a study of the applicability of High Order Languages for on-board software production (assessment and selection of the best candidate among Ada, Modula 2, C, LTR 3, Pascal and HAL/S).

d) Ada detailed assessment for COLUMBUS on-board distributed Data Management System.

ESA/SSADA, two years project (start end 85) investigating three important issues :

- availability of Ada tools (near and mid-term) for the development of distributed application software
- links between Ada features (language and implementation) and specific requirements of a typical space station mission
- specification and development in Ada of a study case software system (derived from space station requirements analysis) which can produce significiant insights on powerful model of future Ada software production environments.

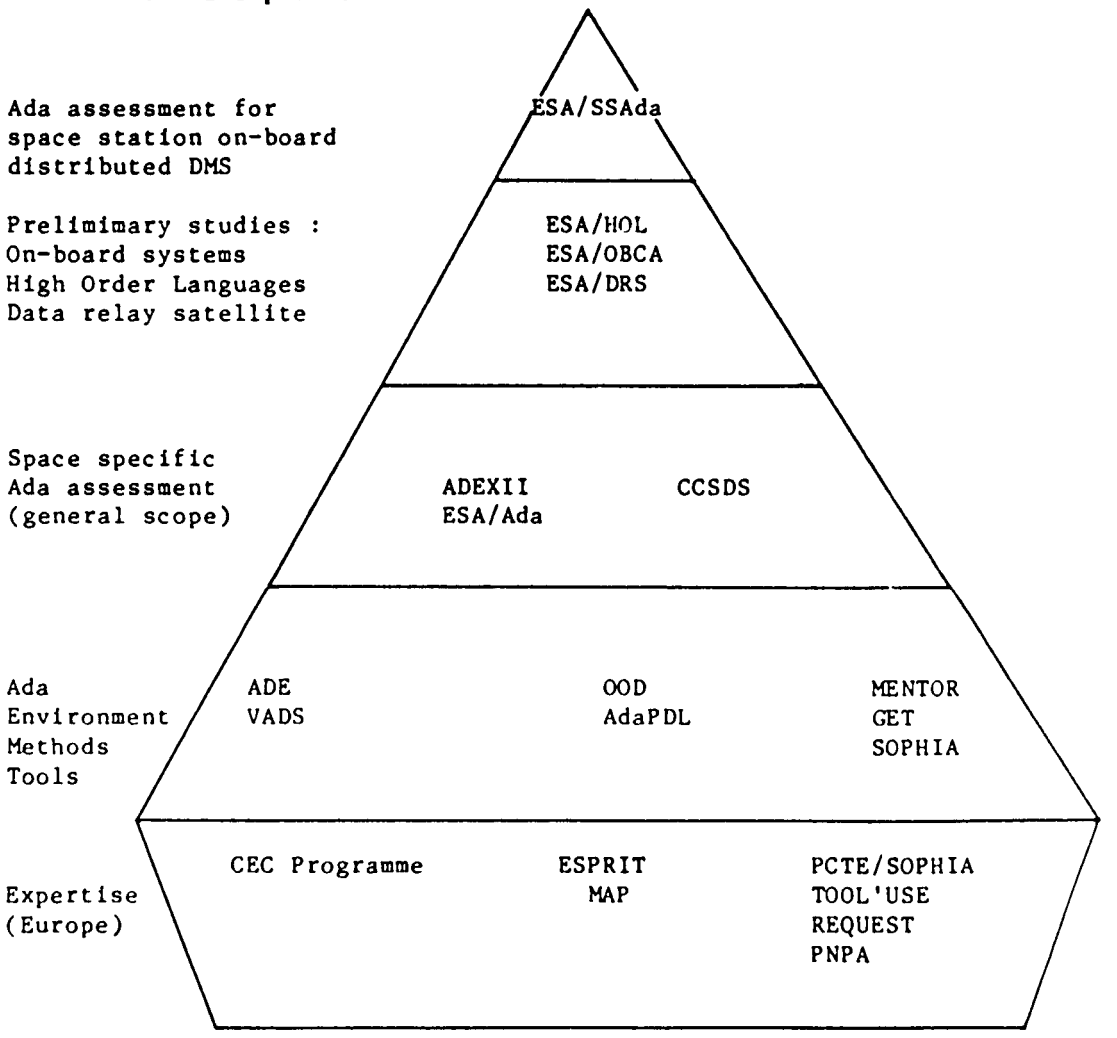


FIG. I. : INVOLVEMENT IN COLUMBUS SUPPORT TECHNOLOGY

3. PRESENTATION OF CNES AND ESA ADA EXPERIMENTS

3.1. CNES ADEXII EXPERIMENT

As previously stated, Informatique Internationale conducted an Ada experiment for the french national space agency (CNES) in Toulouse, France. The experiment main objectives were to provide information on the suitability and effective use of the Ada language for space applications and to locate the potential benefits and possible drawbacks to be expected when introducing Ada into the aerospace industry environment.

As such results and lessons learnt can contribute to a better understanding and management of a space-oriented Ada technology transfer. Education and development methods were especially discussed. The experimental data collected over the project have been extracted from a development effort of six software engineers over two years with a total production of 30 000 Ada source lines (ASL).

The experiment had then to cover two main areas :

- introduction of the language (i.e. how it is used and learned in practice by personnel with different technical backgrounds)

- suitability of the language for applications specific to the aerospace industry, particulary real-time applications.

These topics were futher refined, analyzed and balanced against technical Ada constraints (mainly lack of information and training on Ada software engineering) and three evaluation areas were defined :

- learning and use of the Ada language

- development of Ada software products

- performance and assessment of a validated Ada environment.

To reach these goals within budgetary constraints, it was decided to redesign and redevelop existing Fortran applications, meanwhile monitoring related activities. These applications corresponding to small-scaled projects were preferred to a single large real-time project, due to the high risks implied by such a choice at the time the project started. Previous papers (Labreuille 84 and Papaix 85) give an in-depth discussion of the project tasks and the resources involved.

With respect to the initial objectives, the following conclusions were reached :

Productivity

High productivity ratios have been experienced (up to 1400 ASL per man-month for small Ada developments) but this data should be interpreted with care and balanced against a real industrial context. In this experiment context, the development team was small, motivated, enthusiastic and experiencing the learning process and the use of Ada and programming environment tools.

More than the achievement of good productivity figures over the project, the identification of the main contributors to productivity improvements were pointed out :

- early validation through the use of Ada at the design phase

- automatic recompilation features supported by convenient configuration control system

- reuse of software components

Training

This experiment has proven that acceptable level of proficiency in Ada could be reached rather quickly (in less than a month).
Ada, as a programming language is no more difficult to learn than another language, but making full use of its underlying software engineering principles requires some additional effort. Due to Ada richness, special training is required for "good use" of advanced features, as well as to avoid systematic use of "well experienced" subset.

Environment

The availability of a number of tools is of great help, but one should not forget that learning how to use them effectively is almost as important as learning the language itself and takes time and effort as well.
Evidence was shown that an Ada compiler must be a validated one, tools must be of good quality as well and should be suitable for the development of large Ada programs (more than 10 000 ASL).

Development methodology

Use of Ada impacts heavily on traditional methods through :

- early and continuous use from design

- early validation of design through prototyping and step-wise PDL refinement

- design effort which is increased by up to 50 % while integration is reduced up to 5 times

- effective parallel developement.

3.2. ESA ADA EVALUATION STUDY

As part of its Technical Research Programme, in preparation for using Ada, the European Space Agency has just completed a study to evaluate the use of Ada in a typical space-oriented software project, with particular emphasis on the impacts on METHODOLOGY and the prospects for PORTABILITY, REUSABILITY and developement at multiple sites. The study was based on rewriting in Ada the Attitude and Orbit Control Software and the simulation of the satellite dynamics and operators environment of a recent satellite, which were previously implemented in Assembler and Fortran.

As a result of this study, ESA has now a set of Ada packages which has been used to evaluate many of the existing Ada compilers and Ada supporting toolsets as reported in (Robinson 86). This proved to be a valuable way of identifying some of the key aspects for providing portable software, and for identifying strong and weak features of existing and potential APSES.

The study project was performed by Informatique Internationale (acting as prime constractor) and CESELSA (Spain) under the direction of ESA Technology Centre (ESTEC). The main activity was to rewrite in Ada

a) the Attitude and Orbit Control Equipment (AOCE) software of a recent satellite, from the existing design written in Caine, Farber Gordon PDL and the listings of the RCA1802 Assembler programs,
b) the simulation of the satellite dynamics and operators environment which were previously implemented in Fortran.

The Ada program consists of 6 components as indicated in figure 2. The core of the program is the package P_AOCE containing the satellite software. The RAM is visible to provide access to data for operator display, and part of the RAM (T_RAM1) is available to write telecommands. This package is embedded in a simulation of the real world environment, consisting of telecommand management, hardware interface, dynamics simulation and operator command/display interface.

ESA standards for software life-cycle (ESA 84) were followed to assess their suitability for Ada. These consist of phases for software requirements, architectural design, detailed design and implementation, each phase terminating in a formal review. Full documentation was produced.

The Software Requirements Document was written by Informatique Internationale to pull the requirements together and as a familiarisation task to provide a clear definition of the work to be done.

As an experiment, two Architectural Designs were produced, at both Informatique Internationale and CESELSA. Each consisted of narrative, design diagrams and Ada Specification parts. In addition, the major task structure was prototyped using TEXT_IO to provide a listing of the flow of control, thus demonstrating that the overall architecture is correct, and that the specification parts were consistent and compilable. After the review, the ADD which was based on Object Oriented Design was selected since this provided the more coherent and complete view of the design. It was decided to use OOD on the detailed design of the dynamics part in the next phase to gain more experience of this technique.

The Detailed Design was also repeated by the two contractors, using the same architecture as a baseline for each. The main difference was that Informatique Internationale decided to use SEPARATE compilation extensively in the design of the larger packages. This has the benefit of reducing the time for recompilation due to changes in only one procedure during module testing. It results in more source files and a slightly more complex library structure with therefore more need for Ada Program Library tools to manage the re-compilation and configuration management activities.

To try out the multi-site aspects of the project with a set of independently coded packages, the satellite software was programmed in ESA and the simulation parts were programmed in Spain (CESELSA). These were then integrated at a third site in France (Informatique Internationale), with the help of all parties.

Acceptance was based on 10 test cases from the ESTEC Assembler/Fortran implementation, which produced identical plots in 9 cases and a better result at the 5th significant digit in the 10th case. Differences between computers were therefore insignificant.

The main part of the study produced working software, and the software development lifecycle worked satisfactorily. Module testing at package level lead to easy integration, with good support from the symbolic debugger. There is a clear conclusion that it pays to do module testing, and that the resulting integration effort with Ada is relatively low in that case. A "module" in Ada is defined as package, for which each visible part (data, procedure, functions) is tested.

OOD was found to provide a natural method of producing a clear picture of the design, which leads easily into Ada definition, implementation and integration.

A summary of the statistics of the project is shown below :

Item	Fortran	Ada
Simulator lines	4800	4174
P_AOCE lines	-	2738 = 6912
Lines of test code	-	886 = 7798
Comment lines	1600	3677 = 11475
Compile time	5 min	113 min
Execution time	80 sec.	350 sec.

Phases	Man days	
Requirements	40	
Architectural design	77	
Detailed design	101	
Code, test & integration	152	
TOTAL	370	= 31 lines/day

REFERENCES

[ESA 84] ESA Software Engineering Standards BSSC (84)1

[LABREUILLE 84] B. LABREUILLE, M. HEITZ : "The Introduction of Ada in French Aerospace Industry", ADA-EUROPE, Adatec 1984, Brussels Conference.

[PAPAIX 86] M. PAPAIX, M. HEITZ, B. LABREUILLE : "Two Years of Ada Experiments : Lessons and Results", ADA-EUROPE, 1986, Edimburgh Conference.

[ROBINSON 86] P. ROBINSON : "Ada Evaluation and Transitions Studies", ADA-EUROPE 1986, Edimburgh Conference.

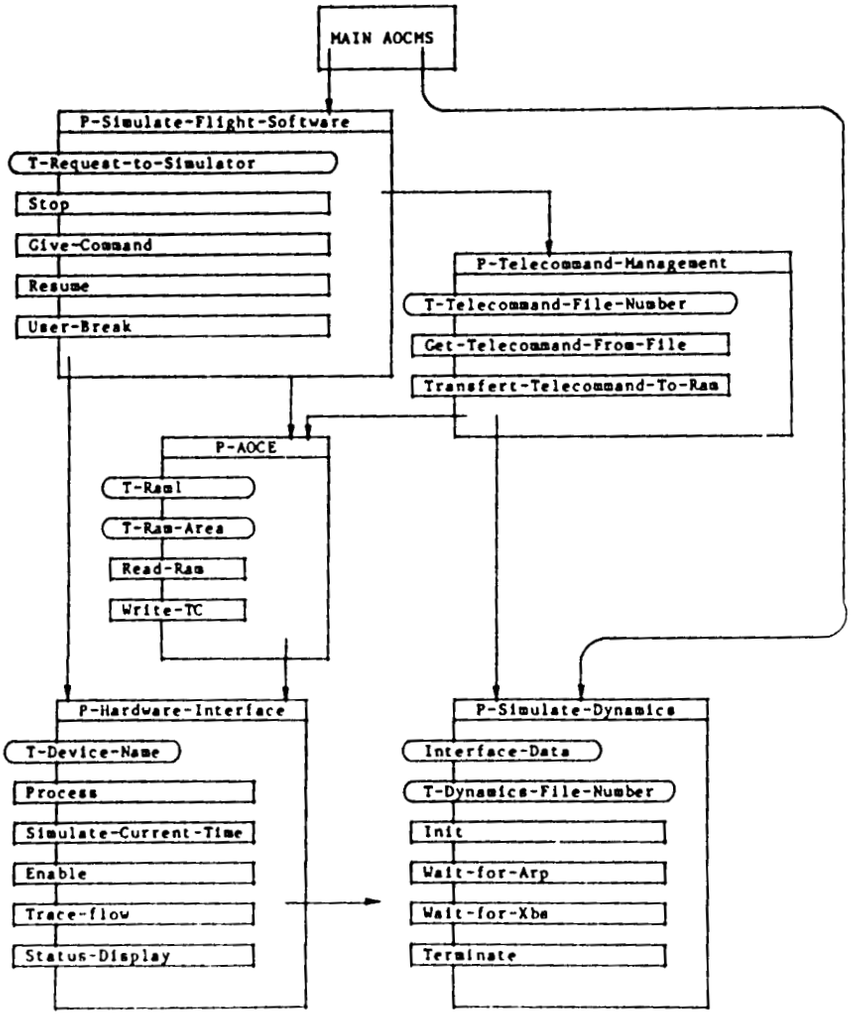


Fig. II : AOCMS Architectural Design

Structuring the Formal Definition of Ada®

Kurt W. Hansen
Dansk Datamatik Center
Lundtoftevej 1C
DK-2800 Lyngby (Copenhagen)
Denmark

Abstract:

The structures of the formal definition of Ada are described in view of the work done so far in the project. At present, a 'difficult' subset of Ada has been defined and the experience gained so far by this work is reported on here.

Currently, the work continues towards the formal definition of the full Ada language.

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

This work has been partly supported by the CEC MAP project on 'The Draft Formal Definition of Ada'. Dansk Datamatik Center - Prime contractor, CRAI - contractor, CNR/IEI - subcontractor, consultants: University of Genoa (Dept. of Mathematics), Tech. University of Denmark (Dept of Comp. Science), and University of Pisa (Dept. of Informatics).

Introduction.

Since the final requirements of Ada (the STEELMAN document) and up to the present Reference Manual for the Ada Programming Language - ANSI/MIL-STD 1815A (RM) the language has been subject to a great deal of discussion, comments, suggestions, and shear critisism.

All of this evaluation has been done on the basis of natural language descriptions, since they are the only ones available. Natural language descriptions of a certain size have a tedency to be ambiguous and contradictory and the RM is no exception to that rule. This has caused some trouble to users, mainly compiler writers.

It is our belief, that having had a formal (mathematical) definition of the language developed together with the natural language description would to a large extent have had avoided these errors in the language design. Not only would it have helped in analysing the complexities of the language which may have altered the design, but it would also have provided an unambiguous definition.

As this was not done, the second best thing is to give a formal definition of the language as it now stands. The number of projects which have attempted this so far [ref INRIA 1982, Bjørner and Oest 1982] strengthen the belief that this work is important, and the fact that none has succeeded in formally defining full Ada also indicates that it is a very difficult task.

In order to gain confidence, and actually prove, that the project is able to formally define the full language Ada, the project has selected two sets of difficult aspects of Ada, in order to show that the expirience and the new methods used are adequate for the task. The reason for having two sets of aspects is, that Ada aspects which are statically difficult are not necessarily dynamically difficult, and vice versa so both modelling static and dynamic semantics were tried out.

At the present stage the project has succesfully finished the trial definition of the Ada subsets, and is now proceeding to formally define full Ada.

This presents the work done, and experience gained in the trial definition of the difficult Ada subsets.

The Overall Structure of the Formal Definition of Ada.

The draft formal definition of Ada has adopted the scheme for defining programming languages as found in VDM [ref Bjørner and Jones 1982]. This means dividing the semantics of the language into two parts: static semantics and dynamic semantics. This gives a good overview of the language features and in this case at the same time complies with the semantics of Ada. As described in the RM two types of rules are identified: rules which describe compile time checks to be performed, and rules describing the dynamic (run time) behaviour of an Ada program. Hence, the static semantics may be seen as the precondition for the dynamic semantics of Ada.

Both static and dynamic semantic definitions are written using the syntax directed approach in a compositional style. Compositional means, that the semantics of a construct is given as a function of the semantics of its subcomponents. Here semantics is understood as a homomorphism (function) from the algebra of syntax into some semantic algebra.

Not only does the compositional style make the writing of the formulae of the semantics of Ada easier as the semantics of each construct is defined in terms of the semantics of its subconstructs, but it also enhances readability as you do not have to remember the semantics of all preceeding constructs in order to understand the semantics of a given construct.

Of course for example in the static semantics you have to use the history to some extent, you have to know the names and types of defined variables in order to perform the type check, but this information is modelled in a separate abstract data type in order not to confuse the overall syntax directed approach.

One may consider the static semantics as the first part of the formal semantics of Ada. Static semantics takes as its input an algebra of syntax which is as ambiguous as the grammar found in the RM. Ambiguous means, that you cannot tell the meaning of a construct without taking into account the context in which it is found. An example is:

```
a := f(x);
```

This is obviously an assignment statement, but the expression f(x) may denote:

- an element of an array
- a function call with one positional parameter
- a type conversion of the expression 'x' to the type 'f'

The ambiguous grammar found in the RM, is translated directly into the algebra of syntax used in the static semantics. The idea is, that only essential information is retained. As an example, in the assignment statement the essential information is the fact that you have a left-hand side name and a right-hand side expression.

The syntactic construct of the assignment statement is therefore in our metalanguage written as:

```
Assignment_stmt :: Name × Expr
```

Static semantics now performs the compile time check on the syntactic constructs found. In the case of f(x), operations on the data type reflecting declarations are used to look up 'f' in order to disambiguate the term f(x). Next overloading is resolved, the static checks for the left-hand side and right-hand side are done, and at last the validity of the assignment statement is tested using the knowledge gained trying to statically check its components (compositionallity). The knowledge could be the fact, that for example the right-hand side is not well-formed at all, and therefore the static check of the whole construct must also fail.

In principle there is no reason why the dynamic semantic should not be able to perform its run time check of and execution on an Ada program on the same abstract syntax the one as used by the static semantics. However in practice this would impose on the dynamic semantics to do most of the work already done in the static semantics over again - like disambiguating syntactic constructs. This would complicate the dynamic semantics considerably, destroying the readability of the final formal definition of the dynamic semantics.

The approach taken in this project, is to impose a transformation on the algebra of syntax used in the static semantics (AS1). This transformation transforms AS1 into an equivalent algebra of syntax (AS2), where the static problems to a large extent have been resolved, and some statically available information is distributed more conveniently (e.g. an aggregate is always given a type).

Resolving the static problems of the syntax means, resolving of syntactic ambiguities, giving unique names to identifiers (apply visibility rules and resolve overloading), adding derived information (attach a type to an aggregate), and removing information not necessary for the dynamic semantics (e.g. the order in which compilation units appear).

The AS2 is then the starting point of the dynamic semantics. In order to improve readability, the AS2 is kept as close to the original Ada program as possible; a user should be able to recognize his program. Furthermore, if a user wants to know some facts about the run time behaviour of his program, he should be able to see the AS2 program

without having to first write an Ada program and then impose the AS1 to AS2 transformation. This of course implies, that the program given to the dynamic semantics must be statically correct, since the successfull application of the static semantics is a prerequisite for the dynamic semantics.

Human Aspects of Structuring.

The writing of formal definitions is still an exercise mostly done in the academic environment since the writing of formal definitions has not yet matured into an engineering practice.

As a reflection of this, most papers found on structuring of formal definitions are aimed at getting the right mathematical structuring, making sure that the whole formula system is correct and consistent. The issue of readability has not been addressed to any large extent. This is one of the facets of structuring that has been studied in this project.

It is our belief, that formally defining Ada is only a worthwhile task to perform, if a large group of people is able to use the definition.

Our good luck has been, that through the last years many more people have become familiar with the notion and uses of formal definitions. Some of the driving force has been the complex problems found in the development of large sofware systems and the users' needs for proven programs, as software move into more and more vital positions of our society. Formal methods provide a tool for analyzing and building such complex systems and some industrial expirience has already been reported on.

Therefore some of the studies laid down in the task of structuring the formal definition of Ada have been in the area of finding out how humans read the formal definition, and what may be done in order to make sure that the reader gets the easiest access to the definition.

In this work, many parameters have been looked into. Some of the parameters have been: what about the size of the reports? model oriented vs. axiomatic descriptions, direct semantics style vs. continuations.

The answer has not always been straightforward, but we believe that we have made the tradeoffs in such a way, that most people with a programming background and a little formal training added, should be able to read and understand the formal definition of Ada.

In the structuring of documents used in this project, each formula has been put into a fixed framework giving the auxiliary information needed in order to read that particular formula. This information includes:

- Identification which directly relates the formula to the RM thereby helping people to understand the formal definition in Ada terms.

- Short description of the objective of the formula.

- The formula itself given either axiomatically or model oriented. As model oriented is believed to be the most readable for computer programmers (it resembles a program) most of the definition is described in a functional style. If a number of concepts can be separated out into a selfcontained abstract data type, it has been done and in many cases the operations performed are described using axioms.

- Natural language explanations of how the formula is supposed to perform its task, and correlation of the formula to the concepts of the RM that the formula describes.

- An extensive cross referencing.

Examples of the above may be found in [ref DDC and CRAI 1986].

Structure of the Static Semantics of Ada.

The subset static semantics of Ada is a homomorphism from the algebra of syntax into the algebra of booleans since separate compilation and hence libraries are not part of the subset. This homomorphism makes heavy use of operations from abstract data types being able to extract information from the program text taken into account until the current point of interrest.

As a mean of breaking the static semantics into useable pieces, the foundation is a hierarchy of abstract data types each aimed at describing an essential Ada concept.

Splitting a definition into data types describing concepts which are carefully highlighted in the RM seems to give the definition two properties: one is that the definition gets broken into manageable size definitions which may be combined, and the other is that breaking the definition into data types which define Ada concepts will give the user who knows about programming languages (maybe even about Ada) a conceptual framework within which to understand the formal definition - facilitating familiarization with and enhancing readability of the definition.

The hierarchy of data types defined, has the following properties: at the bottom of the hierarchy: very basic data types describing integers, identifiers etc. Next level describes types and the strong typing concepts of Ada. This includes operations for the handling of derived types, subtypes, type matching etc. From this data type a new data type is built describing the properties of all entities in Ada which you may declare.

In the same fashion concepts like visibility, overloading, and generics are described in abstract data types in further levels of the hierarchy. The topmost data type is called SUR abbreviated from surroundings. This data type describes the 'static history' of the compilation unit so far, by combining all information from lower level data types. This is done, in order to assemble all static semantics information in one place.

The data types are used in the formation of the homomorphism from the algebras of syntax. This homomorphism is named the well-formed (wf) function(s).

In the subset the 'root construct' is the subprogram body. The type of the function is_wf_Subprogram_body is:

$$\text{Subprogram\_body} \rightarrow \text{SUR} \rightarrow \text{BOOL}$$

but often the check, that a given construct is well formed cannot be performed if the only fact known about the subconstructs is whether they are wellformed or not. Further retrieving of information about the subconstructs is necessary. As an example take the assignment statement: the left-hand side has to be well formed, the right-hand side has to be well formed, but on top of that, the types of the two sides have to be the same. As an is-wf function only returns BOOL, data type operations and auxiliary functions have to be used in order to retrieve the type information from both sides.

Structure of the Dynamic Semantics of Ada.

The dynamic semantics of Ada is modelled using the SMoLCS (Structured Monitored Linear Concurrent Systems) method as described in [Astesiano et al 1985].

Using the SMoLCS method already imposes some structuring on the formal definition of the dynamic semantics. SMoLCS is a layered approach to the description of concurrency. It consists of four layers. At the bottom describing the basic states possible in the system we find a labelled transition system similar to the ones found in for example CCS.

In order to describe the behaviour of the concurrent system, some constraints are applied to the transition system. These constraints fall into three types. First all actions which may result as synchronized operations of processes are identified, next all synchronized actions which may occur in parallel are identified, and the last step defines which actions are possible in the system as a whole.

The above levels constitute what we call step 2. Step 1 of the dynamic semantics, which is using a denotational style is the homomorphism from the algebra of syntax into the semantic algebra defined by step 2. As the metalanguage makes it possible to axiomatically define operations which closely match Ada concepts, the issue is what to define denotationally.

The problem has been solved by structuring the definition of dynamic semantics in such a way, that all concepts described in the RM are defined in denotational clauses, so that no concept of Ada is hidden in an abstract data type.

An argument for moving the concepts from the denotational part could be, that a definition may be written more abstractly by moving some Ada concept modelling out of the denotational part, but for the reason of understanding by the user, it seems more appropriate to split as described above.

A further advantage of the SMoLCS method is the high degree of parameterization. This is used to describe some of the features that previously have been very difficult to describe. These sorts of concepts include implementation dependent features. They may now be modelled by including the appropriate parameters in the definition. A further concept is context clauses. Also here the parameterization scheme helps [ref DDC and CRAI 1986].

Conclusion and Further Work.

The formal definition of the subsets mentioned has assured us, that the task of formally defining the language Ada as described in the RM is feasible and can be done.

During the work with the trial definition we have seen, that in the static semantics the abstract data types had a tendency to become rather large. The problem is overcome by splitting some of them into smaller data types. This is almost also a prerequisite for the second change: the axiomatic modelling of the data types. Currently they are defined by giving a specific model, but breaking the data types into smaller definitions makes an axiomatic definition feasible.

In the dynamic semantics the distinction between operations defined axiomatically and denotational formulae will be studied further. It seems as if the optimal solution (whatever this may be) has not been found yet.

Finally, for both sorts of semantics, some ways of modularizing formulae is needed in order to enhance the readability. The static semantics already to some extent is modularized, but more is needed and the dynamic semantics need more modularizing in step 1. Furthermore, the formal definition has to be updated w.r.t. the commentaries from the Language Maintenance Committee, a task which is timeconsuming and not always straightforward.

References.

Astesiano et al 1985
E. Astesiano, G. F. Mascari, G. Reggio, M. Wirsing
On the Parameterized Algebraic Specification of Concurrent Systems.
TAPSOFT Conf., Berlin
Springer Verlag
Lecture Notes in Computer Science, vol 185, 1985

Bjørner and Oest 1980
D. Bjørner, Ole N. Oest
Towards a Formal Description of Ada
Springer Verlag
Lecture Notes in Computer Science, vol. 98, 1980

Bjørner and Jones 1982
Dines Bjørner and Cliff B. Jones
Formal Specification and Software Development
Series in Computer Science, Prentice Hall 1982

DDC and CRAI 1986
E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, K. W. Hansen, P. Inverardi, E. W. Karlsen, F. Mazzanti, G. Reggio, J. Storbank Pedersen, E. Zucca
Static Semantics of a 'Difficult' Example Ada Subset, and
Dynamic Semantics of a 'Difficult' Example Ada Subset
1986

INRIA 1982
Honeywell inc., Cii Honeywell Bull, and INRIA
Formal Definition of the Ada Programming Language
1982

Recent Trends Related to the Use of Formal Methods in Software Engineering

Søren Prehn
Dansk Datamatik Center
Lundtoftevej 1C
DK-2800 Lyngby (Copenhagen)
Denmark

Abstract:

An account is given of some recent developments and trends related to the development and use of formal methods in software engineering. The paper focuses on ongoing activities in Europe, since there seems to be a notable difference in attitude towards industrial usage of formal methods in Europe and in the U.S.

A more detailed account is given of the currently most widespread formal method in Europe: the *Vienna Development Method*. A currently ongoing project, *RAISE*, aiming at developing a second generation formal method and related tools is described.

Finally, the use of Ada® is discussed in relation to the application of formal methods, and the potential for constructing Ada-specific tools based on such methods is considered.

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

1. Introduction and Background

It is well-known that the increasing use of software systems of an increasingly complex nature imposes greater requirements to the quality of software, its documentation and maintainability. It is also well-known that since the term "software crisis" emerged, little progress has actually been made in industrial software development environments towards meeting these requirements.

In this paper, we advocate the viewpoint that industrial software engineering today really is not *engineering*, and that real progress is to be sought in the maturation of present software production technology into a true engineering discipline.

It is believed that the characteristics of a true engineering discipline are twofold:

- the discipline must have a mathematical foundation

- the day-to-day practises of the discipline are not necessarily truly formal

This is to be understood in the following way. The requirement for a mathematical foundation is triggered by the desire to be able to *reason* about the objects created during software development (such as specifications, programs, and design decisions) in a way that allows one to determine whether any such reasoning is valid or not; in particular one would like to be able to reason about the *functional correctness* of a program with respect to a specification. On the other hand we believe, in particular when one considers industrial software development, that such formal reasoning will mainly take place in order to establish ("once and for all") general rules and techniques whose correctness and soundness are verifiable. On a day-to-day basis there is presently no hope that development of any but trivial (small) programs can be thoroughly reasoned about in a formal way: the combinatorial complexity is simply too high. Thus we advocate the daily use of rules and techniques whose formal correctness and soundness have previously been established.

This is well in accordance with the way established engineering disciplines work. For example, electronics engineering has a rather firm basis in mathematics (e.g.: the use of complex calculus to describe quasi-stationary circuitry) and makes heavy use of various formal notations (such as diagrams, being a language with a precise, mathematical meaning (and a graphical syntax)). In daily life, the electronics engineer goes about his job mainly on the basis of previously established design principles, without considering the formal proofs of their soundness. However, from time to time, it is necessary to bring in formality, to make mathematical analysis and conduct proofs. This typically happens when a completely new sort of circuitry is being considered, or when requirements to circuitry functionality and reliability are particularly strict.

Here it is worth noting that only the fact that electronics engineering has a mathematical basis makes this possible; it would not have worked to base daily practises on informal notions, and then bring in formality from time to time.

The analogy offers another interesting observation: there seems to be two different styles of work involved: one style is based on using sound development rules, another on formally analysing (e.g.: proving the correctness of) an otherwise constructed object (such as the design of an electronic circuitry). We shall return to this dichotomy.

It is not surprising that software development has not yet evolved into a true engineering discipline. The trade is relatively young, and the requirements to the (complexity of the) software systems to be produced are ever increasing. Mathematics and formality has, though, been successfully applied to various aspects of software development. The availability of BNF grammars and parser generators is the classical, convincing example.

The scene is, however, beginning to change. In Europe, information technology industry in general demonstrates a growing interest for formal specification and design languages, for formal development rules, and for formal verification techniques. This, we believe, is in contrast to the trends in U.S. information technology industries, where the emphasis appears to be on tools, workstations, and environments, rather than on the methods they should support.

The purpose of this paper is to outline current trends in Europe. Given the space available, it is impossible to give a complete and covering picture, let alone to go into much technical detail. It is hoped, however, that the material presented will stimulate discussions on introducing formal methods into industrial software engineering environments.

In section 2, an overall scenario is presented, and a number of relevant research and development projects are mentioned. In section 3, an account is given of the so-called Vienna Development Method (VDM), which was the first purportedly formal method to reach any industrial significance, despite its shortcomings. In section 4, an account is given of the RAISE project, whose explicit objective is to provide formal languages and techniques for software engineering (in the above sense) as well as support tools. Finally, in section 5, perspectives specifically concerned with Ada are discussed.

2. The European Scene

Although there has been some industrial interest in formal software development methods in the European information technology industry over the past decade, and even a few successful attempts to seriously apply such methods on "real" projects, formal software development methods have had no pervasive impact. There has been a distinct, and partially well-founded, belief that formal methods were not sufficiently industrialized. Also there has been an assumption that formal methods probably were not worthwhile to apply or even harmful.

However, formal methods are now beginning to come about in industrialized form, and it is becoming increasingly clear to industry that software development practises must be seriously improved if the potential and challenges offered by the continuous hardware technology evolution are to be met.

Also, European academe has a strong tradition for research in the formal methods area, and there is today a strong desire to transfer the acquired knowledge and expertise to industry.

Probably, the most visible evidence of this trend is the joint industrial and academe support of and participation in projects, concerned with formal methods, sponsored by the Commission of the European Communities (CEC). It is interesting to note that these projects typically involve cooperation between some four to six partners, industries as well as universities.

In order to give an idea of the range of activities and institutions involved we list a number of projects, totalling several hundred person years of effort, sponsored under the ESPRIT program [ESPRIT 86] (European Strategic Programme for Research and development in Information Technology). For each project, name, title, and participants are indicated:

FORMAST
  Formal Methods for Asynchronous Systems Technology
    Advanced System Architectures (United Kingdom)
    Erno (West Germany)
    Imperial College (United Kingdom)
    University of Kaiserlautern (West Germany)

GRASPIN
  Personal Workstation for Incremental Graphical Specification
  and Formal Implementation of Non-Sequential Systems
    GMD (West Germany)
    Olivetti (Italy)
    Siemens (West Germany)

PROSPECTRA
  Program Development by Specification and Transformation
    University of Bremen (West Germany)
    University of Saarland (West Germany)

Systeam KG (West Germany)
University of Dortmund (West Germany)
Syseca Logiciel (France)
University of Passau (West Germany)
University of Stratchclyde (United Kingdom)

RAISE
Rigorous Approach to Industrial Software Engineering
Dansk Datamatik Center (Denmark)
Standard Telephone and Cables (United Kingdom)
Nordic Brown Boveri (Denmark)
International Computers Limited (United Kingdom)

METEOR
An Integrated Formal Approach to Industrial Software Development
Philips (Netherlands)
CGE (France)
AT-T & Philips (Belgium)
Stichting Matematish Centrum (Netherlands)
COPS Europe (Ireland)
Tech. Software Telematica (Italy)
University of Passau (West Germany)

GENESIS
A General Environment for Formal Systems Development
Imperial Software Technology (United Kingdom)
Imperial College (United Kingdom)
Philips (Netherlands)

It is not within the scope of this paper to elaborate on the actual contents of the individual projects. However, section 4 describes one of the projects (RAISE) in more detail. Another major project that should be mentioned is the Munich CIP project carried out at the Technical University of Munich [Bauer 76, CIP 85].

In Europe, the interest in formal methods appears to concentrate more on formal specification and formal development than on verification. That is, there is a belief in the transformational programming paradigm: if an implementation is produced solely by applying a series of transformations, each of which are correctness-preserving, to an initial specification, the implementation will necessarily be correct with respect to the initial specification, thus eliminating the need for verification. The interest in this style of development is connected with two concerns: firstly, it tends to eliminate an early introduction of (design) errors, and secondly, recording the series of transformations applied produces invaluable documentation of the system design process.

3. The Vienna Development Method (VDM)

VDM originated in the IBM Vienna Laboratories in the early seventies and was developed in connection with a project aimed at developing a production quality PL/I compiler. The project group initially worked on giving a formal semantics for PL/I; this effort probably constitutes the first example of successfully applying formal techniques to a fairly large-scale problem in an industrial environment [Bekic 74].

During the late seventies, VDM was further developed, and an increasing number of development projects using VDM emerged. Areas in which VDM was applied comprised not only programming languages and compilers, but also databases, operating systems, hardware specification, business applications, etc.

[Bjørner 83] contains an overview of VDM basics and an extensive bibliography.

[Bjørner 82] contains numerous major examples of VDM specifications.

Today, there is a rather pervasive interest in VDM in Europe, as witnessed by the formation of "VDM Europe", an interest group sponsored by the CEC and drawing participants from a fairly substantial number of European industries and universities, and by the formation of an industrial panel in the United Kingdom working towards making the VDM specification language into a British Standard.

Technically, VDM is based on the techniques developed for giving denotational semantics of programming languages. A denotational semantics is given as a homomorphism from an algebra of syntactic objects to an algebra of semantic objects, or, somewhat simplified, maps pieces of syntax onto semantic objects such as state transformations (functions from states to states). The principle readily adapts to numerous applications: many systems may conveniently be characterised by a state, which is manipulated by operations. Names of operations and their arguments are then considered to be syntactic objects.

VDM is *model-oriented*. By this is meant that the objects (syntactic and semantic) are explicitly constructed in terms of given constructors such as sets, lists, maps, and functions. This is in contrast to *property-oriented* specification approaches, such as algrebraic specification approaches, where objects are defined implicitly by the equational rules for the operations that manipulates them.

It is strongly believed that this aspect of VDM has been crucial for larger applications, and for the acceptability of VDM in industrial environments: model-oriented specifications tend to appeal much more to software engineering intuition than does property-oriented specifications. On the other hand it also clear that a model-oriented specification methodology may easily be abused to produce very operational "specifications" and presents a prevalent danger of over-specification.

4. The RAISE Project

The RAISE project (Rigorous Approach to Industrial Software Engineering) is a 115 person-year effort undertaken by a consortium consisting of Dansk Datamatik Center and Nordic Brown Boveri (Denmark), and Standard Telephone and Cables p.l.c. and International Computers Limited (United Kingdom). The project is partially funded by the Commission of the European Communities under the ESPRIT programme, and is carried out in the period 1985 to 1989. An overview of the RAISE project is given in [Meiling 85].

The RAISE project will provide an environment consisting of

- a *wide spectrum language* in which one can express abstract, formal specifications, designs, and algorithms
- means for expressing and affecting *transformations* of such entities
- proof systems and techniques serving to verify the correctness of such transformations
- a comprehensive tool set

Also, the project has been designed to include production of educational, training and technology transfer material alongside with the development of the above.

In RAISE, *Rigorous* hints at the underlying dogma that, although the RAISE language is formally defined and in principle enables the user to proceed strictly formally in developing a software system, practical conditions and requirements force one to choose, pragmatically, to carry out various parts of a development with varying degrees of formality. The philosophy behind the design of the RAISE tool set is to facilitate such a working style rather than to force a user into unmanageable formality.

RAISE encourages development by application of correctness preserving transformations, and allows for the development and verification of such transformations. The choice of using a specifically designed wide spectrum language implies that most of a development can be carried out independently of a perspective implementation language: only a final step in a development will carry a detailed, operational design into code. Typically, the code of a software system will therefore not exploit all the bells and whistles of the implementation language; indeed, it is hoped that only rather well-behaved systems will then result.

In RAISE, *Industrial* hints not only at the above-mentioned pragmatic choices that should be catered for, but also at truly quality tools and man-machine interfaces, usability of methodologies for "real" software systems, including the ability to obtain efficient end-products. In order to ensure conformance with these requirements, the project has been designed to include a number of *industrial trials*, i.e. applications of (intermediate versions of) languages, methods and tools during the course of the project; such industrial trials are to take place in actual industrial projects not otherwise connected with RAISE.

5. Some Future Perspectives

At present, it is fair to say that the industrial use of formal methods in Europe is beginning to happen. There is, though, still a long way to go. The major obstacles we are facing are:

- insufficient matureness of formal methods
- lack of management awareness
- lack of educational material and capacity
- lack of tools

A number of projects have been mentioned which attempt to seriously work towards more mature formal methods, keeping the more pragmatic requirements to the potential for industrial usage in focus. These projects were designed to bring out the best of earlier formal methods, combined with the most recent advances in research. It is believed that the next 2 to 5 years will bring about radical progress.

By the term "management awareness" we primarily think about first and second level managers' willingness to allow or force formal methods to be introduced into projects and divisions. The present, rather widespread conservatism is well understandable: although a number of successful projects having employed formal methods can be identified, it is, in all fairness, characteristic for such projects that they have been carried out in particularly friendly environments. Will formal methods actually port to "real" industrial environments? The most important part of the answer, we believe, is reflected in our next concern.

Availability of educational material and sufficient well-qualified personnel to aid in the introduction of new technology are invariably a major concern in any situation of evolution, and indeed also for the introduction of formal methods. However, we beleive that availability of text books, workshops, and courses is not sufficient. It appears to be a general experience that the introduction of formal methods should happen (1) in connection with a real project, (2) be preceeded by intense education (not just training), and (3) -- crucially -- be supported by on-project consultancy provided by experienced practitioners.

For the moment, few tools supporting formal methods are available. So, basically experiences today have been painstakingly acquired using paper and pencil. And scepticists may reasonably ask whether one can have more confidence in formal specifications and designs not checked by tools than in programs not checked by a compiler. Nevertheless, projects based on 3 levels of paper-and-pencil description (specification, high-level and low-level designs) preceeding the implementation have proved to come up with rather startling net productivity figures and low error rates. With really good tools, we should be able to do even better. It is important to us, however, that method design, understanding and experience preceed the construction of tools.

The Perspective for Ada and Formal Methods

Ada is probably one of the most complicated programming languages ever designed. The complexity is clearly witnessed by the immense amount of resources that has been required to bring about a reasonably debugged reference manual, compilers, and so on.

The complexity mainly stems from the rather large number of language concepts and features and, in particular, their general interaction. An often-noted problem is, as an example, that concurrency (tasking) interfere with the semantics of otherwise well-understood constructs such as function calls in a rather non-transparent way: the effect of tasking is not clearly bound to the syntax of Ada. It is to be feared that the complexity of Ada may impart a serious threat on the ability to construct and maintain correct and reliable software systems. With the widespread acceptance of Ada as the preferred programming language for military and space applications it is more urgent than ever to be serious about true engineering techniques and tools that will enable industrial construction of correct and reliable software.

We believe that there are two (complementary) lines of development to be pursued: adoption of the transformational programming paradigm, and providing usable techniques and tools for analysis (including verification) of programs. These two lines will probably be effective at different points in time: although powerful transformational programming systems are currently being developed, it will invariably take some time before such systems come into widespread use -- hence there is an extremely urgent need for providing tools that can assist in analysing Ada programs having been produced by more traditional techniques.

If such tools are to be of an interesting quality they must be based on a formal understanding of Ada. It is hoped that the completion of the Draft Formal Definition of Ada [Hansen 86] will provide the necessary foundation.

6. References

[Bauer 76] F.L. Bauer: "*Programming as an Evolutionary Process*"; in: Lecture Notes in Computer Science, Vol. 46, Springer Verlag , 1976

[Bekic 74] H. Bekic et.al.: "*A Formal Definition of a PL/I Subset*"; IBM Vienna Laboratories TR25.139, December 1974

[Bjørner 82] D. Bjørner & C.B. Jones: "*Formal Specification and Software Development*"; Prentice-Hall International Series in Computer Science, 1982

[Bjørner 83] D. Bjørner & S. Prehn: "*Software Engineering Aspects of VDM*"; in: D. Ferrari et.al. (eds.): "Theory and Practice of Software Technology", North-Holland Publishing Company 1983

[CIP 85] F.L. Bauer et.al.: "*The Munich Project CIP - Volume I: The Wide Spectrum Language CIP-L*"; Lecture Notes in Computer Science, Vol. 183, Springer Verlag ,1985

[ESPRIT 86] "*ESPRIT Project Synopses, Software Technology*"; Commission of the European Communities, January 1986

[Hansen 86] K.W. Hansen: "*Structuring the Formal Definition of Ada*"; these proceedings

[Jones 80] C.B. Jones: "*Software Development - A Rigorous Approach*"; Prentice-Hall International Series in Computer Science, 1980

[Meiling 85] E. Meiling et.al.: "*RAISE Project: Fundamental Issues and Requirements*"; RAISE/DDC/EM1/v6, 1985-12-10; Dansk Datamatik Center, 1985

MANAGING ADA DEVELOPMENT

by

James R. Green
Manager, Standard Products

Systems and Software Engineering Operations
Dalmo Victor Incorporated
The Singer Company
6365 East Tanque Verde Road
Tucson, Arizona 85715
Phone: (602)721-0500

C-1

Managing Ada Development — James R. Green

Introduction

The Ada programming language was developed under the sponsorship of the Department of Defense to address the soaring costs associated with software development and maintenance. Ada is powerful, and yet to take full advantage of its power, it is sufficiently complex and different from current programming approaches that there is considerable risk associated with committing a program to be done in Ada. There are also few programs of any substantial size that have been implemented using Ada that may be studied to determine those management methods that resulted in a successful Ada project.

As the Manager of Standard Products, I have the responsibility for developing software products that will be offered for sale on the open market. One of the products which has been developed is implemented entirely in Ada and its success demonstrates that a project can be successfully done using Ada. The program itself comprises over 130,000 source lines of Ada code. This project, although not large by today's standards of software development, did cause me to face the frustrations and the difficult management tasks associated with implementing an entire program in Ada at a time when the Ada development environment was less than desirable. The items presented in this paper are my opinions which have been formed as a result of going through this experience. The difficulties faced, risks assumed, management methods applied, and lessons learned, and most importantly, the techniques that were successful are all valuable sources of management information for those managers ready to assume major Ada developments projects.

People - A Key Ingredient

Projects are implemented by people. The right people are definitely a key ingredient to the success of any project. Ada is no different. Management must realize this at the beginning of a project and ensure that "the right people" are selected. Ada has new concepts which are different from other languages. Concepts such as packages, specifications, body, information hiding, generics, instantiation, and multi-tasking are all examples of concepts and features of the Ada language. Since many of these concepts do not exist in other languages, management must be prudent in selecting personnel for assignment to the Ada project itself.

The real power of the Ada language lies in the concepts not necessarily available in other languages. The people in key positions of the project must relate to these concepts and management must ensure that the people that are initially selected do relate to these concepts. It is possible to write code in Ada that utilizes only the most elementary concepts. An analogy to this would be writing Ada code using only the constructs allowed by FORTRAN. Clearly, the code may work, but you will not realize the benefits of the Ada language.

Those people selected to work on an Ada project most probably will have prior working experiences in other languages. Their effectiveness on the Ada project will be related to how easily they accept the new language features and strive to use them effectively. As a manager, you do not want the powers afforded by Ada, to be eclipsed by an engineering staff of parochial vision.

I have found that people who have recent degrees in computer science relate well to these concepts. In addition, personnel who are well versed in Pascal programming, seem to transition quite easily into the Ada world. In my experience, I was fortunate to find talent that related these concepts. At the beginning of the project, no one on my team had any Ada experience, and further, few of them had any knowledge of what Ada was

all about. The success of this project is a tribute to their talents.

Management must seriously scrutinize the qualifications of those people they select to implement Ada projects. Selecting the right people will definitely increase your probability of implementing a successful Ada project.

Training Programs - A Key Ingredient

Ideally, you would want to hire people who have performed successfully on other Ada projects. However, there is a limited number of people who are proficient in Ada and I highlight the word *proficient.* Proficient means that the people understand the complex concepts of this new language and understand how to apply them. This is different than just knowing the syntax and semantics of the new language. There is a large body of software people that are well versed in FORTRAN, JOVIAL, COBOL, and other well established, high-level software languages. Many of these people will be transitioning to work in the Ada environment. Management must provide a means for these people to transition successfully into the Ada world. This leads me to the second key ingredient to success--training programs.

The program which I managed began with people that were unfamiliar with Ada. There was a wide variety of background experience among the people selected for the project. It was clear at the outset that a key ingredient to the success of the program would be the implementation of an effective training program. The training program would provide two benefits. First, it would establish a common baseline of knowledge for all people on the project at that time. The varied experiences of the people, and their knowledge of software engineering was an unknown. By covering these topics in a training course, I could be sure that every one on the project was in synch with respect to vocabulary, concepts, approaches, methodologies, and techniques. The Ada programming language and the concepts and methodologies to be used when designing Ada programs could be covered in detail. In addition, there would be a benefit of discussing

application techniques--that is how to implement certain features using the Ada language in practical applications. The training program that I implemented actually consisted of five courses and comprised 132 classroom hours of instruction.

The courses developed and given were:

Introduction to Software Engineering.....32 hours
Software Design Methodology.....40 hours
Coding Methodology.....16 hours
Ada Programming Support Environment..... 4 hours
Ada Programming.....40 hours

The Software Engineering, Design Methodology and Coding Methodology courses were developed in-house. These courses are specifically designated to provide a sound understanding of the software development process, software life cycle and design methodologies. The Ada courses used books and lecture material that was available at the time. It dwelled primarily on the syntax and semantics of the Ada language.

The length of time that was taken for training may seem excessive, and indeed, at the time I thought it was excessive. However, during later stages of the project, it was clear that the time spent at the front-end of the project for training, was time well spent.

I cannot stress strongly enough the need for the development team to understand good software engineering principles and design methodologies. To fully realize the power of Ada in your program, these principles must be understood and used. Although, in my case, all individuals went through the same level of training, I would recommend different levels of training for different project people. I would recommend two to four weeks of intense training for key technical people on the project, and possibly one to two weeks for junior people. The Ada training for the junior people will be augmented through on-the-job-training and the assignment of tasks under the guidance of the more senior and more experienced project

personnel.

The training issued, and the time and money which should be allocated during the project for training, is quite controversial. There are a number of training programs currently available. However, many training courses are short and cover the syntax and semantics of the language primarily. For the training program to be truly successful, it must include the software engineering and design principles needed by those people designing the Ada program. These people must understand these principles, and they must understand how they relate to Ada and this means that significant time must be spent on the software engineering aspects of software design.

I recognize that the effectiveness of a training program is largely realizable only after you are well into your project.

The 132 classroon hours that I allocated for training at the front-end of my project, was excruciatingly difficult to justify at the time. It appeared for several weeks that the project was making no progress in accomplishing its real objective of designing and implementing a software program. However, I now firmly believe that the time spent on the basic fundamentals of software design reaped enormous benefits later in the program. The issue of training must be taken seriously by management as well as the training programs themselves. How well the skills and methodologies are learned by your personnel will greatly affect the success of the project. The training must be effective and the personnel assigned to the Ada project must be aware that management considers the training crucial to success and that they must take it seriously. I believe that training related exercises may indeed be integrated with initial project tasks in a sort of a real laboratory exercise.

At this point, you as a manager would theoretically have qualified people who are trained and capable of implementing an Ada project. The next issue you may worry about are the schedule issues. How can you best be assured that the project is progressing on schedule and whether the

schedule is realistic. There are numerous models and rules of thumb which apply for FORTRAN and COBOL and other languages as to the relative amount of time that is spent during the requirements and design phase of the project, how much time is spent during the actual coding, and how much time is spent during the test and integration portion of the project. I have traditionally used the 40-20-40 rule-of-thumb, where 40 percent of the time is allocated to requirements and design, 20 percent to code, and 40 percent to test and integration. It is my experience, however, that when implementing a program in Ada, significantly more time and effort should be expended during the requirements and design phase. Possibly as much as 55 percent to 60 percent of the time should be allocated and expected to be spent during the requirements and design phase. Only 15 percent of the time need be allocated to code, and 25 percent to 30 percent of the time should be spent in testing and integration.

It has been the experience of people on my project, that if the Ada code compiles, chances are good that it will run. I had teams of workers implementing different elements of the program. All of the parts of the program had to work successfully together in order for the entire project to work. We have found that the test and integration phase is extremely shortened using Ada. If a module compiles, chances are very high that it will run except if there are design errors which go back to the extra time spent for requirements and design. You must ensure that your design is correct and sound.

There are many features in Ada which will result in the program being accomplished very quickly. One of these features is a concept called "generic." The concept of generics is that you design a template to do a certain function, and each time the template is invoked at various places in the program, it is instantiated or initialized to the values needed for that particular function. Generics are extremely powerful. However, in order to get the most benefit from this feature of the Ada language, a lot of effort must be put in the design of these generic packages. This is an example of how the training and the software engineering elements work together to ultimately benefit you project schedule. Approximately, 50

percent of the program I was responsible for, is implemented in generics, and the success of a particular generic was largely dependent upon the amount of time that was spent in determining the requirements of each instance that generic would be used and ensuring that the design of that generic package was sound for all instances.

Another key consideration for successfully managing an Ada project, has to do with creating an atmosphere which is conducive to accomplishment. When management is planning the schedule for an Ada program, there must be enough time at the front-end for the technical people to be accustomed to and familiar with the new language. Progress on the project may be excruciatingly slow during this time period, but as the technical people become more accustomed to the features and capabilities of the Ada language, they will be able to better apply this knowledge during the actual application required in the project. An atmosphere for accomplishment, I believe, will encourage experimentation and pushing the language to its limits. In my particular instance, the Ada compiler which I had available at the time that training was occurring, was of poor quality and several of my people found that the Ada compiler really did not operate in accordance with the Ada Language Reference Manual. They took it upon themselves, as part of their training exercises, to determine all those features of the Ada Language Reference Manual which did work. I encouraged this sort of activity as it broadened their horizons and it held their interest in the project during the period of time that training was occurring. This atmosphere of accomplishment meant that the technical people were not afraid to try things and risk new methods of implementation. They became less fearful of failure and concentrated more on success. This attitude is extremely important to maintain for a successful Ada project. The technical people will engage in frustrations and difficult things, but they must be able to experiment and they must be able to feel the freedom to try new things. You must develop a "can do" attitude in your technical people.

Risks and Cost Considerations

Managing Ada Development **James R. Green**

An area of major concern to management to committing a project in Ada is the unquantified cost associated with it. There will be cost associated with training, there will be cost associated with new compilers and software tools, computers, and there is not guarantee that the people that are hired on the project will be able to accomplish the project. Indeed, the risks to doing a project in Ada are formidable. In order to control these risks, and maintain the project on cost and schedule, management must aggressively be involved with all aspects of the project. By this, I mean you don't have to know how to program in Ada. Indeed, I do not. However, you must be able to relate and understand those concepts and those methodologies which are successful.

Management should consider that doing a project in Ada will involve many risks that are not present in projects using other traditional languages. The risks to be faced are not unmanageable, and as a result, the aggressive manager--that is one who can quickly spot trends leading to success as well as trends leading to failure and can direct actions appropriate to either trend, will be able to successfully complete an Ada project.

Since risk equates to cost, the Ada project manager will want to reduce risk as much as possible. I believe this may be done through prudent selection of personnel, good training programs and agressive involved management.

This short discussion on Managing a Program in Ada has touched only a few of the elements management must be concerned with. These, however, are keys to success.

LESSONS LEARNED: MANAGING THE DEVELOPMENT OF A CORPORATE Ada TRAINING PROJECT

Linda F. Blackmon
Coordinator, Corporate Ada Training Curriculum
General Dynamics
Fort Worth, Texas

This paper discusses the management lessons learned during the implementation of a corporate mandate to develop and deliver an effective Ada training program to all divisions. The management process involved in obtaining cooperation from all levels in the development of a corporate-wide project is described; The problems areas are identified along with some possible solutions.

Automated Fortran Conversion

Gregory Aharonian
Source Translation & Optimization
P.O. Box 404
Belmont, Ma 02178
617-489-3727

What to do with a million lines of Fortran code? Managers at every major Fortran installation are asking this question every day. Newer programming languages (C and ADA), and newer computer architectures (parallel, data flow) pose a serious dilemma. How will the algorithms and mathematical techniques in tens of thousands of Fortran programs be moved to these environments? Further, since no language will dominate the science and engineering arena, another question arises. With strained programming staffs and budgets, how will algorithms be maintained in multiple languages and architectures?

There are three solutions. The first is to hire additional staff to translate programs across languages, to coordinate and maintain large libraries of subroutines in the different languages using existing software tools. Most of the conversion will be from Fortran to C and ADA, a project with many unresolved issues (in particular array handling). The solution is unfeasible economically, when you consider the number of combinations of environments (a language out of Fortran,C,ADA,any other) with a new architecture (out of Cray, FPS, CSPI, Alliant, etc.). The staff requirements and overhead will be excessive, even if you could find enough people willing to do the very boring work of translating and maintaining software.

The second solution is to develop completely automatic language translation programs, using all of the breakthroughs in software engineering, language theory, and artificial intelligence. The problems here are many. First noone has developed an efficient automatic translation system. The few on the market either are not completely automatic, or produce very ugly and inefficient code. It is impossible for a computer (and even many humans) to translate a piece of Fortran code that operates on different dimensioned arrays passed to the same subroutine with some EQUIVALENCE and COMMON usage. Further you don't want exact translations. Fortran programs were written within the limitations of Fortran, when in the newer languages the algorithms can be expressed more clearly and efficiently.

The third, and most practical solution, which STO and a few others have adopted, uses an intermediate language that is easy to translate Fortran into, and allows for source code in others languages to be generated automatically. The intermediate language is the union of all other programming languages (and the trick is to create a useful union) with some extensions that reflect the nature of the algorithms. The benefits of this approach are many. First the original Fortran program has to be rewritten only once, and then only parts of the program; most Fortran code passes through without any change (i.e. assignment and simple IF statements). Software tools are provided to ease this initial translation. Once in the intermediate language, the algorithm can then be obtained in any other language automatically.

Some of the conversions (as options) include array indice reversal (where A(B(C,D),E(F,G)) in Fortran becomes in C A[E[G][F]][B[D][C]]), many precision support (constants appended with E0,D0 etc., subroutine and function names are suffixed, ABSR, ABSD, ABSC), and insertion of timing/frequency analysis. Manual conversion introduces errors, hindering the testing of the translated programs.

Figure 1 shows an example of a subroutine from the Eispack library in ten different languages. First, the subroutine is rewritten in STO's intermediate language, and is shorter than most of the final programs. Then, the subroutine is automatically generated in the other languages (and back into Fortran). We have successfully converted Linpack (and its test drivers), and produced tested C, Pascal, Basic, and Fortran 77 versions (and if anyone has compilers for other languages, we will provide the code for verification).

What are the disadvantages of this approach? There are two main problems, which are present even if you adopt another solution to converting Fortran programs. The first problem is that many of the newer languages are incapable of supporting numerical algorithms as easily as Fortran does. Pascal does not allow subroutines to accept arrays of different sizes, making subroutine libraries all but impossible (actually some Pascal compilers do, but there are at least two incompatible implementations). Modula-2, a (weak) attempt to fix Pascal, also doesn't allow subroutines to handle different sized multiple dimensional arrays (only 1D). Neither Pascal nor Modula-2 allow complex numbers (the suggested solution of using records and turning arithmetic expressions into series of subroutine or function calls being pathetic). These languages also provide limited multiple precision support, and not the most useful looping control structures. Modula has no GOTO, and while most GOTOs can be removed from Fortran subroutines, some very important subroutines have GOTOs that are extremely difficult to remove. At least in C and ADA you can use GOTOs for these tricky subroutines (like the *INVIT algorithms in the Eispack library). C supports Fortran programs well; its only deficiency is the lack of COMPLEX numbers used with +-*/ (hint ANSI committee!!!).

The other main problem arises with ADA. ADA has many powerful capabilities that forces you to start from scratch to fully take advantage of ADA. Generics, exceptions, and other features can only be generated if the intermediate language is as expressive as ADA, in which case just use ADA/DIANA to begin with. Unfortunately there are many installations with millions of lines of Fortran code that probably don't need all of the power of ADA, in which case automated translation becomes reasonable. Then languages like Occam (for parallel processing) require additional design considerations (in this case to efficiently use the parallel architecture).

At STO, we are undertaking a project to convert SLATEC to multiple languages via the intermediate language; when successful, packages such as Spice, Nastran, and Gaussian 84 will be converted. These projects are quite important to the design of the intermediate langauge in the translation challenges provided. It is important to realize that the recoding is a small part of the translation process. Creating software environments for multi-language software maintenance is the more critical task. To do so will require flexible software generation programs, in particular, those based on the use of an intermediate language.

The approach taken by STO and others (Boyle at Argonne, Waters at MIT,de Maine at Auburn, Diana for ADA, Lexeme) of using an intermediate language and associated software tools will allow Fortran installations to move their Fortran programs into new environments with minimal problems. While not a perfect solution, it is less costly than having larger programming staffs, and more realistic than relying on completely automatic translators.

```
TYPE ARRAY1DR IS ARRAY (INTEGER RANGE <>) OF REAL;
TYPE ARRAY2DR IS ARRAY (INTEGER RANGE <>,
                        INTEGER RANGE <>) OF REAL;

PROCEDURE ORTRNR (N: IN INTEGER; LOW: IN INTEGER;
             HIGH: IN INTEGER; A: IN ARRAY2DR;
             ORT: IN OUT ARRAY1DR; Z: IN OUT ARRAY2DR) IS
    I, J, KL, MM, MP, MP1: INTEGER ;
    G: REAL  ;
BEGIN
--
--
--       EISPACK SUBROUTINE ORTRAN IN ADA
--
--
    FOR J IN 1..N LOOP
        FOR I IN 1..N LOOP
            Z(I,J) := 0.0E+0 ;
        END LOOP ;
        Z(J,J) := 1.0E+0 ;
    END LOOP ;
    KL := HIGH - LOW - 1 ;
    FOR MM IN 1..KL LOOP
        MP := HIGH - MM ;
        IF A(MP,MP - 1) \= 0.0E+0 THEN
            MP1 := MP + 1 ;
            FOR I IN MP1..HIGH LOOP
                ORT(I) := A(I,MP - 1) ;
            END LOOP ;
            FOR J IN MP..HIGH LOOP
                G := 0.0E+0 ;
                FOR I IN MP..HIGH LOOP
                    G := G + ORT(I) * Z(I,J) ;
                END LOOP ;
                G := (G / ORT(MP)) / A(MP,MP - 1) ;
                FOR I IN MP..HIGH LOOP
                    Z(I,J) := Z(I,J) + G * ORT(I) ;
                END LOOP ;
            END LOOP ;
        END IF ;
    END LOOP ;
END ;
```

```
ORTRND (N, LOW, HIGH, A, ORT, Z)
int N, LOW, HIGH ;
double **A ;
double **Z, *ORT ;
{
    int I, J, KL, MM, MP, MP1 ;
    double G ;
/**/
/*

              EISPACK SUBROUTINE ORTRAN IN C

*/
    for ( J = 1; J <= N; J += 1 ) {
        for ( I = 1; I <= N; I += 1 ) {
            Z[I][J] = 0.0E+0 ;
        }
        Z[J][J] = 1.0E+0 ;
    }
    KL = HIGH - LOW - 1 ;
    for ( MM = 1; MM <= KL; MM += 1 ) {
        MP = HIGH - MM ;
        if (A[MP][MP - 1] != 0.0E+0) {
            MP1 = MP + 1 ;
            for ( I = MP1; I <= HIGH; I += 1 ) {
                ORT[I] = A[I][MP - 1] ;
            }
            for ( J = MP; J <= HIGH; J += 1 ) {
                G = 0.0E+0 ;
                for ( I = MP; I <= HIGH; I += 1 ){
                   G = G + ORT[I] * Z[I][J] ;
                }
                G = (G / ORT[MP]) / A[MP][MP - 1];
                for ( I = MP; I <= HIGH; I += 1 ){
                   Z[I][J] = Z[I][J] + G * ORT[I];
                }
            }
        }
    }
}
```

```
      SUBROUTINE ORTRND (N,LOW,HIGH,A,LDA,ORT,Z,LDZ)
      INTEGER LDA, LDZ
      INTEGER N, LOW, HIGH
      DOUBLE PRECISION A(LDA,1)
      DOUBLE PRECISION Z(LDZ,1), ORT(1)
      INTEGER I, J, KL, MM, MP, MP1
      DOUBLE PRECISION G
C
C
C           EISPACK SUBROUTINE ORTRAN IN FORTRAN
C
C
      DO 210 J = 1 , N
          DO 190 I = 1 , N
              Z(J,I) = 0.0D+0
190       CONTINUE
          Z(J,J) = 1.0D+0
210   CONTINUE
      KL = HIGH - LOW - 1
      IF (KL .LT. 1)  GOTO 411
      DO 410 MM = 1 , KL
          MP = HIGH - MM
          IF (A(MP - 1,MP) .EQ. 0.0D+0)  GOTO 400
              MP1 = MP + 1
              DO 290 I = MP1 , HIGH
                  ORT(I) = A(MP - 1,I)
290           CONTINUE
              DO 390 J = MP , HIGH
                  G = 0.0D+0
                  DO 340 I = MP , HIGH
                      G = G + ORT(I) * Z(J,I)
340               CONTINUE
                  G = (G / ORT(MP)) / A(MP - 1,MP)
                  DO 380 I = MP , HIGH
                      Z(J,I) = Z(J,I) + G * ORT(I)
380               CONTINUE
390           CONTINUE
400       CONTINUE
410   CONTINUE
411   CONTINUE
      RETURN
      END
```

```
PROCEDURE: ORTRNR ()
        INTEGER ARG: N
        INTEGER ARG: LOW
        INTEGER ARG: HIGH
        ANY ARG: A
        ANY ARG: ORT/VAR
        ANY ARG: Z/VAR
END PROCEDURE
PUBLIC: ORTRNR

PROCEDURE: ORTRNR
        INTEGER : I, J, KL, MM, MP, MP1
        REAL : G
260 REM
262 REM
264 REM       EISPACK SUBROUTINE ORTRAN IN BASIC
266 REM
270 REM
320     FOR J = 1 TO N
340         FOR I = 1 TO N
360             Z(I,J) = 0.0E+0
380         NEXT
400         Z(J,J) = 1.0E+0
420     NEXT
440     KL = HIGH - LOW - 1
459     IF KL < 1 THEN  GOTO 821
460     FOR MM = 1 TO KL
480         MP = HIGH - MM
500         IF A(MP,MP - 1) = 0.0E+0 THEN 800
520             MP1 = MP + 1
540             FOR I = MP1 TO HIGH
560                 ORT(I) = A(I,MP - 1)
580             NEXT
600             FOR J = MP TO HIGH
620                 G = 0.0E+0
640                 FOR I = MP TO HIGH
660                   G = G + ORT(I) * Z(I,J)
680                 NEXT
700                 G = (G/ORT(MP)) / A(MP,MP - 1)
720                 FOR I = MP TO HIGH
740                   Z(I,J) = Z(I,J) + G * ORT(I)
760                 NEXT
780             NEXT
800         REM END OF IF BLOCK
820     NEXT
821     REM END OF IF BLOCK
840     REM RETURN
END PROCEDURE
```

```
ORTRNR:
        PROC  (N, LOW, HIGH, A, ORT, Z) ;
        DCL (N, LOW, HIGH) FIXED BIN (15) ;
        DCL A(*,*) FLOAT DEC (6) ;
        DCL (Z(*,*), ORT(*)) FLOAT DEC (6);
        DCL (I, J, KL, MM, MP, MP1) FIXED BIN (15);
        DCL G FLOAT DEC (6);
/*

        EISPACK SUBROUTINE ORTRAN IN PLI

*/
        DO J = 1 TO N ;
            DO I = 1 TO N ;
                Z(I,J) = 0.0E+0 ;
            END  ;
            Z(J,J) = 1.0E+0 ;
        END  ;
        KL = HIGH - LOW - 1 ;
        IF KL >= 1 THEN  DO;
        DO MM = 1 TO KL ;
            MP = HIGH - MM ;
            IF A(MP,MP - 1) != 0.0E+0 THEN  DO;
                MP1 = MP + 1 ;
                DO I = MP1 TO HIGH ;
                    ORT(I) = A(I,MP - 1) ;
                END  ;
                DO J = MP TO HIGH ;
                    G = 0.0E+0 ;
                    DO I = MP TO HIGH ;
                        G = G + ORT(I) * Z(I,J) ;
                    END  ;
                    G = (G / ORT(MP)) / A(MP,MP - 1);
                    DO I = MP TO HIGH ;
                        Z(I,J) = Z(I,J) + G * ORT(I);
                    END  ;
                END  ;
            END ;
        END  ;
        END ;
END ORTRNR ;
```

```
PROC ORTRNR (N, LOW, HIGH, A: ORT, Z); BEGIN
    ITEM N S ;
    ITEM LOW S ;
    ITEM HIGH S ;
    TABLE A[*,*] F ;
    TABLE Z[*,*] F ;
    TABLE ORT[*] F ;
    ITEM I S ;
    ITEM J S ;
    ITEM KL S ;
    ITEM MM S ;
    ITEM MP S ;
    ITEM MP1 S ;
    ITEM G F ;
"""
"""
"          EISPACK SUBROUTINE ORTRAN IN JOVIAL"
"""
"""

    FOR J : 1 BY 1 WHILE J <= N ;BEGIN
        FOR I : 1 BY 1 WHILE I <= N ;BEGIN
            Z[I,J] = 0.0E+0;
        END;
        Z[J,J] = 1.0E+0;
    END;
    KL = HIGH - LOW - 1;
    IF KL >= 1;  BEGIN
    FOR MM : 1 BY 1 WHILE MM <= KL ;BEGIN
        MP = HIGH - MM;
        IF A[MP,MP - 1] <> 0.0E+0;  BEGIN
            MP1 = MP + 1;
            FOR I : MP1 BY 1 WHILE I <= HIGH ;BEGIN
               ORT[I] = A[I,MP - 1];
            END;
            FOR J : MP BY 1 WHILE J <= HIGH ;BEGIN
               G = 0.0E+0;
               FOR I : MP BY 1 WHILE I <= HIGH ;BEGIN
                   G = G + ORT[I] * Z[I,J];
               END;
               G = (G / ORT[MP]) / A[MP,MP - 1];
               FOR I : MP BY 1 WHILE I <= HIGH ;BEGIN
                   Z[I,J] = Z[I,J] + G * ORT[I];
               END;
            END;
        END
    END;
    RN;
    END
```

```
TYPE ARRAY1DR = SUPER ARRAY [1..*] OF REAL8;
TYPE ARRAY2DR = SUPER ARRAY [1..*,1..*] OF REAL8;

PROCEDURE ORTRNR (N:INTEGER; LOW:INTEGER;
                  HIGH:INTEGER; VAR A:ARRAY2DR;
                  VAR ORT:ARRAY1DR; VAR Z:ARRAY2DR);
VAR I, J, KL, MM, MP, MP1: INTEGER  ;
    G: REAL8  ;
BEGIN
(*

        EISPACK SUBROUTINE ORTRAN IN PASCAL

*)
    FOR J := 1 TO N DO BEGIN
        FOR I := 1 TO N DO BEGIN
            Z[I,J] := 0.0E+0 ;
        END ;
        Z[J,J] := 1.0E+0 ;
    END ;
    KL := HIGH - LOW - 1 ;
    IF (KL >= 1) THEN  BEGIN
    FOR MM := 1 TO KL DO BEGIN
        MP := HIGH - MM ;
        IF (A[MP,MP - 1] <> 0.0E+0) THEN  BEGIN
            MP1 := MP + 1 ;
            FOR I := MP1 TO HIGH DO BEGIN
                ORT[I] := A[I,MP - 1] ;
            END ;
            FOR J := MP TO HIGH DO BEGIN
                G := 0.0E+0 ;
                FOR I := MP TO HIGH DO BEGIN
                   G := G + ORT[I] * Z[I,J] ;
                END ;
                G := (G/ORT[MP]) / A[MP,MP - 1];
                FOR I := MP TO HIGH DO BEGIN
                   Z[I,J] := Z[I,J] + G * ORT[I];
                END ;
            END ;
        END ;
    END ;
    END ;
END; {ORTRNR}
```

```
CONST NEIG =
TYPE ARRAY1DR = ARRAY [1..NEIG] OF REAL;
TYPE ARRAY2DR = ARRAY [1..NEIG,1..NEIG] OF REAL;

PROCEDURE ORTRNR (N:INT ; LOW:INT ; HIGH:INT;
                  A:ARRAY2DR; VAR ORT:ARRAY1DR;
                  VAR Z:ARRAY2DR);
VAR I, J, KL, MM, MP, MP1: INT  ;
    G: REAL  ;
(*

         EISPACK SUBROUTINE ORTRAN IN MODULA-2

*)
BEGIN
    FOR J := 1 TO N DO
        FOR I := 1 TO N DO
            Z[I,J] := 0.0E+0 ;
        END ;
        Z[J,J] := 1.0E+0 ;
    END ;
    KL := HIGH - LOW - 1 ;
    IF (KL >= 1) THEN
    FOR MM := 1 TO KL DO
        MP := HIGH - MM ;
        IF (A[MP,MP - 1] <> 0.0E+0) THEN
            MP1 := MP + 1 ;
            FOR I := MP1 TO HIGH DO
                ORT[I] := A[I,MP - 1] ;
            END ;
            FOR J := MP TO HIGH DO
                G := 0.0E+0 ;
                FOR I := MP TO HIGH DO
                    G := G + ORT[I] * Z[I,J] ;
                END ;
                G := (G / ORT[MP]) / A[MP,MP - 1] ;
                FOR I := MP TO HIGH DO
                    Z[I,J] := Z[I,J] + G * ORT[I] ;
                END ;
            END ;
        END ;
    END ;
    END ;
END
```

```
TYPE ARRAY1DR : ARRAY 1..* OF REAL
TYPE ARRAY2DR : ARRAY 1..*,1..* OF REAL

[illegible] (VAR N: INT, VAR LOW, VAR IGH: INT,
                    A:ARRAY2DR, VAR ORT:ARRAY1DR,
                    VAR Z:ARRAY2DR)
VAR I, J, KL, MM, MP, MP1: INT
    G: REAL
%
%
%         EISPACK SUBROUTINE ORTRAN IN TURING
%
%
    FOR J : 1..N
        FOR I : 1..N
            Z(I,J) := 0.0e+0
        END FOR
        Z(J,J) := 1.0e+0
    END FOR
    KL := HIGH - LOW - 1
    IF KL >= 1 THEN
    FOR MM : 1..KL
        MP := HIGH - MM
        IF A(MP,MP - 1) NOT = 0.0e+0 THEN
           MP1 := MP + 1
           FOR I : MP1..HIGH
               ORT(I) := A(I,MP - 1)
           END FOR
           FOR J : MP..HIGH
               G := 0.0e+0
               FOR I : MP..HIGH
                  G := G + ORT(I) * Z(I,J)
               END FOR
               G := (G/ORT(MP)) / A(MP,MP - 1)
               FOR I : MP..HIGH
                  Z(I,J) := Z(I,J) + G * ORT(I)
               END FOR
           END FOR
        END IF
    END FOR
    END IF
END ORTRNR
```

```
PROC ORTRNR = (INT N, INT LOW, INT HIGH,
               [,]REAL A, REF []REAL ORT,
                 REF [,]REAL Z ) VOID:
BEGIN
CO

      EISPACK SUBROUTINE ORTRAN IN ALGOL-68

CO
    INT I, J, KL, MM, MP, MP1 ;
    REAL G ;
    FOR J FROM 1 TO N DO
        FOR I FROM 1 TO N DO
            Z[I,J] := 0.0e+0 ;
        OD ;
        Z[J,J] := 1.0e+0 ;
    OD ;
    KL := HIGH - LOW - 1 ;
    IF KL GE 1 THEN
    FOR MM FROM 1 TO KL DO
        MP := HIGH - MM ;
        IF A[MP,MP - 1] NE 0.0e+0 THEN
            MP1 := MP + 1 ;
            FOR I FROM MP1 TO HIGH DO
              ORT[I] := A[I,MP - 1] ;
            OD ;
            FOR J FROM MP TO HIGH DO
              G := 0.0e+0 ;
              FOR I FROM MP TO HIGH DO
                G := G + ORT[I] * Z[I,J] ;
              OD ;
              G := (G/ORT[MP]) / A[MP,MP - 1];
              FOR I FROM MP TO HIGH DO
                Z[I,J] := Z[I,J] + G * ORT[I];
              OD ;
            OD ;
        FI ;
    OD ;
    FI ;
    RETURN: ;
END
```

GSFC Ada Programming Guidelines

Daniel M. Roy, Robert W. Nelson

1 INTRODUCTION

A significant Ada effort has been under way at Goddard for the last two years. To ease the center's transition toward Ada (notably for future space station projects), a cooperative effort of half a dozen companies and NASA personnel was started in 1985 to produce programming standards and guidelines for the Ada language.

2 APPROACH

Two parallel tracks were pursued:

1. Coding style and Ada statement format.
2. Portability, efficiency and whole life cycle issues.

Two documents have been produced so far, one for each track followed. This paper more specifically deals with the second one. Both documents are similar in structure (closely modeled on the Ada LRM) and were greatly influenced by Nissen and Wallis guidelines ([NW]). Other documents also had some influence:

- The rationale for Ada [Rationale].
- The IEEE Ada PDL recommended practices document [IEEE-990].
- Intermetrics BYRON user's guide [Intermetrics].
- Ada in practice (Ausnit, Cohen, Goodenough, and Eanes) [Softech].
- Using Selected Features of Ada [NTIS].
- Intellimac's Ada style [Intellimac].
- Regulation for the management of computer resources in defense systems (MIL-STD-2167) [2167].

Both drafts are currently being merged into an Ada Style document for use by all projects at the NASA Goddard Space Flight Center.

3 STRUCTURE OF THE DOCUMENT

It was decided early on to model our guide on the Ada Language Reference Manual (LRM) for the following reason:

1. The LRM gives us a frame of reference that is a standard.

2. By following the LRM, we can reasonably expect to be thorough.

3. We intend to illustrate the LRM jargon with good Ada code examples.

Therefore, the document follows the numbering of the LRM as closely as possible, including the appendices. However, in spite of this convention, our Ada Programming Guidelines are sufficiently self contained that they can be read without the LRM.

Chapters 1 to 14 of our document closely follow the corresponding LRM sections.

Appendix A of the document (Language Attributes in the LRM) describes the recommended documentation keywords both for design (user oriented) and code (programmer oriented).

Appendix B of the document (Predefined Pragmas in the LRM) illustrates the usage of pragmas.

Appendix C of the document (Predefined Language Environment in the LRM) gives the Ada source code of a decision deferral package (package TBD).

Appendix D of the document (Glossary in the LRM) is a glossary of terms used in the guide and not defined in the LRM.

Appendix E of the document (Syntax Summary in the LRM) is a place holder for the definition of "Ada_LINT", an Ada style and programming practice analyser. After a consensus has been reached about the specification of the tool and its command language, this appendix will include:

1. The APSE tool command language syntax and semantics definition.

2. The directives embedded in Ada documentation, style specification files, etc.

Appendix F (Implementation Dependent Characteristics in the LRM) identifies the links, waivers or modifications to the company standards made necessary by these guidelines.

Appendix G is a place holder for the definition of a "pretty printer" utility. After a consensus has been reached about the specification of the tool and its command language, this appendix will include:

1. The APSE tool command language syntax and semantics definition.

2. The directives embedded in Ada documentation, format specification files, etc.

Appendix H is an annotated bibliography.

The illustrated, recommended practices and guidelines suggest rules and provide examples of good Ada design and coding formats to promote readability, maintainability and, therefore, portability and reusability of Ada code.

An effort was made to alleviate the bureaucratic burden (that so often mars software standards) by concentrating on the programmer's "need to understand" and relying on automated tools for the mechanical (and subjective) aspects of programming such as indentation, alignment of tokens, etc. Most such rules are to be localized in an Appendix (Pretty_printer Definition).

Automated support from simple code templates and comment constructs to the definition of APSE tools are also considered.

4 EXCERPTS FROM THE GUIDELINES

Figure D.1.4-1 introduces the recommended comment constructs that allows simple tools to extract PDL or documentation from the Ada design or code.

The document strives to complement the LRM by illustrating its jargon with examples whenever possible. Unless the rule is particularly obvious, a rationale is given (possibly in the form of a bibliography reference), and an example is proposed. The rules are classified as either suggestions or strong recommendations. The latter are underlined for emphasis.

Figure D.1.4-2 to D.1.4-5 show the typical format of the rules given.

The document also draws on the IEEE 990 document (Ada as a Design Language) to show the smooth progression from Ada design to Ada code where practical. Figures D.1.4-6 and D.1.4-7 show two examples adapted from the IEEE document.

Finally, because efficiency issues pervade the LRM, the guide addresses the tradeoffs between readability, portability and

efficiency where appropriate.

5 CONCLUSION

The great richness of the Ada language and the need of programmers for good style examples, make Ada programming guidelines an important tool to smooth the Ada transition.

Because of the natural divergence of technical opinions, the great diversity of our government and private organizations and the novelty of the Ada technology, the creation of an Ada programming guidelines document is a difficult and time consuming task. It is also a vital one.

Steps must now be taken to ensure that the guide is refined in an organized but timely manner to reflect the growing level of expertise of the Ada community.

Daniel Roy is a senior member of the technical staff at Century Computing Inc. where he has been working since 1983. He received the Diplome d'Ingenieur Electronicien (MSEE) from ENSEA in 1973 and the Diplome d'Etudes Approfondies en Informatique (MSCS) from the University of Paris VI in 1975.

Robert W. Nelson is a member of the technical staff in the Software Engineering Section at NASA's Goddard Space Flight Center. He received a B.S in Mathematics from Drexel Institute of Technology and an M.S. in Numerical Science from Johns Hopkins University.

Authors current address:
Century Computing, Inc., 1100 West street, Laurel, Md., 20707.
Tel: (301) 953 3330.

Goddard Space Flight Center, Code 522, Greenbelt, Md. 20771.
Tel: (301) 344 4751.

2.7 COMMENTS

Comments should convey information not directly expressible in Ada. The conventions given below are used throughout this document.

(a) Use "--|" to indicate documentation [Intermetrics].

See Appendix A for the recommended documentation template.

(b) Use "--*" to indicate PDL construct [Intermetrics].

Using Ada as a PDL has numerous advantages. See [IEEE-990].

In the example of a function stub below, the three lines of the function specification are both documentation and PDL.

```
subtype INQUIRED_VAR_TYPE is TBD.SOME_TYPE;
function INQUIRE_INT (          --| Emulate DCL verb for integers --*
  PROMPT : STRING               --| --*
  ) return INQUIRED_VAR_TYPE is       --| --*

  type TRY_RANGE is range 1 .. TBD.MAX;       -- Nr try
  INQUIRED_VAR : INQUIRED_VAR_TYPE := 0;      -- Value returned
--
begin --* INQUIRE_INT
  --* Displays "prompt (min..max): "
 ERROR_LOOP:  --* Until good data or nr errors > max
  for TRY in TRY_RANGE loop   --*
    --* Get unconstrained value
    --* Validate and translate unconstrained value
    return INQUIRED_VAR ;     --*
  end loop ERROR_LOOP;        --*
end INQUIRE_INT ;     --*
```

See Appendix C for the definition of the decision deferral package (Package TBD).

Figure D.1.4-1: Rule for comments.

3.2.2 Number declarations

(a) Do not use numeric literals except in a constant declaration or when a number is obviously more appropriate.

This yields more readable and more maintainable code since a change in value will be localized to the constant declaration.

```
-- Circle object characteristics
RADIUS : constant := 10.0;              -- meters (constant object)
PI : constant := 3.14159;               -- (This is a named number)
CIRCLE_AREA := PI * (RADIUS ** 2);      -- (2 better than "TWO")
```

As a rule, using a constant object is better than using a named number which itself is better than using a numeric literal [NW].

Figure D.1.4-2: Illustrating the LRM jargon.

4.4 EXPRESSIONS

(a) Use parentheses to enhance the readability of expressions [NW].

```
X := (A + B) * (C / ((D ** 2) + E));
```

(b) Use static universal expression for constant declaration [NW].

Universal expressions maximize accuracy and portability. Static expressions eliminate run time overhead.

```
SMALL_STUFF : constant := 12  -- Better than "constant INTEGER :="
KILO : constant := 1_000;
MEGA : constant := KILO * KILO;
```

Note that the declaration of object "MEGA" would be less portable had KILO been declared as INTEGER since INTEGER'LAST could be less than one million on some target systems.

Also note that the following declarations are more readable than they would be using the constants MEGA and KILO above.

```
type MASS_TYPE is FLOAT range 1.0 .. 1.0E12;  -- Grams
GRAMS : constant MASS_TYPE := 1.0;
KILOGRAMS : constant MASS_TYPE := 1_000.0 * GRAMS;
TONS : constant MASS_TYPE := 1_000.0 * KILOGRAMS;
```

Figure D.1.4-3: Discussing the rules.

CHAPTER 9

TASKS

(a) Use a task for:

- o modeling concurrent objects (such as airplanes in an airport simulation).
- o asynchronous IO (other tasks may run while the IO task is blocked).
- o buffering or providing an intermediary link between asynchronous activities (buffer, active link between two passive tasks).
- o hardware dependent, application independent functions (device drivers, interrupt handlers).
- o hardware independent, application dependent functions (monitors, periodic activity, activity that must wait a specified time for an event, vigilant activity, and activity requiring a distinct priority).
- o programs that run on a distinct processor.

It is imperative that the methodology selected to develop multitasking systems minimize the number of tasks and provide guidance in the usage of the numerous tasking features of Ada. See [Cherry-84] for details.

Figure D.1.4-4: Rules and bibliography.

(b) Encapsulate priorities in a package [NW].

The LRM does not specifiy the number of priority levels.

```
with SYSTEM;use SYSTEM;        -- Makes sense here to shorten declarations
package PRIORITY_LEVELS is     --| Implementation dependent
--| Raise:
--|    The following declarations can raise CONSTRAINT_ERROR on
--|    some implementations since the number of priority levels
--|    is not defined in the LRM.
--| Purpose:
--|   Encapsulate implementation dependent priority definitions.
--| Portability:
--|   Some declarations may have to be modified for systems featuring
--|   less than 16 levels.  For instance *_HIGH and *_MED priorities
--|   may have to become equal to *_LOW in an 8 levels system.
--| Notes:
--|   Change Log:
--|   Daniel Roy      1-mar-86        Baseline

  LOWEST : constant PRIORITY := PRIORITY'FIRST;
  HIGHEST : constant PRIORITY := PRIORITY'LAST;
  NR_PRIORITY_LEVELS : constant POSITIVE := HIGHEST - LOWEST + 1;
  AVERAGE : constant PRIORITY := NR_PRIORITY_LEVELS / 2;

  IDLE : constant PRIORITY := LOWEST;
  BACKGROUND_LOW : constant PRIORITY := AVERAGE - 6;
  BACKGROUND_MED : constant PRIORITY := AVERAGE - 5;
  BACKGROUND_HIGH : constant PRIORITY := AVERAGE - 4;
  USER_LOW : constant PRIORITY := AVERAGE - 3;
  USER_MED : constant PRIORITY := AVERAGE - 2;
  USER_HIGH : constant PRIORITY := AVERAGE - 1;
  FOREGROUND_LOW : constant PRIORITY := AVERAGE + 1;
  FOREGROUND_MED : constant PRIORITY := AVERAGE + 2;
  FOREGROUND_HIGH : constant PRIORITY := AVERAGE + 3;
  SYSTEM_LOW : constant PRIORITY := AVERAGE + 4;
  SYSTEM_MED : constant PRIORITY := AVERAGE + 5;
  SYSTEM_HIGH : constant PRIORITY := AVERAGE + 6;
end PRIORITY_LEVELS;  --|

-- Using priorities
with PRIORITY_LEVELS;
task NASCOM_SERVER is --| Distribute NASCOM blocks
  pragma PRIORITY (PRIORITY_LEVELS. SYSTEM_LOW);
.....
end NASCOM_SERVER;
```

Figure D.1.4-5: Adding to Nissen and Wallis.

10.2.1 Example of subunits

The following example is adapted from [IEEE-990] and shows how to defer decisions at design time, using Ada as a PDL.

```
with TRACKER_DATA_TYPES; use TRACKER_DATA_TYPES;
procedure TARGET_TRACKER is   --| Radar echo processing

  ECHO : ECHO_TYPE;
  SMOOTHED_RANGE : SMOOTHED_RANGE_TYPE;
  SMOOTHED_ANGLES : SMOOTHED_ANGLES_TYPE;

  package FILTERING_ALGORITHMS is      --| Could be later extracted from
                                       --| here and "with'ed"
    function RANGE_SMOOTHING (         --|
      RAW_ECHO : ECHO_TYPE             --|
      ) return SMOOTHED_RANGE_TYPE;    --|

    function ANGLES_SMOOTHING (        --| May be a generic SMOOTHING
      RAW_ECHO : ECHO_TYPE             --| function could be written.
      ) return SMOOTHED_ANGLES_TYPE;   --|

  end FILTERING_ALGORITHMS;            --|

  -- The following postpone implementation decisions
  -- Simple stubs could be written
  function IS_ECHO_VALID (             --|
    RAW_ECHO : ECHO_TYPE               --|
    ) return BOOLEAN is separate;      --|
  package FILTERING_ALGORITHMS is separate;

begin --* TARGET_TRACKER
  .....
  if IS_ECHO_VALID (ECHO) then         --*
    SMOOTHED_RANGE := FILTERING_ALGORITHMS. RANGE_SMOOTHING (ECHO); --*
    SMOOTHED_ANGLES := FILTERING_ALGORITHMS. ANGLES_SMOOTHING (ECHO); --*
  else        --* decoy ?
    --* log decoy candidate coordinates
    null;
  end if;     --* IS_ECHO_VALID
  .....
end TARGET_TRACKER;   --|
```

Figure D.1.4-6: Using subunits and the TBD package.

Note that all types from the TRACKER_DATA_TYPES package may have been fully described (using Ada as a data definition language and TRACKER_DATA_TYPES as a data dictionary). Another solution is to use the TBD package:

```
with TBD;
package TRACKER_DATA_TYPES is --| data dictionary
--| Notes:
--|   Preliminary design

  subtype ECHO_TYPE is TBD.RECORD_TYPE;
  subtype SMOOTHED_RANGE_TYPE is TBD.REAL_TYPE;
  subtype SMOOTHED_ANGLES_TYPE is TBD.ARRAY_TYPE;
  .....
end TRACKER_DATA_TYPES;        --|
```

Figure D.1.4-6 (cont.): Using subunits and the TBD package.

(b) Use generics as a decision deferral technique during design. [IEEE-990]

```
generic                          --| Decision deferral
  type LIST_TYPE is private;     --| Don't want to bother with details now
function SORT (                  --|
  LIST : LIST_TYPE               --|
  ) return LIST_TYPE;            --|
--| Notes:
--|   Preliminary design

function SORT (                  --| --*
  LIST : LIST_TYPE               --| --*
  ) return LIST_TYPE is          --| --*
--| Notes:
--|   Preliminary design stub
  SORTED_LIST : LIST_TYPE;
begin --* SORT
  SORTED_LIST := LIST;
  return SORTED_LIST;            --*
end SORT;                        --| --*
```

The above generic unit can be further refined at detailed design time using the same kind of technique:

```
-- Adapted from [IEEE-990]
generic                          --| Decision deferral
  type ELEM_TYPE is private;     --| Members of the list
  type INDEX_TYPE is (<>);       --| Can be INTEGER or ENUMERATION type
  type LIST_TYPE is array (      --| We know more about type now
      INDEX_TYPE range <>        --|   but we still defer decisions
      ) of ELEM_TYPE;            --|   about index and element types
  with function "<" (            --| We now know we'll need to overload "<"
    LEFT : ELEM_TYPE;            --|   for our type.
    RIGHT : ELEM_TYPE            --|
    ) return BOOLEAN;            --|
function SORT (                  --|
  LIST : LIST_TYPE               --|
  ) return LIST_TYPE;            --|
--| Notes:
--|   Detailed design
```

Figure D.1.4-7: Using generics to defer decision.

Ada EDUCATION IN A SOFTWARE LIFE-CYCLE CONTEXT

Anne J. Clough
Ada Office
The Charles Stark Draper Laboratory, Inc.
555 Technology Square
Cambridge, Massachusetts 02139
(617) 258-2748

ABSTRACT

This paper describes some of the experience gained to date from a comprehensive educational program undertaken at The Charles Stark Draper Laboratory to introduce the Ada[1] language and to transition modern software engineering technology into the development of Ada and non-Ada applications. Initially, a core group, which included managers, engineers and programmers, received training in Ada. An Ada Office was established to assume the major responsibility for training, evaluation, acquisition and benchmarking of tools, and consultation on Ada projects. As a first step in this process, an in-house educational program was undertaken to introduce Ada to the Laboratory. Later, a software engineering course was added to the educational program as the need to address issues spanning the entire software life cycle became evident. Educational efforts to date will be summarized, with an emphasis on the educational approach adopted. Finally, lessons we have learned in administering this program will be addressed.

Introduction

Early in 1984, a laboratory-wide committee was set up at the Charles Stark Draper Laboratory, Inc. in Cambridge, Massachusetts, to assess the impact of Ada and the advances in software technology that this new DoD-mandated language would impose on the development of software. As a result of recommendations of this committee and support of upper-level management, a concerted effort is being undertaken to bring this technology in-house. A multi-level education and training program has been set up, Ada products are being evaluated and procured, consulting and support services are being provided as Ada projects become a reality at the Laboratory. This paper will concentrate on the education and training efforts to date.

---

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

At the heart of Draper's educational plan was the formation of a small, highly motivated and qualified group of individuals responsible for supporting the introduction of Ada technology throughout the Laboratory. A team of instructors from Raytheon/Mid-Atlantic Systems Facility and Raytheon/Equipment Development Laboratories assisted in this effort. Two courses were offered - a 16 to 20-hour Fundamentals of Ada tutorial for managers and an 80-hour Designing and Programming with Ada course for engineers and designers. Twenty managers and thirty engineers/designers participated in this initial phase. This group, chosen from a wide cross-section of projects in the Laboratory, continues to provide support to Ada activities. An Ada Advisory Committee chosen from this core group provides essential advice, feedback and support to the overall effort.

In order to coordinate, plan and implement all Ada-related activities, an Ada Program Office was established. Education, training, and the acquisition of basic tools were first priorities. Video courses and computer-aided instructional aids were evaluated and purchased to supplement more formal education. An in-house course was developed, and compilers and other support tools were evaluated and acquired. In addition, the Ada Office has followed closely and participated in the larger Ada community and publishes an Ada newsletter to keep the Draper technical staff informed of developments in this area. Figure 1 presents the initial plan for the acquisition of Ada technology at the Laboratory, and in fact, quite accurately describes what has happened during the past two years.

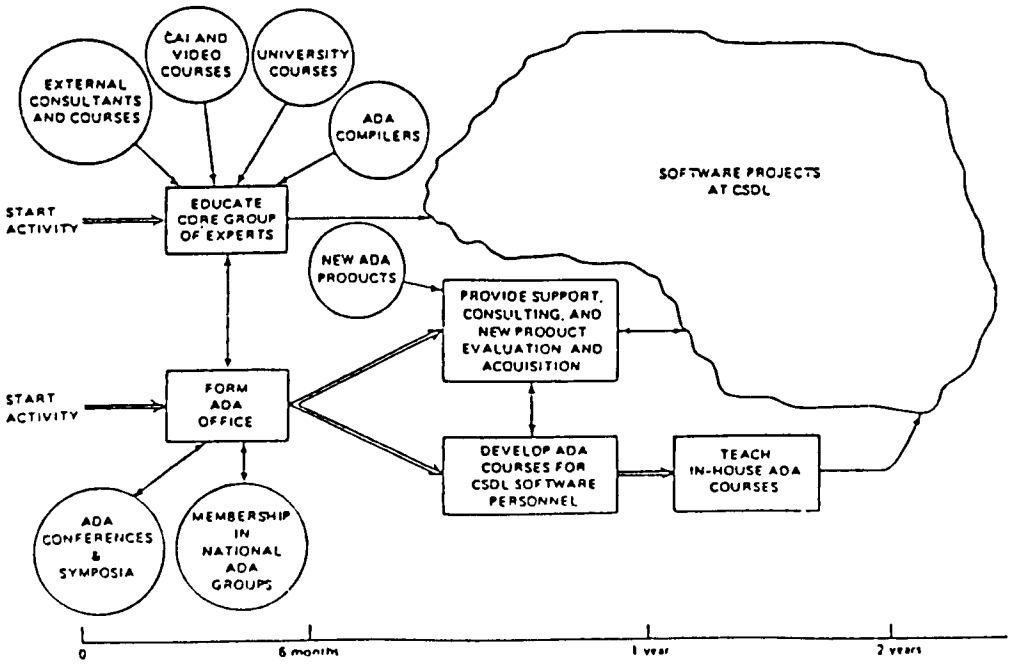


FIGURE 1. ADA TECHNOLOGY PLAN OVERVIEW

Developing an In-House Ada Curriculum

Because Ada is a very large language and at times complex, it was felt that traditional training techniques might not prove adequate. A three-tiered method was adopted which essentially takes a top-down approach to introducing the language. The first "pass" through the language presents an overall view. It concentrates on the need for a new approach in developing software and presents the history, development environment, and features of Ada. Initial exposure concludes with a look at simple, but complete, examples. The second pass studies Ada's features in more detail, but still does not emphasize syntax or grammar rules, or the more obscure, difficult, or infrequently used aspects of any language feature. A third and final pass then carefully examines each feature in detail with sufficient time allowed for discussion, questions, and programming practice.

In practice, this approach has proved to be very effective for several reasons. First, because of the structure of the course, it is possible for students to choose the level of participation desired. Participants who attend the first portion of the course receive an overview of the goals and features of Ada. Administrators, for example, often choose this level and find it appropriate for their purposes; they can exit the course with a cohesive set of knowledge. Those attending the first two segments of the course will learn to develop and recognize high quality software design in Ada from a conceptual viewpoint, rather than with an emphasis on detailed rules. This might be an appropriate level of detail for software project managers. Those participating in the entire course receive thorough hands-on training in the effective use of Ada, an essential requirement for the software practitioner.

A second reason that this approach proved effective is the direct result of the richness and complexity of the language. It is necessary to understand language features at a high level. "Why do we have this feature?" "How will it benefit me as a developer of software to be able to use this feature? "Where - in what context - will it be used?" If the instructor is not careful to address these issues at the beginning, it becomes very difficult to differentiate the forest from the trees, or lose sight of the trees themselves while we focus on a small portion of one tree. In addition, the very fact that we "visit" a language feature at least three times during the entire course makes the practitioner ultimately comfortable with that feature. Initially, he/she may be struggling with the concept itself ("just what is a generic?"), but ultimately it becomes familiar and the software developer can begin to realize and appreciate the extra capabilities that many of these unfamiliar Ada features provide to the developer.

Texbooks selected for this course are: "Software Engineering with Ada" by Grady Booch and "Programming with Ada" by J. G. P. Barnes. These are supplemented by pertinent articles and materials throughout the course. The bibliography at the end of this paper lists some of the materials that have been used both in this course and in a separate software engineering course.

Homework is an integral part of the course. Students design and implement Ada applications of increasing complexity as the course progresses. Though first sessions of the in-house course and the core course that preceded it were hampered by the lack of a validated compiler or even a compiler that could handle the full Ada language, the availability of a DEC VAX/VAX compiler now makes assignments more meaningful. Certainly hands-on work using a competent, fully-validated compiler is essential. Certificates are awarded to all participants in the course who satisfy homework requirements. This certificate is added to their personnel records, thus providing more incentive to complete all homework assignments and enabling the Laboratory to identify those staff members with Ada expertise.

Sixty hours of instruction are required for the entire course. Classes meet for 2 1/2 hours two mornings a week during working hours. Three sessions of the entire course have been given - approximately 110 people have participated, 45 have completed the full course.

Developing a Software Engineering Curriculum

Ada education at Draper Laboratory is very definitely software engineering with Ada. The emphasis throughout is on "engineering" software for large systems and all features are introduced and taught in that context. Ada, of course, is unique in that it has been expressly designed with features to encourage modern programming and software engineering practices. Designed for portability and reuse, providing effective encapsulation and data abstraction facilities, Ada has the potential to substantially change the way software is produced. As such, it is imperative that the importance of software design, the development of an appropriate Ada style, and the proper use of this language be emphasized in any Ada educational effort. Developing the "Ada mind-set" is important. As emphasized by many Ada experts and practitioners, a syntax-driven educational approach will not work and will most likely produce poorly constructed programs, disappointing results, and consequently negative feelings about the language itself. Software engineering therefore becomes a priority in our educational efforts throughout the entire Ada course, with each language feature discussed within this context. In addition, special sessions deal with Ada as a program design language, object-oriented design techniques, and investigating whether or not, and how well, Ada does meet the goals of software engineering.

Having emphasized that our Ada educational approach is heavily software engineering driven, it is nevertheless necessary to assert that one course cannot do it all. It is not possible to provide in a single course of any reasonable length a complete treatment of Ada and a comprehensive treatment of software engineering at the same time. Nothing less than changing the model of software design, development and maintenance acquired from previous language experience will suffice. Each sequential phase of the life cycle must be evaluated in terms of what

skills are required for effective and efficient production of software and the proper use of Ada.

The Ada course introduced software engineering concepts that may not have been consciously considered by students before that time. However, the need for more software engineering knowledge became apparent. To that end, comprehensive software engineering training, not foreseen in the original Ada program plan, is being developed by the Ada Program Office.

A software engineering course which deals with the entire life cycle has been added to Draper's educational program. Topics ranging from system definition, software costing and software standards to requirements analysis, design, testing, maintenance and configuration management are covered. Tools that can aid or automate various portions of the life cycle are presented.

The course was initially conceived as having a complete Ada orientation, both because it grew out of the Ada course and because it is being developed by the Ada Program office. However, widespread interest in software engineering by both Ada and non-Ada software developers led to a course that has both language-independent and Ada-dependent portions.

An integral part of this course is a workshop that allows participants to apply both software engineering principles and Ada implementation techniques to a real application as the course progresses. A space station command and control problem, adapted from an application designed and implemented for MITRE Corporation by a Boston University, College of Engineering student team,[2] was used for this purpose. An exercise had to be chosen that could be completed in a three-month time span but yet would be interesting enough and challenging enough to motivate the workshop members. Teams of approximately eight members each are given the documentation that has resulted from the system definition and scheduling phase of a project . This documentation is not complete; therefore one of the first things each team must do is get back to the "customers" -- (the instructors in this case) -- and fill in the gaps that remain in the system description. Each team then develops the application -- conducts requirements analysis, designs the software architecture, does low-level algorithmic design, codes and tests the solution. At this point, the two teams swap software and documentation, and each verifies the other team's software. Since the application is developed in Ada, the design portion of the course concentrates heavily on design methodologies and techniques suitable for developing Ada applications. Software requirements reviews, preliminary design reviews, detailed design reviews as well as testing and final reports are presented during regularly scheduled class sessions so that all members of the class can benefit from seeing the application progress through all stages of the life cycle.

---

[2] Ruane, Michael F. and Vidale, Richard F., Assessing Ada: Implementation of Typical Command and Control Software Functions.

Each presentation of a life cycle topic is completed before the workshop group begins work in that portion of the life cycle. Classes meet for 2-1/2 hours two mornings a week during working hours for thirteen weeks. The workshop then continues for an additional month at which time the entire class reconvenes to review testing and final reports by the workshop participants. The workshop schedule mirrors a 30-30-15-25% life cycle model -- one month for requirements analysis, one month for design, 2 weeks for coding and 3 weeks for testing.

As in the Ada course, members can choose their level of participation consistent with their own requirements and schedules. A participant can take part in the language independent portions only or in the entire course with or without the workshop. Exercises are provided so that all participants, whether or not they are members of the workshop, will gain experience applying the concepts that are presented. Certificates will again be presented to indicate participation and fulfillment of course requirements.

Lessons Learned -- Ada Education

A very pleasant outcome of the Ada effort thus far is an ever-growing group of people within the Laboratory who are being exposed to Ada and who are becoming enthusiastic about the language. This group includes people at all levels and across a wide variety of application areas. Many were frankly skeptical initially and have been impressed by Ada and its power and promise, especially in the area of the mission-critical embedded systems that are an important part of the Laboratory's activities.

At this point, we have had enough experience in Ada education that we can begin to assess its effectiveness. We can look critically at our course materials and see where they have been successful and where improvement is needed. We listen carefully to the comments of our students and attempt to tailor this course so that it meets our current and future needs. Some of what we have learned in this process follows.

In the Ada course, two areas of difficulty for the beginning student have caused us to make adjustments in the presentation of course material. The first, the strong typing of Ada, which is initially frustrating, actually becomes one of the first pleasant surprises for the student. Ada allows us, actually urges us, to define our own data types. An object is given a type when it is declared. Thereafter, an object's type is invariant throughout program execution. Values of one type cannot be assigned to variables of another type. Standard operators cannot be used with variables of different types. For the student accustomed to working with languages that do not have strong typing features, this seems very restrictive and he is at least initially annoyed every time the compiler flags a typing error and makes him explicitly convert values from one type to another before an operation can be performed or an assignment statement can be executed. However, the first time that the compiler catches a typing error that would have formerly

slipped through and become an elusive bug in a less strongly-typed language, a new convert to strong typing is won.

The second difficulty for the new student involves the input/output feature of Ada. This has necessitated more emphasis on input/output early in the course in recognition that even simple programs need some rudimentary I/O. Since input/output in Ada uses packages and generics as well, I/O can be confusing and can seem needlessly awkward to a student accustomed to working with a language with built-in input/output facilities. Time spent familiarizing the student with exactly how this works in Ada not only eases his frustrations but also provides him with an example and model of an use of packages and generics. This can be very helpful in understanding and using these concepts later on.

The problem of presenting topics in an optimum sequence is not a trivial one, both in terms of maintaining class interest and applying the concepts presented. As an example, in order to cover Ada types completely, much material must be presented. However, if some effort is not made to disperse this material throughout the detailed portion of a course, rather than present it in a single block, it will surely be difficult to maintain interest. The "divide and eventually conquer" approach to Ada's typing topics also benefits the student when derived types are presented at enough distance from the concept of subtypes so that the two do not become hopelessly muddled. An additional consideration is that of allowing sufficient time to apply those features covered at the end of any course. This can be a serious problem if tasking is the last topic presented as is the case in many Ada curriculums. Since most programmers tend to think in a sequential manner and tend to have the most difficulty dealing with concurrency and the issues concurrency raises, putting this topic to the end of a course will not give the student sufficient time to apply these new concepts. Not only will students be unable to appreciate Ada's tasking facility, but they will also have real hesitancy to use this feature at all when beginning to design systems in Ada.

We have found that students at all levels want more complete and concrete examples of good Ada systems. "Real" work-related examples are especially helpful. The experienced programmer wants to concentrate on the unique features of Ada -- he prefers to learn on his own the simple statements, constructs and expressions that are similar to those found in most high order languages. Students would like each Ada construct to be accompanied by many examples of its use -- the Language Reference Manual syntax format supplemented by many more examples would be useful. The Ada-unique packaging and generic features have proven to be accessible to most students who very quickly perceive their power and begin to use these features effectively. Exception handling, and the way Ada implements it, always initiates lively discussion. While quite valid concerns about the misuse of this feature are often expressed, students soon produce code that uses the exception handling feature effectively.

Tasking is perhaps the most difficult feature for new students of Ada. Many traditional languages do not have features that allow parallel processing. Because of this, most programmers have a great deal of experience -- or all of their experience -- in sequential programming.

This lack of experience in programming concurrent processes, coupled with the unique problems that can arise such as deadlock, starvation and timing considerations, make this feature a difficult one to both teach and learn. It has been necessary to expand this portion of the curriculum. A tool developed at Draper, which graphically shows tasks operating concurrently and explicitly shows such things as actuation, suspension of tasks, rendezvous, and termination, has helped the educational effort in this area. However, an advanced Ada course which would concentrate in large part on tasking should perhaps be considered.

Lessons Learned -- Software Engineering Education

Although we have not had as much experience with the software engineering course as with Ada education, the first session of this course has been very successful. Participation, as has also been the case in our Ada educational efforts, has included a wide cross-section of the Laboratory both in terms of application areas and job level. Managers are participating both in the course and in the workshop, as are entry-level engineers and programmers. A great deal of enthusiasm centers around the workshop approach as this provides a convenient mechansim to apply techniques and tools discussed in class in an essentially "no-risk" situation. There is a great deal of learning that takes place in the workshop groups, as people with diverse backgrounds and experience are taking part. In the classroom as well, much information exchange is taking place and a wide range of expertise is being tapped. This combination has resulted in a very effective learning forum. The Ada Program Office is coordinating the course and supplying most of the instruction; however, a number of presentations given by experts both within the Draper community and outside as well, have greatly enhanced the course offerings. Through the very active participation of its members, all participants in the course are being challenged to think about the way they are currently developing software. In addition, any new methods being presented, whether they be requirements analysis, design or testing methods, are subjected to the most rigorous scrutiny. "Will this method work as advertised by its proponents?" "Will it work in the type of application that I develop?"

As mentioned earlier, participants can choose their own level of participation. Though course developers had assumed that members who had no familiarity with Ada would choose to participate in the language-independent portions only, in actuality most members have opted for the entire course. Because of this, several sessions were added to familiarize non-Ada participants with Ada's unique features. An unexpected side effect appears to be a group of people interested in registering for our next Ada course.

Have we presented these two courses in the correct order? Shouldn't a software engineering course precede a course in Ada? Although this will be the case for the group of people just mentioned, in general, the opposite approach has worked quite well. First of all, the Ada course has a good amount of software engineering content. In addition, having

the majority of course participants conversant with Ada has enabled us in this second course to consider a number of design methodologies uniquely suited to Ada, has enabled us to conduct a non-trivial workshop in Ada and has allowed us to deal with some of the more advanced and difficult aspects of the language, especially in the tasking area.

Future Plans

As Ada applications continue to be introduced into the Laboratory, the Ada Program Office will continue its efforts in education and its efforts to provide a more supportive programming environment. In Ada itself, an advanced course concentrating heavily on the tasking aspects of the language and providing more guidance on developing embedded applications may need to be added to the curriculum already developed. Utilizing the low-level features of the Ada language may need closer examination as well. For the course already "on the shelf," tuning and tailoring for Draper's particular requirements will be a continuing process. The top-down, three-level approach has proved quite effective. Perhaps a separate course for administrators or a separate course for managers will need to be given at some point in the future -- our essentially modular approach would make that very easy to prepare. Continuing seminars sponsored by the Ada office provide an opportunity for those who will not be using Ada immediately to keep their Ada skills up to date and enable those presently involved in Ada applications to keep informed about new Ada methodologies, techniques and tools.

Software engineering will continue to be emphasized. Growing interest within the Laboratory ensures a repetition of the software engineering course discussed in this paper. In future sessions, different applications may be given to each team so that, in the testing phase, teams can test applications that they have not developed. Since the major thrust of the software engineering course is on the requirements analysis, design, implementation and testing portions of the life cycle, further courses or intensive seminars could be added on the system definition and scheduling phase. The software planning phase and software cost analysis could be covered in more detail. Review techniques, maintenance, security and configuration management are other possible topics for future in-depth coverage. Possibilities for further growth in training and education surely exist.

Conclusions

Experiences in education and training at Draper Laboratory illustrates the effectiveness and long-term benefit of establishing an in-house capability in this area. Many training offerings are available that provide intensive, short-term training in Ada; fewer offerings are available in software engineering. The long-term effect of some of these offerings is often questionable. Certainly a five-day or two-week

intensive "hands-on" approach to teaching Ada will not really allow students to either become comfortable with the new concepts presented or to grapple with the more difficult issues. A course spread over a longer period of time -- our courses traditionally have a 3-4 months span -- allow the student time to assimilate new ideas, raise questions and most importantly get real hands-on experience with non-trivial applications. In addition, having in-house support for Ada and software engineering ensures that, long after a course has been completed, the instructor or instructors are available for consultation and assistance. This latter advantage cannot be overemphasized when new technology is being introduced if the desire is to truly assimilate and integrate that technology into the software development process.

BIBLIOGRAPHY

1. Albrecht and Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation", IEEE Transaction on Software Engineering, Vol. SE-9, No. 6, November 1983, pp. 639-648.

2. Ausnit, Cohen, Goodenough & Eanes, "Ada in Practice", Springer-Verlog, 1985.

3. "An Object Oriented Design Handbook for Ada Software", EVB Software Engineering, Inc., 1985.

4. Barnes, J.G.P., "Programming in Ada", Addison-Wesley Publishing Co., 1984.

5. Boehm, B., "Software Engineering Economics", Prentice-Hall, 1981.

6. Boehm-Davis & Ross, "Approaches to Structuring the Software Development Process", General Electric Company, October, 1984.

7. Booch, Grady, "Software Engineering with Ada", Benjamin/Cummings Publishing Company, Inc., 1983.

8. Brooks, Frederick, "The Mythical Man-Month", Addison-Wesley Publishing Company, 1975.

9. Buhr, R.J.A., "System Design with Ada", Prentice-Hall, Inc., 1984.

10. Fairley, Richard, "Software Engineering Concepts", McGraw Hill Book Company, 1985.

11. Freeman & Wasserman, "Tutorial on Software Design Techniques", IEEE Computer Society Press, 4th Edition.

12. Helmbold & Luckham, "TSL: Task Sequencing Language", Proceedings of the Ada International Conference, 1985, Cambridge University Press, Cambridge, pp. 255-274.

13. Mardrioli, Zicari, Ghezzi and Tisato, "Modeling the Ada Task System by Petri Nets", Computer Language, Vol. 10, No. 1, pp. 43-61, 1985.

14. Myers, Glenford, "The Art of Software Testing", John Wiley & Sons, 1979.

15. Pressman, Roger S., "Software Engineering: A Practitioner's Approach", McGraw-Hill Book Company, 1982.

16. Ruane, Michael F. & Vidale, Richard F., "Assessing Ada: Implementation of Typical Command and Control Software", Boston University, College of Engineering, Boston, MA, 1984.

17. Snaufer, Joseph B., "Practical Guidelines for Testing Ada Programs", Master's Thesis, Arizona State University, 1985.

18. Szulewski & Sodano, "Design Matrics and Ada", Proceedings of the 1st Annual Washington Ada Symposium, Sponsored by ACM, 1984, pp. 105-114.

19. Wegner, Peter, "Self-Assessment Procedure VIII", Communications of the ACM, Vol. 24, No. 10, October, 1981.

20. Weiner & Sinovec, "Software Engineering with Modula-Z and Ada", Wiley & Sons, 1984.

VIDEO TAPES

1. Ichbiah, Barnes and Firth on Ada, Alsys, Inc., 1984.

2. Software Engineering Training Curriculum, R.S. Pressman & Associates, Inc., 1985.

CAI

1. Lessons on Ada, Volume I and II, Alsys, 1983 and 1984.

Towards A Software Profession

by Edward V. Berard, EVB Software Engineering, Inc.

"Between the amateur and the professional ... there is a difference not only in degree but in kind. The skillful man is, within the function of his skill, a different integration, a different nervous and muscular and psychological organization ... A tennis player or a watchmaker or an airplane pilot is an automatism but he is also criticism and wisdom."

Bernard De Voto
from "Across the Wide Missouri"
[1947]

"Liberty trains for liberty. Responsibility is the first step in responsibility."

William Edward Burghardt Du Bois
from "The Legacy of John Brown"
[1909]

"The absurd man is he who never changes."

Auguste Marseille Barthelelmy
from "Ma Justification"
[1832]

"A professional makes it look easy."

Source Unknown

"Old age and treachery will overcome youth and skill."

Julian Levi
Motto for the "65 Club"

"Computer programming," as we know it today, is a little more than 35 years old. You might even say that, as an occupation, it is in its "late adolescence." Programmers. themselves, have been known to exibit all the symptoms of adolescents, e.g., arriving at work at odd hours, dressing in a unconventional manner, spouting technical gibberish that is seldom understood by anyone other than another programmer, referring to themselves as "gurus" or "wizards," and an extreme loathing to accept anything that even vaguely resembles responsibility. These items may be collectively referred to as the "Real Programmers Don't Eat Quiche" syndrome.

To be fair, an increasing number of programmers have attempted to change their image. They have made it plain that they wish not only to be taken seriously, but they also wish to be regarded as "professionals." Even the term "programmer" has become passe'. Many programmers, and their companies, now refer to programmers as "software engineers." (Note that this change in nomenclature is seldom accompanied by a corresponding change in job description.)

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

Put simply, changing the image of "programming" and programmers is a "tall order." Both the software and the people who deal with it suffer from a severe case of "Rodney Dangerfield Syndrome," i.e., they get little if any respect. Hardware professionals often look at software as something that one "slathers on the hardware" to get the real product (i.e., the hardware) out the door. Even programmers have few qualms about stealing software. People, in general, have a hard time recognizing software as a product.

The attitude that "anyone can be a programmer" is still very prevalent in our culture. The only credentials one seems to need to call oneself a programmer are a general familiarity with the syntax of a programming language, a rudimentary knowledge of a text editor, and enough exposure to an operating system to invoke a compiler. Most programmers are totally lacking in skills such as software design, software testing, software maintenance, software quality assurance, error analysis, metrics, and configuration management.

Our work seems to be cut out for us. If we wish software professionals to be considered professionals in every sense of the word, two of the major obstacles we will have to overcome will be: the inability to think of software as a product, and the idea that little or no skill is required to create and handle software throughout its life-cycle.

Professions and Professionals

If we are going to address the issue of *professionalization*, we must first define what it is we mean by a profession and by a professional. A logical place to start is the dictionary. The 1979 version of *Webster's New Collegiate Dictionary* provides two common definitions for a **profession**:

1. "a calling requiring specialized knowledge and often long and intensive academic preparation," and

2. "a principle calling, vocation, or employment."

Unfortunately, the second definition more accurately describes the "software profession" as it exists today. There are, however, a small, but growing, number of organizations where the first definition is more appropriate. These organizations have found that an engineering approach to the software life-cycle is not only less chaotic, but cost-effective as well.

Our dictionary also provides two definitions for a **professional**:

1. "one that engages in a pursuit or activity professionally," i.e., one who conforms "to the technical or ethical standards of a profession," and

2. "engaged in by persons receiving financial return."

If you have any doubt that the second definition more accurately reflects the "software professional" of today, merely ask a software professional to list (or give a specific reference to) "the technical or ethical standards" of the profession. The Computer Society of the Institute for Electrical and Electronics Engineers (IEEE-CS), has made a good start at defining some of the technical standards for the software profession. The Institute for Certification of Computer Professionals (ICCP) requires that those who pass a written examination *and* "subscribe to the *ICCP Codes of Ethics, Conduct, and Good Practice*" may use the designation "CCP" after their names. These last two points illustrate that attempts already have been made to establish technical and ethical standards for software professionals.

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

We should also note that there will probably be a need for **paraprofessionals** in the software industry for some time. (Webster's defines a paraprofessional as "a trained aide who assists a professional person.") While we will acknowledge the probable need for paraprofessionals in the software industry, we will not discuss their required qualifications in this article.

Characteristics of a Profession and Professionals

If you were to interview a number of different professionals (doctors, lawyers, teachers, nurses, airline pilots, electrical engineers, and certified public accountants), you would find that their professions placed a number of requirements on anyone who wished to be considered a professional, including:

- *a minimal level of training for entrance to the profession* . Many professions require a minimum of a four-year college degree from an accredited institution. Even those that do not specifically require a college degree often require many hours of training which may take years to accomplish.

- *some form of formal certification.* The classic examples are the bar examination for lawyers, the CPA examination for accountants, and board certification for medical doctors. *In many professions, certification is not a one-time affair, with professionals having to re-certify every one to three years.*

- *some form of continuing education or training* . Just because a professional has acquired a college degree does not mean that he or she is finished with formal education. Teachers, lawyers, nurses, and other professionals are often required to take a minimum number of courses per year to maintain their certification. (Even if college courses are not required, most professionals must keep current with their profession. Imagine an accountant who is unaware of the most recent changes in the tax laws, or a doctor who was not up-to-date on the latest findings on a particular antibiotic she was prescribing.)

- *some minimal level of proof of performance.* Proof of performance can take many forms. For example, it shows up as "publish or perish" for college and university professors, successful diagnosis and treatment for doctors, and the won/lost record for attorneys. *Professionals must demonstrate that they can practically apply the training required for their profession.*

- *conformance to professional standards* . Prior to admission to a profession, a candidate will probably be made aware of the standards (e.g., methodologies, metrics, and levels of quality) for the profession. The certification process will most likely test the candidate's knowledge of these standards. Upon being accepted into the profession, the professional will be expected to conform to the existing standards, and to keep abreast of any changes to these standards.

- *adherence to professional ethics.* Webster's defines ethics as "a set of moral principles or values." Professional ethics involve such items as the professional's obligations to his or her client, the social responsibility of the professional, the relationship of professionals to their employers, and acts which might discredit or degrade the profession.

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

- *the taking of responsibility and acceptance of liability.* Professionals take direct responsibility for their actions. The profession, as a whole, usually provides some form of established guidelines and acceptable limitations on responsibility to guide the professional. Professionals may also be required by some municipal, state, or federal laws to take on some amount of additional responsibility. Examples of professional responsibility can be found almost daily in any newspaper account of an event which caused harm to one or more individuals, or in which some law was broken. Classic examples include plane crashes, bank failures, and medical malpractice cases.

The profession itself typically provides a number of benefits to its members, including:

- *the establishment of a number of professional societies.* Professional societies provide a number of services for their members. (In fact, when people speak of a profession they are often referring to the professional societies for that specific profession.) They sponsor continuing education for their members, publish professional journals and other periodicals, provide a forum for the members to express opinions and influence the profession itself, and generally represent the interests of their members.

- *providing protection for its members.* One of the the most common defenses used in a malpractice suit is that the professional was "following generally accepted professional procedures and guidelines." Professions also provide guidance in such areas as rights of ownership, items which will directly affect current practices, and career advancement.

- *maintaining public respect for the profession as a whole.* The words "profession" and "professional" usually have positive connotations in the mind of the public. This is no accident. Professions (both the professional societies and the membership in general) continually strive to maintain, and improve, the image of the profession in the mind of the general public. *This translates into increased status and financial gain for the professionals themselves.*

Cook and Winkle (in their book, *Auditing Philosophy and Technique*) observe that: "Professions are characterized also by the performance of intellectual services, as contrasted with manual and artistic labor. In addition, professions recognize a duty of public service and adopt a code of ethics generally accepted as binding upon their members." Later, in the same book, they make an interesting observation regarding the American Institute of Certified Public Accountants (AICPA): "Frequently, state regulations are modeled after AICPA pronouncements. Many court decisions use the statements from the AICPA as criteria to evaluate public accountants and their work." *This is a powerful statement.* It illustrates a precedent for a profession to directly influence outside governmental regulation of the profession.

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

Professionalization

Professionalization may be loosely defined as the establishment, and adherence to, a professional model where one either did not previously exist, or was not firmly entrenched. In a professional model, the individuals who refer to themselves as professionals exhibit most, if not all of the characteristics of professionals mentioned in the previous section. Further, there is a "professional atmosphere" which is established jointly by both the individual members of the profession and the existing professional societies. This professional atmosphere exhibits the characteristics of the profession, also described in the last section.

A number of steps occur in the process of professionalization. These steps need not occur sequentially (i.e., some may occur concurrently and some may even occur "out of order"), and (ideally) they should make use of work which has already been done. The following list of steps is usually required for professionalization:

1. recognition of the need for professionalization. This usually occurs when the need for highly-skilled, uniformly-trained individuals is recognized, i.e., the work performed by those already in the field becomes increasingly critical in nature.

2. the formal definition of the profession. This will include establishment of standard terminology for the profession, creation of job titles and descriptions for profession members, and identifying relationships (e.g., profession to profession, professional to client, and the relationship of the profession to the general public).

3. the identification of key professional societies. These societies will hopefully have already achieved some degree of formal status (e.g., the respect of the professional community, the publication of useful periodicals, and the conducting of local and national meetings). These societies will prove invaluable in aiding the rest of the professionalization process.

4. the establishment of minimal entrance criteria. This must include such issues as: minimal formal education, minimal experience (apprenticeship), and a certification process.

5. the establishment and recognition of education and training programs. These can include existing college curricula and profession-approved continuing education programs.

6. the establishment of formal certification (and re-certification) procedures. The certification process must be based on the minimum amount of useful skills and knowledge required by a "typical" professional. The re-certification process should be directed towards the career paths available to the professionals. The certification and education processes will obviously directly affect each other.

7. the collecting and promulgation of professional standards. Professional standards encompass such items as: procedures, methodologies, metrics, acceptable performance levels, and tools.

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

8. the identification of relevant professional ethics. A code of ethics for the profession must be established, made known, and adhered to by the profession. Ideally, the profession's code of ethics will be incorporated into the certification process.

9. the establishment of minimal, quantifiable performance goals for the profession and the professionals. Professionals must be expected to meet (and hopefully exceed) some minimal set of performance goals to maintain their professional status. They must have some way of knowing how well they are doing in their chosen field. The profession must continually strive to improve itself as a whole (e.g., a decrease in the average error rate per member).

10. the identification of required continuing education (and a continuing education process). No professions are static, especially the technical professions. The useful life of much professional knowledge continues to shrink. For example, it has been said that the technical knowledge of the human race doubles every four years. Typically, professionals are required to take a minimal number of prescribed courses per year to maintain their professional status.

11. the identification of professional responsibilities and liabilities. Professionals must be aware of their responsibilities to their clients, their employers, other professionals, and to the profession in general. Further, they must be aware of the liabilities which come along with responsibility. Only people who are legally insane are not held accountable for their actions.

12. the establishment of mechanisms for filing of grievances, removal of individuals from the profession, and appealing both. The status of a profession is diminished by the inclusion of individuals who no longer meet standards established by the profession.

13. the establishment and maintenance of a positive public image. A profession with a positive public image can command better benefits for its members, including higher levels of compensation.

While all of the above obviously take time to occur, they can be accomplished in a relatively short time, say within three to four years. This time can be further shortened by a focused effort on the part of the professionals themselves.

Examples of Professionalization

Examples of professionalization abound. Outside of the software industry, we have as examples: the legal profession, the accounting profession, the medical profession, and the teaching profession. The professionals in these areas are lawyers, certified public accountants, doctors and nurses, and teachers respectively. Even a casual conversation with any individual associated with these professions would reveal that most, if not all, of the points covered in the previous section are relevant to their profession. The mechanisms and the nomenclature may vary from profession to profession, but the points themselves still remain relevant.

Professionalization occurred in these, and other, disciplines for a number of reasons. In some cases, it was the desire to disseminate knowledge, skills, and techniques in a uniform manner. For others, professionalization was forced (or threatened) by some form of government. Quite surprisingly, professionalization does not seem to occur at any consistent time in the existence of a discipline. For example, it took literally thousands of years before civil engineering became professionalized in the modern day sense, while electrical engineering became professionalized almost as soon as it was recognized as a separate discipline.

A major factor in the speed with which a discipline becomes professionalized seems to be the environment in which it functions. For example, when electrical engineering first came into existence, other scientific and engineering professions were already firmly in place. These established professions provided paradigms for the creation of the electrical engineering profession. When one views the handling of software (and related issues) throughout its life-cycle as an engineering problem, we can easily see that paradigms already exist for the professionalization of those who are directly responsible for software. Further, this professionalization process should be taking place now, i.e., definitely much before 1990.

Some of the steps necessary for professionalization are already in place. We have a number of professional societies. Computer science curricula at colleges and universities have been in place for more than twenty years. (Although some schools offer some "software engineering" courses at the undergraduate level, it appears that none are offering undergraduate degrees in software engineering, and only a few are offering advanced degrees in the topic.) The IEEE-CS is actively involved in defining standards for the engineering of software. The Institute for the Certification of Computer Professionals (ICCP) and the Certified System Professional Program (CSPP), among others, have established model programs for the certification of software professionals. The Ada Joint Program Office (AJPO) has established a working group in Ada and software engineering education. One of the issues that this group is looking at is the certification of Ada professionals. Finally, the Europeans are also exploring the idea of certification of computer professionals.

Software Engineering and Computer Science

For purposes of this article, I will restrict our attention on professionalization to three general categories of potential software professionals: computer scientists, software engineers, and software engineering management. It is not our intention at the moment to provide in-depth definitions of each of these professions. Instead, we will provide quick sketches of each, and leave the details to a later article.

In discussing computer scientists and software engineers, we will differentiate the two using the paradigm of more "conventional" engineers and scientists. A scientist is chiefly concerned with explaining current phenomena, predicting future phenomena, and generally improving the state of technical knowledge available to the human race. An engineer takes the information supplied by scientists, and others, and uses this information to produce cost-effective, paragmatic solutions to real-world problems. (These are obvious oversimplifications, but they will suffice for now.)

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

A computer scientist might be looked upon (most simply) as an applied mathematician. Some would rightly say that, with the disappearing differences between hardware and software, that some areas of computer science encompass a good deal of computer hardware technology. For our purposes, a computer scientist has a minimum of a four-year degree in an approved computer science curriculum (e.g., the ACM 1978 curriculum) from an accredited college or university. (Remember, the degree alone is not enough to make one a professional.)

A computer scientist might specialize in compiler design, queueing theory, operations research, or language-driven hardware architectures. While a computer scientist might focus on the details of some aspect of computer science (e.g., algorithm design), he or she might not have an immediate practical application for the technology they are uncovering. It is very likely, though, that this uncovered technology can be used in some existing, or soon to exist, practical application. (Just as a physicist studying the quantum mechanics of molecular collisions might produce results which have applications in gas lasers.)

Just as "conventional" scientists build on the work of other scientists, a computer scientist most often builds on the work of other computer scientists. *This means that an error introduced via carelessness or faulty analysis on the part of one computer scientist can have drastic consequences even outside of that computer scientist's immediate area of specialization.* Keep in mind that software permeates our very existence. (It has been said that the average American comes into contact with at least "two dozen computers" every day.) In addition, computer science technology is being used in increasingly critical application areas (e.g., pacemakers, cruise missile guidance systems).

Software engineering, like any engineering discipline, involves a mixture of technologies. The basic background of a software engineer requires: computer science, mathematics, engineering disciplines (e.g., design methodologies, metrics, error analysis), communication skills, imagination, problem solving skills, ingenutiy, and a respect for simple, pragmatic solutions to real-world problems. Software engineers, like their "more conventional" counterparts, has a minimum of a four-year degree from an accredited college or university. Unfortunately, the few software engineering degree programs that currently exist are almost exclusively graduate programs.

Software engineers may be charged with any number of tasks, e.g., the developing, testing, maintaining, measuring, or assuring the quality of a particular software system. They constantly find themselves integrating different technologies, making tradeoff decisions, and dealing with many different types of people (users, other engineers, managers). Most of the time they have a very tangible goal in sight. This goal must be reached within the (often unreasonable) limits on time, money, and other resources established at the begining of the project.

Like computer scientists, software engineers build on the work of other software engineers *(with the same serious implications)*. Like computer science, software engineering is a rapidly growing, rapidly changing discipline. The software engineer is being asked to apply his or her skill to increasingly critical applications, e.g., life-support systems or the Strategic Defense Initiative ("Star Wars").

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

Even with the best-trained computer scientists and software engineers, the success of a project is not guaranteed. Poor management can ruin any project. The benefits of a truly professional technical staff cannot be realized without the support of professional management. While a successful professional software project manager need not be a technical wizzard, he or she must know who to hire, what to ask for, and what can reasonably be done with existing technology. We must be just as concerned with the professionalization of technical management as we are with the professionalization of the technical staff.

As you can see, I have described a huge task: professionalization of software personnel, i.e., computer scientists, software engineers, and technical managers. I want to now further narrow the scope of this discussion to the professionalization of software engineers. This is done primarily to keep this article from "becoming a short novel." However, much of what we have to say about software engineers will hold true for computer scientists and technical managers.

The U.S. Department of Defense (DoD) has advocated a software engieering approach to their Ada® effort. In fact, Ada is but one small piece of the DoD's Software Technology for Adaptable Reliable Systems (STARS) effort. We will use this a basis for our discussion for the professionalization of software engineering. Virtually everything we have to say will apply to the professionalization of all software engineers, regardless of whether they use Ada or any other language.

The Motivation for the Professionalization of Ada Software Engineers

There are several people who have a vested interest in the professionalization of Ada software engineers:

- the contracting office,
- the software engineer's employer,
- the software engineers themselves, and
- the general public.

One of the most difficult tasks for a contracting office is determining the *real* capabilities of a potential contractor. A university degree is somewhat meaningful, but often the contracting office is more interested in the actual "on-the-job experience." On-the-job experience is usually measured in the types of projects the personnel have previously been associated with, and the length of time the personnel have logged on each project. These are, unfortunately, very crude metrics.

If software engineers were professionalized, however, the contracting office's job would be somewhat easier. For example, if a potential contractor identified an individual as a software engineer, the contracting office might be able to make the following assumptions about that individual (depending on how the profession and its professionals have been defined) :

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

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

- *he or she has had a minimal amount of education at an accredited institution*. Further, this education covered a specific set of known topics which were directly relevant to their job.
- *he or she has known professional standards and guidelines to follow,*
- *he or she has gone through some known form of certification (and re-certification) process,*
- *he or she must abide by a known set of professional ethics,*
- *he or she has had to exhibit some minimal level of performance in order to remain in the profession,*
- *he or she will take direct responsibility (and liability) for their work,* and
- *he or she will be required to take a minimal amount of continuing education each year.*

In essence, the software engineer becomes more of a known quantity. (*It is important to realize that professionalization guarantees only minimal levels of quality.* While this might not seem like much, remember two things. At present there are no guarantees of any level of quality for any software "professional." Second, establishing a "floor for performance", tends to raise the "ceiling of performance" for the profession as a whole.)

The employer of the software engineer has a number of reasons for being extremely interested in the professionalization of software engineers, including:

- *all of the reasons listed previously for the contracting agency.* This makes hiring much easier.
- *while a professional might cost more than a non-professional they are usually much more cost-effective (i.e., productive) than non-professionals.* This does *not* mean that all non-professional software engineers are not very productive. It means, depending on the effectiveness of the professionalization process, that the odds are greater that a professional will be more productive because he or she will most likely have been exposed to productivity increasing techniques.
- *a professional is more likely to have a more mature, business-like (i.e., professional) attitude.*

Software engineers will, of course, be interested in professionalization. Some of the more important reasons, include:

- *the ability to know, in advance, the minimal criteria for entrance and advancement in the profession.* There will be a number of secondary benefits along this line. For example, it will be easier to coordinate college and university curricula with the demands of the job market. In addition, the requirements for advancement along a specific career path will be better defined.

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

- *the ability to determine how much an individual software engineer has improved over time.* At present, few software engineers know how they "stack up" against their fellow software engineers. They also have little idea about how to improve their status (worth) in their chosen field.

- *the chance to learn new things, which are relavent to their profession, on a continuing basis.* An active re-certification program will encourage (and obligate) software engineers to remain current in their field.

- *the protection and advice of the profession.*

- *known standards, guidelines, and practices which are established by the profession (i.e., not by some organization with little, or no, familiarity with current technology),*

- *the respect given to professionals, by the public, and by other professionals.*

The general public will be interested in the professionalization of software engineers for a number of reasons, including:

- *as taxpayers and consumers, the public is keenly interested in acquiring high quality software at the lowest possible price.* We should not have to belabor the point that software is consuming an ever-increasing chunk of every tax dollar, and of every new modern appliance.

- *professionalization reduces the chance of major (and minor) disasters which are the result of erroneous software.* If the general public had any idea how much of their daily lives, and their national security, depended on software, there would be an immediate large public outcry for professionalization.

- *the public, as a whole, is more comfortable dealing with professionals (e.g., airline pilots, doctors, lawyers).*

Who and What Needs To Be Certified

The item which will probably evoke the most controversy in the professionalization process is that of certification. Before we go any further we should define what we mean by certification. The certification process for the software engineers and technical managers themselves will probably be not unlike that currently used in other professions, i.e.:

- It will require that the candidates have a minimum level of formal education.

- Candidates may have to serve an apprenticeship (residency) for some pre-specified period of time.

- A written examination, possibly spanning several days, will definitely be a requirement.

- Personal and professional references may have to be supplied.

- The candidates will have to sign a document saying they will adhere to a professional code of ethics.

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

- Other documents that may have to be signed might address items such as professional conduct, acknowledgement of professional responsibility and liability.

- The candidates may have to demonstrate that they are covered by any appropriate insurance policies (e.g., malpIractice insurance).

- If this is a re-certification process, the candidate will have to demonstrate that he or she has taken appropriate continuing education courses within the time limits specified by the profession.

If, for example, we focus on the Ada community, we find that certification can be applied to a number of items, including:

- Ada and software engineering courses,

- Ada and software engineering curricula,

- the instructors for these courses and curricula,

- the graduates of these courses and curricula (managers as well as technicians) and,

- the software, standards, and procedures created by, or used by, members of the profession.

How Can Professionalization Be Accomplished

We have previously discussed a number of things that will be necessary for professionalization. To assure that the process itself is as effective as possible, we will have to consider the following:

- *There must be some form of quality assurance for the entire process.* Transcripts will have to be verified. The quality and appropriateness of any written tests will have to be monitored.

- *Someone will have to track and analyze the results of the professionalization process.* For example, certified professionals will have to be interviewed to identify weaknesses in the system.

- *Industry, academia, and the government must be constantly polled for constructive feedback.*

- *The process must be updated in a regular and timely manner.*

The Impact of Professionalization on the Ada Education Process

The first large impact of the professionalization process will probably be in the area of Ada education. Why? The thrust of Ada technology is not the Ada language itself, rather it is the overall improvement of the handling of software throughout its life-cycle. An examination of the STARS effort shows that a large part of that effort is focused on the improvement of human resources.

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

Although Ada educators have been aware that Ada had something to do with software engineering, most have given token attention to the topic, e.g., they mentioned the terms "abstraction" and "information hiding" frequently during their courses, but failed to address topics like software quality assurance, testing, design methodologies, and software engineering metrics. *Professionalization will undoubtedly require an increased emphasis on software engineering, mathematics, and computer science in Ada curricula.*

One of the major mistakes made by Ada educators is the assumption that software engineering "will be taught in a separate course immediately prior to (or following) the 'Ada course'." Professionalization will require that software engineering (along with ethics, standards, and mathematics) permeate the entire Ada curriculum. *This will have a definite impact on the selection of instructors, and students, for these courses.*

Instructors will have to exhibit some qualifications in addition to a knowledge of the syntax of the Ada language. Indeed, the qualifications of an "Ada technology instructor" will be have to be quite varied. Probably the least important part of the instructor's qualifications will be the knowledge of Ada syntax. Further, these instructors will have to go through some sort of certification process before they are allowed to teach.

The students in a professional-oriented Ada curriculum will find that they must meet some entrance criteria before being admitted. In addition, they will most likely be graded during the course of their training, and may fail, i.e., not be given credit, even though they attended the training.

The Re-Certification Issue

Re-certification does not mean giving the same test over again. Neither does it mean giving a "slight" variation on the same test to an individual who has previously taken, and passed, an earlier version of the test. Re-certification involves two broad areas: the recognition that software technology is extremely dynamic, and that a software professional may wish to advance along any one of a number of career paths.

It has been said that, in 1963 the technical knowledge of the human race was doubling every *ten* years. In 1983, someone observed that the technical knowledge of the human race seemed to be doubling every *four* years. There is little doubt that our knowledge of software technology is increasing at a faster pace than technology in general. (Ironically, most software practicioners seem to be "stuck" somewhere in the 1960s in terms of the way they deal with software.) Therefore, like other professionals, software professionals will have to demonstrate that they are aware of the significant current trends in their industry. Further, they will have to demonstrate proficiency in some of this new technology, i.e., that which is directly related to their immediate job.

This can be accomplished in a number of ways. Software professionals will, of course, be required to take and pass a number of continuing education courses on a yearly basis. There will also have to be some way that they can successfully demonstrate proof of performance at their jobs. (As yet we have few metrics for this, e.g., management and peer evaluation.) An actual re-certification test will be required on a regular basis (possibly yearly, or every eightteen months).

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

Like any other professional, software professionals may wish to advance along a given career path, or change paths. Here one of the advantages of professionalization becomes obvious. The requirements for advancement along a given path, or for changing career paths will be well-defined (at least much better defined than they currently are). In the event that a software professional wishes to change career paths, the re-certification process will require continuing education and a re-certification test as before. However, the demonstration of proof of performance may now encompass an apprentice period in the area of the new career path.

The Difficulties Involved In Professionalization

Even those who are favorably disposed towards professionalization will admit that a number of barriers to the process exist. It is important to realize, however, that most, if not all of these barriers can be successfully overcome. Here are a few of the more interesting problems we can expect to encounter:

- *Deciding on methods, procedures, and metrics* will be one of the first obstacles we will encounter. The important thing to remember here is that there is no guarantee that we will recognize the best methods, procedures, and metrics when we see them. This means we will have to pick some "good-looking" trial examples and be prepared to change them.

- *The cost and logistics of professionalization* will be staggering. Yet even these costs will pale in comparison to the costs of ignoring the need for professionalization, e.g., the cost of malpractice suits and insurance.

- *The development of meaningful tests* for certification will be complicated by the number of areas for which they will be needed, i.e., we will have to develop separate tests for software engineers, computer scientists, and technical managers.

- *Defining and determining success* will be one of the biggest problems. Fortunately, we will probably have to define success as part of the justification process for professionalization. This means that this problem will be considered early (and solved early).

- *Extreme initial resistance from those currently involved with software* , i.e., current programmers and managers, will be a major (if not *the* major) difficulty we can expect to encounter. However, the impact of this difficulty will be lessened by two facts: most programmers and managers want to do the best possible job, and programmers are very goal oriented. Don't forget the advantages we listed earlier.

- *The interaction of certified professionals with non-certified management* will prove to be an interesting problem. For example, what recourse does a certified professional have when he or she is instructed to do something which violates existing professional standards or ethics?

- *Maintaining a high level of quality throughout the entire professionalization process* is yet another item. This will be a classical "who will watch the watchers" problem. It can be solved by appointing an independent group of quality assurance people at the begining of the process, and giving them the authority needed to accomplish their mission.

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

Alternatives to Professionalization

One of the more obvious questions is: "What happens if we ignore the issue of professionalization? Will it just go away?" Unfortunately the answer is no. In addition to not realizing all of the benefits listed earlier in this article, there are two main problems we will have to deal with: the very real possibility of government regulation of the software industry, and an increased impetus for the automation of the software process.

One of the main reasons the AICPA was founded was the realization that the federal government was seriously considering the regulation of the accounting profession. By accomplishing formal professionalization (via the AICPA) accountants realized two goals. First, the accounting profession was able to provide its own regulation, instead of being regulated by those with little knowledge of their industry. Second, whenever municipal, state, or federal governments must develop laws which directly or indirectly affect the accounting profession, they consult the AICPA. *It is not uncommon to see AICPA rules and regulations quoted directly in laws governing the accounting industry.* It would not be unusual to assume that the software industry could accomplish the same goals via its own professionalization.

The forces that shape technology are seldom technological. They are more often political, economic, or sociological. If the software industry generally refuses to advance itself through professionalization, the public may react by placing an increasing emphasis on automating as much of the software process as possible, e.g., the increased use of fourth generation languages and off-the-shelf software. This can, and will, mean a direct loss of jobs in the software industry. (This will occur anyway. However, we do not necessarily wish to accelerate the process.)

Recommendations

What recommendations are to be made? The following will serve as a starting set of recommendations:

- *We must begin at once with positive results to be visible within two years*. Specifically, the "average programmer" and the "average manager" should be affected by the professionalization process before the end of 1987.
- *We must solicit input from many sources*. Included must be programmers, analysts, managers, educators, government, professional societies, and other, more established professions.
- *The process must be publicized and highly visible.*
- *A professionalization maintenance committee must be established.* The job of this committee will include tracking changes to the professionalization process, introducing these changes in an orderly manner, and acting as a "supreme court" for any professionalization matter disputes.
- *Encourage the establishment of software engineering curricula on an undergraduate level.* Further, encourage the concept of professionalism on all forms of software education.

5301 Spectrum Drive
Frederick, Maryland 21701 (301) 695 - 6960

OMIT

NASA Training Programs for Ada

Bob MacDonald
NASA Lyndon B. Johnson Space Center

This presentation will provide a report on the status of the NASA training for the Ada Programming Language.

The Impact of Common APSE Interface Set Specifications on Space Station Information Systems.

by
Jorge L. Diaz-Herrera and Edgar H. Sibley
George Mason University, Fairfax, VA

ABSTRACT

Certain types of software facilities are needed in a Space Station Information Systems Environment; the Common APSE Interface Set (CAIS) has been proposed as a means of satisfying them. This paper discusses how reasonable this may be by examining the current CAIS, considering the changes due to the latest Requirements and Criteria (RAC) document, and postulating the effects on the new CAIS 2.0. Finally a few additional comments are made on the problems inherent in the Ada (*) language itself, especially on its deficiencies when used for implementing large distributed processing and database applications.

1. INTRODUCTION

Certain types of software facilities are needed in a Space Station Information System Environment (SSISE). Not the least of these are:

a. the distribution of the target and host facilities for both the run-time and development environment,
b. the absolute need for good configuration management methodology to control the development and use of the many versions of the software and tools,
c. the need to develop and modify systems within distributed environments using sophisticated terminal interfaces,
d. a consequent need for good interfaces and standards, abstract data typing in a distributed system (including development and run-time bindings),
e. a real-time distributed software development methodology, and corresponding language support and operating environment and tool constructs,
f. good human to human and machine to machine communication techniques.

---

* Ada is a Registered Trademark of the U.S. Government, Ada Joint Program Office

Because SSISE development will use Ada as its implementation language, it would be extremely unfortunate if its needs were not addressed in the Ada environments now under specification and development: the Common APSE Interface Set (CAIS). This paper is structured around the following three major aspects:

1. How well are these needs addressed within the current CAIS specification? Indeed, would a poor fit have a bad effect on the Space Station software?
2. What improvement can be expected due to changes mandated by the latest Requirements and Criteria (RAC) document?
3. Will this truly affect the next CAIS (version 2.0)?

2. SPACE STATION INFORMATION SYSTEMS ENVIRONMENT NEEDS AND THE CAIS

The Space Station Software Working Group and NASA software specialists have recently defined their needs for support of space station software development [Dixon 85], and produced a definition of the space station software support environment requirements [Chevers 86] in early 1986. The major issues include aspects about generic elements of the environment, tool characteristics and consideration of the following major questions:

- Should a uniform NASA Software Development Environment for space station be defined and developed? Issues relative to this include:
  * Software development for the space station will be highly distributed, with no localized single development group.
  * Major software portions will be managed by various centers and not by a single NASA center.
  * Important functional differences exist between major software systems; these need completely separate software environments.
- How much of the space station software development environment should be furnished by NASA?
  * This had a major impact because NASA has never developed its own SDE.

2.1 THE SSISE AND ITS REQUIREMENTS

Despite the fact that the specification of a single standard environment may involve solving many problems, the working group felt that the potential advantages far outweigh the difficulties. There was therefore a recommendation for the definition of a well-defined development environments with capability for two classes of user:

- SDE interfaces to support software developers and their managers. These were to consist of:
  * Mail and Telecommunication support (e.g., editors, file systems, communications aids, etc.)
  * Technical management/control aids (e.g., cost models, project management and planning systems)
  * Data base support (e.g., file management, retrieval, control, etc.)
  * Modeling/simulation aids (e.g., Architecture models, testing aids, etc.)
  * Prototyping aids (e.g, requirements, specs, man/machine interface, etc.)
  * Document preparation aids
  * Requirements specification validation and analysis aids
  * Design specification aids (e.g, PDL analyzers, data dictionary, etc.)
  * Code construction and control aids (compilers, linkers, configuration managers, etc.)
  * Program analysis/testing and integration aids (path coverage/test generators, symbolic executors, etc.)
  * Metrics (quality, complexity, cost and reliability measures)
  * Man-machine interface support (interface and use of the environment, help, tutorial, etc.)
- An SDE interface to support NASA software managers responsible for software requirements/acquisition/acceptance; this required essentially the same capabilities as those above, with changes in emphasis or tailoring the relative importance, complexity of function and response needs. Thus the management controls should be more heavily directed toward schedules, planning, project management, and PERT, while the modeling, prototyping, and simulation aids would be minimal or unnecessary.

These two interfaces can thus result from a single CONFIGURABLE environment which is tailored to the specific needs of each work station and locale.

2.2 THE CURRENT CAIS

Several needs in the above list have not been addressed in the CAIS 1.4 specification. These issues have been discussed at length in KIT (KAPSE Interface Team) and KITIA (KIT Industry and Academia Support) group meetings but are, as yet, only partially resolved. In fact, many of these were deliberately excluded from discussion in the current CAIS document. They are:

* A particular Configuration Management Methodology
* Sophisticated Device Control and Resource Management Capabilities
* Distributed Environments
* Inter-tool Interfaces
* Interoperability
* Typing Methodology
* Archiving

These and other issues are each discussed in the detailed sections below.

2.3 THE EFFECT OF THE RAC

Although the requirements of the first version of the CAIS were never explicitly defined, they were a mixture of the specification and partial implementation of the ALS provided by Softech and the AIE under design by Intermetrics. Thus, because these two efforts were already funded, they introduced several problems because the CAIS specification team were attempting to provide as much compatibility as possible with these two, somewhat different, architectures of an environment (with differences also in their scope). In general this attempt may have introduced problems of upward compatibility. Thus the future CAIS will either have to ignore the normal needs of a "standard" in dealing with a required "upward compatibility" or else admit to serious deficiencies and possible poor interfaces in future systems due to lack of adequate controls and functions.

The new requirements were written to allow more flexibility and better interfaces, with an attempt to have better functionality. Thus the Entity Management Support (section 4. of the RAC) requires a support that parodies the description of a normal database management system without specifically saying that it is needed. Some of the needs are quite specific and (though open to interpretation) quite encompassing; e.g., "impose a lattice structure on the types which includes inheritance of attributes, attribute value ranges (possibly restricted), relationships and allowed operations."

Another type of problem arises due to wish to allow the CAIS to be operable on almost any current commercial and experimental operating system: viz, "The CAIS specification shall be machine independent and implementation independent. The CAIS shall be implementable on bare machines and on machines with any of a

variety of operating systems." This could restrict the design in many unfortunate ways.

2.4 THE NEXT CAIS

It is difficult to peer into the future, and thus the following predictions for the next CAIS may prove incorrect, however, the degree of effort and choice of contractor (Softech) allows us to make some early assumptions.

First, it seems unlikely that the contractor would make a new specification that would not allow the current ALS to be considered an "almost complier with" or "minimal fix away from" the new CAIS.

Second, the level of funding and staffing is not one that would be expected to allow anything but the narrowest extension of the current CAIS.

Third, it is somewhat doubtful whether the politics of the situation would allow a large diversion from the Army's ALS.

Fourth, the contractor has already suggested that divergence from some of the old CAIS specifications to go to the RAC statements would be difficult. The discussion of such issues at recent KIT/KITIA meetings has not been encouraging to a feeling of extension of the role of the CAIS.

3. SPECIFIC DEFICIENCIES

3.1 CONFIGURATION MANAGEMENT

The lack of a particular Configuration Management Methodology means that several vendors could provide incompatible but "standard" systems. These issues seem, primarily, to devolve on a need for a long time naming continuity and, in general, software configuration management. The first issue is that of providing "Unique Names" across geographic and time boundaries. The term "unique name" (UN) has been used to define an immutable name for an entity; e.g., a compiler should be uniquely identified by a UN, which neither changes nor is "recycled". Thus a UN is given out once to an entity and remains its name; if the entity is deleted/removed, then the UN will still identify the

entity, but an attempt to retrieve it will result in a statement that it is no longer available.

There are two possible problems:

1. Is any sort of change allowed to an entity without its UN changing?
   Normally, the contents of the entity may be altered, but this could mean that it is no longer even similar to it previous "parent" entity. Certainly, it seems reasonable that a program may be debugged without changing its name for each error detected. This would suggest that the unique name was really a run-time UN, which could be said to remain constant during programming and debugging. However, if the UN were for a data entity, the effect of a change in any one of its values would be a new "version" of the entity, and this could be important enough to be considered a new "entity" though the normal way of dealing with this is to consider the data entity to be "time and data stamped" with an audit trail to allow the previous entity to be reconstructed (e.g., for roll back).
2. How are UN related for the same (but changed) entity?
   There must be a method for data entity reconstruction -- roll back from an audit trail, however, the data in a traditional database must not be called by physical location, but by "name pointers" or indexes or "logical" keys -- these might be considered the UN for data. On the other hand, the only "audit trail" for programs is normally provided by the configuration management system (CMS). In fact, the idea of version in a CMS is another way of looking at the unique name; i.e., the UN is logically equivalent to a user name concatenated with the version number (or equivalent).

What has been suggested above about a UN for both program and data could also hold for control structures.

3.2 SOPHISTICATED DEVICE CONTROL

Some of the biggest problem are undoubtedly going to be the introduction of more sophisticated input/output and other special device dependent interfaces (e.g., for a mouse). This will be a problem when there are unusual but sophisticated interfaces to devices and sensors. Unfortunately, this issue will require too much discussion to fit here and requires a paper of its own.

3.3 DISTRIBUTED ENVIRONMENTS

The development of Space Station Information Systems is bound to be highly distributed with no single group solely responsible for the required software systems. This could result in difficulties when looking at large and complex development and run-time environments. Discussions on space station software development must address Distributed Environments (Host and Target) and particular ways to distribute data as well as control. The Ada Programming Support Environment (APSE), however, does recognize such a need, and states that additional software tools are necessary in order to allow "independent" programs to communicate with each other dynamically, in a "natural" and controlled way. The RAC states, however, that: "CAIS program execution facilities shall be designed to require no additional functionality in the Ada Run-Time System (RTS) from that provided by Ada semantics. Consequently, the implementation of the Ada RTS shall be independent of the CAIS"...

There are some problems here with Ada itself. A distributed system can be designed and implemented in Ada from two different points of view, namely as a single program or as a collection of cooperating programs. The first of these alternatives, single program, is particularly useful when considering tightly coupled multiprocessor systems. Inter-processor communication and synchronization can then be naturally achieved by using rendezvous. The second alternative is to design the system as several independent programs (one per processor). The Ada language, however, does not support the idea of independent programs dynamically cooperating with one another (i.e., no constructs are provided for inter-program communication).

Both approaches require further support from the environment. For example, Specific target-oriented tools (e.g., loaders) are needed, to assist in the actual implementation on the distributed architecture. An Ada solution to these problems may be in the form of a set of inter-program communication primitives provided at the APSE level in the program library. In general, the design and implementation of a multi-processing system as a collection of independent programs present a number of inconveniences resulting in the following issues:

- Creation of "linguistic" facilities to enable interprogram communication
- Provision of a methodology for designing Distributed Systems using these higher-level primitives.

3.4 INTER-TOOL INTERFACES, INTEROPERABILITY, AND TYPING METHODOLOGY

These three issues represent the generic problem of the tool builder. When several tools must interchange data, they must either do it via the standard interfaces or else be designed as a suite of tools with total knowledge of the data requirements and functionality of the other tools in the suite. In general, there are problems in defining inter-tool interfaces, because a change to one tool may cause a ripple effect. However, reliance on interoperability interfaces entails passage of abstract data types across tool interfaces. This could have serious security and integrity repercussions.

Interoperability also has severe impact on distributed systems, where the passage of abstract types may be essential for accurate and reliable data interchange between the various nodes. Without a good typing methodology, it is obviously impossible to provide such features or to deal effectively with data base management and similar issues. The alternative to such methodology is of course straight ASCII interchange, with negligable checking. Again, these topics deserve a paper of their own.

3.5 ARCHIVING

This is an important issue in any configuration, but more so in a distributed environment of the kind mentioned here. However, for the purposes of this paper, it will be left as another undiscussed issue.

3.6 CENTRALIZATION AND DECENTRALIZATION ISSUES

The really tough problems of unique names of any of the types of entities occurs when they are (in some way or another) decentralized. As an example, when a compiler is moved to a new node, does its UN change? And whether it does or not, which node controls or restricts the change? Obviously, the answer to such questions involves policy and method of control. It is important that the controlled use of a distributed environment be effected through distributed kernels operating locally. It is conceivable that one or more nodes would be designated as decision making kernel(s), while other nodes will be merely servers. This seems to provide a reasonable compromise between centralized (high communication costs and high vulnerability) and decentralized (with its unnecessary control burden on every node).

4. ADA LANGUAGE ISSUES

As discussed earlier, there are some severe problems in using Ada in multiprocessing and distributed systems. From Ada's point of view, a multiprocessor system which uses a common memory can be viewed as a "uniprocessor system which implements multitasking in a more efficient manner." In this case, the entire system is designed and built as a single Ada program with certain procedural abstractions implemented as tasks. Each of these tasks represents the work of one logical processor, and may eventually run on a dedicated physical processor. Inter-processor communication and synchronization can then be naturally achieved by using rendezvous. However, before the program is run on the target multi-processor environment, the different tasks need to be "assigned" to their corresponding processors. And this is not explicitly supported by the language. The use of PRAGMAS has been suggested here. On the other hand, a distributed system may be supported by Ada as a collection of Ada programs communicating through intermediaries. One way would be to provide library packages to maintain "mailboxes" and whose "procedures" (which could be implemented as tasks) can be called from several programs. In any case a standard protocol is needed.

An Ada solution to these problems may be in the form of a set of inter-program communication primitives provided at the APSE level in the program library. Basically, what we are talking about here is a general facility by which programs can communicate and synchronize their activities. These facilities must be designed in such a way that they could be applied in a number of situations using different programs. Thus, the specification must be general enough as to hide the identification of the programs involved, and yet provide ways to identify a particular situation. Ada's generic units provide the answer. They are general at the definition level, and particular at the instantiation level.

Unfortunately, the use of generics here presents a number of inconveniences since the identity of the actual programs using the tools is not known at the time of writing the tool, these tools cannot be tasks themselves. The Ada tasking model defines an asymmetric inter-task communication mechanism in which the identity of the callee must be known to the caller. In other words, to have true library tasks (where the identity of the callers/callees is not revealed),

we need to introduce extra programs. For example, if we want to connect two library programs and run them in parallel, we have to do so through a third intermediary program. This is feasible because the identity of this third program is known to the other two. The fact that these units run in parallel is an implementation decision, which is best hidden inside the unit body (an added benefit).

The first alternative seems more effective, since we could use the full power of the language at compile time (e.g., type checking) and at run-time (at least the system can be tested on a uniprocessor environment), and it does not require any "special" features from the programming language (in fact, most available implementations will not even support multi-processor targets directly). The second alternative, however, may be more convenient and elegant, reflecting the real world situation (i.e., independent parallel programs each running on its own CPU), but requires a well-defined STANDARD distributed systems methodology.

5. CONCLUSIONS

Accommodating heterogeneity in a software development environment requires that the system be written for a number of different machines and be able to support numerous software packages associated with various operating and run-time systems. It is postulated that control of such system must be effected through **distributed kernels** operating on a local basis. The run-time system is best organized following the layered model provided that we are able to highlight:

- the relationship between the distributed and local operating systems
- the relationship between the different types of decisions made by the juxtaposition of the two control domains (i.e., local and global)
- the visibility necessary to effect the various implementation issues

Obviously, the APSE approach is the way to go, but perhaps it will need to be modified to resolve distributed computing issues such as:

- network transparency at the user level
- interprogram (internode) communication mechanism
- exception handling mechanisms encompassing distributed characteristics
- awareness of application objectives

- fault tolerance strategy over the placement and updates of back-up copies of information

What we need here therefore is an extra layer, the DAPSE, in between the MAPSE and KAPSE. This will provide a standard interface for such a system support environment.

6. REFERENCES

Chevers,E. "NASA Space Station Software Requirements" (JSP, Jan. 1986)

Dixon "Open Forum on Space Station Software Issues" (NASA, Jonhson Space Center Houston, Texas, Feb. 1985)

KIT/KITIA "DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS) September 1985

KIT/KITIA "Military Standard Common APSE Interface Set (CAIS) Version 1.4 October 1984.

7. ACKNOWLEDGEMENT

The authors would like to thank Dr. Ann E. Reedy of Planning Research Corporation for discussion of many of the Standards and Unique Naming issues.

A Risk Management Approach to CAIS Development

Hal Hart
Judy Kerner
Tony Alden
Frank Belz
Frank Tadman

TRW Defense Systems Group
Redondo Beach, California 90278

ABSTRACT

The proposed DoD standard Common APSE Interface Set (CAIS) has been developed as a framework set of interfaces that will support the transportability and interoperability of tools in the support environments of the future. While the current CAIS version is a promising start toward fulfilling those goals and current prototypes provide adequate testbeds for investigations in support of completing specifications for a full CAIS, there are many reasons why the proposed CAIS might fail to become a usable product and the foundation of next-generation (1990's) project support environments such as NASA's Space Station software support environment. The most critical threats to the viability and acceptance of the CAIS include performance issues (especially in piggybacked implementations), transportability, and security requirements . To make the situation worse, the solution to some of these threats appears to be at conflict with the solutions to others.

TRW's CAIS development is a risk-managed approach planned to gather information early about critical threats, and, based on that information, to identify and pursue risk-reduction development approaches. This is an application of Barry Boehm's "Spiral Model" of the software development process, which integrates risk management into a generalization of systems development processes. Risk-managed approaches typically include prototyping to expedite acquisition of information in critical risk areas. TRW's initial assessment of risks led to a comprehensive design phase for the prototype before

coding based on two principal reasons:

1. the necessity to avoid a "narrow" prototype that accomplished some objectives while impeding others (or at least to reduce such conflicts in the initial implementation and to reduce and assess costs in expanding the prototype to serve broad risk-reduction objectives), and

2. incomplete information about how to accomplish that in a prototype (or even what the threats really were and hence what the objectives should be).

This prototype design phase was the first traversal of the Spiral Model. The near-term benefit of this approach is to direct initial prototyping activities toward areas with highest payoff in risk-reduction information while retaining compatibility with pursuance of other areas. The ultimate payoff of the TRW approach will not be in rapidness of prototype simulation of the initial CAIS, but in gathering information for specification and implementation of a viable 1990's CAIS (and perhaps even putting the CAIS prototype on the direct evolutionary path toward such a production-quality implementation).

Following are some of the risk-reduction directions determined by the TRW CAIS prototype design activities:

- **Performance:** a key fact is that the CAIS is more complex than typical 1980's operating systems, offering direct tool and user support in many areas not well (or directly) supported in most operating systems (e.g., configuration management support, inter-program communication and synchronization, access control, etc.). Early intense effort is needed in such key areas to develop efficient algorithms and/or architectures in these not-so-well-supported areas. Simulation has been identified as a time-saving approach to assess performance of newly developed CAIS algorithms or architectures without the complete expense of tool building or porting (and sometimes without completely implementing the CAIS algorithms). Additionally, the tough goal of piggybacked implementations (atop existing

operating systems) is aggravated by CAIS portability concerns

- **Transportability of CAIS Implementations:** the TRW CAIS design is based on a mapping of CAIS functionality directly to a machine-independent underlying model called the "tool portability layer". This means that most of the CAIS functionality can be implemented without regard to the underlying host. This approach isolates into the "inner portability layer" of the CAIS those functions that are most host-dependent. This ties in with the goal of efficiency by allowing development of host-dependent optimizations in the inner layer, and host-independent higher-level optimizations in the outer tool portability layer.

- **Security:** due to the time and expense of developing a certifiably secure CAIS (as on a bare machine), TRW's initial efforts will be investigations into using components from TRW's Army Secure Operating System (ASOS) project (scoped for A1) as a Trusted Computing Base upon which to implement the inner portability layer. This looks like a promising compromise between development costs of secure systems, and CAIS transportability and performance goals (because of reuse of the tool portability implementation layer and its optimizations).

As demonstrated in the list above, a risk-managed approach can find development strategies which simultaneously work toward solutions of the multiple critical threats to CAIS viability. A prototype implementation approach incorporating these is ongoing now, with a basic subset of the CAIS now implemented. Progress will be reviewed against the risk list later this year, at which time risks may be re-assessed, new alternative approaches hypothesized, and new directions selected based on information acquired in this phase of prototyping. This prototyping, risk re-assessment, and replanning will constitute another traversal of the Spiral Model.

Extending the Granularity of Representation and Control for the MIL-STD CAIS 1.0 Node Model

Kathy L. Rogers
Rockwell International
Space Station Systems Division

Introduction

The Common APSE (Ada[1] Program Support Environment) Interface Set (CAIS) [DoD85] node model provides an excellent baseline for interfaces in a single-host development environment (see Figure 1). To encompass the entire spectrum of computing, however, the CAIS model should be extended in four areas. It should provide the interface between the engineering workstation and the host system throughout the entire lifecycle of the system. It should provide a basis for communication and integration functions needed by distributed host environments. It should provide common interfaces for communication mechanisms to and among target processors. It should provide facilities for integration, validation, and verification of test beds extending to distributed systems on geographically separate processors with heterogeneous instruction set architectures (ISAs). This paper proposes additions to the PROCESS NODE model to extend the CAIS into these four areas.[2]

Rationale

The intent of the CAIS is to promote transportability and interoperability. The user interface should provide the same view of the system for a remote workstation connected through a network as for a directly connected terminal. Accessibility and finer granularity of the PROCESS NODE and QUEUE file information could provide processor performance measures during the design phase of the project, debugging information during the coding phase, and assessments of hardware and software changes during the

---

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

[2] It is the intent of this paper to discuss some of the topics which were explicitly deferred in MIL-STD CAIS 1.0.

Extending the Granularity of Representation
and Control for the MIL-STD CAIS 1.0 Node Model

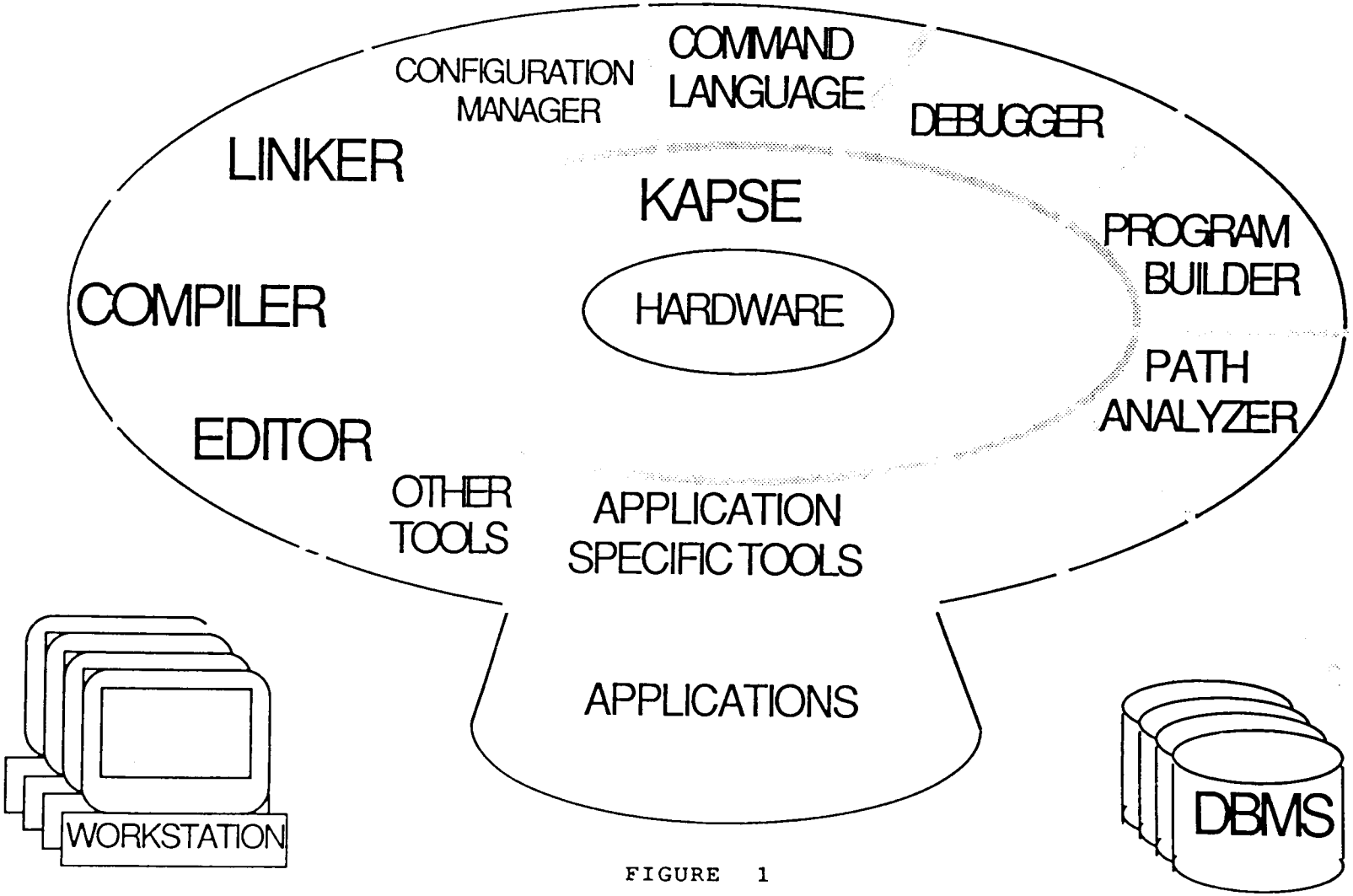


FIGURE 1

maintenance phase.

CAIS-provided code and data sharing could provide services (and entities) in a cost-effective manner to more than one application or user. To implement sharing, the node model must be able to manage data for dictionary driven processes, maintain version and revision information for library units, and provide security to maintain the integrity of the system. Data management requires information such as location, format, and access control of sharable resources. It might also extend to "knowledge" regarding the use of data, so that data may be relocated to facilitate convenient access. PROCESS NODES should be able to take advantage of code (such as common packages) that can be shared.[3]

The PROCESS NODE model should accommodate the communications necessary in a distributed environment. Five types of communication interfaces should be added to the current model: communication between parts of a process executing on separate processors, between processors (extending to processors with different ISAs), between the CAIS and the PROCESS (in both the host and the target environment), between different CAIS implementations, and among PROCESS NODES. In order to satisfy the Ada Language Reference Manual [LRM83] requirement that "several physical processors (may) implement a single logical processor"[4] effective information interchange is vital. Information must be communicated in an understandable format between heterogeneous ISAs. "Hooks" should be established so that individual elements of a test bed, as well as the integrated test bed, can be monitored. The CAIS should be extended to interact with other CAIS

---

[3] Multiple copies of packages, such as TEXT_IO, would be eliminated in favor of all processors at a site accessing the same copy. In a heterogeneous distributed environment, this can extend to shared copies of SYSTEM packages and STANDARD packages, if a common data representation scheme is used.

[4] Ada Language Reference Manual, Chapter 9, paragraph 5.

implementations.[5] When processes executing under the auspices of two different CAIS implementations interact and require CAIS services, a standard method should be used to determine which CAIS should be called. Interfacing to communication mechanisms, especially in a geographically separate system, is an important aspect of the CAIS.

Annotations for "non-functional"[6] directives could be handled by the PROCESS NODE model. These directives include desired degree of fault-tolerance, scheduling priority, desired level of status information, recovery processes, performance measures, special hardware requirements, and/or amount (and detail) of information to be promoted. Fault tolerance could be supported to ensure that sufficient resources are utilized to maintain the level of integrity required by the process. Scheduling of processes according to priorities should be considered; algorithms for serving processes according to their priorities could be provided in a straightforward manner. Directives stating the granularity of information required for a PROCESS (which determines the amount of overhead incurred) should be flexible.[7] Directives should also provide error recovery and rollback to the last "safe" state at a level of overhead which is appropriate for the PROCESS. Performance measures should be provided, especially for "time critical" processes which may need to be routed to a processor based on the speed and level of services available. The need to know the execution efficiency of processes on target processors is a major reason the CAIS services should be available in the target environment. In some configurations,

---

[5] Oberndorf, Patricia, Prototyping CAIS [Obern86].

[6] "Non-functional" is used here to denote constraints on functionality beyond those which are explicitly written into the code.

[7] For example, information pertaining to the current/last instruction or procedure executing might be requested. In the same way, the status of entities ranging from register values to values of user variables might also be requested.

security will b^ important; security directives should be provided and enforceable [LeGra86]. The CAIS can be continually extended by providing additional handlers to accommodate future "non-functional" directives.

The PROCESS NODE model should include capabilities to query and to negotiate with other nodes. Negotiation may be required in the case of a remote procedure (subprogram) call where the size of the parameters exceeds the capabilities of the receiving processor. Query and negotiation procedures could detect this problem and establish a piecewise transmission of data. Processes executing on processors with different ISAs could negotiate a standard data format for transmitting data. Query capabilities are vital for processes which have very specific processor needs. Query and negotiation capabilities should be provided to determine the optimal processor configuration to execute a process. Library management, in a system containing heterogeneous ISAs and specialized processors, creates demand for information such as version/revision, intended ISA, special processing needs or priorities, and other required support.[8] Check out, with locking mechanisms, must be maintained for library units. Security for the items being managed is also a concern. The level of access required to read or update information must be established, including altering access requirements after updates. Creation and maintenance of multiple copies must be addressed with respect to update[9] procedures.

Recommendations

The current CAIS node model should be enhanced in four ways. First, the PROCESS NODE state information should be more descriptive. Second, there should be a PROCESS NODE representation of the status of each

---

[8] Other support may include speed, space, and/or security requirements, etc.

[9] Update is being used here to encompass all modification functions, addition, modification, deletion, etc.

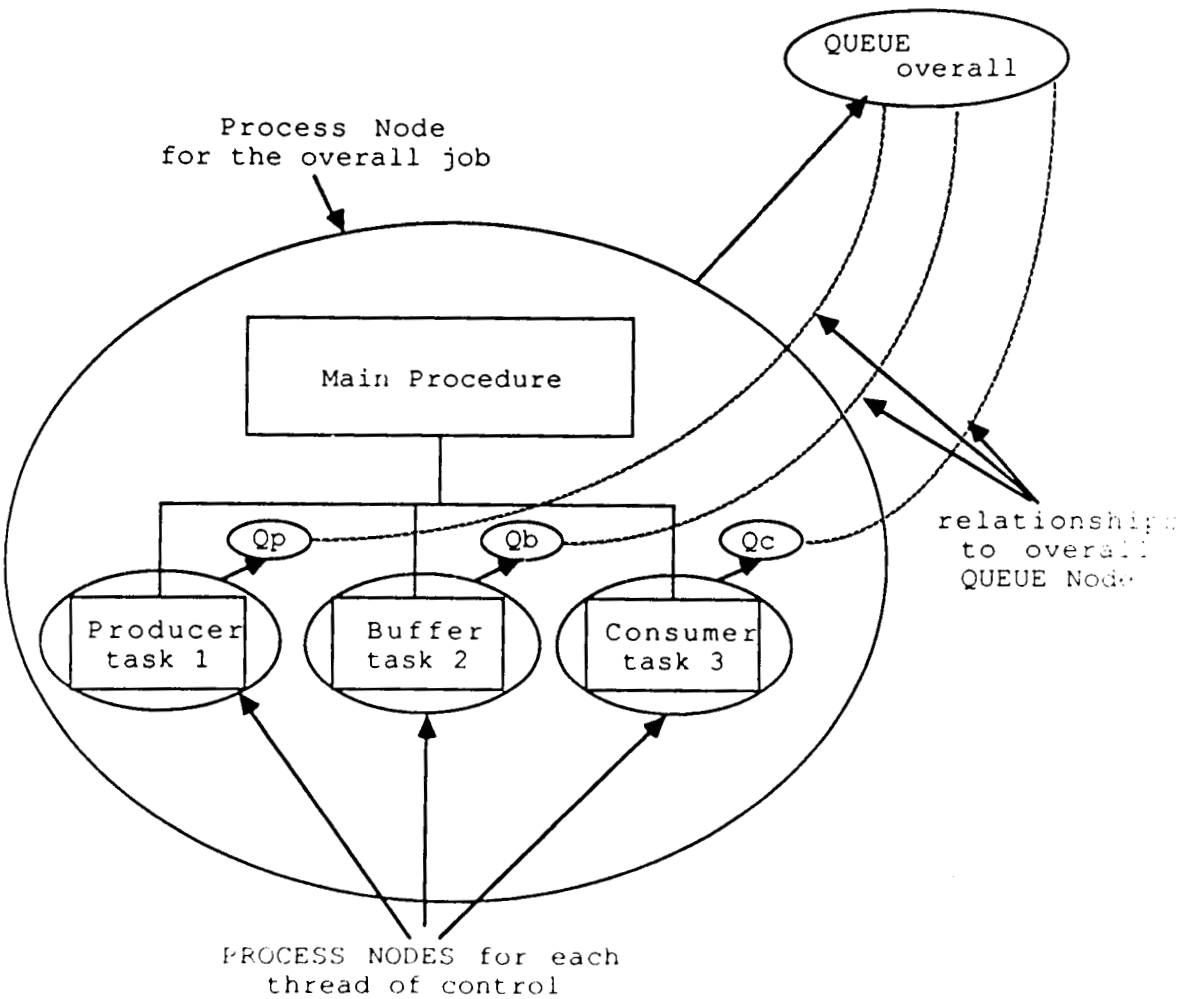


FIGURE 2
Snapshot of four PROCESS NODES

thread of control extending to any level of decomposition, and a QUEUE associated with each PROCESS NODE. Third, the QUEUE NODE[10] should be able to provide accessible status measures beyond those which are "hard coded" into the process. Finally, the QUEUE NODE model should provide capabilities to act on the information received.

As an example of the implications of the above recommendations, consider a PROCESS that spawns three subordinate tasks: a producer, a buffer, and a consumer. Figure 2 is a snapshot of the four PROCESS NODES; it represents the state of each thread of control currently executing on behalf of the "main" process. The overall job, as well as each subordinate task is depicted as a PROCESS NODE, with an associated QUEUE NODE. Each PROCESS NODE has several predefined attributes including: CURRENT_STATUS, PARAMETERS, and RESULTS. Other information, such as the logical name of the site where the process is executing, may also be available. Each QUEUE NODE representing one of the subordinate tasks has a relationship to the QUEUE NODE associated with the PROCESS NODE for the overall job. Note that when the subordinate tasks terminate, their respective PROCESS and QUEUE NODES cease to exist.

In order to augment the PROCESS NODE, the process states should consist of "meta-states" as well as "micro-states". In addition to the current "meta-states" READY, SUSPENDED, ABORTED, and TERMINATED, a new meta-state, RUNNING, should be added. The meta-states should also have micro-states to provide additional information. The READY meta-state should include the micro-states WAITING (for resources), COMPLETE (but not terminated), and BLOCKED (awaiting rendezvous). The TERMINATED meta-state should include the micro-states NORMAL and ABNORMAL.

To increase the granularity of the PROCESS model, the PROCESS NODE, which represents the overall job should also provide PROCESS NODES for each "thread of control". That is, a PROCESS NODE should be associated with every body of a subprogram, task, or package in a state of execution. All PROCESS NODES should be of the

---

[10] The term QUEUE NODE is used (rather than QUEUE FILE) in order to describe the QUEUE as an entity.

same form (complete with proposed extensions).[11] The PROCESS NODE for each thread of control should have an associated QUEUE NODE. Information from QUEUE NODES should be promotable upward to the QUEUE NODE representing the next higher level of decomposition, based on the amount of information required by the higher level PROCESS/QUEUE pair. In this way, the current CAIS PROCESS NODE is maintained on the job level, but is also decomposable to provide more specific information when needed.

Status information provided to the QUEUE[12] should be usable by other processes. In the current model, data, procedures, or tasks in one process cannot be directly referenced from another process.[13] QUEUE files are currently used as holders of PROCESS information.[14] The level of detail for status messages and the amount of overhead incurred, should be able to be specified. Other specifiable information includes the amount of information that should be promoted from a QUEUE NODE at any level to a QUEUE NODE related to a PROCESS at a higher level. Extensibility of the QUEUE NODE model can be provided by viewing the node as a database which can be queried by applications (or engineers). Additional information could be added to the database in the future, which could be utilized by processes which are aware of the enhancements. Status information generated independent of the process (or processor) is necessary in a distributed system, in the event of process (or processor) failure.

The QUEUE should be more than a passive information receptacle. It should be capable of being used to initiate procedures, such as recovery upon

---

[11] The PROCESS NODES should extend to any level of decomposition necessary.

[12] CAIS Rationale and Criteria document.

[13] MIL-STD CAIS 1.0 p. 14.

[14] Three types of QUEUE files are defined. The QUEUE files can operate in SOLO (write append, destructive read), COPY (SOLO QUEUE with initial contents), and MIMIC (dependent upon another QUEUE file) modes.

used to initiate procedures, such as recovery upon detection of a fault in the system. Facilities such as those necessary to terminate processes which are not performing correctly could also be provided. Early warning regarding process failure (rather than fault detection upon request for service) provides the calling process with a potentially greater number of recovery possibilities. Action in the event of failing processes is essential in environments which require fault tolerance, especially in unattended systems or in those systems where life and property depend on continuous, correct functioning of hardware and software.

Conclusion

The potentially long lifetime and large number of host development environments and target processor configurations, using Ada, require a CAIS that promotes transportability, interoperability, communication, and extensibility. The CAIS should provide a constant view (at an appropriate level of detail) of the supporting hardware and the APSE tools. This view should be provided to an engineer at a workstation, as well as to a secure, fault-tolerant distributed process. The CAIS should be extended to provide query and negotiation capabilities among nodes. It should include mechanisms for handling "non-functional" directives (in order to address the spectrum of processing complexity). It should also accommodate sharing code and data, as well as communication interfaces. These enhancements are necessary to accommodate the potential changes that will occur throughout the lifecycle of Ada applications. Some extensions to the CAIS model are necessary. Recommendations include maintaining more descriptive PROCESS state information; viewing the current PROCESS NODE model as a description at the overall job level (and providing PROCESS nodes for subprograms, tasks, and packages, while they possess a thread of control); viewing the QUEUE as a resource NODE rather than a logging file, and enhancing the QUEUE NODE to make it responsive to processing requirements. The proposed extensions to the CAIS model maintain the job level view of the original CAIS design and enhance it by providing decomposition to a finer level of granularity.

Extending the Granularity of Representation and Control for the MIL-STD CAIS 1.0 Node Model

The author gratefully acknowledges the contributions of the Joint NASA-JSC/UH-CL APSE Beta Test Site Team members, especially the expertise and encouragement of the research team's technical director, Dr. Charles W. McKay.

References

[DoD85] Department of Defense, Military Standard Common APSE Interface Set (CAIS), 31 January 1985.

[LeGra86] LeGrand, Sue and Richard Thall, The CAIS 2 Project, Softech Incorporated, Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, June 1986.

[LRM83] Reference Manual for the Ada Programming Language MIL-STD 1815A, 1983.

[McKay84] McKay, Charles PhD., University of Houston at Clear Lake lecture notes, Fall 1984.

[Obern85] Oberndorf, Patricia, Prototyping CAIS, presented by Hal Hart at the Los Angeles SIGAda, February 1986.

[Roger86] Rogers, Patrick and Dr. Charles McKay, Distributing Program Entities in Ada, University of Houston at Clear Lake, High Technologies Laboratory, Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, June 1986.

Experience with the CAIS

Michael F. Tighe
Intermetrics, Inc.
Cambridge, Mass.

1. Overview

Intermetrics is currently using an earlier version of the CAIS (based on CAIS 1.2) in the implementation of it's line of Byron[1]/Ada[2] APSE products. This proto-CAIS provides all the Byron tools, Ada compiler, linker-driver, Ada program library manager, etcetera with a standard interface to the underlying operating system. Written in Ada and using Ada language features to separate specification from implementation, this proto-CAIS is currently implemented on four different operating systems, representing two different machine architectures including

- VAX/VMS (Digital Equipment Corporation)
- MVS/370 (IBM)
- CMS/370 (IBM)
- UTS 3.5 (Amdahl UNIX[3] derivative running under VM on the 370).

In progress is the task of moving the proto-CAIS (thus the Byron APSE) to the Sperry 1100 series (a third hardware class and fifth operating system).

Intermetrics is using this technology to permit the primary development team to proceed doing main-line development work on the Byron APSE, while *rehost* teams take either source or object modules (depending on the target hardware) and install the most recent version of the Byron APSE on these new machines for testing and

2. Ada is a registered trademark of the U.S. Government Ada Joint Program Office.
1. Byron is a registered trademark of Intermetrics, Inc.
3. UNIX is a trademark of Bell Laboratories.

demonstration to customers. This process allows the various rehost teams to follow the primary development team very closely, at times being only two or three weeks behind the primary team in terms of supported capability. Each rehost team continues work on developing contract- or host-specific features for the rehosted compiler.

2. Proto-CAIS Usage in the Byron/Ada Compiler

The proto-CAIS is the primary database used by the functions that implement the Ada Program Library functions of the Byron APSE. The Program Library contains various representations of the Ada program as the compiler translates it from text to object code. The retention of these representations in the Program Library is under user control but usually include

- an *abstract syntax tree* (AST),
- a **Diana** representation of the program,
- an internal form used for code generation (called **Bill**),
- and the linkable object module.

Each form of the program representation is kept for each smallest compilable unit of the language, as the programmer can present his source to the Ada compiler in any of a variety of sequences and portions. It is necessary to organize these representations in an orderly fashion which is related to the name of the entity that they represent. Additionally, there are inter-relationships between the representations. For instance, each specific object module is derived from a corresponding specific Bill representation which is derived from a corresponding specific Diana representation which in turn depends on its specific abstract syntax tree representation. Compilation dependencies of the **with** statements in the source are represented as dependencies between the corresponding Diana representations. Date/time of compile and other information is kept as well.

As the Ada Program Library is also a foundation for a distributed configuration management system, it organizes the users compilations into collections and catalogs. A *collection* is a set of catalogs, and represents a higher level of grouping than the catalog. A *catalog* is a set of Ada sources (and their subsequent representations) that represent a single unit of related work chosen by the programmer. Each compile is made in within the context of a *primary* catalog which refers to various *resource* catalogs. Catalogs can created at any time, or can be *derived* from an existing catalog to form a new (incremental) version of the original catalog. Additionally, resource catalogs can be specified as *interface* or *implementation* catalogs, corresponding to the concepts of Ada **spec** and **body**. Multiple catalogs can exist as an implementation catalogs for a single interface catalog. At link time, the programmer is allowed to choose a specific implementation catalog to match a specific interface catalog.

Catalogs are composed of *units* which represent the smallest unit of compilation.

Each unit is composed of a *spec* and a *body*, with the possible inclusion of a *subunit*. Each spec or body is composed of the underlying representations *(form)* of the source (AST, Diana, Bill, Objmod).

One specific implementation detail is that all file objects (the CAIS *file node*), which represent the Diana or AST or Objmod, descend from the collection. The spec and body forms of the unit have secondary (rather than primary) links to the file nodes.

This grouping of compilation information into catalogs with all the various representations and attributes for each compilation unit represents the set of data managed by the proto-CAIS. This information is stored by the proto-CAIS in underlying host files. Each representation (AST, Diana, Bill, Objmod) of a compilation unit is kept as a separate file on the host. Relationships and attributes are stored in a single database represented by three files. The accompanying diagrams are intended to be suggestive of the use that the program library makes of the proto-CAIS rather than an exhaustively complete example.

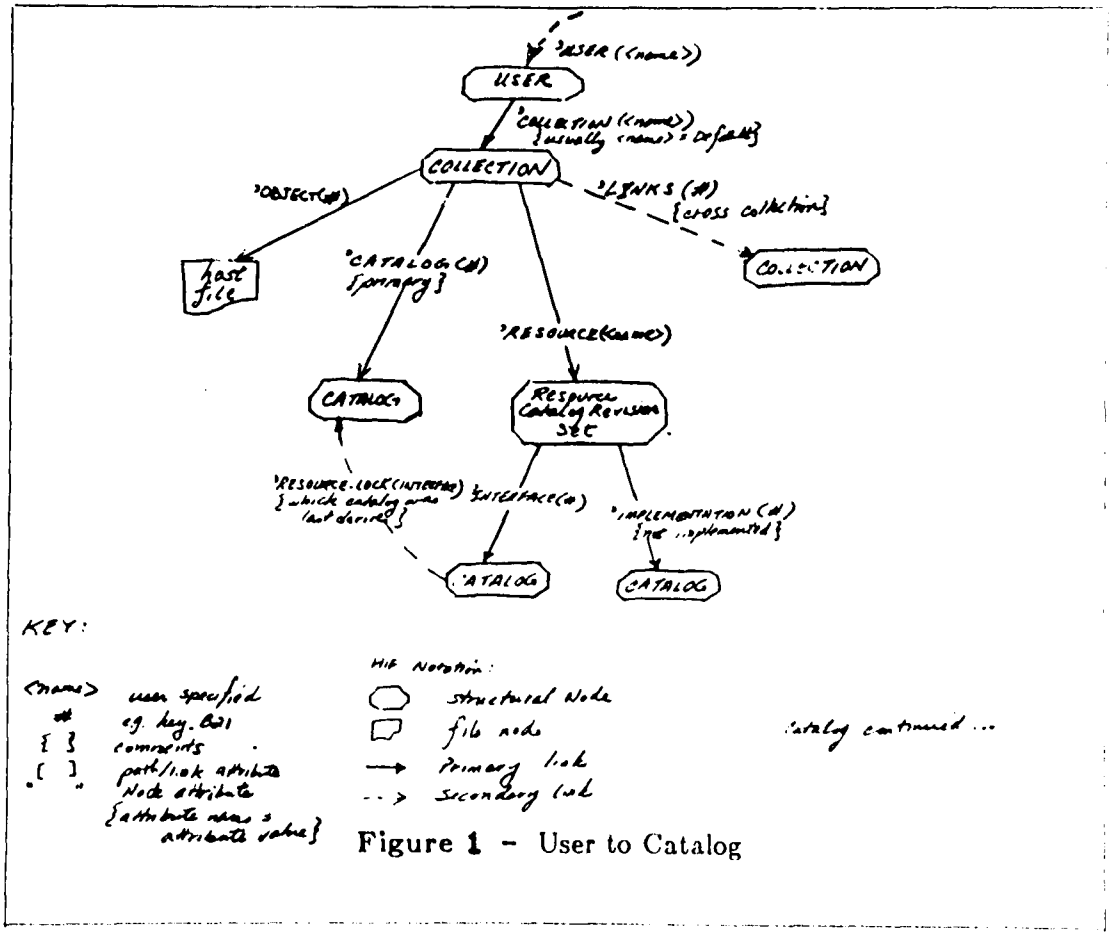


Figure 1 – User to Catalog

3. Implementation Details

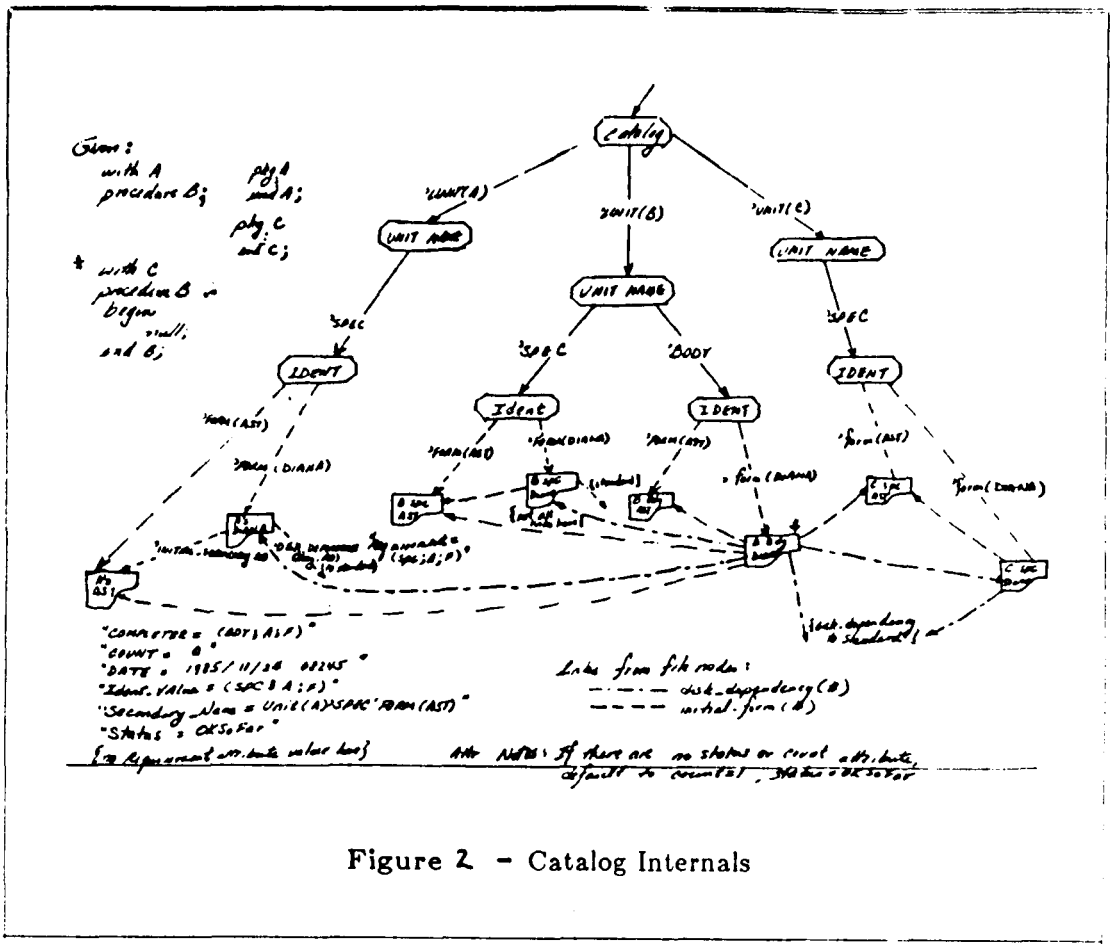


Figure 2 - Catalog Internals

Intermetrics has recently had experience in replacing the proto-CAIS with a totally re-written implementation to improve performance of the tool set using the proto-CAIS. Preliminary performance analysis indicated that the initial implementation of the proto-CAIS was a drain on the performance of the system, and it was targeted for a major upgrade in performance. The entire underlying implementation of the proto-CAIS was redesigned and reimplemented in light of the performance data, and the new implementation is currently installed in the latest version of the Ada compiler and its tools.

At present the new implementation is performing up to expectations with no anomalies reported due to differences in implementation. It is important to note that only minor changes (less than 500 lines of Ada code, excluding the new proto-CAIS code) were made in the compiler and tool sources (400KSLOC) to allow this new implementation to be installed. Most of these source changes were required by changed functionality of the new implementation of the proto-CAIS which were intended to improve performance without loss of portability. Some small number of changes were made to clear up anomalies in the preliminary implementation and definition of the proto-CAIS. Had no changes in functionality been required, there

would have been no source changes required in the sources of the tool set.

4. Conclusion

Intermetrics experience is that the Ada package construct, which allows separation of specification and implementation allows specification of a CAIS that is transportable across varying hardware and software bases. Additionally, the CAIS is an excellent basis for providing operating system functionality to Ada applications. By allowing the Byron APSE to be moved easily from system to system, and allowing significant re-writes of underlying code, Ada and the CAIS provide portability as well as transparency to change at the application/operating system interface level.

THE CAIS 2 PROJECT

Sue LeGrand
SofTech, Houston, Texas

and

Richard Thall
SofTech, Waltham, Massachusetts

ABSTRACT

The Common APSE Interface Set (CAIS) is a proposed MIL-STD intended to promote the portability of Ada Programming Support Environment (APSE) tools written in Ada. The standardized interfaces define a virtual operating system, from which portable tools derive their basic services, e.g., file management, input/output, communications, and process control. In the Ada world, such a virtual operating system is called a Kernel Ada Programming Support Environment (KAPSE). The CAIS is a standardized interface between KAPSEs and tools. The CAIS has been proposed as a starting point for standard interfaces to be used in the NASA Software Support Environment (SSE) for the Space Station Program. This paper describes the status of the CAIS standardization effort and plans for further development.

BACKGROUND

The proposed standard [1] was prepared, largely, by a volunteer group composed of members of the KAPSE Interface Team (KIT) and KAPSE Interface Team from Industry and Academia (KITIA). The KIT/KITIA is composed of 60 representatives from U.S. government, industry, and universities, as well as foreign governments and institutions. One seat on the KIT has recently been created for a NASA representative. A small core group of dedicated KIT/KITIA volunteers was responsible for producing the proposed CAIS 1 standard issued in January 1985. Public review of CAIS 1 is now being completed as part of the normal military standardization process. Unless insurmountable objections are recorded, CAIS 1 will become a MIL-STD. A number of prototype implementations of CAIS 1 have been or are being constructed for experimentation and validation of the design. No significant test of the design with tools that use the interfaces has yet been undertaken.

Design of a comprehensive interface set is a large, complex problem. Since the resources of the original volunteer group were severely constrained, CAIS 1 effort was focused on the problem of defining the major structural elements of the virtual interfaces, i.e., the data structuring model, the process control model, and input/output. Many subjects could not be addressed in the requisite time. These include configuration management issues, device control, resource management, issues related to distribution, interfaces between Ada tools, data exchange between environments, data typing in the file structure, and graphically oriented input/output.

In late 1985, a contract was awarded to SofTech, Inc. of Waltham, Massachusetts for the continued development of the CAIS. The enhanced CAIS is called CAIS 2. Compusec, Inc. of San Diego, California is a consultant to SofTech for issues relating to multilevel-secure operating systems and access control. The Naval Ocean Systems Center (NOSC) in San Diego, California, is the contracting organization acting in behalf of the Ada Joint Program Office. NOSC provides the technical lead for all KIT/KITIA and CAIS-related programs.

CAIS 2 DEVELOPMENT

The primary goals of the CAIS 2 project are to produce a standard that:

- meets practical requirements,
- is technically superior,
- is developed with responsive public review, and
- has adequate supporting material.

The major products of this project are a draft CAIS 2 Standard and a final CAIS 2 Standard. These are currently slated for publication in early 1987 and 1988, respectively. Experience has shown that it is exceedingly difficult to understand a software interface standard in the absence of considerable supporting documentation as well as an operating model. For this reason, CAIS 2 will be accompanied by a Rationale, a Guide for CAIS Implementors', a Formal Semantic Description of CAIS 2, and a prototype. Rationale documents will be published with the draft and final CAIS 2. Other supporting items will become available in the year following the publication of the final CAIS 2 Standard.

Public review meetings are planned after the publication of the draft and final CAIS 2 Standards as one method for obtaining constructive criticism from a wide audience. A more formal comment mechanism will also be available during these periods. At other times, the KIT/KITIA acts as the sounding board for design issues. As a guide for CAIS 2 design, the KIT/KITIA has published requirements for CAIS 2 [2]. These requirements are also subject to public review and comment.

CAIS 2 REQUIREMENTS

A few of the major CAIS 2 requirements are paraphrased below with commentary relating to NASA issues.

General Requirements

The CAIS services are intended to be sufficient to support tools used for software development. There are no requirements for real-time services as might be required by many NASA applications. Except for some aspects of communications, software development has no time-critical component. Support for testing of applications which have time-critical features is not addressed by the requirements.

The CAIS shall be independent of any specific operating system or computer. However, a reasonable level of modernity is assumed.

When implemented with sufficiently sophisticated hardware and software, the CAIS shall be capable of supporting multilevel secure operations. In other words, CAIS access control mechanisms must be sufficiently robust to provide for the partitioning of data, users, and devices which are commingled in a common system, but operating with differing levels of security clearance. Some data and devices will be shared, others must not be. This requirement is critical for Space Station operations where classified military and proprietary industrial applications must all share a common facility.

The CAIS shall incorporate existing standards to the greatest extent possible.

The CAIS shall be designed to allow tools to operate in distributed environments. The least constrained model of distribution is a network of computers, each having independent memory and file storage. The database can be shared among the nodes of the network. Computers in the network may be of the differing types. This model should be sufficient to support the SSE.

Data Base Requirements

The requirements mandate support for a sophisticated file system, very close to what is usually called a database management system (DBMS). The DBMS is to support a very general structuring capability, e.g. an entity-relationship model.

No specific configuration management capability is required; although it is tacitly assumed that the structuring capability be general enough to support almost any configuration management method.

A database typing mechanism is required to control the name space of the data objects, the attributes possessed by data objects, and the nature of relationships that can be created among objects.

Robust access control is required. In addition to the conventional discretionary access control, mandatory access control for multi-level classified material is required.

CAIS 2 is required to supply a mechanism by which certain database operations trigger the execution of user-defined procedures.

CAIS 2 will supply a means for grouping database operations into transactions. When transactions are used, the database is permanently modified only when an entire transaction succeeds. Failing transactions result in no change to the database. It is also a requirement that the effect of running transactions concurrently shall be the same as running them in some serial order.

CAIS 2 is required to supply a mechanism for collecting and storing information about how database objects were generated. For example, the history of an object module would include the names and revision numbers of all source files used in the compilation, the name and revision number of the compiler used, and the parameters given to the compiler.

A standard data interchange format is required.

Program Execution Requirements

One executing program can start, stop, suspend, and resume other processes to which it has access.

The CAIS will supply a means for interprocess synchronization and communication.

The CAIS will supply a means whereby one process can monitor the execution of another process. This is useful for debuggers and other dynamic analysis tools.

CAIS 2 PLANS

The design of CAIS 2 has progressed to the point where some general statements can be made about CAIS 2 and its relationship with CAIS 1. We expect CAIS 2 to address all issues explicityly deferred by the CAIS 1 team. We expect simplifications in some areas. However, since the scope of CAIS 2 is significantly larger than CAIS 1, the overall complexity level may be similar. The issue of inter-tool interfaces will be addressed by proposed standard representations for textual and graphical data. CAIS 2 designers do not believe that standardization of inter-tool interfaces more specific than these are within the purview of the effort.

CAIS 2 will not alter the basic structure of the CAIS 1 database model. We expect, however, to conceptually simplify the discretionary access control mechanism. A typing mechanism will be superimposed upon the present entity-relationship model. This mechanism will allow new types of objects to be created by reference to existing types. There will be one base type for all objects, so that tools which operate on all database objects will not be affected by the creation of new types of objects. As a minimum, the typing mechanism will manage the name space of database objects as well as the allowed attributes and relationships. It is not clear if the typing mechanism will deal with the representation of database objects. A few additions to CAIS 1 database services are expected for support of distributed databases.

CAIS 2 will maintain the present process model, i.e. tree structured process creation with a few embellishments. It is likely that some changes will be needed in the area of interprocess communication and control in order to support distributed environments.

The entire input/output model of CAIS 1 will be streamlined. The present model has built-in services for specific classes of devices, e.g. scrolling terminals, page terminals, and form terminals. Not only does this approach proliferate the number of interfaces, but it fails to promote the notion of device independence. For example, given a tool written for a page terminal, it could be difficult to redirect output from that tool to a scrolling terminal. While it may not be possible to achieve satisfactory operation of a screen editor from a Model 33 Teletype, we do not want the interfaces to encourage the construction of device-dependent tools. To accommodate differing devices, we intend to propose the notion of a logical device driver (LDD). An LDD is a program fragment that converts information in a standard representation to and from a stream of commands for a known device, producing the best rendering possible, given the device constraints. Under this proposal, CAIS 2 will have specific interfaces for LDDs, in addition to the normal tool interfaces. If the LDD interfaces can be defined with the correct blend of flexibility and specificity, it should be possible to write LDDs in Ada and transport them from one CAIS implementation to another. In other words, tools would be largely device independent but would depend upon a specific collection of device drivers. Tools would be ported with their associated LDDs, if the LDDs are not already present on the new host. New devices would require new LDDs to be created; however, a new LDD should allow most existing tools to be used with the new device, unless the tool or the device has some unusual characteristic. The LDD concept would allow CAIS 2 implementations to utilize new devices without circumventing the Standard, or necessitating a change to the Standard.

Like CAIS 1, CAIS 2 will supply a bridge to I/O facilities defined by Chapter 14 of the Ada language standard. These facilities are sufficient for a large number of tools, many of which will exist prior to or outside of CAIS 2 implementations. This bridge will allow such tools to be imported into CAIS 2 environments with minimal source code alteration.

We have proposed that CAIS 2 define a few standard data representations sufficient for a large proportion of tools. One representation would be used for sophisticated text. It would encompass the conventional ASCII

character stream, but augment it to support multi-font, multi-format, multi-color realizations. This representation would be bipartite, separating the text stream from the description of how the stream is to be displayed. We have also proposed a standard representation for graphical images. This representation would subsume the sophisticated text representation. Finally, we have proposed a standard language for describing how physical file layout corresponds to the Ada file specification. This would allow files to be converted so that data can be moved across the boundaries between computers, operating systems, KAPSEs, and compiling systems. A standard interchange representation will complete the capability for moving data between CAIS implementations. This capability is key to the success of heterogeneous distributed systems such as SSE.

CAIS 2 designers hope to be able to apply the concepts of standard representations and LDDs to other areas of the CAIS in order to build some resilience into the Standard. A standard as comprehensive as CAIS cannot survive unless it can be rapidly adapted to changing hardware technology as well as the demands of sophisticated applications such as the Space Station.

REFERENCES

[1] Military Standard Common APSE Interface Set (CAIS); 31 January 1985; Department of Defense, Washington, D.C. 20302.

[2] DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS); 13 September 1985; Ada Joint Program Office, Washington, D.C.

TRANSPORTABILITY, DISTRIBUTABILITY, AND REHOSTING EXPERIENCE WITH A KERNEL OPERATING SYSTEM INTERFACE SET

F. C. Blumberg, A. Reedy, and E. Yodis
Planning Research Corporation
1500 Planning Research Drive
McLean, Virginia 22102

INTRODUCTION

For the past two years, PRC has been transporting and installing a software engineering environment framework, the Automated Product Control Environment (APCE), at a number of PRC and government sites on a variety of different hardware. The APCE was designed using a layered architecture which is based on a standardized set of interfaces to host system services. This interface set, called the APCE Interface Set (AIS), was designed to support many of the same goals as the Common Ada™[1] Programming Support Environment (APSE) Interface Set (CAIS): transportability of programs; interoperability of data; and distributability of the environment processes and data. However, the evolution of the AIS has been quite different than that of the CAIS. The AIS was designed to support a specific set of lifecycle functions and to provide maximum performance on a wide variety of operating systems.

The APCE was developed to provide support for the full software lifecycle. Specific requirements of the APCE design included: automation of labor intensive administrative and logistical tasks; freedom for project team members to use existing tools; maximum transportability for APCE programs, interoperability of APCE database data, and distributability of both processes and data; and maximum performance on a wide variety of operating systems. The functions supported by the APCE include: configuration management for lifecycle products; traceability; change and release control; project control and reporting; management for all levels of testing including integration and system testing; and support for standards enforcement. The AIS design is critical in supplying transportability, interoperability, and distributability. The AIS design is also critical in providing the basis for APCE performance.

This paper gives a brief description of the APCE and AIS, a comparison of the AIS and CAIS both in terms of functionality and of philosophy and approach, and a presentation of PRC's experience in rehosting the AIS and transporting APCE programs and project data. Conclusions are drawn from this experience with respect to both the CAIS efforts and the Space Station plans.

[1]Ada™ is a registered trademark of the U.S. Government Ada Joint Program Office.

ENVIRONMENT FUNCTIONS

The APCE has been designed based on a separation of concerns between the functionality of the environment framework or architecture and the functionality of tools. The environment provides control, coordination, and enforcement of standards and policy and acts as repository for information (including software lifecycle products). The tools assist the project members in the actual creation or modification of the products (software and associated documentation and lifecycle products).

The APCE supports a software lifecycle process paradigm. The software lifecycle is viewed as a series of development or maintenance projects. Project members fall into three board categories: managers, developers, and testers. Developers include all project members who create or modify lifecycle products: requirements analysts, designers, coders, etc. Testers include the traditional categories of configuration management and quality assurance personnel and personnel involved in product reviews and audits. Projects have phases which can be defined in terms of the products developed during each phase. The APCE requires a testing process for the products of each phase. The paradigm is illustrated in Figure 1 which uses Mil-STD-2167 phases and products as an example. The APCE is configurable for different phases and products as well as for different methods of integrating products (software or documents) from components.

The functions provided by the APCE framework include:

- o configuration management of software, documentation, and test procedures;
- o automated status reporting and tracking of product components, work packages, and changes;
- o maintenance of traceability from requirements through development to code;
- o automated test bed generation and support for testing from unit testing through system testing;
- o maintenance of project database;
- o automated integration and release control for products;
- o enforcement of selected standards and procedures through testing;
- o project specific environment configuration.

The user interface consists of a set of menus for the major subsystems. The functions provided by the five major subsystems are summarized below.

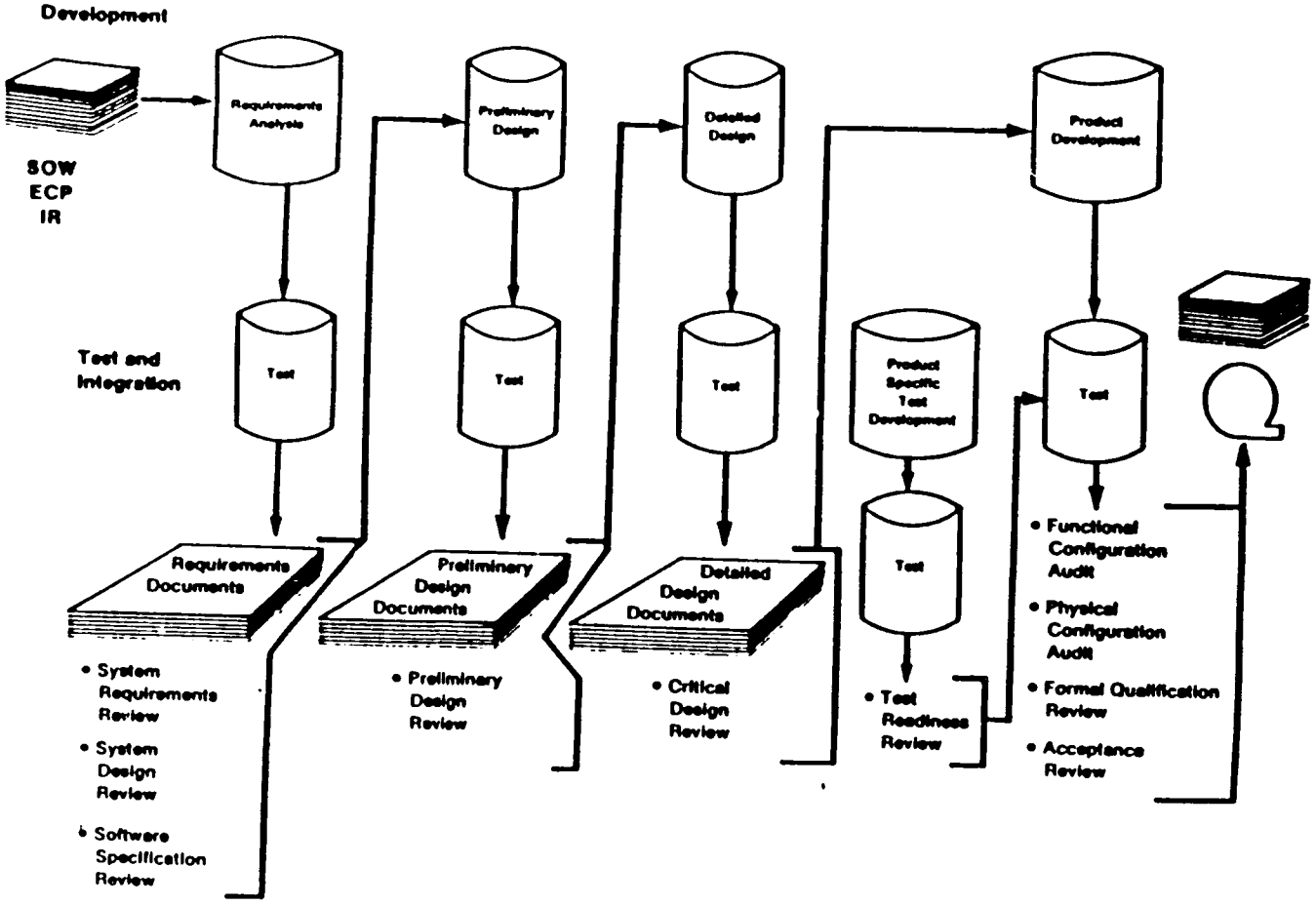


FIGURE 1: APCE DoD DOCUMENTATION AND REVIEW SEQUENCE

Generation Subsystem: The generation subsystem allows selected privileged users to configure the APCE to the specific project in terms of user groups and organization, work packages and schedule, project phases, products, and product integration structure. The APCE can be reconfigured to reflect changes to the project structure and organization as needed. The generation subsystem uses this information to organize the project database.

Development Subsystem: The development subsystem allows developers to select the data or products associated with their task and to return their finished products back into the database when they are ready for testing. The developers can use the software tools available on their host system or workstation to work on the products. The current version of the APCE does not directly control the use of these tools.

Test Subsystem: The test subsystem supports the testers in the building, execution, and reporting of the product tests. The test subsystem allows the testers to create test procedures, which are then managed by the APCE. The APCE will build test beds and integrate product components for the testers, who will then execute tests. The testing process provides the methods for enforcement of standards and policies. The testers report the test results through the test subsystem. Testers are also responsible for system release in the APCE paradigm and the test subsystem performs this function.

Change Control Subsystem: The change control subsystem allows managers to enter change requests into the system and to define stop dates for release support.

Report Subsystem: The report subsystem allows managers and other APCE users to get reports on the current status of changes, test procedures, and releases. It also gives reports on project status by task or by product component. Additional reports provides impact analysis for proposed changes and other traceability information.

ENVIRONMENT GOALS

The goals of the APCE design are:

- to provide management and control for the full software lifecycle process;
- to automate the labor intensive administrative and logistical overhead functions;
- to allow full use of existing hardware, operating systems, file management/DBMS, and communication resources.

The last goal implies a series of subgoals. An environment should be distributable across heterogeneous operating system configurations, heterogeneous file management/DBMS facilities and use the available communications facilities as well as heterogeneous hardware configurations.

The control framework must be easily transportable to new hardware and host systems at reasonable cost. The environment database, including the lifecycle products and their relationships and attributes, must be easily moved between environment instances. There must be no performance penalties for using the environment. It must cooperate at some level with existing operating systems to take advantage of their security and performance features. Finally, the environment must allow the use of existing software tools and allow flexibility for retooling as necessary.

ROLE OF THE APCE INTERFACE SET

The basic architecture of the APCE is best described as "Stoneman inspired but data coupled". The system is layered as illustrated by Figure 2. The host system (s) provide basic services such as operating system services, file management system/access mechanism or database management system, access controls, and communications mechanisms as needed for the configuration. The communications facilities are needed if distribution, workstations, or remote test beds are desired. The software engineering environment instance based on the APCE is layered on top of these services. The instance provides users with project specific tools and procedures which will usually exercise the host services directly and the APCE major subsystems which exercise the host services through the APCE Interface Set (AIS).

Since the APCE major subsystems use AIS calls, the APCE is transported to a new hardware/OS/DBMS configuration by rehosting the AIS. Thus, the AIS provides the Kernel interface described by Stoneman and supports the goal of distribution. Since all database accesses must be made through the AIS, the AIS also supports the interoperability of project data.

IMPLEMENTATION PHILOSOPHY

The AIS design reflects the implementation philosophy of the APCE as a whole. The architecture of the APCE is data coupled. That is, the APCE subsystems do not interface directly with each other; rather, they interface via the AIS to the project database. The APCE adopts an open system approach towards the use of third party tools. The APCE controls lifecycle products which are entered into the database through user interaction with APCE subsystems. Thus, there are no constraints on the tools used to develop the products. For maximum performance, the AIS is designed to function in conjunction with a modern operating system rather than on a bare machine. Tools do not have to be rehosted to the AIS in order to be used.

The AIS was developed by defining a set of transportability rules that provide the maximum independence for applications (tools, programs, etc.) from the run time environment. **For maximum transportability**, it was determined that **the application must have a logical view** of the operating services, the database services, communications services and the data it uses. The industry is evolving toward this conclusion, however, only a step at a time.

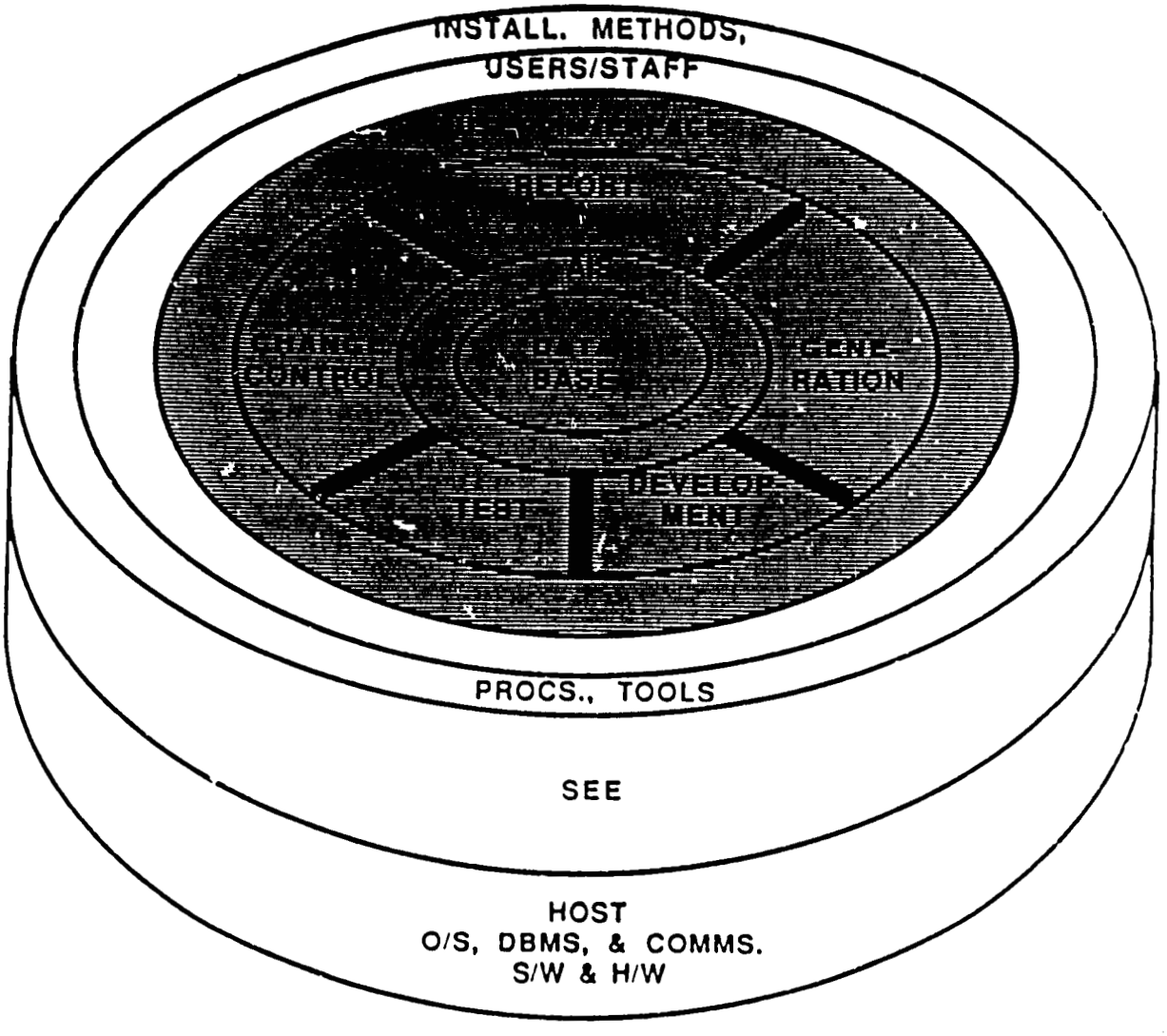


FIGURE 2: APCE STATIC VIEW

As an example, currently UNIX™[2] is considered transportable and it does provide hardware independence. However, it does not provide application independence any more than any other operating system. Accepting a operating system as the basis for transportability provides the application a highly constrained set of system services, database services, and communication services which may adversely affect the applications performance. Therefore a set of logical service interfaces was implemented that can be mapped to any operating system, file management/database management system and communication protocols.

This AIS implementation has been proven transportable over a wide range of operating systems, file management/database management systems, and hardware. The AIS design approach assumes that the host system has been developed by the vendor to take full advantage of the hardware features of the computer. The host system should provide performance achievable only through intimate study of the hardware system. The AIS takes advantage of the host system performance and does not try to duplicate it. The performance the AIS should be the same as that of the services supplied by the host system.

The AIS assumes that the following features are supplied by the host system:

- o file management system/access mechanisms or database management system;
- o access controls;
- o command processor with command script feature;
- o communications mechanism (e.g. VAX™[3] DECnet) between host(s)/workstations(s)/targets(s) if distribution or remote workstations or remote test beds are desired.

CAIS/AIS COMPARISON

The CAIS had no impact on the APCE development, however both the CAIS and the AIS had similar goals. The intent of both interfaces sets was to achieve transportability of tools between environments and to achieve interoperability of data between environments. The CAIS was in response to a need in the DoD for cost reduction and commonality of tools for software development. The same requirement fostered the AIS developed within PRC. PRC has many software development contracts running concurrently, and each contract has different required hardware, tools, and methods. Therefore, PRC requires an environment that is adaptable, transportable and allows interoperability of data and excellent performance on any host system.

The AIS strategy is based on a layering of system services rather than on a specific system service interface model (such as the node model of the CAIS).

[2]UNIX™ is a registered trademark of Bell Laboratories.
[3]VAX™ is a registered trademark of Digital Equipment Corporation.

The APCE software is based on an interface into which the host system services that satisfy the interface specifications are mapped. The AIS design is based on the expected availability of certain host system services. If a service is not directly available, then extra layers of software which provide the needed enhancement are created below the interface layer to satisfy the requirement.

Both the CAIS and the AIS attacked the problem at the interface layer between operating system services and the application programs. See Figure 3, AIS/CAIS Comparison, for AIS/CAIS comparison. As the diagram illustrates, the AIS provides services at a slightly higher level of abstractness than the CAIS. In addition, the AIS already has additional interfaces operational (DBMS, Communications) that the CAIS has not implemented as can be seen in Figure 4, CAIS/AIS Major Functions. The CAIS also requires a significantly greater number of functions primary because of the node management requirement. The AIS terminal I/O implementation currently only handles form management functions, and therefore does not provide as rich a set of features as the CAIS terminal I/O provides.

The primary difference between the AIS and the CAIS is the concept of the node model. The node model provides a method of organizing files, directories, devices, queues, and processes into a form that can be manipulated by any APSE tool on any host that implements the CAIS. The node model is similar to the implicit node model within the UNIX™ operating system with some extensions. The AIS embraces the concept that applications (programs, tools) require only a logical view of the services. Therefore, the interface functions should be mapped into the existing system services providing these capabilities.

The AIS provides only the logical view of the system services to the application which accomplishes two goals, total application independence and improved performance. Figure 5, CAIS/AIS Implementation Differences, illustrates each implementation.

Application independence is attained because dependence on structural or physical implementation of each service has been removed from the applications domain. This has not been attained in the CAIS because each application has knowledge of the node model and therefore any change to the node model will require a change to all applications dependent upon that structural knowledge.

The direct mapping of AIS services to system services enables an AIS implementation to operate as efficiently its host system. The CAIS, however, superimposed a control structure (the node model) on top of existing services that may limit performance on a given CAIS implementation.

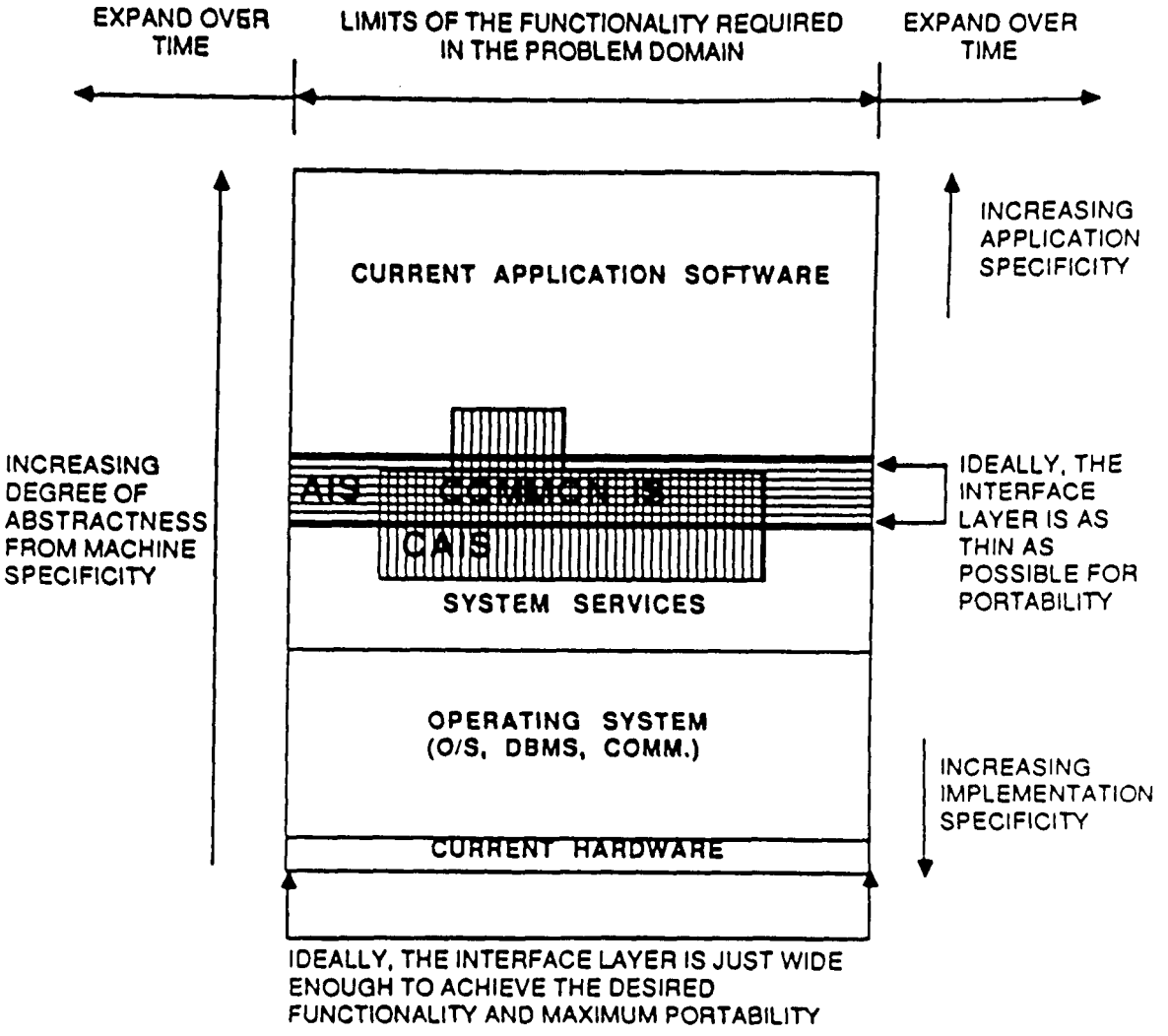


FIGURE 3: AIS/CAIS COMPARISON

	CAIS MAJOR FUNCTIONS	AIS MAJOR FUNCTIONS
SYSTEM MANAGEMENT (Node Management)	73	NONE. These functions are accomplished differently by each O/S, DBMS, and Communication suite.
PROCESS		
Host	32	15
Distributed	Not Implemented	37
INPUT/OUTPUT		
File Access/DBMS	38	54
Terminal	106	14
Tape	20	Host Operating System Provided
Distributed	Not Implemented	29
UTILITIES	86	26
TOTAL	355	175

FIGURE 4: CAIS/AIS MAJOR FUNCTIONS

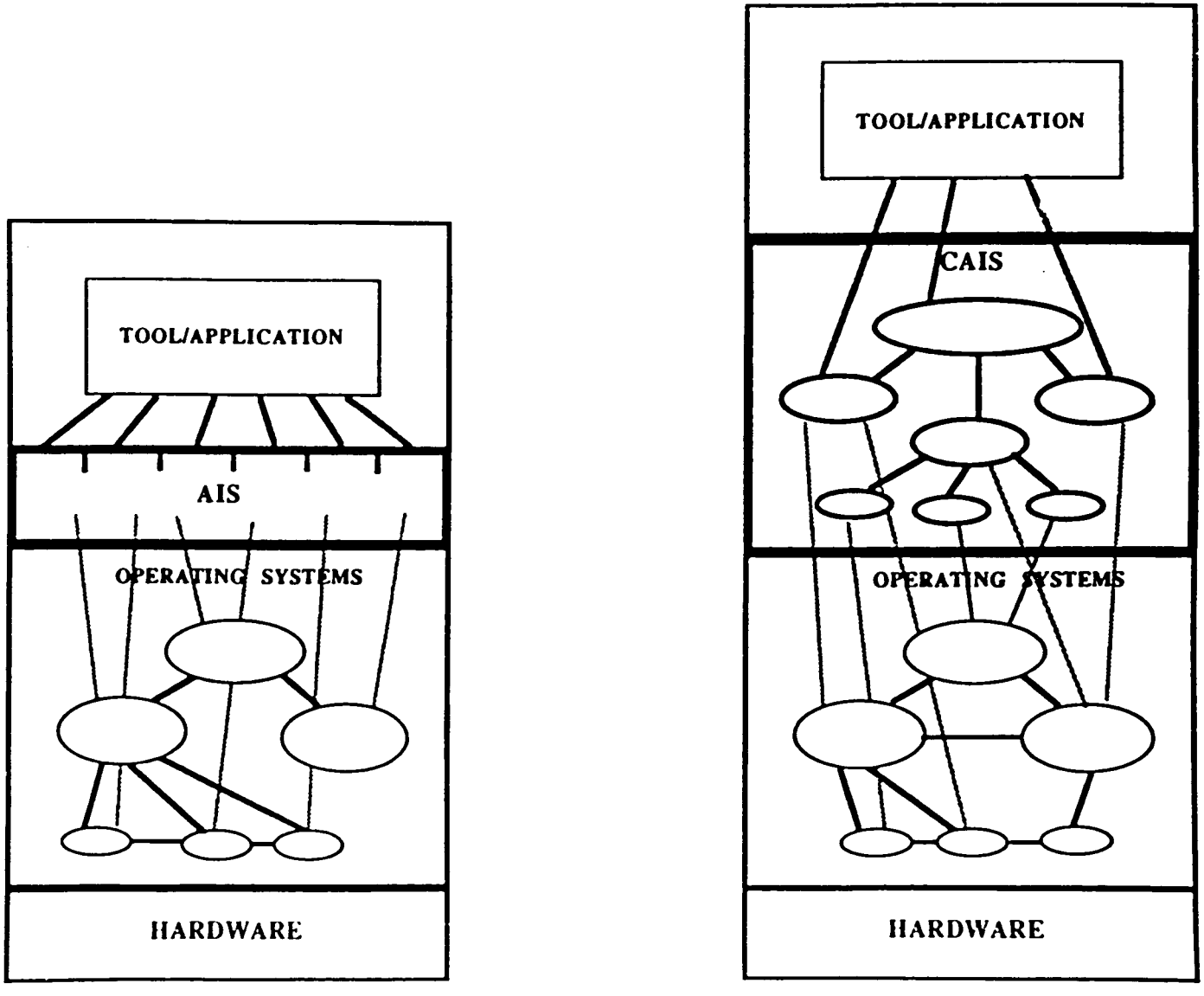


FIGURE 5: CAIS/AIS IMPLEMENTATION DIFFERENCES

EXPERIENCE TO DATE

The APCE is currently available on six different computer systems: VAX/VMS™[4], ROLM/AOS.VS, IBM/MVS™[5] and VM, and Intel 310 with XENIX™[6]. APCE processes can be distributed to the Macintosh™[7] and soon to the IBM PC. The rehosting process for the AIS takes approximately 2 calendar months for a mainframe and 1 month for a mini- or micro-computer. Figure 6, Current AIS Rehosts, illustrates the current systems the APCE is available on and the time it took to accomplish this, both in months and staff months.

APCE transportability has been attained using the AIS and a 'C' compiler. All APCE framework applications were designed using Ada™ PDL and implemented in 'C'. This was done because the Ada compilers were not available on all the hosts targeted for the APCE. The use of 'C' has not been without problems. Current implementations are using five (5) different 'C' compilers and as each new compiler has been introduced a 'C' subset has been defined. All APCE applications must be normalized to any new subset. This has entailed a five to ten percent code modification for each new subset. However, all new applications use the subset and are completely transportable. Because PRC must validate each 'C' compiler used for APCE code, the APCE will be recoded in Ada™ when validated compilers are available.

CONCLUSIONS

The APCE has the advantage that it can be installed in an existing configuration with minimal distruption of the current way of doing business. It provides a clear transition path into a better disciplined engineering process and allows new advances in automated tools to be incorporated. It does not, however, shield the users from a need to understand the native operating system or tool command language. This is not viewed as a disadvantage at this time since standardization of these features does not seem to be possible. Premature standardization of these features by an environment may ensure its technical obsolescence or, at best, enforce a delay while new tools are rewritten or rehosted. Such standardization is also not possible for a software house which works with a wide client base with widely differing requirements and standards for their software development and maintenance projects.

The APCE also does not provide the tight integration of tools. The user is still responsible for ensuring that the output of one tool is suitably modified to be acceptable as input for the next. This is one of the areas in which future work needs to be done to relieve the users of the more clerical types of work.

---

[4]VMS™ is a registered trademark of Digital Equipment Corporation.

[5]IBM/MVS™ is a registered trademark of International Business Machines, Inc.

[6]XENIX™ is a registered trademark of Microsoft Corporation.

[7]Macintoch™ is a registered trademark of Apple Computer, Inc.

REHOST		SCHEDULE TIME	STAFF MONTHS
Rolm MSE 800	AOS/VS	1.2 Months	4 Staff Months
INTEL 310	XENIX	1.5 Months	4.5 Staff Months
IBM	MVS	2.5 Months	12 Staff Months
IBM	VM/CMS	5 Months	13 Staff Months
Macintoch		1.5 Months	1 Staff Month

FIGURE 6: CURRENT AIS REHOST

The APCE framework provides significant advantages and can be used by a project without new hardware or significant retooling. It provides an immediate benefit without locking out future advances in software tools and techniques by managing the process and products rather than focusing on tools. The APCE provides a different approach to the software engineering environment problem.

PRC has been successful in rehosting the APCE to six different operating systems, with 4 different file management/database management systems that use 2 different sets of communication services without affecting the APCE applications. Since these different APCE instances can exchange project data and any APCE application is transportable between APCE instances, **the AIS attains true application independence.**

The benefits of using **an AIS like interface opens the options for the Space Station Software Support Environment (SSE)** configurations. **No longer constrained to only hardware independence** by operating system transportability; now a truly heterogeneous SSE can be configured. This environment will be **able to take advantage of all the required technology** while maintaining a consistent single environment through the SSE applications (tools and framework). The SSE will be **truly evolvable** since host services are divorced from the SSE itself therefore allowing new services (O/S, DBMS, communication and hardware) to be introduced and obsolete services to be retired **without disruption to operations.**

CONSTRUCTING A WORKING TAXONOMY OF FUNCTIONAL Ada SOFTWARE COMPONENTS FOR REAL-TIME EMBEDDED SYSTEM APPLICATIONS

Robert Wallace
Reseach Triangle Institute
Research Triangle Park, North Carolina

A major impediment to a systematic attack on Ada software reusability is the lack of an effective taxonomy for software component functions. The scope of all possible applications of Ada software is considered too great to allow the practical development of a working taxonomy. Instead, for the purposes of this paper the scope of Ada software application is limited to device and subsystem control in real-time embedded systems. A functional approach is taken in constructing the taxonomy tree for identified Ada domain. The use of modular software functions as a starting point fits well with the object oriented programming philosophy of Ada. Examples of the types of functions represented within the working taxonomy are real time kernels, interrupt service routines, synchronization and message passing, data conversion, digital filtering and signal conditioning, and device control. The constructed taxonmy is proposed as a framework from which a need analysis can be performed to reveal voids in current Ada real-time embedded programming efforts for Space Station.

Visualization, Design, and Verification of Ada® Tasking Using Timing Diagrams

by

R.F. Vidale*, P.A. Szulewski**, and J.B. Weiss**

ABSTRACT

This paper recommends the use of timing diagrams in the design and testing of multi-task Ada programs. By displaying the task states vs. time, timing diagrams can portray the simultaneous threads of data flow and control which characterize tasking programs. This description of the system's dynamic behavior from conception to testing is a necessary adjunct to other graphical techniques, such as structure charts, which essentially give a static view of the system. A series of steps is recommended which incorporates timing diagrams into the design process. Finally, a description is provided of a prototype Ada Execution Analyzer (AEA) which automates the production of timing diagrams from VAX/Ada debugger output.

1.0 Introduction

Concurrent programming brings another dimension of complexity to the problem of software design and testing. Unlike sequential programming, where functional decomposition allows the designer to concentrate on one module at a time, concurrent programming in Ada requires the coordination of many modules (tasks) executing in parallel. The requirements for task sequencing must be established early in the design, and carried through into the traditional domain of detailed design. An incomplete understanding of the task sequencing requirements or their erroneous implementation is an invitation to disaster.

* Boston University, Boston, MA
**The Charles Stark Draper Laboratory, Inc., Cambridge, MA
® Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

Most available software development tools and techniques, based on functional decomposition, do not adequately portray time dependency and thus do not help the developer visualize, design, and verify task sequencing. Tasking, as a programming technique, presents opportunities to improve productivity, maintainability and portability, but also introduces the possibility of programming errors unique to tasking. Incorrect design or implementation of tasking will produce unintended task sequencing which at best degrades system performance, at worst results in deadlock, deadness, or starvation.

Within the past three years, a number of object-oriented design methods have been proposed specifically for Ada. See Booch [BOOC83], Buhr [BUHR84], and Cherry [CHER85], for example. These methods all use the structure-chart type of diagram to describe the architecture of an Ada program. With the exception of Buhr, whose diagrams include some temporal notations, these representations are essentially static, and as such are of limited use in visualizing the overall sequencing of task interactions intended for a design. Buhr does make limited use of timing diagrams in his book [BUHR84] to illustrate the rendezvous, but does not include them in the design process.

It is the opinion of the authors that timing diagrams are a necessary adjunct to structure charts and should be used in conjunction with them first to design an Ada tasking program, then later to verify that it is behaving as expected. Tai [TAIK86] has also recognized the value of timing diagrams (rendezvous graphs, in his terminology) for debugging Ada tasking programs but does not advocate their use in the design process.

2.0 Timing Diagrams in Program Development

Timing diagrams are useful to Ada program developers at several phases in the life cycle. Data flow sequencing must be considered during the requirements analysis, preliminary design, detailed design, debugging, and testing. With tasking the proportion of time devoted to design, in relation to implementation, is much greater than for sequential programs. We propose the following steps for multi-task Ada program development for gaining confidence in the design before and after implementation.

1. Visualize objects and data flows using "cloud diagrams" to represent objects in the problem domain. Single threads of data flow can be shown by numbering them in sequence, but multiple, interacting threads are difficult to show.

2. Use preliminary timing diagrams, which do not show directions of calls, to show scenarios of required task interaction. Steps 1 and 2 are problem-domain representations.

3. Define Ada data structures and code and compile global data types.

4. Transform the problem-domain objects into Ada program units and portray these with structure graphs showing caller-callee relationships. Refine the preliminary timing diagrams to show caller-callee relationships with task ready/blocked state information.

5. Code the structure graphs in Ada as program unit specifications.

6. Code control skeletons in the program unit bodies to implement the task interactions visualized in the timing diagrams and annotated structure graphs.

7. Execute the code skeletons and generate a timing diagram.

8. Compare timing diagrams against desired behavior.

9. Revise design as necessary.

10. Complete Detailed Design of program unit bodies.

11. Generate timing diagrams to verify.

3.0 Automated Timing Diagram Generation

Automated support for the timing diagrams described in the preceeding section is not, to these authors' knowledge, publically available, but would require two forms: predictive and actual.

The preliminary timing diagrams would be predictive of the program's behavior. These diagrams would be drawn before any code is written to guide the developer in constructing the first level of task interaction. Successive, actual timing diagrams would be derived by simulating or executing program units and automatically extracting task trace information.

To date, no work has been done to develop automated support for the predictive diagrams, which is still a manual process. It is, however, feasible that a system, using formal specification and an assertion checker, could be developed to support this activity. There has, however been some work done by the authors of this paper in the development of a tool for generating actual timing diagrams of multi-task Ada programs.

4.0 The Ada Execution Analyzer Prototype

The Ada Execution Analyzer (AEA) Prototype has been developed at The Charles Stark Draper Laboratory, Inc. (CSDL), to explicitly show the relationship of time, concurrent operations, and task communication using the timing diagram format for multi-task VAX/VMS Ada programs. The AEA provides the capability to visually monitor the runtime execution of multitask Ada programs developed in the DEC VAX/VMS Ada Development Environment. The AEA is run as an extension to the VAX/VMS Symbolic Debugger, and thus provides all the capabilities of that debugger plus a graphic display of task execution. The AEA produces both an overview timing diagram which shows up to 20 Ada tasks, and a detailed timing diagram which shows up to 5 selected tasks. An example Overview Timing Diagram is shown in Figure 1 and an example Detailed Timing Diagram is shown in Figure 2. The symbology used in both diagrams is defined in Tables 1 and 2.

The impetus for developing the AEA was the inability of the conventional DEC/Ada debugger to provide visibility into concurrent task behavior. Even though multi-task information is available from the debugger, it is not easily converted to a useful format by manual means. It is a time consuming and error prone process.

The AEA provides graphic timing diagrams on demand from a program run, significantly reducing the debugging time for multitask programs. The availability of such a tool make practical the method outlined in Section 2.0.

The AEA Prototype is written in VAX/Ada and was released for internal use at CSDL in December 1985. As a rapid-prototype, the AEA was produced quickly in order to allow users some functionality and the opportunity to suggest enhancements. To date, the AEA has been used to debug some small tasking programs for both real projects and in-house Ada training problems. User acceptance of the tool has been generally favorable and the tool will likely be maintained as a corporate resource.

5.0 Future Extensions

Extensions to the AEA fall into three categories: short-term, medium-term, and long-term. Short-term extensions (within 6 months) will focus on making the current AEA implementation more user friendly and including some options to reduce clutter in the diagrams by selectively blanking tasks from the diagram.

Medium-term extensions (within 18 months) will focus on transporting the AEA to an embedded microprocessor development environment in order to extract timing diagrams from a target processor.

Long-term extensions (beyond 18 months) might include automatic task sequence checking and automatic generation of program unit body control skeletons. These extensions require the use of a formal specification technique like the Task Sequencing Language (TSL) [HELM85] during development.

6.0 Conclusions

Ada tasking adds a new dimension of complexity which is hard to visualize using established graphical design methods. With this added complexity, it is essential to work out the required task sequencing early in the design and have a means for verifying task sequencing behavior during testing.

Timing diagrams are a natural, easily understood means of visualizing task sequencing in the conceptual and testing phases of concurrent program development. Timing diagrams can evolve with the data-flow picture of a system. They can show time explicitly and can illustrate multiple threads of control including the effects of time slicing. In this manner they can be used to identify serious tasking errors like deadlock, race conditions, and starvation.

A prototype Ada Execution Analyzer, which produces timing diagrams from VAX/Ada debugger output, has demonstrated the value of timing diagrams in understanding the behavior of an Ada program with multi-tasking. The authors believe that the expanded role for timing diagrams suggested in this paper will result in fewer design errors in multi-tasking applications using Ada.

REFERENCES

[BOOC83] Booch, G., Software Engineering with Ada, Menlo Park, CA, Benjamin/Cummings Publishing Company, 1983.

[BUHR84] Buhr, R.J.A., System Design with Ada, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[CHER85] Cherry, G., and B. Crawford, "The PAMELA Methodology," Thought Tools, Inc., Reston, VA, November 1985.

[TAIK86] Tai, K.C., "A Graphical Notation for Describing Executions of Concurrent Ada Programs," ACM Ada Letters, Vol. VI, No. 1, Jan., Feb. 1986.

[HELM85] Helmbold, D., and D. Luckham, "TSL: Task Sequencing Language," Proc. of the Ada International Conference, Paris, France, May 1985.

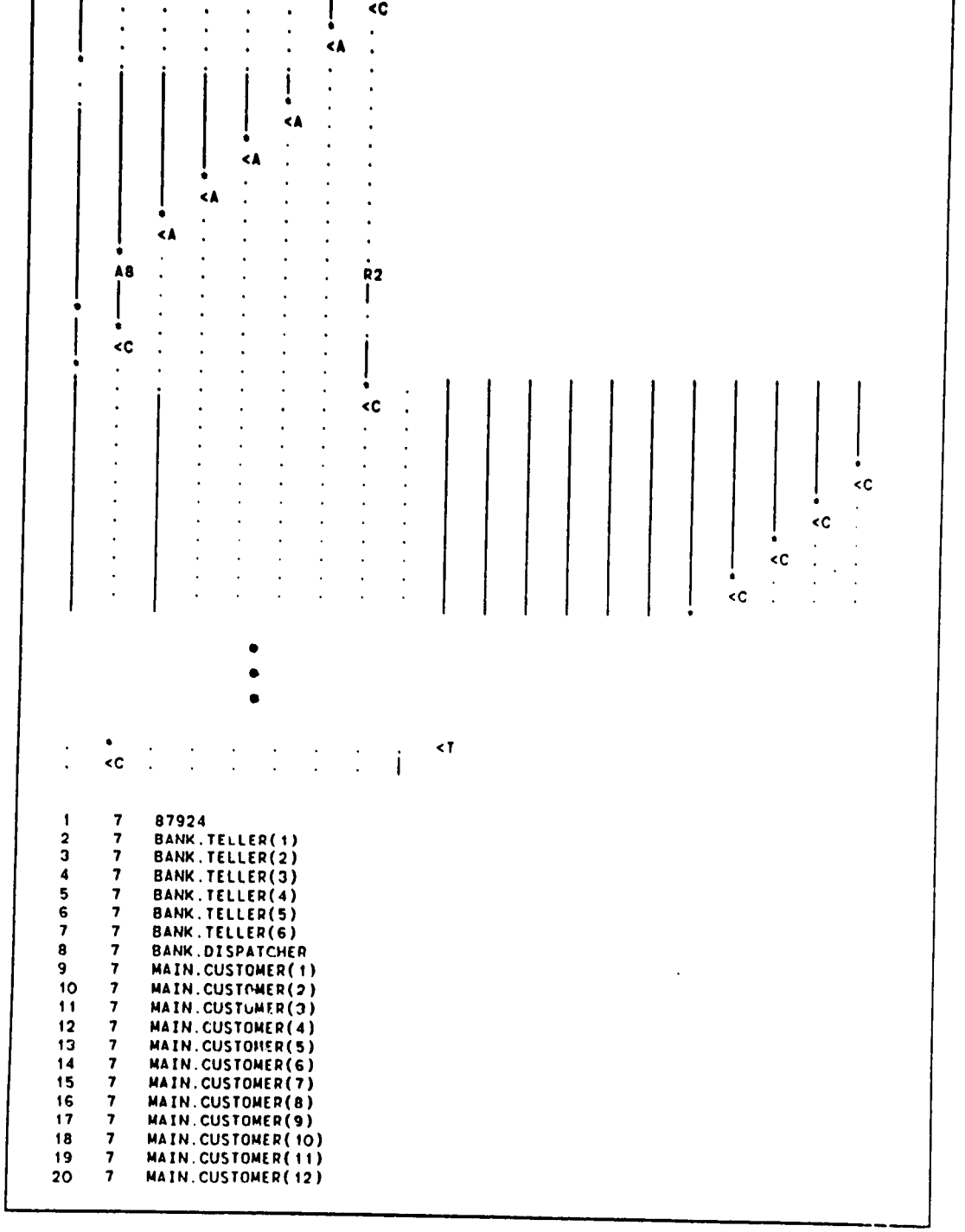


Figure 1. AEA Overview Diagram

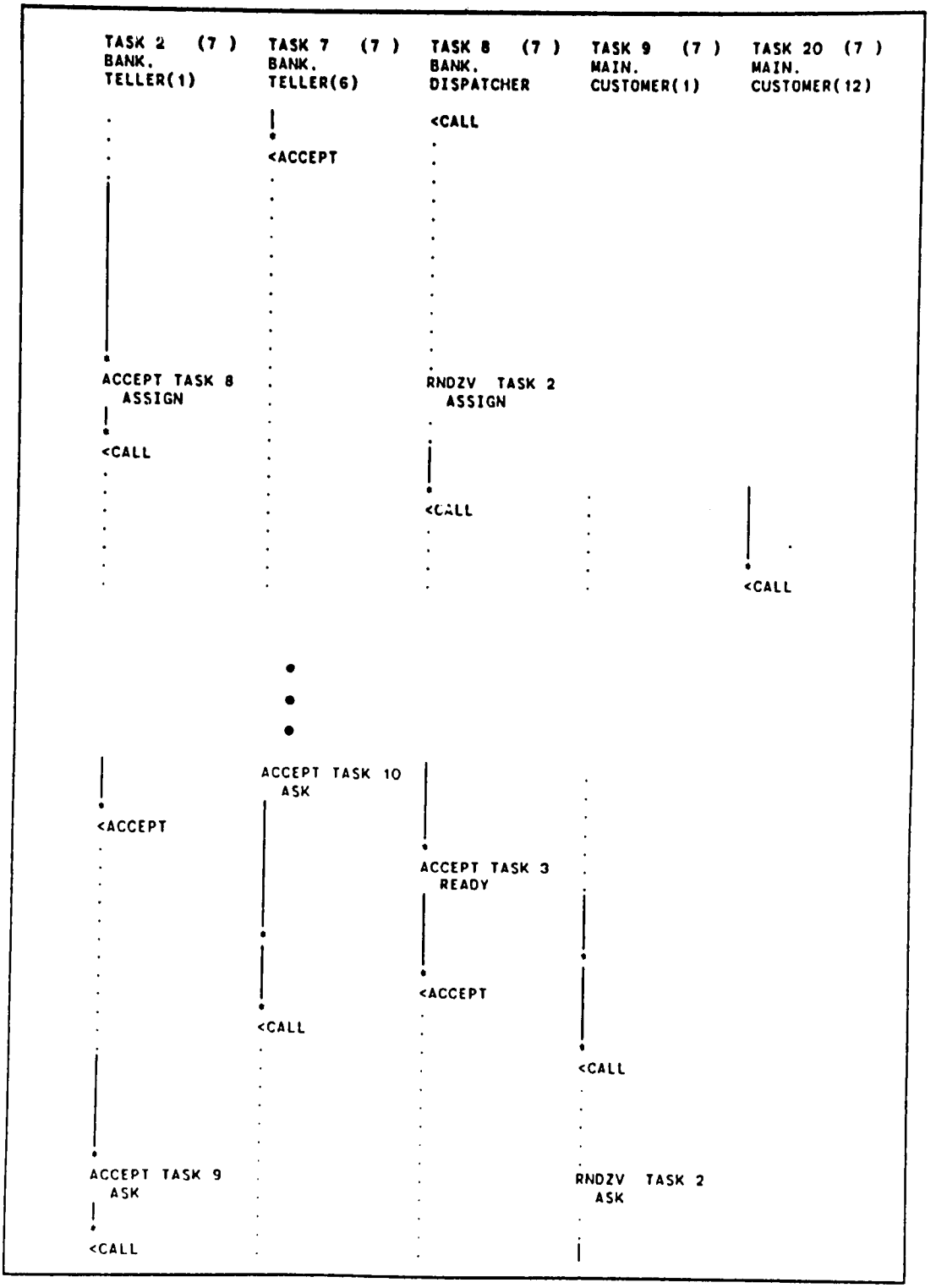


Figure 2. AEA Detailed Timing Diagram

Table 1. AEA Overview and Detailed Diagram States

TASK STATES		
TIMING DIAGRAM SYMBOLS	OVERVIEW DIAGRAM SYMBOLS	MEANING
TASK N (#)	TN P#	Task number N with priority #
UNIT.TASK_NAME		Logical name of program unit that declares TASK_NAME
		POINTS OF RENDEZVOUS:
RNDZV TASK #	R#	Task has rendezvoused with task #
ENTRY_NAME		Task #.ENTRY_NAME
ACCEPT TASK #	A#	Task has accepted call from task #
ENTRY_NAME		Accept ENTRY_NAME
		TASK STATES:
*	*	Task is running
*	*	Task is ready to run
|	|	
•	•	Task is suspended
<TERM	<T	Task has terminated

Table 2. AEA Overview and Detailed Diagram Substates (Part 1 of 2)

TIMING DIAGRAM SYMBOLS	OVERVIEW DIAGRAM SYMBOLS	TASK SUBSTATE	MEANING
<ABORT	<AB	Abnormal	Task has been aborted.
<ACCEPT	<A	Accept	Task is waiting at an accept statement that is not inside a select statement.
<Completed[ab	<CA	Completed[abn]	Task is completed due to an abort statement, but is not yet terminated. In Ada, a task awaiting dependent tasks at its "end" is called "completed". After the dependent tasks are terminated the state changes to terminated.
<Completed[ex	<CE	Completed[exc]	Task is completed due to an unhandled exception, but is not yet terminated. In Ada, a task awaiting dependent tasks at its "end" is called "completed". After the dependent tasks are terminated, the state changes to terminated.
<Completed	<CO	Completed	Task is completed. No abort statement was issued, and no unhandled exception occured.
<Delay	<DL	Delay	Task is waiting at a delay statement.
<Dependents	<DP	Dependents	Task is waiting for dependent tasks to terminate.

Table 2. AEA Overview and Detailed Diagram Substates (Part 2 of 2)

TIMING DIAGRAM SYMBOLS	OVERVIEW DIAGRAM SYMBOLS	TASK SUBSTATE	MEANING
<Dependents[e	<DE	Dependents[exc]	Task is waiting for dependent tasks to allow an unhandled exception to propagate.
<CALL	<C	Entry call	Task is waiting for its entry call to be accepted.
<Invalid State	<IV	Invalid state	There is a bug in the VAX Ada run-time library.
<I/O or AST	<IO	I/O or AST	Task is waiting for I/O completion or some AST. (Asynchronous system true).
<Select or del	<SD	Select or delay	Task is waiting at a select statement with a delay alternative.
<Select or Ter	<ST	Select or term.	Task is waiting at a select statement with a terminate alternative.
<SELECT	<S	Select	Task is waiting at a select statement with neither an else, delay, or terminate alternative.
<Shared resour	<SR	Shared resource	Task is waiting for an internal shared resource.
<Terminated[a	<TA	Terminated[abn]	Task was terminated by an abort.
<Terminated[e	<TE	Terminated[exc]	Task was terminated because of an uhandled exception.
<Terminated	<TN	Terminated	Task terminated normally.
<Timed entry	<TI	Timed entry call	Task is waiting in a timed entry call.

Ada* and Cyclic Runtime Scheduling

Philip E. Hood
SofTech Inc.

Abstract

An important issue that must be faced while introducing Ada into the real time world is efficient and predictable runtime behavior. One of the most effective methods employed during the traditional design of a real time system is the cyclic executive. This paper examines the role cyclic scheduling might play in an Ada application in terms of currently available implementations and in terms of implementations that might be developed specifically to support real time system development.

The cyclic executive solves many of the problems faced by real time designers, resulting in a system for which it is relatively easy to achieve appropriate timing behavior. Unfortunately a cyclic executive carries with it a very high maintenance penalty over the lifetime of the software that it schedules. Additionally, these cyclic systems tend to be quite fragile when any aspect of the system changes.

This paper presents the findings of an ongoing SofTech investigation into Ada methods for real time system development. Section 1 discusses cyclic scheduling in general - what it is and why it is used. Section 2 examines how cyclic scheduling might be applied to Ada real time systems. Methods of introducing cyclic schedulers into applications without violating Ada semantics is explicitly discussed. Several classes of cyclic schedulers will be evaluated on their compatibility with the Ada world. Section 3 briefly examines how future systems might use a cyclic scheduler without paying the high price levied upon current systems. The topics covered include a description of the costs involved in using cyclic schedulers, the sources of these costs, and measures for future systems to avoid these costs without giving up the runtime performance of a cyclic system.

1.0 Cyclic Executive Description

A cyclic executive provides a mechanism for enforcing a predetermined ordering of processing events in a system. All processing to be performed is arranged within a schedule of finite duration. This schedule is repeated at a specified rate called the major cycle. The major cycle is broken down into a number (usually a power of two) of equal minor cycles. Each minor cycle is assigned a processing frame containing a list of processing elements (routines) to be performed during the associated minor cycle. An example of the basic cyclic executive structure is shown in Figure 1.

---

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

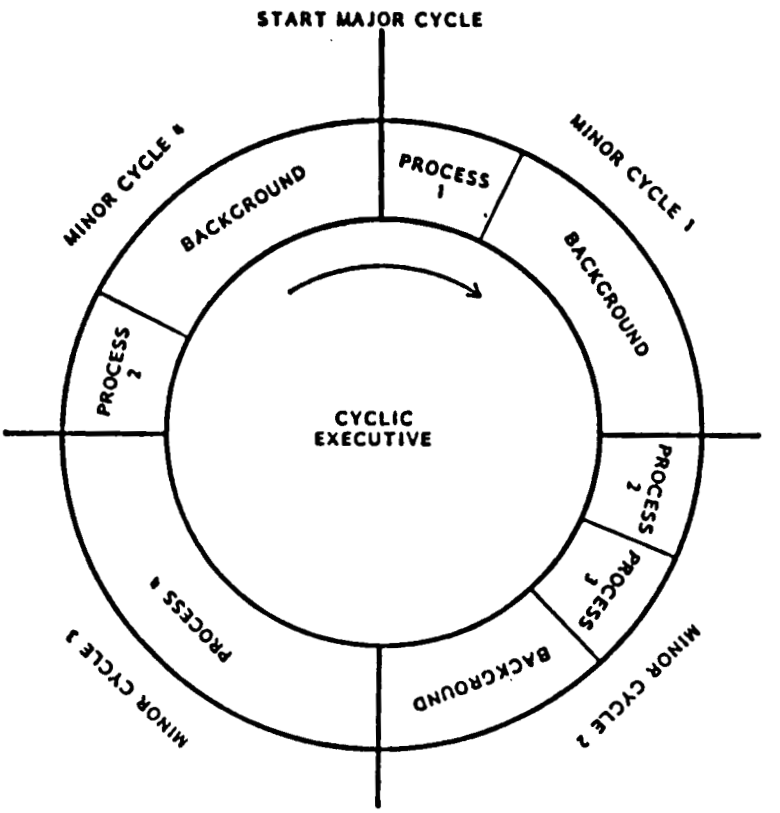


Figure 1 - An Example Cyclic Executive Structure

Although all cyclic executives share the structure we have described, they vary in almost every other aspect. Many types of cyclic executives have been developed to support various applications, and each one is different from the last. Some of these variations, such as mode changes, varying frame assignments and handling frame overruns, are discussed below.

1.1 Mode Changes

One of the advantages of a cyclic executive is that the static schedule can be tuned to optimize the system's timing performance for the expected load conditions. The load on the system, however, may not be constant. A change in the system load may cause the cyclic executive to allocate run time in a very inefficient manner (a job with a long allocated run time may have little or no processing to perform).

To solve this problem mode changes are introduced into the system. A mode change can change both the processing to be performed and the cyclic schedule. The more variation possible in the loading of the system, the more mode changing operations will be necessary. Each mode change is expensive in terms of new coding and tuning that must be performed and in terms of the damage to the program structure that always accompanies tuning operations.

1.2 Varying Frame Assignments

Schedule variations do not always require a mode change. If the variations can be localized to one frame, then that frame can use a local scheduler to resolve the problem. This solution of course, requires the overhead of some run time scheduling. Moreover, every possible scheduling possibility must be verified during system tuning.

1.3 Handling Frame Overruns

The greatest amount of variation between cyclic executives lies in the handling of frame overruns. We will consider the following four methods, by no means a complete list (many variations and hybrids exist): overruns ignored, overruns logged, overruns suspended, and overruns terminated.

1.3.1 Overruns Ignored

In some systems the problem of frame overrun can be adequately addressed during system debugging; these systems may choose to ignore overruns during runtime. The designer is responsible for verifying that overruns can never occur. This type of executive is typical of systems with either very simple software or over-confident designers.

1.3.2 Overruns Logged

This strategy is used in systems where no runtime action is appropriate when a frame overruns. The overrun is recorded for handling off line. This approach results in a very realistic executive for any system in which tuning issues can be adequately addressed. In a properly tuned cyclic executive application, frames should not be overrunning. Thus if this type of scheduler is inadequate, it implies that a cyclic schedule is not capable of providing a reliable schedule for that application and must be enhanced.

1.3.3 Overruns Suspended

When a frame overruns in this type of system, it is suspended and the next frame is allowed to start on time. When there is free time the suspended frame is allowed to complete.

This method greatly complicates data access in the application software. A built-in efficiency of a cyclic executive is the synchronization implied by static frame assignments. Additional synchronization is normally unnecessary during shared data references. When frame suspension is introduced, the implied synchronization is disrupted, and consequently references to shared data must include the appropriate synchronization mechanisms.

1.3.4 Overruns Terminated

When overruns occur in a system using this strategy, the overruning frame is terminated. It is restarted from the beginning at its next scheduled start time. This mechanism avoids the synchronization problems of the suspension mechanism but introduces its own problems. Software components that could possibly overrun frame boundaries must be written very carefully so that valuable data is not lost. There is also a potential problem with data that is incompletely updated when the frame is terminated - if this data is used by other components, serious problems could arise.

2.0 Ada Implementation of Cyclic Executives

Some varieties of cyclic executive fit very well into Ada, others do not map so naturally into the language.

The basic cyclic structure is fairly easy to implement in Ada. MacLaren [1] and Hood [2] show how to write simple cyclic executives in Ada. The basic cyclic scheduler for this type of executive is shown in Figure 2. This type of executive ignores the issues of varying loads and overrunning frames.

```
with Frame_Package;
package body Executive is

  task Cyclic_Scheduler is
    entry Minor_Cycle_Tick;
    for Minor_Cycle_Tick use at 8#1000054#;
  end Cyclic_Scheduler;

  task body Cyclic_Scheduler is
  begin
    loop  --forever
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_1;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_2;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_3;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_4;
    end loop;
  end Cyclic_Scheduler;
end Executive;
```

Figure 2. A Simple Cyclic Executive

The simple structure can be easily expanded to incorporate mode changes and variable frame assignments. Figure 3 shows a cyclic executive with mode changing. Each mode is represented by a complete list of frames to be scheduled in that mode. At the beginning of each major cycle, the executive decides which schedule to run. Varying frame assignments require no change to the cyclic scheduler; instead a local scheduler is created in the varying frame as shown in Figure 4.

Overruns can be logged by adding a task to receive the periodic interrupt and to check whether or not the previous schedule has completed. This type of scheduler is shown in Figure 5.

None of the cyclic variations discussed so far has been difficult to implement in Ada. The last two variations, namely overrun suspension and termination, are considerably more difficult. In both cases, these executives could only be written if they were heavily supported by the underlying run time system.

The only asynchronous scheduling point provided by Ada occurs when an interrupt is received, so this fact must be used in both the suspension and termination variations. Asynchronous response to an interrupt is not guaranteed by the Ada specification, however any Ada implementation that has any value in the development of real time systems have to provide asynchronous interrupt handling. The only ways to terminate an executing piece of Ada code are either to raise an exception or to abort the task. Asynchronous exceptions are not allowed in the Ada semantics, leaving only the abort statement. The abort statement is not guaranteed to stop the aborted task from executing at any particular time. Therefore, a frame termination executive could be written in Ada only if the underlying implementation guarantees the immediate termination of aborted tasks.

The overrun suspension executive has similar problems. The only way to ensure the new frame will have precedence over the old one is to introduce the new frame as a task with higher priority than the old frame task. This technique works for the frames in a given major cycle, but when the first frame is reintroduced at the beginning of the next major cycle, it must wait for all the frames from the previous major cycle to complete before starting. This behavior is clearly not desirable. In order to implement this type of executive in Ada, the implementation must provide some sort of dynamic priority mechanism. Standard Ada priorities are not dynamic, thus a additional priority scheme must be introduced. These new priorities can not interfere with the workings of the Ada priority system but can be used to assign relative priority to tasks that either have no standard Ada priority, or have the same standard priority.

In general, these executives require more control over the processing resources than can be obtained using a single thread of control (single Ada task). The resulting cyclic executive must be implemented using Ada tasking facilities. Ada tasking facilities, however, lack support for the primitive (and often dangerous) functions necessary for these variations of the cyclic executive.

```
with Frame_Package;
package body Executive is

  task Cyclic_Scheduler is
    entry Minor_Cycle_Tick;
    for Minor_Cycle_Tick use at 8#100054#;
  end Cyclic_Scheduler;

  task body Cyclic_Scheduler is
    type Mode_Type is (Mode_1, Mode_2);
    Mode: Mode_Type := Mode_1;
  begin
    loop  --forever
      case Mode is
        when Mode_1 =>
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_1;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_2;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_3;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_4;
        when Mode_2 =>
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_1;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_2;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_1;
          accept Minor_Cycle_Tick;
          Frame_Package.Frame_2;
      end case;
    end loop;
  end Cyclic_Scheduler;
end Executive;
```

Figure 3. A Cyclic Executive with Mode Changing

```
with Frame_Package;
package body Executive is

  task Cyclic_Scheduler is
    entry Minor_Cycle_Tick;
    for Minor_Cycle_Tick use at 8#100054#;
  end Cyclic_Scheduler;

  task body Cyclic_Scheduler is
    Status: Frame_Package.Status_Type := Frame_Package.Status_Type'First;
  begin
    loop  --forever
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_1;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_2;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_3;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_4 (Status);
    end loop;
  end Cyclic_Scheduler;
end Executive;

separate (Frame_Package)
procedure Frame_4 (Status : in Status_Type) is
begin
  case Status is
    when Good =>
      Application_1;
      Application_2;
    when others =>
      Application_1;
  end case;
end Frame_4;
```

Figure 4. A Cyclic Executive with Frame Level Scheduling

```
with Error_Handling_Package;
with Frame_Package;
package body Executive is
  task Tick_Handler is
    entry Clock_Tick;
    for Clock_Tick use at 8#1000054#;
  end Tick_Handler;

  task Cyclic_Scheduler is
    entry Minor_Cycle_Tick;
  end Cyclic_Scheduler;

  task body Tick_Handler is
  begin
    loop  -- forever
      accept Clock_Tick;
      select
        Cyclic_Scheduler.Minor_Cycle_Tick;
      else
        Error_Handling_Package.Log_Overrun;
      end select;
    end loop;
  end Tick_Handler;

  task body Cyclic_Scheduler is
  begin
    loop  --forever
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_1;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_2;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_3;
      accept Minor_Cycle_Tick;
      Frame_Package.Frame_4;
    end loop;
  end Cyclic_Scheduler;
end Executive;
```

Figure 5. A Cyclic Executive with Overrun Logging

4.0 Avoiding the Cost of Cyclic Scheduling

Cyclic scheduling is very costly over the lifetime of a software system. The reason is very simple: cyclic systems require software to be developed in modules according to their time consumption rather than according to functional considerations. Two phases of development are totally dominated by the timing structure of a cyclic executive: detailed design/coding stage and tuning. During detailed design, frame assignments are designed and coded specifically to fit into their assigned time slots. Functionality is traded back and forth between routines and frames, depending on where there is time.

Tuning can be though of as temporal debugging, during which timing errors are found and corrected. The correction methods include dividing up existing routines and shifting functionality between frames and routines. The end result is a very fragile schedule which meets the timing requirements but suffers several drawbacks: minor changes are likely to have sufficient impact on the schedule to require complete system retuning. Functional components are so dispersed that to understand any single component requires knowledge of the entire system. The structure of the software has been totally lost and maintenance efforts can only degrade the structure further. Finally, one has a system in need of constant and expensive maintenance.

In order to reduce the cost of this type of system, the creation of the cyclic structure must be separated from the creation and maintenance of the software components. Modern software engineering techniques can be applied to the system development and maintenance issues, with an extra step added to derive a cyclic implementation from a more general design. The code developed would be structured according to functional rather than timing considerations. The timing of the system would move from the detailed design and coding steps into a new precompilation step.

This extra step might be implemented as a machine-assisted (programmer directed) set of program transformations which parallel the cyclic design process that would normally take place during the software design. The transformational sequence as well as the the untransformed source would be save for future rederivations after necessary program maintenance is performed.

A tool assisted tuning system need not be limited to cyclic transformations. While there may always be a class of real time systems requiring cyclic runtime performance, there is an equally large number of systems that do not require such extreme measures. Many of these systems would benefit from the flexibility of an Ada style runtime scheduler. This type of scheduling allows more flexibility in dealing with runtime loading variations, and is far more robust when maintenance changes are made. These systems still require tuning, although not to the same extent. For these systems, other types of tuning transformations can be made available, such as replacing monitor tasks with semaphores or simplifying groups of tasks using program inversion techniques [3]. By applying these techniques, a system can be tuned until the appropriate level of predictability and efficiency has been reached.

Conclusion

There will always be systems that have a need for the runtime performance of cyclic scheduling. Many of the cyclic scheduling models fit well within the Ada language. In order for the cost of a cyclic system to be brought under control, new methods must be developed to for their creation. These methods ought not be limited to the creation of cyclic systems; however, they should provide a more general approach to the development of real time systems, with cyclic scheduling as one of many options for achieving real time performance.

References

[1] MacLaren "Evolving toward Ada in Real Time Systems," SIGPLAN Notices, 1980

[2] Hood, Philip and Grover, Vinod, Designing Real Time Systems in Ada, Technical Report 1123-1, SofTech Inc., Waltham, MA, January 1986.

[3] Rajeev, S., On Applying Ada to Real-Time Systems: The Inversion Technique and Some Examples, Technical Report TP 148, SofTech, Inc., Waltham, MA, March 1983.

[4] Rajeev, S., Certain Optimizations in Ada Tasking Implementations Technical Report 9074-2, SofTech, Waltham, MA, January 1983.

[5] Gonzalez, M.J., Jr., "Deterministic Processor Scheduling," ACM Computing Surveys, Vol. 9., No. 3., September 1977.

Choosing a software design method for real-time Ada applications: JSD Process Inversion as a means to tailor a design specification to the performance requirements and target machine.

James V. Withey, Intermetrics, Inc.

Abstract

The validity of real-time software is determined by its ability to execute on a computer within the time constraints of the physical system it is modeling. In many applications the time constraints are so critical that the details of process scheduling are elevated to the requirements analysis phase of the software development cycle. It is not uncommon to find specifications for a real-time cyclic executive program included or assumed in such requirements. We have found that preliminary designs structured around this implementation obscure the data flow of the real world system that we are modeling and that it is consequently difficult and costly to maintain, update and reuse the resulting software.

A cyclic executive is a software component that schedules and implicitly synchronizes the real-time software through periodic and repetitive subroutine calls. It guarantees a consistent processing rate but not homogeneous data. Ada tasking on the other hand, can assure the consistency of the data but not a stable execution frequency. Each scheduling paradigm has its disadvantages: race-conditions and maintainability for the cyclic executive, jitter and nondeterminism for Ada tasking.

We therefore seek a design method that allows the deferral of process scheduling to the later stages of design. The designer must be able to chose the appropriate scheduling paradigm given the performance constraints, the target environment and the software's lifecycle. Ada design methods must, in order to support the tasking features of Ada, initially specify the software design as a set of interconnected concurrent sequential processes. They should also provide a verifiable transformation that allocates this design specification to modules based on either a periodic or event-driven scheduling paradigm.

Jackson System Design is an example of such a design method. When desired, it provides the means to transform a specification comprised of many sequential processes into one serial program. Although details of this transformation are subtle, it essentially exploits the subroutine mechanism to suspend and resume the execution of a sequential process.

This paper explores the concept of "process inversion" with respect to the cyclic executive. The transformation has been presented in the literature as a means for more than one process to execute on a single processor. Whenever multitasking is provided, it may be used when analysis shows for example, that the timing uncertainty of Ada tasking is unacceptable for a

certain family of processes. The paper elaborates by example on the subtleties of JSD process inversion with respect to time, and attempts to abstract the technique to other Ada design methods. The paper concludes with an analysis weighing the long term software cost savings of using methods which provide this capablity.

IMPLEMENTATION OF AN ADA* REAL-TIME EXECUTIVE - A CASE STUDY

James D. Laird
Dr. Bruce A. Burton
Mary R. Koppes

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

ABSTRACT

Current Ada language implementations and runtime environments are immature, unproven and are a key risk area for real-time embedded computer systems (ECS). This study provides a test-case environment in which the concerns of the real-time, ECS community are addressed. A priority driven executive is selected to be implemented in the Ada programming language. The model selected is representative of real-time executives tailored for embedded systems used in missile, spacecraft, and avionics applications. An Ada-based design methodology is utilized, and two designs are considered. The first of these designs requires the use of vendor supplied runtime and tasking support. An alternative high-level design is also considered for an implementation requiring no vendor supplied runtime or tasking support. The former approach is carried through to implementation.

* Ada is a Registered Trademark of the U.S. Government (AJPO)

INTRODUCTION

Since the inception of the common DoD High Order Language (HOL) effort in the mid-70's, the Ada programming language has remained a cornerstone of the government effort at producing software in a cost-effective manner. Validated Ada compilers are becoming available on a variety of different computers with at least 17 validated compilers now available and more slated for validation during the current year. There are currently 37 different defense programs using Ada, and this number is anticipated to exceed 120 during the next four years[1]. While this progress is encouraging, the success of the Ada language in meeting the needs of specific applications will hinge on the consideration of the potential risks that face the implementors of a given system.

This process of risk identification should be followed by development of risk minimization and avoidance strategies tailored to meet the needs of the system. The emphasis of this paper is in

the area of technical risk identification and resolution for real-time ECS applications. While the Ada programming language is intended for real-time applications, current compilers and runtime systems are unproven for these types of programming efforts. Consequently, the impact and implications of using the Ada language and Ada-oriented methodologies in embedded real-time development efforts should be assessed. While it is necessary to examine how well and to what extent the built-in real-time features of the language meet the needs of ECS applications, additionally, we must re-evaluate the standard approaches to solving real-time problems in light of the new capabilities and assess the impact, if any, on the way we design and implement these solutions in software.

SCOPE

Perhaps the major consideration with regard to the use of the Ada programming language for real-time ECS applications is the cost of doing so in terms of memory and processing overhead. The relative costs associated with the use of Ada and its real-time features is especially relevant to small embedded computer system applications given the physical and temporal constraints imposed on these types of applications. The determining factor in the decision to utilize a particular high order language (HOL) feature is often the efficiency of its implementation. It is important to know what the utilization of Ada with its real-time tasking primitives, representation specifications, exception handling, and various other features translates to in terms of program size, speed, and efficiency. The ability to selectively include runtime support and its resultant overhead for these features on an "as needed" basis is another important consideration. During the course of this investigation, answers to fundamental questions such as these were sought.

BACKGROUND

It is important to stress the significant conceptual differences between the two approaches investigated with regard to this case study implementation of a priority driven Ada executive. Figure 1 serves to illustrate the alternative approaches and concepts and their implications for the developer of an Ada executive.

The terms O.S., executive, and runtime support or system (RTS) are often used rather loosely when ECS topics are discussed. The ambiguity of this terminology in the ECS environment is primarily due to the overlap in functionality provided by different implementations for different applications. An application residing on a bare machine may interface with software providing minimal scheduling and memory management. This software is often referred to as an "executive" or runtime

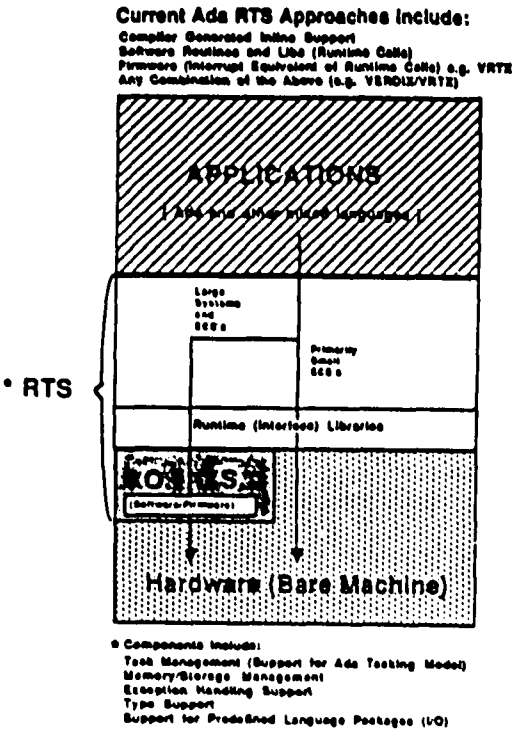


FIGURE 1
RUNTIME SUPPORT (RTS)
APPROACHES

kernel whereas the same services provided on another system may be obtained from software referred to as an O.S. The primary difference in terminology is attributable to the variety and nature of the services provided by the support software in question. The more minimal the services provided, the more likely that the terms runtime support, runtime kernel, or executive will be applied. True operating systems in the strict sense are distinguished by two major factors. They are typically developed independently of any compiler-/applications software and are acquired independently rather than as a part of a given compiler system or package.

The other major distinction is in the comprehensiveness of the services provided by an O.S. for the target machine; services that may be targeted and utilized by a variety of differing applications and tools as well as many different compiler systems. The minimal runtime support for applications developed under a single compiler system may interface to, and utilize, the comprehensive services provided by an O.S. Therefore, the RTS for an ECS can be thought of as providing the minimal required subset of O.S. services needed for a given application. As stated, this minimal subset can be provided by direct access to the underlying machine or through the utilization of the services provided by an underlying comprehensive O.S. The former case is the most typical for embedded computer systems. The term "executive" is most often used to refer to that part of the RTS that performs the basic scheduling and memory management. Other portions of the RTS may include I/O control, timer/clock management, and a certain amount of system level runtime error and interrupt trapping.

The RTS of an ECS supports the execution of application programs and the programming language features utilized to develop those programs. As illustrated in Figure 1, this support can be implemented in hardware, microcode, through direct calls to an O.S., through the use of runtime support libraries, or by compiler

generated (in-line) code. The operating system and RTS needs of small embedded computer systems are typically modest. All that such small ECS targets usually require is an "executive" consisting of little more than a basic scheduler, memory manager and some type of I/O manager or controller. Obviously, different applications may have specific needs relative to memory management, I/O, or clock services which will be reflected in the "executive/-O.S" software.

APPROACH

This paper addresses two basic options or approaches to the implementation of an Ada executive and briefly discusses ongoing as well as proposed work in a third area of related investigation. The first of these approaches is explored in depth (through to implementation) and consists of a combination of a "pseudo executive" or scheduler at the applications layer in concert with vendor supplied executive software at the runtime system level. The obvious benefits of such an approach - imposing an additional layer of control upon the runtime system scheduling mechanism - include ease of portability, and relative target independence with respect to the underlying scheduling algorithm at the RTS layer. These benefits as well as the tradeoffs in overhead and consistency from implementation to implementation will be discussed in detail.

The second option is explored at a high level only. This alternative, termed the bare machine approach, is consistent with the traditional approach to avionics-based executives and is considerably more limited in scope than the first in the sense that it assumes no underlying vendor supplied runtime support. This executive performs all necessary support for the execution of user jobs or "tasks". However, this approach is significantly more restrictive than the first with respect to the nature of what constitutes a "task" as well as to the use of certain Ada language features involving both the Ada tasking model and dynamic memory management and certain other real-time aspects of the language.

The third option is considered only in terms of current and ongoing investigative work and proposed future studies based upon the results of past investigations. This approach diverges from the others in that it proposes a migration to the runtime system layer in order to probe the issues of efficiency and risk reduction for real-time Ada applications. This option emphasizes the tailoring and optimization of the executive functions provided at the RTS layer.

A multi-phased approach beginning with a requirements specification was utilized for the design and development of the priority driven executive. The functional capabil-

ities that were to be provided were extracted from an existing avionics executive implemented in a combination of FORTRAN and Assembly language. It was determined that these same functional capabilities would be provided within the executive being implemented in the Ada language.

While providing substantially the same functionality, the Ada equivalent constituted a complete re-design utilizing Ada concepts and features where possible. For this reason, the Ada executive posed some unique problems from the outset with respect to use of the new Ada concepts and features such as the Ada tasking model. These issues are addressed in the RESULTS section of this paper.

The Ada priority driven executive was to provide facilities for the creation of active tasks via a scheduling mechanism. The scheduling mechanism would provide time-dependent scheduling capabilities, precision timing of task activation as measured by time base generated (TBG) epochs, and signal dependent scheduling capabilities. The Ada priority driven executive would perform prioritized tasking and would have the option of enabling and disabling interrupts. The capability to directly connect to a real-time clock interrupt would be provided. In the absence of such a facility, the real-time clock interrupt would be simulated with the smallest granularity possible. In short, the Ada priority driven executive was required to be a real-time, multi-tasking process manager with interrupt handling and both cyclic and asynchronous scheduling capability.

Integral to the design of the Ada priority driven executive was the selection and application of a state-of--the-art, Ada-based design methodology. A somewhat novel design approach was selected that was based upon Object Oriented Design[2] with enhancements and modifications specific for real-time embedded systems[4]. The methodology derived was termed Real-Time Object Oriented Design (RTOOD) and drew upon another real-time, systems--based design methodology called Design Approach for Real-Time Systems (DARTS)[5]. The steps utilized in this hybrid methodology are outlined in Figure 2.

I. Definition/statement of the problem

II. Informal strategy (Modified specification)

III. Identify objects and attributes

IV. Identify Operations

V. Identify concurrency *(DARTS)

Decomposition into tasks/packages based on:

The asynchronous nature of major transforms
- sequential vs. concurrent

specifically:

Vo dependency
time critical functions
computational requirements
function cohesion
temporal cohesion
periodic execution

VI. Establish the interfaces

VII. Implement the operations

* (DARTS)
Design Approach for
Real-Time Systems

FIGURE 2
REAL-TIME OBJECT ORIENTED DESIGN (RTOOD) METHODOLOGY

Similarly, a high level design was developed for the alternate approach - termed here "the bare machine approach" - to the development of an Ada executive. The "bare machine" model implements its own concurrency through the executive while disallowing the use of the Ada tasking model per se as well as any difficult, and potentially risk-prone, dynamic storage management. The potential benefits and risks of each of these approaches was examined with the former approach being carried through to implementation and limited utilization.

RESULTS

I. ADA EXECUTIVE WITH VENDOR RUNTIME SUPPORT

The capabilities of the FORTRAN/Assembly language implementation and the Ada language implementation are summarized in Table 1. The Ada language version consists of two major components - the program code and the vendor supplied runtime system. In both implementations the scheduling primitives are provided by the executive, but the ultimate responsibility for cyclic/acyclic task scheduling lies with the user (application) tasks. Note, however, that the task interleaving and task waiting in the Ada language version is strictly under the control of the Ada runtime system and not under the control of the executive as in the FORTRAN/-Assembly implementation. Furthermore, although tasking could be prioritized dynamically (changed) in the FORTRAN/Assembly implementation, priorities at the runtime system level are static in the Ada language version.

Functional Summary

Figure 3 depicts the major functional components of the Ada equivalent prototype developed for the case study investigation. The major distinction between the Ada implementation and the FORTRAN/Assembly model depicted in Figure 4 involves the interaction of the Ada runtime system with the priority driven executive functions.

TABLE 1 FORTRAN/ASSEMBLY VERSUS Ada IMPLEMENTATION

CAPABILITY	FORTRAN/ASSEMBLY EXECUTIVE	Ada EXECUTIVE	Ada RUNTIME SYSTEM
CYCLIC/ACYCLIC TASK SCHEDULING	Provided	Provided	
TASK DISPATCHING	Provided	Provided	
TASK INTERLEAVING	Provided		Provided
TASK WAIT	Provided		Provided
PRIORITIZED TASKING	Provided	Provided	Provided
TBD INTERRUPT HANDLING	Provided	Provided	

While the FORTRAN/Assembly model managed all state transitions for user tasks from inactive to executing and all information associated with these state transitions, the Ada implementation utilizes the Ada runtime support system (for the tasking model) to manage the active processing phase of any user task as well as the body of information associated with a tasks' active execution. Specifically, the Ada runtime

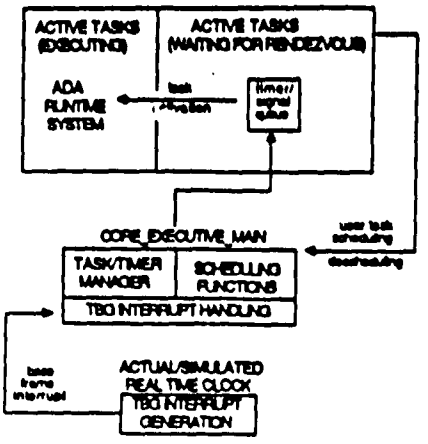


FIGURE 3
ADA PRIORITY DRIVEN
EXECUTIVE FUNCTIONAL
SCHEMATIC

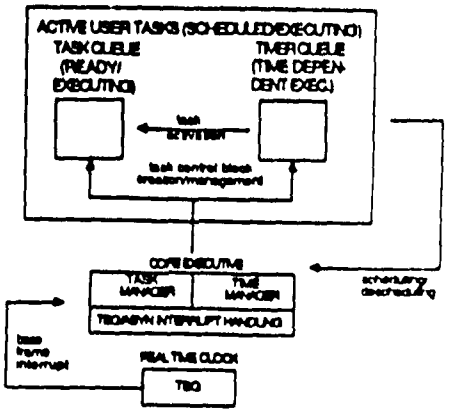


FIGURE 4
FORTRAN/ASSEMBLY
EXECUTIVE FUNCTIONAL
SCHEMATIC

system manages the interleaving or time-slicing of concurrently executing user tasks and is responsible for management of the associated task activation information. The start of a user tasks' scheduled execution phase is strictly under the control of the Ada priority driven executive at the applications layer, yet, the management of the transfer of control between any number of concurrently executing user tasks is by definition under the control of the vendor supplied Ada runtime system.

To satisfy the requirement for a cyclic capability, the executive was required to have some method for specifying fixed-rate scheduling. This was provided on two levels. In keeping with the scheme utilized in the original model, the facility for scheduling a task for execution is provided. Active tasks currently executing may therefore utilize this facility to re-insert themselves into the schedule for future execution, or this may be done by some other active user task.

In the original model a voluntary, non pre-emptive scheduling scheme was utilized among the user tasks that enforced the notion that no transfer of control or context switching among tasks could occur unexpectedly. Bearing in mind that within an Ada environment the underlying operating or runtime system utilizes another level of scheduling for the interleaving of currently active tasks, a task prioritization scheme among these tasks is then required to enforce the notion that a particular task is incapable of having its scheduled execution interrupted once it begins.

In short, we have a scheduling scheme at the user task level to specify fixed--rate triggering of a tasks' processing and the Ada pragma "PRIORITY" enforced at the underlying operating or runtime system level to ensure uninterrupted completion of that processing.

The major potential point of failure with respect to this type of approach to task scheduling at the applications level is at the underlying runtime system level. The issue is one of consistency from implementation to implementation with respect to time slicing of concurrently executing processes of equal priority. While fixed rate triggering of task execution can be guaranteed via a combination of algorithmic control, prioritization, and interrupt handling through the "psuedo executive", no such guarantee can be made with respect to the method of time slicing utilized by the underlying runtime support for concurrent tasks of equal priority. This will vary from implementation to implementation although adhering to the so-called "FAIR" requirement dictated by the language specification. Given the stringent nature of typical ECS performance and reliability requirements, this potential inconsistent behavior across implementations could pose a significant risk.

Static prioritization of Ada tasks may be a problem in some instances of task scheduling or interrupt handling since external events often dictate a need to dynamically change priorities. The Ada rendezvous occurs in a first in, first out manner using a queue structure for multiple entry calls issued for any given task entry point (ACCEPT statement). There is no way to reorder and influence the position a calling task may occupy in such a queue. It is possible that with dynamic task prioritization this could be programmer controlled.

Efficiency: Space and Time The FORTRAN/ Assembly language implementation used as a model in this case study was coded in a little over 1 K (bytes) of memory and accounted for somewhat less than two percent of the entire system. While the entire Ada system consisted of just over 700 lines of code, the space requirements varied with respect to the host machine. The Ada version required anywhere from 27 K to 38 K bytes of memory for the applications code alone. The runtime kernel on one machine imposed an additional penalty of 200 K bytes to utilize the Ada tasking model. It should be noted, however, that the executive was developed for functional realism and was not optimized for minimal program size. The runtime kernels were large, as much as 200K bytes, but the runtime kernels were intended for a main-frame environment, not a typical ECS application.

The significant lessons learned were in what options were available to optimize the size and speed of the execu-

table image. Significant savings - approximately 100K - were available via a selectively loadable tasking kernel in at least one implementation while other options resulting in savings were no runtime checking (1-2K savings), and no debugging instrumentation (5K savings). In one particular implementation, the option for space optimization was offered yet yielded no appreciable difference in the size of the executable image.

While there is no strict linear relationship with respect to overhead between host and ECS environments, the significant savings realized through configurability within the host environments has significant positive implications for ECS environments where efficiency constraints are paramount.

It was found that the total storage penalty to include a minimal exception handling capability within each Ada program unit was on the order of 4-5 percent of the total program storage while the cpu overhead to invoke an exception handler ranged from 30-500 microseconds. This represents an acceptable cost in either a host mainframe or embedded environment.

The overhead in terms of time to utilize the rendezvous mechanism within the host environment was rather high, being approximately 11-12 milliseconds. Given the relatively rapid frame times of many real-time applications (on the order of 40-100 milliseconds), a feature that uses approximately one tenth of the frame time poses serious risk[3]. However, based upon current investigations with Ada for embedded 16 and 32 bit targets, the case can be made that this is a problem somewhat localized to the mainframe environment.

II. THE BARE MACHINE APPROACH

The alternate design approach proposed in this study for the Ada priority driven executive (see Figure 5) is intended for a bare machine environment with no resident operating system nor any vendor supplied Ada runtime support. The design of such an executive raises some important issues with respect to validation when considering what must be provided to support the execution of an Ada application on such a bare target. The implications of the traditional model of an executive, such as the original FORTRAN/Assembly language implementation used as a basis for this study, are considered.

This approach differs greatly from that which utilizes an underlying runtime system. This approach implies that beyond the generation of native machine instructions from the HOL by some generic translator or compiler, it becomes necessary to provide programmer supplied support for any HOL language features not directly implementable through primitives on the bare hardware. It therefore becomes the task of the

runtime supervisor or executive software to provide this underlying support for things such as concurrency or multi-tasking, I/O, dynamic storage and memory management to name a few. In addition, this executive must not, in turn, rely on some underlying support for its own execution.

Validation is certainly an issue with respect to this kind of subset Ada approach. While recognizing the incompatibility of this approach with the notion of validation, we choose not to address the topic in any detail other than to acknowledge the conflict. Our focus is on technical risk identification and minimization.

The design of this bare machine executive was purely hypothetical and no specific embedded target was selected. For that reason, only a high-level design was iterated. Currently, typical vendor supplied Ada runtime support packages facilitate things such as: system elaboration or initialization, task communication and scheduling, exception handling, interrupt, I/O, and type support. The amount of overhead varies with each vendor's implementation. The design proposed is for an Ada executive function that would minimally support the execution of other Ada software constituting jobs or "tasks". However, the Ada tasking model is not supported by the proposed subset Ada implementation for a bare ECS target.

As in the traditional model, concurrency is achieved via the executive utilizing a non pre-emptive, voluntary context switching mechanism. Control over scheduling is therefore explicit and known to the programmer. In addition, any dynamic data or storage management is restricted to that which supports the execution of the executive functions only.

It must be noted that the notion of an "all Ada executive" at this level is fallacious. A certain amount of privileged accessing of register and stack contents by the executive functions to facilitate the basic context switching and memory management would be required. This is not directly achievable from within the Ada language. Therefore, a component of the executive software (e.g. the Control_Transfer_Package) would by necessity be implemented in a lower level programming language. In current commercial Ada runtime systems for embedded targets such as the 1750A, this accounts for approximately two percent of the vendor supplied runtime support. Ada packaging concepts facilitate the encapsulation and isolation of such machine context sensitive components.

The rationale for the approach to concurrency presented is straightforward. While explicit context switching can be considered risky, it has certain potential benefits. It avoids the necessity of excessive locking since the programmer knows

exactly when context switches are to be performed. Another benefit is realized when a high priority event occurs that must be handled rapidly as is the case in many real-time systems. While handling such an event, it may be deleterious to release the processor. Finally, the avoidance of unnecessary context switches and/or checking results in greater efficiency[8].

Admittedly, however, it is reasonable to question the feasibility and advantages of using Ada without its tasking features and other real-time components versus using any other high-level programming language. It should also be noted that, with some re-working of the design, there is nothing to explicitly prevent the use of the Ada tasking model and rendezvous concept, provided that the necessary runtime support is supplied at an acceptable cost in memory overhead and execution efficiency. This is the motivating concept driving our current and future investigations with respect to Ada real-time systems and will be discussed in the following section.

<u>Current and Future Investigations</u> The rationale for an approach such as the bare machine option is that given the present state of tasking support in an environment that supports full Ada tasking, exception handling and other HOL features, the resultant program size may be unsuitably large for an embedded application. While the applications level strategy and the bare machine approach represent two available options, an additional alternative exists that holds some promise for the design of compact, efficient real-time systems and is the focus of our current and future investigative work. This consists of a migration to the RTS layer in pursuit of optimization and risk reduction at this level while maintaining the complete (or nearly complete) functionality of the language. The focus is on tailorable, configurable runtime support for the design of efficient real-time systems in Ada.

It is highly likely that the full functionality of the traditional model of a priority driven executive can be achieved in this manner by minimizing the role of a programmer supplied executive and relying on the efficient implementation of the Ada tasking model at the operating or runtime system level. While it may still be necessary to provide customized runtime/executive support, this can be provided primarily through tailoring of existing systems at the RTS level to meet specific performance requirements rather than exerting additional control at the applications layer.

Our current efforts are focused foremost on proof of concept - that we can design and implement fast, compact, efficient, real-time systems in Ada - with a secondary emphasis on the validation issues. The steps we have identified as being necessary

to the success of this effort include:

- Obtain Valided Vendor Supplied RTS
- Maintain Stable RTS Interface
- Modify Internals to gain Required Performance
- Address (Re) Validation issues

CONCLUSION

Although several of the issues that face developers of real-time ECS applications in Ada are design issues or primarily resolved through education, training and good programming technique, many issues remain that pose risk to the development of real-time systems in Ada.

We have identified a number of key risk areas and issues for real-time ECS applications and have explored these issues, and solutions, within the context of a specific Ada language application. With respect to the issues that were successfully addressed within the scope of this case study, the following conclusions can be made.

Many issues of concern exist due to the immaturity and quality of Ada language implementations and uncertainties regarding performance. The performance of the code generated by early compilers may be poor and may result in poor system performance. However, as Ada language systems mature and currently available optimizing technology is employed, large runtime overhead with respect to memory utilization and execution speed should certainly become less of an issue. This is in fact the case with some of the Ada language systems currently under development.

Current investigations with a variety of differing compiler systems and runtime environments for 16 and 32 bit embedded targets have revealed that kernel runtime systems currently exist that appear to be providing the minimal, configurable support necessary to accommodate Ada language features in a timely and efficient manner. Standardized kernel runtime support on the order of 2K provided by minimal system service interfaces is currently available (e.g. VRTX) and can be targeted and utilized efficiently by Ada compiler systems for a variety of embedded targets.

Problems remain with the non-support among many Ada implementations of certain real-time features of the Ada language. A case in point is the vectoring of interrupts to task entries via the Ada representation specification. This continues to be a concern to the real-time applications community although it is somewhat localized to the mainframe environment. Additional problems are rooted in the language specification itself (MIL STD 1815A) which fails to provide certain features desirable in typical real-time systems.

While alternatives exist, this lack of certain explicit language primitives poses unique problems for many types of real-time applications. Specifically, the lack of explicit language primitives to allow dynamic "disconnection" and "connection" to interrupts without the termination or creation of a program unit (task) and the inability to utilize dynamic task prioritization are of major concern to ECS developers. Furthermore, the lack of precision in the specification of exact delays as well as the lack of alternatives or ability to time-out during initiated rendezvous' may be an impediment to the development of efficient, reliable real-time systems in Ada.

There is a continuing need for a clear, concise design methodology for real-time embedded Ada applications that includes a criteria for the identification of concurrency and a graphic means of depicting concurrent relationships with timing and synchronization information at any given point in the system. While helpful, the hybrid method utilized during this case study falls short of fulfilling such a broad requirement.

We are currently continuing our real-time investigations to evaluate the effectiveness of Ada language systems for real-time embedded applications within realistic host and target environments. This work is being carried out with a focus on the 1750A and 68000 compiler and runtime environments.

The focus of our initial case study was at the applications level although an alternative was proposed for a prohibitively restrictive Ada executive that fulfilled a subset of the runtime responsibilities to support the execution of concurrent Ada programs. The current approach calls for migration to the RTS level to investigate optimization and tailoring of existing systems to allow efficient use of the Ada tasking model and other real-time features within realistic target environments. It is in this manner that we will attempt to address and seek additional information and solutions to those issues left unanswered in our preliminary Ada real-time investigations.

Options for future efficiency improvement and risk-reduction include:

- Highly Configurable Runtime Support Systems
- Standardized Runtime Support Systems
- Support in Silicon
- Custom RTS Components Libraries

ACKNOWLEDGMENTS

The authors wish to acknowledge the support and advice of the personnel at Intermetrics, Inc. in the preparation of this manuscript.

REFERENCES

1. Judge, J.F., "Ada Progress Satisfies DoD", Defense Electronics, June 1985.

2. Booch, Grady, Software Engineering with Ada, Benjamin/Cummings, Menlo Park, California, 1983.

3. Davis, R., "FDA Program Conclusions", Intermetrics Inc., Huntington Beach, California, August, 1985.

4. Laird, James D., "Implementation of an Ada Real-Time Executive: A Detailed Analysis", Intermetrics Inc., Huntington Beach, California, March, 1985.

5. Gomaa, H., "A Software Design Method for Real-Time Systems", Communications of the ACM, Vol. 27, No. 9, September 1984.

6. Temte, Mark, "Object Oriented Design and Ballistics Software", ACM Ada Letters, Vol. IV, No. 3, November/December, 1984.

7. United States Department of Defense, Reference Manual for the Ada Programming Language MIL STD 1815A, Ada Joint Program Office, March, 1983.

8. Binding, Carl, "Cheap Concurrency in C", ACM SIGPLAN NOTICES, Vol. 20, No. 9, September 1985.

(Informal Presentation)

Real Time Ada in a MC68XXX System

Dick Naedel, President
Intellimac
Rockville, Maryland

This presentation will present recent results of running Ada programs in a Motorola based embbedded computer system.

OBJECT-ORIENTED DEVELOPMENT

by

Donald G. Firesmith
Software Methodologist

Magnavox Electronic Systems Company
Advanced Software Systems Division
1313 Production Road
Fort Wayne, IN 46808
(219) 429-4327

1) WHY IS OBJECT-ORIENTED DEVELOPMENT (OOD) IMPORTANT?

OOD is one of the extremely few software development methods actually designed for modern Ada (*) language, real-time, embedded applications.

OOD is a significant improvement over more traditional functional decomposition and modeling methods in that OOD:

a) Better manages the size, complexity, and concurrancy of today's systems.
b) Better addresses important software engineering principles such as abstract data types, levels of abstraction, and information hiding.
c) Produces a better design that more closely matches reality.
d) Produces more maintainable software by better localizing data and thus limiting the impact of requirements changes.
e) Specifically exploits the power of Ada.

2) WHAT IS OOD?

OOD is a systematic, step-by-step software development method that:

a) Has an optimal domain of application -- the development of modern Ada applications (e.g., real-time, embedded software).
b) Covers all, or a major portion, of the software life-cycle.
c) Supports extensive parallel development.
d) Manages the complexity of large development efforts.
e) Is supported by detailed standards and procedures.
f) Requires training and support to be effective.

OOD is:

a) Object-oriented.
b) Ada-oriented.

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

c) Based upon modern software engineering.
d) Recursive, globally top-down, hierarchical COMPOSITION method.
e) Revolutionary in approach.
f) A "grab and go" method.
g) Relatively easy to learn.
h) Being successfully used by several companies.
i) Still evolving (see Figure 1).

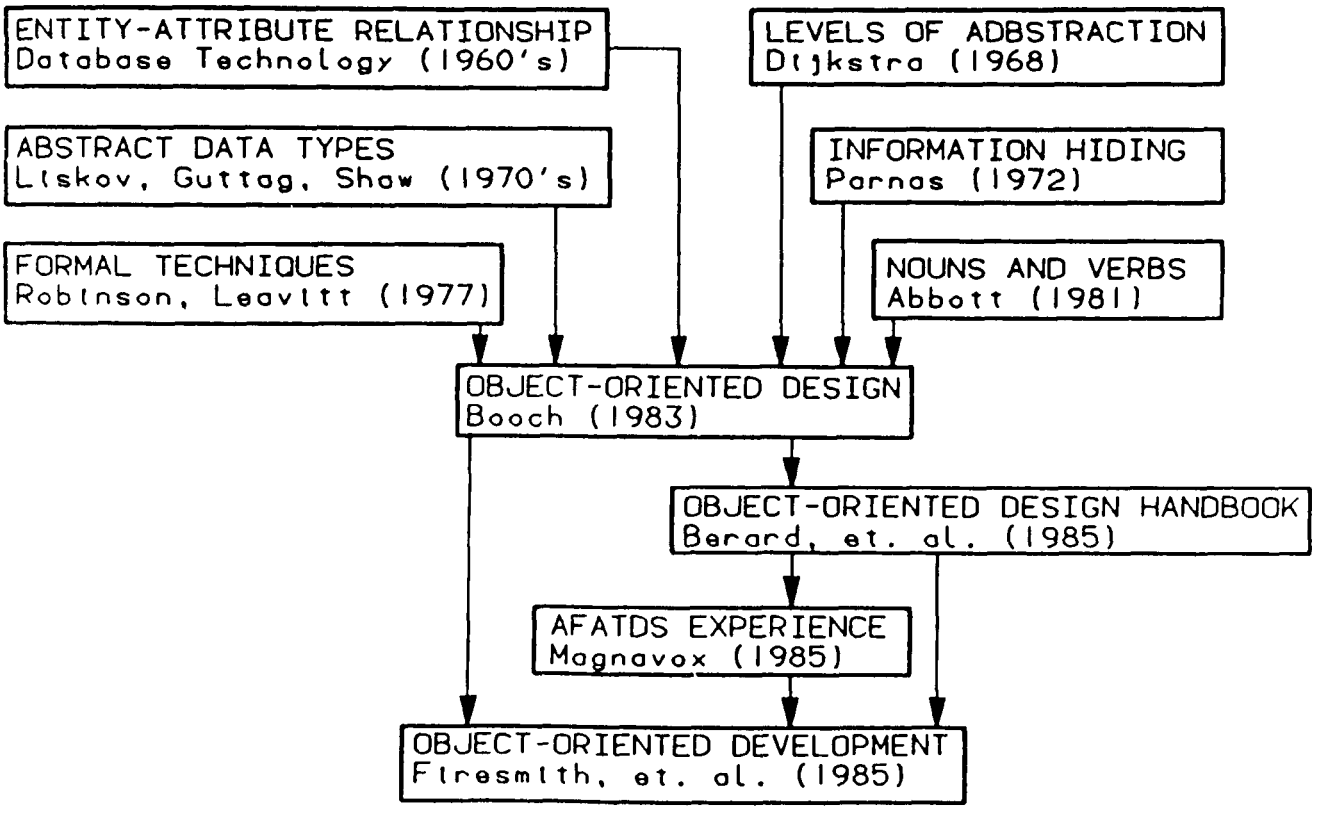


Figure 1: The evolution of OOD

OOD is NOT:

a) A functional, hierarchical DECOMPOSITION method.
b) A modeling method.
c) Easily mated with such methods.
d) Effective without adequate training.
e) Constrained to the classical "waterfall" lifecycle.
f) Consistent with DOD-STD-2167 and related pre-Ada standards.
g) Standardized across the industry.
h) Yet adequately supported by commercially available software tools.

3) OOD IS OBJECT-ORIENTED.

An OBJECT is an entity that:

a) Has a value (e.g., data) or state (e.g., Ada task).
b) Suffers and/or causes operations.

OOD produces:

a) Ada objects that correspond to objects in the real world.
b) Ada types (i.e., object templates).
c) Operations that operate on these objects.

OOD emphasizes the implementation of objects in terms of ABSTRACT DATA TYPES. OOD groups, in the same Ada package:

a) A single type and
b) All operations that operate upon such objects.

OOD produces a substantially different software architecture than traditional functional decomposition methods such as Structured Design which generate units, each of which implements some FUNCTION of the requirements specification.

4) OOD IS ADA-ORIENTED.

Ada is an object-oriented high-level language.

Packages, which are the main building blocks of properly designed Ada software, are also the main building blocks produced by OOD.

The physical design produced by OOD is top-down in terms of Ada:

a) Nesting and
b) Context (i.e., the Ada "with" statement).

OOD separately develops Ada specifications and bodies.

OOD's low-level testing naturally accounts for Ada compilation order constraints.

OOD Diagrams clearly identify the various Ada programming units.

Ada PDL is an integral part of OOD's design and coding steps.

The objects produced by OOD are implemented in Ada as:

a) Constants and variables
b) Abstract data types
c) Tasks

The operations produced by OOD are implemented in Ada primarily as:

a) Subprograms
b) Task entries

5) OOD IS BASED UPON MODERN SOFTWARE ENGINEERING.

OOD specifically addresses each of the following software engineering principles or concepts:

a) ABSTRACT DATA TYPES.
b) ABSTRACTION LEVELS.
c) Cohesion.
d) Concurrency.
e) Coupling.
f) INFORMATION HIDING.
g) Localization.
h) MAINTAINABILITY.
i) MODULARITY.
j) Organizational Independence.
k) Readability.
l) Reusability.
m) Structure.
n) Testability
o) Verifiability.

6) OOD IS RECURSIVE, GLOBALLY TOP-DOWN, HIERARCHIAL COMPOSITION METHOD.

Traditional software development methods are restricted to the classic "waterfall" life-cycle (see Figure 2) in which:

a) The software requirements are analyzed first.
b) The preliminary design is developed second.
c) The detailed design follows.
d) And so on.

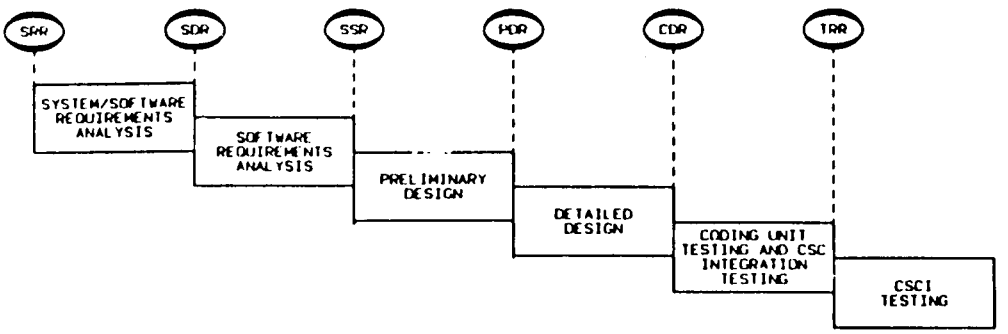


Figure 2: The classic "waterfall" life-cycle

Because the software is developed in a top-down manner only within the boundries of each life-cycle phase, these methods are at best only locally top-down.

OOD is a recursive, globally top-down, hierarchial composition method. Its software life-cycle (see Figure 3) differs significantly from the classic "waterfall" life-cycle because it is based upon recursion and two concepts unique to OOD: the Booch and Subbooch.

A BOOCH is the collection of all software resulting from the recursive application of OOD to a specific set of coherent software requirements -- requirements that specify a single well-defined problem.

A SUBBOOCH is a small, managable subset of a booch that is identified and developed during a single recursion of OOD. See Figure 4.

Note that these two concepts have no obvious natural relationship to the DoD hierarchical decomposition entities CSCI, TLCSC, and LLCSC.

Beginning with the highest abstraction level and progressing steadily downwards in terms of nesting and "withing", the booch is designed, coded, and tested in increments of a subbooch. Thus, the software grows top-down, subbooch by subbooch, via the recursive application of OOD until the entire software tree is completed.

Locally, however, OOD employs the appropriate technique (top-down or bottom-up) depending upon the specific requirements of each individual development activity.

This allows very significant parallel development based upon the "Design a little, code a little, test a little" concept.

7) RESPONSIBILITIES.

The following personnel have OOD responsiblities (see Figure 5):

a) Management
b) Software Development Teams, each consisting of a:
   - Designer
   - Programmer
   - Tester
c) Metrics Collectors
d) Software Quality Evaluation
e) Software System Engineering

8) SUBBOOCH DEVELOPMENT

The subbooches that comprise each booch are recursively developed in a globally top-down fashion. The development of each subbooch consists of the following three subphases:

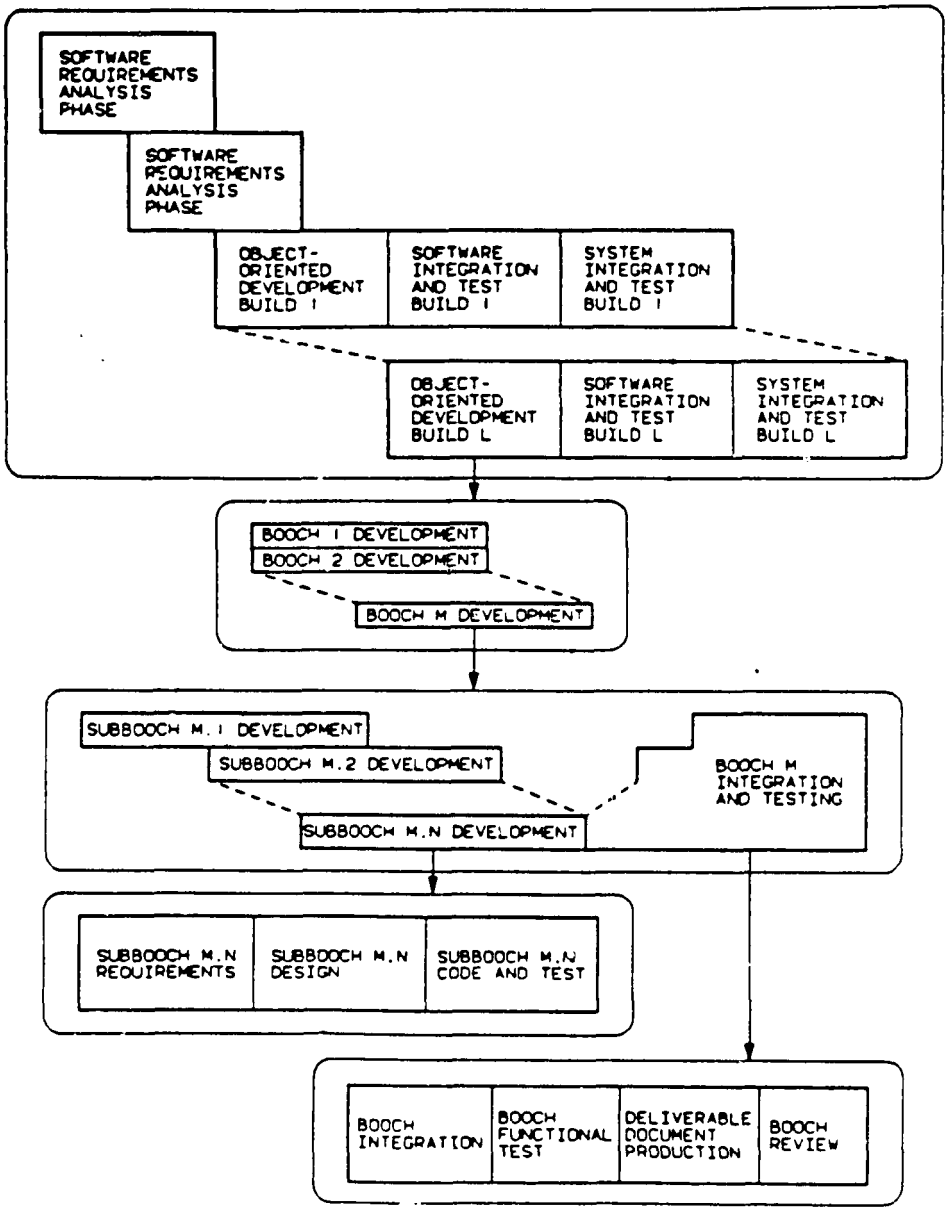


Figure 3: The OOD software life-cycle

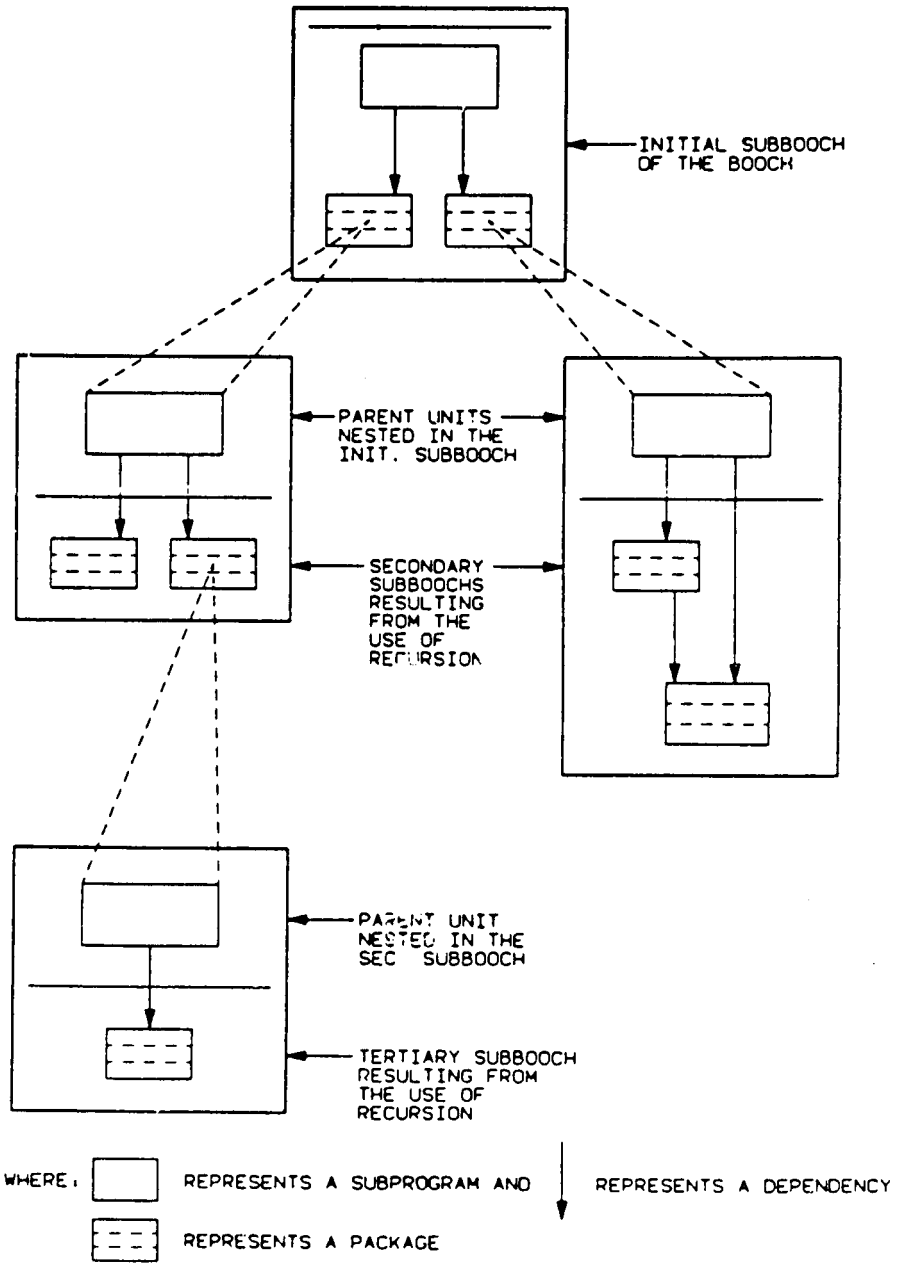


Figure 4: Sample Booch structure

Step	Object-Oriented Development Process Title	MGMT	Software Dev. Team D	P	T	MC	SQE
1	INITIATION OF BOOCH DEVELOPMENT	1					4
2	SUBBOOCH DEVELOPMENT						
2.1	SUBBOOCH REQUIREMENTS SUBPHASE						
2.1.1	Initiation of Subbooch Development	1					4
2.1.2	Initiation of the SDF	3	1				4
2.1.3	Problem Statement	3	1	2	2		4
2.1.4	Requirements Analysis	3	1	2	2		4
2.1.5	Subbooch Requirements Inspection	1	2	2	2		4
2.2	SUBBOOCH DESIGN SUBPHASE						
2.2.1	Logical Design	3	1	2	2		4
2.2.2	Object Analysis	3	1	2	2		4
2.2.3	Operation Analysis	3	1	2	2		4
2.2.4	Unit Id., Org., and Dependencies	3	1	2	2		4
2.2.5	Subbooch Preliminary Design Inspection	3	2	1	1		4
2.2.6	Design Analysis	3	1	2	2		4
2.2.7	Coding of Unit Specifications	3	1	2	2		4
2.2.8	Subbooch Detailed Design Inspection	3	2	1	2	1	4
2.3	SUBBOOCH CODE AND TEST SUBPHASE						
2.3.1	Coding of Unit Bodies	3	2	1	2		4
2.3.2	Subbooch Test Plan	3	2	2	1		4
2.3.3	Subbooch Test Software	3	2	2	1		4
2.3.4	Subbooch Test Procedures	3	2	2	1		4
2.3.5	Subbooch Code Inspection	3	1	2	2	1	4
2.3.6	Initial Subbooch Testing	3	2	2	1		4
3	BOOCH INTEGRATION AND TESTING						
3.1	BOOCH INTEGRATION	3			1		4
3.2	BOOCH FUNCTIONAL TESTING	3			1		4
3.3	BOOCH DELIVERABLE DOCUMENTATION	2	1	1	1		4
3.4	BOOCH REVIEW	1	2	2	2	1	1

MGMT = Management
D = Designer(s)
P = Programmer(s)
T = Tester(s)
MC = Metrics Collector(s)
SQE = Software Quality Evaluation

1 = Primary or major responsibility
2 = Secondary responsibility
3 = Managerial responsibility
4 = Independent audit responsibility

FIGURE 5: OOD Responsibilities

a) Subbooch Requirements.
b) Subbooch Design.
c) Subbooch Code and Test.

The SUBBOOCH REQUIREMENTS SUBPHASE has the following steps:

INITIATION OF SUBBOOCH DEVELOPMENT - The Manager initiates subbooch development by identifying the members of the associated Software Development Team and tasking them to meet an assigned schedule of subbooch milestones.

INITIATION OF SOFTWARE DEVELOPMENT FILE (SDF) - The Designer initiates the associated SDF by obtaining an empty SDF binder and inserting the initial Software Engineering Forms (SEFS) that make up the coverpages.

PROBLEM STATEMENT - The Software Development Team jointly state in a single sentence the problem to be solved during the current recursion.

REQUIREMENTS ANALYSIS - The Software Development Team jointly collect, analyze, clarify, organize, and identify the subbooch requirements.

SUBBOOCH REQUIREMENTS INSPECTION - The Designer prepares the SDF for inspection. The Manager schedules the associated meeting. The Manager, the Programmer, and the Tester perform the inspection. The Software Development Team takes any appropriate corrective action.

The SUBBOOCH DESIGN SUBPHASE has the following steps:

LOGICAL DESIGN - The Software Development Team (under the leadership of the Designer) develops in a single paragraph a logical design that properly solves the problem of the current recursion and identifies the relevant objects and operations.

OBJECT ANALYSIS - The Software Development Team (under the leadership of the Designer) analyzes all relevant objects in the logical design paragraph, determines and documents their relevancy, and provides the relevant objects with valid Ada identifiers, brief descriptions, and a list of associated attributes.

OPERATION ANALYSIS - The Software Development Team (under the leadership of the Designer) analyzes all relevant operations in the logical design paragraph, determines and documents their relevancy, and provides the relevant operations with valid Ada identifiers, brief descriptions, and a list of associated attributes.

MODULE IDENTIFICATION, ORGANIZATION, AND DEPENDENCIES - The Software Development Team (under the leadership of the Designer) organizes all relevant objects and operations

by types, identifies the non-nested units for each such organization, nests the organized objects and operations within these units, and determines the visible dependencies between these units.

SUBBOOCH PRELIMINARY DESIGN INSPECTION - The Designer prepares the SDF for inspection. The Programmer and Tester perform the Inspection. The Software Development Team takes any appropriate corrective action.

DESIGN ANALYSIS - The Software Development Team (under the leadership of the Designer) analyzes the design, identifies the type of the nested units, common software, and nested units requiring recursion, etc.

CODING OF UNIT SPECIFICATIONS - The Software Development Team (under the leadership of the Designer) implements and compiles, in a bottom-up manner in terms of unit dependencies, the Ada specifications of all units. This includes the development of specification headers, PDL, comments, and code from skeleton unit specifications.

SUBBOOCH DETAILED DESIGN INSPECTION - The Designer prepares the SDF for inspection. The Metrics Collector collects, summarizes, and reports the subbooch design metrics. The Programmer and Tester perform the Inspection. The Software Development Team takes any appropriate corrective action.

The SUBBOOCH CODE AND TEST SUBPHASE has the following steps:

CODING OF UNIT BODIES - The Software Development Team (under the leadership of the Programmer) implements and compiles, in a top-down manner in terms of unit dependencies, the Ada bodies of all units to be implemented during the current build. This includes the development of body headers, PDL, comments, and code from skeleton unit bodies using the technique of step-wise refinement.

SUBBOOCH TEST PLAN - The Software Development Team (under the leadership of the Tester) develops the test plan by determining, creating files of, and documenting the test input and expected test output data required for all subbooch testing and documenting the allocation of these test cases to specific subbooch tests.

SUBBOOCH TEST SOFTWARE - The Software Development Team (under the leadership of the Tester) designs, implements, and compiles all test software programs required for subbooch testing scheduled for the current build.

SUBBOOCH TEST PROCEDURES - The Software Development Team (under the leadership of the Tester) develops the detailed step-by-step procedures for performing all subbooch tests scheduled for the current build.

SUBBOOCH CODE INSPECTION - The Programmer prepares the SDF for inspection. The Metrics Collector collects, summarizes, and reports the subbooch code metrics. The Software Development Team perform the inspection. The Software Development Team takes any appropriate corrective action.

INITIAL SUBBOOCH TESTING - The Software Development Team (under the leadership of the Tester) perform and document the results of all initial subbooch tests.

9) PRACTICAL EXPERIENCE.

The use of OOD at Magnavox on the AFATDS Project (over 50K lines of Ada code so far) has resulted in the following lessons learned:

a) Avoid overspecifying the requirements with explicit or implicit design information of a functional decomposition nature.

b) If a functional decomposition method is used to produce the top-level design, it will be incompatible with the design produced by OOD at the lower-levels and numerous interface problems will result.

c) Replacing the previous functional decomposition mindset is difficult, primarily among the more experienced designers.

d) The concept of recursion is fairly difficult to master.

e) OOD training and support in the method needs to continue beyond the classroom.

f) OOD needs to be further refined, primarily in the area of object-oriented requirements analysis.

g) Ada-oriented test training is as necessary as training in Ada-oriented design and programming.

h) OOD improves designs due to:

- Proper abstraction levels.
- Proper information hiding.
- High modularity.
- Improved interfaces.
- Good support for strong typing.
- Good correspondance to the real world.

i) OOD improves productivity due to:

- Enhanced parallel development.
- Reuse of code.
- Easy coding from design information.
- Easy modification of design and code.