TR 203
N83

STEPS TO AN ADVANCED ADA PROGRAMMING ENVIRONMENT

Richard N. Taylor
Thomas A. Standish

Reprinted from IEEE 1984 PROCEEDINGS OF THE
7TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING

IEEE COMPUTER SOCIETY REPRINT

IEEE COMPUTER SOCIETY
1109 Spring Street, Suite 300
Silver Spring, MD 20910

IEEE
COMPUTER
SOCIETY
PRESS

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS INC

TR 203

ARCHIVES
Z
699
C3
no. 203

Steps to an Advanced Ada* Programming Environment

*Richard N. Taylor and Thomas A. Standish***

Programming Environment Project
Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717 U.S.A.

ABSTRACT

Conceptual simplicity, tight coupling of tools, and effective support of host-target software development will characterize advanced Ada programming support environments. Several important principles have been demonstrated in the Arcturus system, including template-assisted Ada editing, command completion using Ada as a command language, and combining the advantages of interpretation and compilation. Other principles, relating to analysis, testing, and debugging of concurrent Ada programs, have appeared in other contexts. This paper discusses several of these topics, considers how they can be integrated, and argues for their inclusion in an environment appropriate for software development in the late 1980's.

Key Words: programming environments, Ada, analysis, debugging, concurrency

Introduction

In this paper we present concepts and ideas we think should characterize an advanced APSE (Ada Programming Support Environment). These range from already proven concepts to emerging ideas currently being explored and prototyped. Several important issues remain unresolved. Our intention is to explore issues, problems and opportunities.

Brief Background

Ada is a new programming language designed to promote reliability and maintainability of embedded computer software. Such software is often large (upwards of 100,000 instructions) and long-lived (15-25 years from conception to retirement). It must often meet requirements for concurrent processing and real-time performance.

If we are to produce and maintain Ada software that is *reliable, affordable*, and *adaptable*, the characteristics of Ada may not be the only important matter to consider. In addition, the characteristics of Ada software development environments may well be critical. [Stan82] makes the case that it is time to seize the opportunity to conceptualize what sort of advanced programming support tools should populate a mature APSE of high utility and effectiveness. In this context, consideration arises of support tools for modern programming practices, software reuse, interactive programming, software project management, project database support, improved testing and verification, and improved program understanding techniques.

At Irvine an experimental approach is being taken, using a prototype interactive Ada programming environment called *Arcturus*. We are using Arcturus as a platform for investigation and resolution of issues such as:

- Is interactive Ada a valuable concept? If so, what is needed to make it work to advantage by way of debugging facilities, coupling of interpretation and compilation, and other interactive support features?
- Given that Ada is a comparatively extensive language, is it reasonable to have a large, highly-integrated support environment, somewhat like Interlisp [Teit81], or is it better to follow the composable, small-tool fragment philosophy of Toolpack [Oste82]? What are the performance implications of these two philosophies?
- How are we going to set sail on the uncharted waters of *verification, testing, and debugging* of Ada tasking programs? Are there effective, practical approaches for checking them out, especially in cases where they may be prepared on a host and

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

** This work was supported in part by the Defense Advanced Research Projects Agency of the United States Department of Defense under contract (ONR) N00039-83-C-0567 to the Irvine Programming Environment Project. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

executed on a different target with unknown scheduling algorithms?

In this paper we will present our approaches to exploring these matters and discuss the lessons learned to date. Particular attention will be paid to the analysis and debugging of tasking programs. It will be clear that we have not addressed all the issues and that more investigation is needed. A tutorial on interactive Ada features and template-assisted Ada editing is given in [Stan83].

Arcturus: Current Status

At the moment, an experimental prototype of Arcturus exists, has been used for the past two academic years (1982-1984), and has been released to over 45 outside user groups. This prototype version of Arcturus offers an approach to the integrated use of compiled and interpreted Ada, template-assisted Ada text editing, an Ada Program Design Language (PDL), Ada program performance measurement with color profiles, formatted printing of Ada programs with useful listing options, and automated stepwise refinement from Ada program designs written in Ada PDL into executable Ada. In this section, we briefly cover some of the features of the current working prototype.

Interactive Ada

Interactive Ada provides immediate execution of Ada program fragments and convenient conversational management of interactive programming sessions. Arcturus employs interactive techniques such as breakpoints, tracing, interruption and resumption of computations, queries, and pretty-printing. No matter whether Arcturus is at the top-level awaiting a command from the user or whether an Ada computation is suspended at a breakpoint, the full power of the interpreter is available. One can execute statements (such as assignments, procedure calls or multi-line blocks), evaluate expressions (such as Boolean conditions or function calls), elaborate declarations, or call any operating system service. Full Ada has not been implemented; we are currently at the Pascal superset stage.

Template-Assisted Editing

An approach to interactive, screen-oriented Ada editing is incorporated which integrates five ideas: filling in holes in Ada program templates, syntactic completion starting with unique prefixes, arbitrary character-by-character editing at arbitrary cursor positions, help menus, and file inclusion (which permits users to define and name their own templates, extending the set initially available).

Performance Measurement

Measurement of Ada program performance can be accomplished by using traditional execution-time profiles displayed as histograms or by using color spectra. For example, the frequently executed parts of an Ada program text are displayed in red (the hot parts) and the infrequently executed parts in blue (the cool parts).

Mixed Compilation and Interpretation

Ada programs can be compiled using an Ada compiler donated by the *Irvine Computer Sciences Corporation.* (Originally we started to implement our own compiler, but decided to use the ICSC compiler when it was offered to us in order to release more of our resources for exploration of Ada support environment issues). Currently we can develop Ada programs in interactive mode and then compile them after putting them through an *export laundry.* This transforms interactively developed Ada programs into programs suitable for compilation, by, for example, inserting incomplete type declarations required in mutually recursive declarations. It is planned (though not yet implemented) to be able to execute mixes of compiled Ada compilation units and interpreted units to combine the advantages of compilation and interpretation during program development. Compilation yields a factor of 35 to 50 times better execution speed than interpretation and compiled code is considerably more compact than interpreted code (which requires elastic, information rich representations). However, interpretation promotes rapid composition and debugging at the expense of these space-time performance penalties.

Program Design Language

An Ada *Program Design Language* (PDL) is integrated into the interactive Ada environment so that the same tools which process normal Ada programs can operate on PDL program *designs* as well. For instance, PDL programs can be recognized by the syntax analyzer, pretty-printed, and refined stepwise into executable Ada by expanding user-definable Ada macros [Smit82]. The Ada PDL uses normal Ada syntax forms in which quantities in braces {such as this} can be substituted for names, types, declarations, expressions, or statements.

Development Details

The current version of Arcturus runs on a VAX* under Berkeley Unix**. It was built with approximately six person years of effort by four graduate students and one professional programmer. The source consists of about 24,000 lines of C. When the system is initially loaded, the bare Arcturus code and data consume 238,732 bytes. Adding in the additional space necessary to run user programs, the system requires 310,749 bytes. When Arcturus runs a 300 line Ada program for a "knights tour," the total system size is

*VAX is a trademark of Digital Equipment Corporation.

**Unix is a tradmark of Bell Telephone Laboratories.

368,200 bytes. The internal form used consists of nested records linked together by pointers into a tree and annotated with symbol tables, suitable for execution by a recursive interpreter.

Concepts and Issues Being Explored

Issue of Complexity

One challenge we face relates to the complexity of Ada. Ada is an ambitious language that can express a greater range of concepts than many previous languages (e.g, tasks, generics, overloading, packages, representation specifications, and exceptions). A price must be paid for this wide conceptual coverage. The problems we encounter in building an interactive environment for Ada are substantially greater than they are for languages with simpler, more uniform representational substrates, such as Lisp. Handling all of Ada's cases and the interactions between them appear to require much greater computer resources than handling a simpler language. It remains to be determined whether the whole language can be supported in an interactive environment of acceptable efficiency and size. Building and experimenting with environments such as Arcturus may help us resolve this issue.

Since we have not yet tried out Arcturus on large-scale software development, we cannot be certain that it will scale up and work equally well for "programming-in-the-large." We speculate, however, that it could be effective for large projects where the work is broken down into modules of reasonable size with well-specified, stable interface characteristics. Arcturus is demonstrably effective in rapidly composing and developing Ada modules in the range of two thousand lines or less. Of course the use of Arcturus for such large projects must be coupled with effective large-project support disciplines, such as the use of "automated unit development folders" [Boeh81]. We are working in partnership with a group at TRW who are applying their expertise in large-project management to add software management support tools to an environment based on Arcturus. By this means we hope to address the large-scale software production issues that lie beyond the boundaries of single-user, interactive programming techniques.

AVOS: An Ada Virtual Operating System

Another issue we are exploring is the concept of an Ada Virtual Operating System, *AVOS* [Whit82]. The goal of AVOS is to provide a completely uniform user interface and a machine-independent substrate for Arcturus-like environments. We are seeking to show how all operating system commands and programs can be expressed in Ada, and how Ada concepts can be used in place of conventional operating system notions, such as files and directories. The potential advantage of this is one of conceptual simplicity — it is simpler to use a single substrate of Ada concepts to express operating system commands than to learn and use a separate operating system command language.

User Interface Issues

Can we identify a family of compatible man-machine interfaces to serve as user-interfaces to a wide variety of tools? Can these interfaces be simple and uniform? One approach to this is to catalogue powerful user-interface techniques and move them out of individual tool interfaces into a common interface. (This probably cannot be done independently of the device type, so we may need one approach for each substantially different class of user interface devices, such as line-at-a-time printing terminals, glass teletypes, and bit-mapped displays). Examples of capabilities are uniform ways of: (a) redrawing a line and deleting the last character typed, (b) completing commands from initial prefixes and prompting for parameters, (c) menuing, (d) windowing, (e) including files, (f) getting help, (g) switching contexts rapidly (with screen regeneration on return to an interrupted context), and (h) typescript management and template-assisted editing. If we move powerful capabilities into a common interface that sits between the user and the tools we can perhaps enjoy one simple way of interfacing to a variety of tools.

Our current exploration of these issues is taking the form of a prototype common user-interface called *Adash* (which stands for "Ada Shell").

Mixing Interpretation and Compilation

We are exploring approaches for mixing compiled and interpreted Ada program units. One issue that arises when we try to do this is, "What level of granularity shall we be allowed to use when mixing such units?" A natural level of unit to begin with is the Ada compilation unit, but mixing at the level of procedures or statements may also be appropriate.

Our approach to this problem is to begin with compilation units and to develop "export and import laundries". These will: (a) export an interpretively developed Ada program to a compiler by filling in missing information such as incomplete type declarations needed to handle forward referencing (as described earlier), and (b) import a compiled package into the interpretive environment by interfacing to its visible entities making it possible to call procedures and access variables from the interpreter.

Evolving Toward a Layered System

We are evolving toward a layered system such as that pictured in Figure 1. The user may have to interface to several distinct devices, each with its own well-adapted user-interface properties. These properties are exhibited by the user-interface paradigms provided by *Adash* (and perhaps others) which furnish services such as typescript management, template-assisted editing, and command completion. At the next level are tools such as the mixed interpreter/compiler, the Castor

[Smit82] macro package for mapping Ada PDL into concrete Ada, verification and testing tools, and program explanation tools. At the base sits an Ada Virtual Operating System, providing a foundation level of services that frees the Ada environment from dependencies on particular operating systems and renders services linguistically accessible in Ada conceptual terms.

Not only are we convinced that these ideas are noteworthy and valuable in themselves, but the Arcturus concept (if not its current implementation) is a necessary and excellent platform for further investigation of the issues associated with the development of Ada programs. One of the most challenging areas is analysis and debugging of tasking programs. The following section considers this in detail and outlines the approach we are taking in the ongoing development of Arcturus.

Analysis, Testing, and Debugging of Tasking Programs

Typically, embedded software must satisfy real-time requirements and is often composed of multiple concurrent tasks. The software is usually developed on a host system providing program development services and then retargeted for execution on (usually dissimilar) embedded hardware. Assuring the reliability of embedded software is significantly complicated by these factors, yet it is of critical concern. In this section we present some techniques specifically aimed at assuring the reliability of multi-tasking Ada software developed in a host-target environment. A static analysis technique, a dynamic analysis technique, and an interactive debugger are considered. Though these techniques can be considered independent agents, our theme is that closely integrated application of all three is needed. (Other analysis aids are surely necessary too. No claim is made that the techniques presented are sufficient.) Furthermore, housing these analytic capabilities in a framework which provides the general program development services described in the preceding sections is essential to their effective application. Tasking and real-time issues cannot be considered only on their own: development and analysis of tasking programs involves looking *both* at the tasking and sequential aspects of the program. There should be no artificial boundary separating investigation of these concerns. Analysis and testing work is therefore an important part of the Arcturus project.

Static Analysis

Static analysis is often a desirable technique because execution of the subject program is not required, the analysis is often quickly performed, and

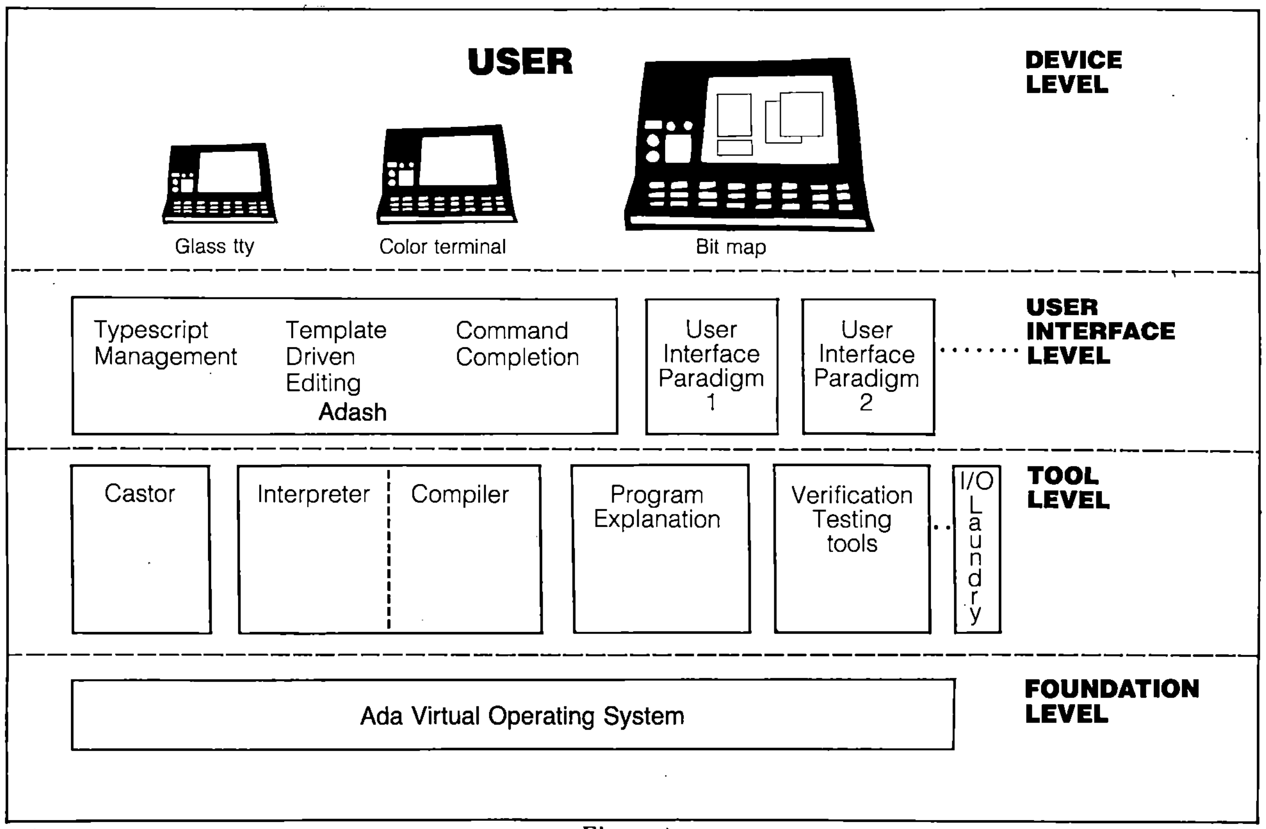

Figure 1
Layers of Arcturus support

the results cover all paths through the program. These characteristics are particularly valuable in the analysis of concurrent programs. It is unable to check deep semantics, however, and some overly pessimistic analysis reports may be included with the valuable ones. The static analysis technique with which we are concerned is described in detail in [Tayl83a]. Its objective is to determine, for a given program, all possible sequences of concurrency related events. From these sequences information regarding several aspects of a program's synchronization structure may be derived. Included are identification of all the rendezvous that are possible, detection of any task blockages (deadlocks) that may occur, and listing of all program activities that may occur in parallel.

These sequences of concurrency related events are expressed in terms of *concurrency states.* A concurrency state displays the next synchronization-related activity to occur in each of a system's tasks. A sequence of states presents a history of synchronization activities for a class of program executions. The analysis algorithm can develop a representation of all possible concurrency histories.

We will illustrate the concepts with an example. Figure 2 presents an Ada program designed to solve a version of the Dining Philosophers problem. Five philosophers are seated at a circular table, alternately eating and thinking. In order to eat, a philosopher must acquire the fork to the left of his plate and the fork to the right. In Figure 2 each philosopher is a separate task, as is each fork. The philosopher tasks request the fork resources by issuing entry calls. The program presented is a poor one in the sense that it is possible for deadlock to occur: if each philosopher is able to acquire the fork to his left, then they will all starve while waiting for the fork to the right. This possibility can be detected using static analysis.

The situation where all the tasks are active, the philosophers are all requesting the left fork, and all the forks are *ready to accept a call on "Up"* (i.e. "physically" the forks are all down on the table) is shown in the following concurrency state:

Main	Philosophers					Forks				
Task	A	K	B	T	S	0	1	2	3	4
end	U_0	U_1	U_2	U_3	U_4	U'	U'	U'	U'	U'

Here we abbreviate each philosopher's name with its first initial, the entry calls on "Up" are shown with a "U" and subscripted to indicate which fork is requested, and the accept statments in the forks are marked with an apostrophe (to distinguish them from entry calls). The main thread of control is shown at "end", indicating it is ready to terminate when all its dependent tasks terminate. Among many possible actions, the system may progress from this state to

Main	Philosophers					Forks				
Task	A	K	B	T	S	0	1	2	3	4
end	U_1	U_1	U_2	U_3	U_4	D'	U'	U'	U'	U'

```
procedure Dining_Philosophers is

type Seat_Assignment is Integer range 0..4;
task type Fork is
   entry Up;
   entry Down;
end Fork;
task body Fork is
begin
    loop
        accept Up;
        accept Down;
    end loop;
end Fork;
type Array_of_Fork is array (0..4) of Fork;

Forks: Array_of_Fork; --this declaration results
      -- in the activation of the 5 fork tasks

generic
   N: Seat_Assignment;
package Philosopher is
   task T;
end;
package body Philosopher is
   task body T is
   begin
      loop
          --acquire left fork
        Forks(N).Up;
          --acquire right fork
        Forks((N+1) mod 5).Up;
        delay 1.0; --eating time
          --put down left fork
        Forks(N).Down;
          --put down right fork
        Forks((N+1) mod 5).Down;
        delay 1.0; --thinking time;
      end loop;
   end T;
end Philosopher;

package Aquinas is new Philosopher(0);
package Kierkegaard is new Philosopher(1);
package Bonhoeffer is new Philosopher(2);
package Tilich is new Philosopher(3);
package Schaeffer is new Philosopher(4);

--This instantiation of each specific package results
--in the activation of the task contained within
--the package. Each task is activated with the
--generic actual parameter (0, 1, ..., 4) in place
--of the formal parameter N
begin
   null;
end Dining_Philosophers;
```

Figure 2
Dining Philosophers, Reserved Seating

implying that Aquinas acquired $Fork_0$ and is now requesting $Fork_1$, as is Kierkegaard. $Fork_0$ is now waiting to be put down. Supposing that next Kierkegaard acquires $Fork_1$ the system can progress to

Main	Philosophers					Forks				
Task	A	K	B	T	S	0	1	2	3	4
end	U_1	U_2	U_2	U_3	U_4	D'	D'	U'	U'	U'

The key item to note is that the static analysis algorithm will calculate all the possible states, exploring all eventualities. The particular states that arise during actual execution will be determined by the scheduler algorithm, processor speeds, and the like. (To simplify the presentation of this example we have not shown the relative position of entry calls on the entry queues.)

Further consideration of this example reveals that, after a series of rendezvous, the following state is possible:

Main	Philosophers					Forks				
Task	A	K	B	T	S	0	1	2	3	4
end	U_1	U_2	U_3	U_4	U_0	D'	D'	D'	D'	D'

This represents the deadlock described earlier. Simple, automatic analysis of this state will cause the deadlock to be reported. It is noteworthy that this state is a common successor of many different earlier states. Moreover it may not occur until after an extended period of "eating and thinking". All these possible sequences of states are revealed by the static analyzer. This history information can be used as an aid in further investigation of the anomaly, such as investigation to ensure that the history does not involve an unexecutable path.

It is important to note the limitations of the technique. First, static analysis must assume that each intra-task path is executable. This presents no problem in the example shown, but surely would introduce some spurious event sequences in a real program. Second, static analysis is only accurate when individual program objects (like tasks or entries) can be identified statically; program features potentially causing dynamic identification, such as access values (pointers) and subscripts, may be inadequately handled. Again, in this example there was no problem because of the use of the generic (compile-time) parameter to determine the "seating arrangement". If the program had been constructed so that seating positions were assigned dynamically, as suggested by the program fragment of Figure 3, then analysis would not have been so useful. The static analyzer would have been forced to compute all possible concurrency states, not knowing the value of "Seat". Even though the program may guarantee that no two philosophers simultaneously have the same value of "Seat", the static analyzer would nevertheless compute such outcomes. Literally thousands of spurious states would result. This observation is a key motive for developing a strategy to integrate static analysis with other techniques, and is considered in more detail below.

Finally, since the analysis conducted is independent (ignorant) of the target execution environment, the implications of delay statements, non-zero execution times, and scheduler algorithms are not taken into account. This restriction, of course, is also an advantage: the results produced do not rely on any possibly erroneous assumptions about the target environment.

A final note concerning this static analysis technique is in order. Regarding complexity, the algorithm is $O(n^T)$, where T is the number of tasks in the system, and n is the number of concurrency related statements [Tayl83b]. Usually a very large number of states will be generated, and such generation may take considerable time.

```
task type Philosopher;

task body Philosopher is
   Seat: Integer range 0..4;
begin
   loop
      get_seat_assignment(Seat); --this call
        -- dynamically returns the number of
        -- an unoccupied seat
      Forks(Seat).Up;
      Forks((Seat+1) mod 5).Up;
      delay 1.0; -- eating time
      Forks(Seat).Down;
      Forks((Seat+1) mod 5).Down;
      vacate_seat(Seat); --this call returns
        -- "Seat" to unoccupied status
   end loop;
end Philosopher;

Aquinas, Kierkegaard, Bonhoeffer,
 Tilich, Schaeffer: Philosopher;
     -- in this program the tasks are not contained
       -- within packages, but are explicitly
     -- declared and activated here
```

Figure 3
Dining Philosophers, General Admission

Dynamic Analysis

German, Helmbold, and Luckham have described a dynamic analysis technique having several of the same goals as the static analysis technique above [Germ82]. The basic idea of the technique is to transform an Ada program P into another Ada program P' such that P and P' have the same set of possible deadness errors, but, during execution, P' will detect the imminency of a deadness error, report the condition, and allow the possibility of evasive action. The transformation accomplishes this by adding a monitor task to P which maintains, essentially, the current concurrency state. Each task in the system updates the monitor, telling it the task's next tasking activity, such as entry call, accept completion, or task completion. By doing a one state look-ahead the monitor can detect a deadness error "just before it happens" and can thus raise an exception in the offending task, again "just before" the error would occur.

The dynamic technique is not beset by the same restrictions as the static technique. The dynamic technique functions correctly in the presence of nearly all Ada tasking constructs. The use of access values and subscripts presents no problems. No spurious errors are reported. Dynamic analysis of the program in Figure 3 would not present any difficulties: only legal seatings would occur. (The deadlock possible in that program may never occur with a particular scheduler, however.)

The instrumentation process itself is potentially efficient; some parts of the monitor task do not even require recompilation with each new program to be monitored.

Some interesting issues arise regarding run-time efficiency however. Since dynamic analysis is being discussed it is clear that the impact of the real-time environment is felt, including the effect of delay statements. Unfortunately the error monitoring instrumentation imposes a potentially substantial amount of overhead. Not only is another task included in the program (the monitor) but the number of entry calls occurring leaps by a factor of three or more. Sensitive timing properties may therefore be disturbed. Then too the overhead induced by the instrumentation may cause an observed phenomenon to disappear (under the same set of external conditions) though the *potential* for that error still remains. An example of this is shown in [Germ82].

The prime limitation of error monitoring, of course, is that a batch of error-free runs does not allow one to infer much about the correctness of the program, even with respect to the limited scope of deadness errors. A change in the underlying implementation, such as a new scheduler, may cause a crop of errors to appear, even when the program is run on the old test data.

The apparatus used to perform this error monitoring can be easily augmented to document many run-time events of interest. Since the monitor is notified of all rendezvous, task initiations, terminations, and so forth, it is a simple chore for it to produce a trace listing of these events. The implementation described in [Helm83] allows this. Many other obvious extensions are possible, including emission of the current concurrency state, length of actual delay at delay statements, length of blockage at entry calls, and so forth.

Cooperative application of static and dynamic analysis

Clearly the static analysis and dynamic analysis techniques presented have complementary characteristics. Static analysis results can be definitive and informative, accounting for all possible program actions. But the application of the technique is limited to a subset of Ada and some of the analysis results require scrutiny to determine if the reported phenomena are indeed possible. Dynamic analysis has fewer limitations, but the meaning of the results obtained is not so clear (unless an error is discovered). This complementary character of the individual techniques suggests that if they are applied in concert, several of their deficiencies may be eliminated.

The most obvious joint use of the techniques is to employ dynamic analysis in the further investigation of phenomena detected by the static analyzer. It may be, for example, that the scheduler used in a particular implementation or the semantics of the guards on select statements preclude a (statically) reported deadlock from occurring.

Certain concurrency states may not be possible for other reasons: the static analyzer may have assumed the executability of an unexecutable path, for instance. Dynamic analysis could monitor these conditions, watching for the error during testing. Moreover the concurrency histories may be of substantial use in attempting to develop test data to force execution of the anomalous sequences. In general, the concurrency histories provide a guide to the development of a battery of test cases. (Other automated tools, such as symbolic executors, may be useful aids in this process as well.) In the restricted sense of testing a program's synchronization structure, the complete concurrency history can be used as a yardstick in evaluating the thoroughness of a test regimen.

It is also possible for static analysis results to help in reducing the overhead incurred by the dynamic techniques. If a subset of a program's tasks can be conclusively shown to be error free, then the instrumentation used for their monitoring may be reduced or eliminated.

Potentially one of the most significant problems with static analysis is the large number of concurrency states that a program may have. There may be so many that it would be infeasible to generate them all. It is possible, however, to use static analysis to examine all the possible successors of an interesting concurrency state that had been captured dynamically. Used in this way the static analyzer would not generate the complete history for a program, only the the histories rooted at the state supplied by dynamic analysis. An appropriate state for the program of Figure 3 would be when all the philosophers have received a seating assignment, but before any of the forks have been acquired. (Knowing what an appropriate state is may require significant knowledge of the problem domain, though.)

This cooperative application strategy is more fully described in [Tayl83c].

Debugging

A variety of issues are associated with the debugging of concurrent programs. The papers and discussions at the workshop on "High-Level Debugging" highlighted several of these [HLDB83]. While not denying the importance of other kinds of support, our approach is based on three distinguishing premises. First, we believe a debugging system should permit focus on both task interaction concerns (synchronization) and sequential program concerns. Effective investigation of a phenomenon may involve looking in both domains. Second, support must be provided for debugging using Ada language concepts. Tools for focusing on other levels of abstraction, higher and lower, are needed too, but Ada level support seems key. Third, the analyst must be able to direct the course of a debugging run with precision; complete control over the scheduler and processor speed seems a must. At any desired point full visibility into the system should be available.

Our goal here is not to rehash all the desirable capabilities of a debugger, particularly as they relate to sequential code, but to stress the importance of integrating a debugger with the static and dynamic analyzers described above. Our contention is that this union provides substantial debugging power. This integration will be shown to be particularly valuable in a host-target environment.

The functioning of a debugging system designed with these principles in mind is indicated in Figure 4. "Single-stepping" is the fundamental capability of the system. A path through a concurrent program is determined by three factors: the data processed, the choices of the scheduler, and timing issues. In the timing category are matters of processor speed and the arrival time of data values. The analyst is given control over all three factors. Data values are supplied on the necessary input channels as with sequential debugging. Scheduler issues and timing concerns are controlled either through use of concurrency states (the notion described above under "Static Analysis") or by direct interaction. Using a sequence of concurrency states, the analyst may precisely state what actions are to occur in the debugging execution. A sequence of states completely governs the progress of each task, including their *relative* progress. Time is thus "controlled" as well. Intra-task paths are, of course, determined by the test data. Input received from a resource shared by multiple tasks must also be governed. This can be accomplished if the resources are identified before execution and included directly in the concurrency states.

The utility of technique integration can be seen by considering one approach to investigation of an error discovered during testing of the program running on the target machine (bench testing). Assume for this discussion no resources are shared and time of data arrival is immaterial. From the low level final state of the target machine (taken perhaps from a memory dump) an Ada level final state must be constructed. (This reconstruction is machine dependent and, of course, may not always be possible.) From the Ada level state the last concurrency state reached may be derived. Call that state S. The test data and S are transferred to the host, where a debugging execution is initiated. Paths internal to tasks are driven in the debug run by the test data. The synchronization and timing issues can be controlled as follows. The static analyzer is used to generate the set of all concurrency histories for the program that contain S. When the debugging execution reaches a point where a scheduling decision must be made, a choice consistent with the history graph is made. If this choice later proves to be infeasible (i.e.,

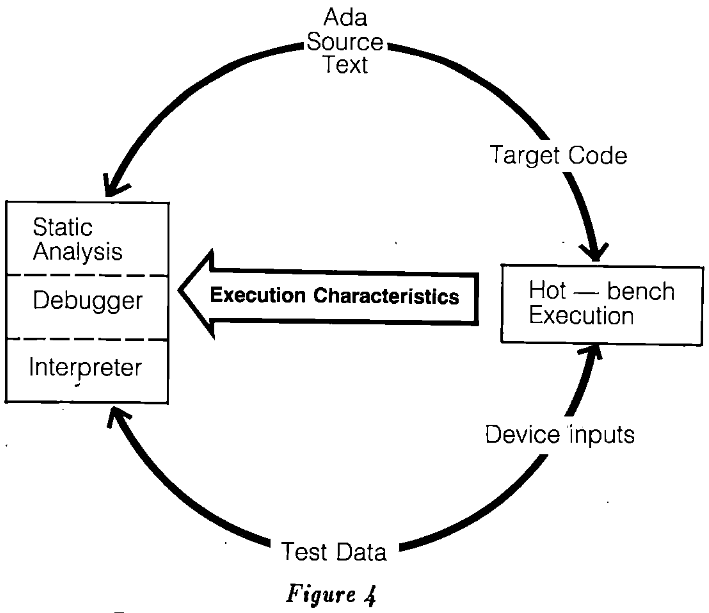

Figure 4
Debugging in a host-target environment

not leading to S) then execution is backed up and another selection made. In effect a depth-first traversal of the history tree guides the execution until S is duplicated in the debugging run. Visibility into the debug run is always available, using standard debug features.

If the bench execution contained any instrumentation in support of dynamic analysis, then the information gathered may significantly speed up this process for finding a path to S. The instrumentation, for instance, may capture an (acceptable) machine state prior to the fault. Then the debugging execution could be started from that point. If only a concurrency state was captured, it could be used to prune the set of allowable concurrency histories.

Other debugging strategies are possible in this general framework as well. It is noteworthy that many of the normal limitations of static analysis (such as dealing with pointers) can be overcome in this integrated context, since information from the debugging execution (or target execution) can be used to help refine or build the set of concurrency histories.

One of the more exciting possibilities associated with debugging tasking programs is in the design of animation features and user interface notions to aid in the process. The current version of Arcturus uses color and intelligent terminal operations in a variety of ways. Bit-mapped terminals offer many opportunities; we are currently exploring how to best extend and exploit these features. A more detailed presentation of our approach may be found in [Tayl83d].

Conclusion

The Arcturus system is a platform for exploring issues associated with advanced Ada programming environments. We believe many exciting concepts have been demonstrated. Interactive program development and debugging, the use of Ada as a command language, template-assisted editing, and program design language facilities combine to yield an effective environment for module development. We have learned that even the current fragile, incomplete prototype of Arcturus demonstrates that interactive Ada programming techniques are effective and valuable. Even though the pieces of the current Arcturus are not completely integrated, Arcturus has been used with considerable success in five compiler construction clases for 140 students at Irvine in which (small) working educational compilers have been written in Ada. It has also been used in rapid prototyping of programs written in Ada in which software productivity has been measured to be about 19 times greater than the nominal estimate given by [Boeh81]. In one case the rapid prototyping techniques involved use of the Ada PDL for design, use of interactive Ada for rapid composition and debugging, and a 62% software reuse factor in constructing the prototype system [Stan83, Tayl82].

Our initial experience is also leading us to conclude that a high degree of integration of *some* tools has a high payoff. Rapid composition and debugging of code frequently involves small-grained, intermixed use of the editor, interpreter, and debugger. Tight integration (sharing of address space and data structures) reduces the conceptual overhead in tool-switching and execution efficiency is improved as well (*if* adequate machine resources are available). We are not claiming that such tight integration is always effective. Indeed we believe that the analysis and testing tools discussed will best be fashioned from several small, separable tool fragments. This would allow, for example, resource intensive jobs — like static concurrency analysis — to be spun off as background processes. Currently we are exploring how we may support a "mixed strategy" in which some tools are tightly coupled and others are not. In particular we wish to allow dynamic, and perhaps automatic, environment configuration, in which appropriate tools are tightly or loosely coupled in response to functional requests, availability of machine resources, and availability of previously developed information.

We are particularly concerned with expanding Arcturus to support the analysis, testing, and debugging of real-time, tasking programs. Stand-alone techniques have been developed to aid in these activities. In this paper a strategy has been advanced for applying the individual techniques in an integrated fashion, making them even more valuable as they are mutually reinforcing and complementary in important ways. The strategy presented offers a disciplined approach to the problem of analyzing errors uncovered during real-time target machine testing. Presenting all these capabilities in a cohesive framework will result in their most effective application.

Acknowledgments

Several individuals have been responsible for the design and implementation of Arcturus, including Ray Klefstad, Craig Snider, Frank Tadman, Steve Whitehill, and Steve Willson. We are indebted to Dave Luckham, Ed Falis, Steve German, and Ralph London for several insights. Thanks also to the referees for their constructive comments.

References

[Boeh81]
Boehm, Barry W. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, N.J., 1981.

[Germ82]
German, Steven M., David P. Helmbold, and David C. Luckham. Monitoring for deadlocks in Ada tasking. Proceedings of the AdaTEC Conf. on Ada, Arlington, VA (October 1982), pp. 10-25.

[Helm83]
Helmbold, David P. and David C. Luckham. Run-time detection and description of deadness errors in Ada tasking (Draft). Computer Systems Laboratory, Stanford University, Stanford, California (1983).

[HLDB83]
Proceedings of the ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-Level Debugging, Asilomar, CA. (March 1983). Appeared in *Software Engineering Notes, 8,* 4 (August 1983).

[Oste83]
Osterweil, L.J. Toolpack- An experimental software development environment research project. *IEEE Transactions on Software Engineering, SE-9,* 6, 673-685 (November 1983). pp. 166-175.

[Smit82]
Smith, David Andrew. *Rapid Software Prototyping.* Ph.D. dissertation, Computer Science Dept., UC Irvine, TR-187 (May 1982).

[Stan82]
Standish, Thomas A. The importance of Ada programming support environments. *Proc. 1982 NCC, 51,* AFIPS Press, Montvale, NJ, 333-339.

[Stan83]
Standish, Thomas A. Interactive Ada in the Arcturus Environment. *Ada Letters, III,* 1, 23-35 (July, August 1983).

[Tayl83a]
Taylor, Richard N. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM, 26,* 5, 362-376 (May 1983).

[Tayl83b]
Taylor, Richard N. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica, 19,* 57-84 (1983).

[Tayl83c]
Taylor, Richard N. Analysis of concurrent software by cooperative application of static and dynamic techniques. In Hans-Ludwig Hausen (editor), *Proceedings of the Symposium on Software Validation,* Darmstadt F.R.G. (September 1983), North-Holland Publishing.

[Tayl83d]
Debugging real-time software in a host-target environment. Technical Report #212, Department of Information and Computer Science, University of California, Irvine (November 1983).

[Tayl82]
Taylor, Tamara, and Thomas A. Standish. Initial thoughts on rapid prototyping techniques. *Software Engineering Notes, 7,* 5, 160-166 (March 1982).

[Teit81]
Teitelman, W. and L. Masinter. The Interlisp programming environment. *Computer, 14,* 4, 25-33 (April 1981).

[Whit82]
Whitehill, Stephen B. An Ada Virtual Operating System. Proceedings of the AdaTEC Conf. on Ada, Arlington, VA (October 1982), pp. 238-250.