Title
Debugging real-time software in a host-target environment

Author
Taylor, Richard N.

Publication Date
1984

Peer reviewed

Debugging Real-Time Software in a Host-Target Environment

Richard N. Taylor

Technical Report #212

Programming Environment Project
Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717 U.S.A.
Telephone (714) 856-7202

Debugging Real-Time Software in a Host-Target Environment

Richard N. Taylor

Programming Environment Project
Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717 U.S.A.
Telephone (714) 856-7202

ABSTRACT

A common paradigm for the development of process-control or embedded computer software is to do most of the implementation and testing on a large host computer, then retarget the code for final checkout and production execution on the target machine. The host machine is usually large and provides a variety of program development tools, while the target may be a small, bare machine. A difficulty with the paradigm arises when the software developed has real-time constraints and is composed of multiple communicating processes. If a test execution on the target fails, it may be exceptionally tedious to determine the cause of the failure. Host machine debuggers cannot normally be applied, because the same program processing the same data will frequently exhibit different behavior on the host. Differences in processor speed, scheduling algorithm, and the like, account for the disparity. This paper proposes a partial solution to this problem, in which the errant execution is reconstructed and made amenable to source language level debugging on the host. The solution involves the integrated application of a static concurrency analyzer, an interactive interpreter, and a graphical program visualization aid. Though generally applicable, the solution is described here in the context of multi-tasked real-time Ada* programs.

Keywords: debugging, host-target development, concurrency, real-time software, Ada, environments

1. Introduction

Real-time software is often developed on a host machine and then recompiled for execution on a target machine. The host machine is typically much more powerful than the target, providing a variety of program development services. Target machines are frequently "bare machines", having no support software at all — not even operating systems.

The difficulty with this development model is in testing software on the target machine. Some testing must be done on the target, as host machine testing is grounded upon some assumptions about the target. For example, host testing often involves use of a target machine emulator. Target machine testing is necessary to ensure that the emulator correctly reflected the target's

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

characteristics. The difficulty is in determining the cause of an error detected during target testing: most likely there are no tools to aid in this determination. The analyst may have only a memory dump from which to work.

This unfortunate situation is greatly aggravated when the software being developed contains multiple concurrent tasks, or when its functionality is determined by real-time considerations. Target machines are often embedded processors, executing in a real-time feedback loop. When this is the case, several additional factors, such as the following, may cause target machine executions to deviate from host executions:

- The real-time input simulators on the host may not operate at the same rate as the actual inputs to the target.
- The real-time clock may be less (or more) precise.
- The number of physical CPU's may vary between host and target, affecting the execution rates of separate tasks.
- The relative speed of the multiple processors may vary.
- Though the same scheduler algorithm may be used on both machines, different behavior may be observed because of differences in processor construction.
- Different scheduling algorithms may operate on the host and the target.

Because of these matters, a very real possibility is that a concurrent program may execute correctly on the host, but deadlock on the target, even though it is processing the "same" data. Thus straightforward attempts to debug on the host may be fruitless.

The intent of this paper is to present a technique for host debugging of failed target machine executions which addresses all the relevant concerns.

1.1. Objective

The initial objective of the technique we present is to reconstruct, with fidelity, the target execution on the host. This means determining the exact sequence of (target) machine state transitions. Once this reconstruction is achieved, a secondary objective is to provide debugging techniques that enable effective investigation of the behavior of concurrent, real-time programs. These techniques should be provided at the level of source language concepts (e.g. Ada rendezvous), not assembly language or, worse yet, machine code instructions. Furthermore, the techniques must enable a program to be viewed from different perspectives, and the analyst must be able to move smoothly from one to another. The perspectives we have in mind are: 1) looking within a single task to investigate its particular behavior, and 2) looking at the system of interacting processes to study the task interactions that occur. The focus of this paper will be on the second of these perspectives, as techniques for debugging single process, non-real time programs can be used for looking within a single task. It must be remembered, however, than any implementation of the overall technique must provide both capabilities.

Achieving these objectives is difficult, and the technique presented below is not perfect. A key characteristic of the technique, which is of interest in its own right, is that it involves the integrated application of several sophisticated tools. To be used effectively these tools must be housed in a programming environment. This will be considered more fully at the end of the paper.

To provide focus for the discussion, attention will be restricted to concurrent, real-time Ada programs. Ada provides several high-level facilities for describing multi-tasked systems [Ada83]. The technique has broad applicability, however, and could be used in debugging, for example, CSP [Hoar78], HAL/S [Mart77], or Industrial Real-Time Basic [IRTB81] programs.

Several other research groups are investigating the problems of debugging concurrent and distributed systems. A variety of promising work is described in [HLDB83]. Other, more closely related research is referred to in the following presentation.

2. Solution scheme

We are proposing a two-step approach to the problem of locating an error in a failed target machine execution. The first step, and the most difficult one, is recreating the target machine's execution back on the host. The second step is to analyze that execution, using a powerful debugger, to isolate the fault.

The first step involves several operations. After listing them here we will then consider them in more detail.

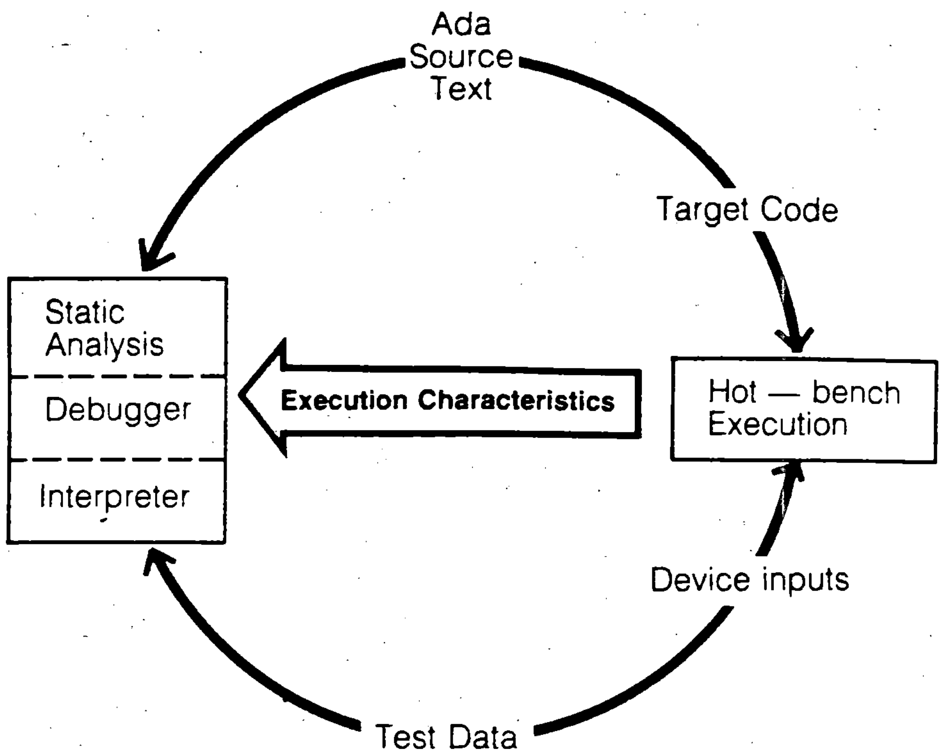
- From the final target machine state (possibly given by a memory dump) derive the corresponding Ada-level machine state.
- Extract from the Ada state the final *concurrency state*. (I.e., determine the final concurrency related action taken by each task.)
- Determine the full range of possible *concurrency* and *real-time* program actions that could lead to the final concurrency state.
- Prune the range of concurrency actions potentially leading to the final state on the basis of knowledge of the target's execution. (This may be a null step.)
- Find a viable sequence of concurrency and real-time actions, using a process of "depth-first execution".
- Initiate a detailed debugging execution driven by the sequence of viable actions.

The processes and data flows involved in these activities are indicated in Figure 1.

The limitations of the technique will become painfully obvious in the remainder of the presentation. Here we simply note a few of them.

- It must be possible to reconstruct (key portions of) the final Ada-level machine state from the target machine's final state. Therefore errors whose penultimate action is to wipe out all of memory cannot immediately be addressed.
- It must be possible to capture the sequence of data values that were read by the target machine, though it is *not* necessary for them to be time-stamped. (But time-stamps may be useful, however.)
- The process of reconstructing the execution may be slow, though it certainly will be more efficient than having a person attempt the same.

A technology key to the entire process is static analysis of concurrent Ada programs. This technology is described in detail in [Tayl83a]. The following subsection summarizes the key



Overview of Host-Target Debugging
Figure 1

techniques.

2.1. Static analysis of concurrent programs

The objective of this analysis technique is to determine, for a given program, *all possible sequences of concurrency related events.* 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. 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. For the purposes of this paper, though, it is the existence of a representation of all possible histories that is important.

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" is shown in the following concurrency state:

Main	Philosophers					Forks				
Task	A	K	B	T	S	0	1	2	3	4
end	Up_0	Up_1	Up_2	Up_3	Up_4	Up'	Up'	Up'	Up'	Up'

Here we abbreviate each philosopher's name with its first initial, the entry calls on "Up" are 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	Up_1	Up_1	Up_2	Up_3	Up_4	Down'	Up'	Up'	Up'	Up'

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

```
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
         Forks(N).Up; --acquire left fork
         Forks((N+1) mod 5).Up; --acquire right fork
         delay 1; --eating time
         Forks(N).Down; --put down left fork
         Forks((N+1) mod 5).Down; --put down right fork
         delay 1; --thinking time;
      end loop;
   end T;
end Philosopher;

package Aquinas(0) is new Philosopher;        --This instantiation of each specific package results
package Kierkegaard(1) is new Philosopher;    --in the activation of the task contained within
package Bonhoeffer(2) is new Philosopher;     --the package. Each task is activated with the
package Tilich(3) is new Philosopher;         --generic actual parameter (0, 1, ..., 4) in place
package Schaeffer(4) is new Philosopher;      --of the formal parameter N
begin
   null;
end Dining_Philosophers;
```

Figure 2
Dining Philosophers, Reserved Seating

Main	Philosophers					Forks				
Task	A	K	B	T	S	0	1	2	3	4
end	Up_1	Up_2	Up_2	Up_3	Up_4	Down'	Down'	Up'	Up'	Up'

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 considered 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	Up_1	Up_2	Up_3	Up_4	Up_0	Down'	Down'	Down'	Down'	Down'

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.

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 non-realizable event sequences in a real program. Second, static analysis is accurate only 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, then analysis would not have been as useful. The static analyzer would have been forced to compute all possible concurrency states, not knowing the value of "N". Even though the program may guarantee that no two philosophers simultaneously have the same value of "N", the static analyzer would nevertheless compute such outcomes. Literally thousands of spurious states would result.

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.

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 a key advantage: the results produced do not rely on any possibly erroneous assumptions about the target environment. In fact it is this very characteristic which guarantees that the set of histories produced by the static analyzer *includes* the history which led to the failed target execution that we are attempting to debug. The problem then, is to determine which history is the one.

2.2. Path finding strategy

The problem of reconstructing the failed target execution back on the host is now considered in some detail. The procedure described below makes few assumptions about communication between the host and target. Necessarily the resulting analysis is potentially costly. After presentation of the basic procedure several optimizations are described. At the expense of increasing the communication between the machines and constraining the structure of the target, substantial speedup of the reconstruction process is obtained.

Working from perhaps a memory dump from the target execution, the first task is to reconstruct the final state of the program in Ada-level terminology. Ideally the complete *program state* F would be "unloaded", yielding the last value of all variables as well as knowledge of what tasks were in existence, their status (running, blocked, etc.) and which instruction in each of these tasks was to be executed next. However a useful debugging exercise can be conducted even if only the final *concurrency state* C can be reconstructed. The specifics of this unloading process will vary from target to target and, as noted earlier, may not always be possible. When it is possible, though, the reconstructed state is handed over to the host-resident tools which reconstruct the execution path.

The first step in path reconstruction is static generation of all concurrency histories H leading from the start state to C, the final concurrency state. Those are the *only* histories to be generated. The static analysis technique described earlier can easily be used to do this. The next step is determination of which of these histories describes the failed execution. This can be determined as follows. A host machine execution of the subject program is initiated. This execution uses as input data the data values used by the target execution. These values need not be time-stamped, and could be captured by simple hardware monitors on the target machine. Whenever the host execution reaches a point where a scheduler decision or a time-dependent activity is required, a decision or activity consistent with a concurrency history $h_1 \in H$ is made. Execution then resumes. This process continues until F (and thus C) is reached, in which case a candidate valid history has been found and the process terminates, or else the debugging execution cannot continue in accordance with h_1.

This later situation can be thought of as follows. Let $h_1 = s_1\, s_2\, s_3 \ldots s_n\, s_{n+1} \ldots C$ where s_i is a concurrency state in history h_1, s_n is the last concurrency state reached in the host execution, and the transition from s_n to s_{n+1} is impossible in the host execution. This means that the data processed by the program demands that some other concurrency state s' be reached from s_n (perhaps because of a path within a particular task). If indeed s_n has another possible successor that leads eventually to C, then that history $h_2 = s_1\, s_2 \ldots s_n\, s' \ldots C$ is pursued, again until reaching C in the host execution or until the process can continue no further. If once again the process stalls, another possible history is chosen and pursued. This may involve backing up before s_n. We are in fact suggesting that H be traversed in a depth-first manner to guide the scheduler in exploration of all feasible histories until the desired one is found.

Note: If the path reconstruction process uses only the final concurrency state C and not the complete final state F, then h, the concurrency history "found", may not be the history h' that occurred during the target's execution. It will be an "interesting" history though, as it characterizes an execution with properties close to h'. Specifically, if h' resulted in a tasking error such as deadlock, then h is a possible execution (with respect to the same input data) that will also result in that error. If F is used instead of C then h is *more likely* to be h' since the value of program variables can be used to determine the need for further depth-first executions. But since complete intermediate program states are not compared between the target execution and the host, one cannot guarantee that the two are identical.

This entire process poses many difficulties and is potentially expensive. Following are some comments briefly addressing some of the serious issues.

- If two or more tasks in the program can reference the same input channel, then all references to that channel must be shown in the concurrency states of H. In so doing, all possible patterns of reference to that shared resource can be examined.
- If the data values read by the target are time-stamped, then these time-stamps can be used to prune H so that it only includes histories consistent with the observed patterns of reference. To take advantage of these time-stamps all references to input devices must be shown in the concurrency histories.
- Central to the above strategy is driving the host execution in accordance with a concurrency history. This implies the host's scheduler must be completely controllable, and accept a history as controlling input.
- When the debugging execution cannot proceed any further in connection with a given history, execution is "backed up" to a previous concurrency state and resumed along another history. This implies the (virtual) saving of complete intermediate program states by the host. Such states would not actually have to be saved at all concurrency state — concurrency state transitions, however, as states could be recomputed.
- The static analyzer is limited in its ability to generate histories for programs using pointers to name tasks. It can be directed to generate histories based on all possible references, however, and rapid pruning will occur when dynamically generated information is supplied.

2.3. Speed-up through dynamic analysis

The above process is somewhat brute force and inelegant. But the problem is difficult and the solution scheme only assumes that the target's final state can be unloaded and that input values can be captured. Substantial speedup can be obtained by weakening restrictions on the degree of host-target communication. Namely, if some information describing the progress of the target can be gathered during its execution, that information can be used to prune substantially the set of histories H that have to be explored on the host.

Most desirable, of course, would be a detailed description of the activities of the target's scheduler. If it emitted a message describing its every activity then that would completely describe the concurrency history. A scheduler that does this has been constructed at Stanford [Germ82]. If messages were only issued intermittently then they could be used in the history-pruning process. Less desirable but still very helpful would be snapshots of the target's memory, or portions thereof. It may be impossible to obtain any of this additional information, however, because of constraints the target machine may impose.

2.4. Execution visualization and intra-task debugging

The result of the path reconstruction process is specification of the concurrency history which occurred on the failed target execution. Once this is obtained the analyst has available complete knowledge about what events took place on the target. The concurrency history details the scheduler and time-related phenomena, while the test data determines the actions within individual tasks. Based on this information a detailed debugging execution can be initiated, with the purpose of determining the cause of the error.

Debuggers provide the ability to investigate program activities in detail: initially one is concerned with seeing what happens during execution. Further understanding is often obtained by modifying the execution, such as by changing a variable's value, then observing the effect of the change. We believe that aids which help visualize (or animate) the execution of concurrent programs are particularly helpful, and briefly present here a few ideas which we think hold promise. (Some issues association with construction of such an animator are briefly presented in a later section.) It is not sufficient, however, to just provide information about task interaction. The analyst must also have the ability to look in detail *within* a given task, and readily move between these perspectives. We will not consider techniques for debugging within a task, however, as that technology has been described many places [HLDB83].

With respect to animation of concurrent executions, we envision the following features. The analyst will use a bit-map terminal, preferably with color display capabilities. One window, always visible on the screen, is a control menu. The bulk of the screen is devoted to displaying task interactions. When a task comes into existence a new rectangle (window) appears on the screen. This task window is linked to its parent task by an arc, indicating the task dependency relationship. The priority at which a task runs is indicated by its color: high priority tasks glow red while low priority is shown as violet. (A full color spectrum would be used.)

Each individual task may have further attributes displayed. A candidate set of default attributes may be as follows. Within a task's window five lines of program text are displayed: the first two lines are the two previously executed statements, the third is the current statement, and the fourth and fifth are the two statements textually following the current statement. (The size of this window could of course be varied to display more or less.) Each task may own entries. Each entry owned by a task would be shown as a labeled rectangle attached to the outside of the task. When an accept statement for a given entry is eligible for execution (such as when it appears in a select statement and its guard is true) the entry rectangle would be highlighted. Entry calls issued by a task would appear as dashed arcs from the task issuing the call to the entry rectangle on the task owning the entry. The order of entry calls in the queue would be shown by ordering the arcs on the entry rectangle. When a call is accepted (a rendezvous begins) the appropriate arc would change from dashed to solid. The arc would disappear on rendezvous completion. Delay statements (and timed entry calls) would cause the appearance of a countdown clock in the task rectangle. Finally, when a task became eligible for termination its terminate block would glow. Termination would result in removal of the block from the screen.

With the amount of information listed here it is clear that the ability to shrink and grow rectangles is important. Furthermore when the analyst wishes to look in detail within a single task, then that task's rectangle should fill the screen, its internal data values should become accessible, and so forth.

Three key capabilities would be controlled by the menu items. The first is control of the speed of the animation. The ability to slow and halt execution is necessary. The second is to initiate and control intra-task debugging. The third is to initiate and control inter-task breakpoints. Breakpoints could be set at specific rendezvous, or particular task elaborations, for instance. Ability could also be given for the user to direct execution down another concurrency history (different from the one which occurred on the target machine). Such directions could correspond to the effects of applying a different scheduler algorithm.

We are guilty of the charge that the preceding list of capabilities is a "wish-list". We have not implemented these yet, though we are convinced that it is a very feasible task. The important point is that provision of these types of features could make a tremendous impact in the understanding and debugging of concurrent programs. Program visualization is an important concept and we are attempting to delineate worthy goals. (Some noteworthy related work concerning application-specific animation is being carried out at SRI by Mark Moriconi [Mori83]. Our work is with application-independent (structural) animation.)

3. Implementation

A host-target debugging system built along the lines suggested will never achieve its full potential unless it appears within a comprehensive programming environment. The full range of debugging activities includes text editing, file manipulation, and all the subtasks associated with interpretive or incrementally compiled execution. Furthermore the scheme we are proposing involves the integration of an unloader, a static concurrency analyzer, a "pruner", an interpreter or compiler, a display driver, and (potentially) dynamic analysis tools. Efficient application of the technique will require a well-designed tool framework.

We are currently engaged in constructing such a system for Ada programs called Arcturus [Stan83]. Two prototype implementations have been created. The current system provides interactive Ada programming, a break package (an intra-task debugger), template-assisted Ada program editing, command-completion using Ada as a command language, an integrated program design language/rapid prototyping system, and performance profiling. We are now studying ways of implementing the host-target debugging paradigm and its associated tools.

3.1. Some implementation issues

Listed below are a few of the more interesting implementation details that arise and which must be investigated further.

- The process of unloading target machine states is, as mentioned earlier, target-dependent. A particular problem here is determining the names of tasks in the machine state, corresponding to the names used on the host (such as by the static concurrency analyzer). The unique task id technique of [Germ82] potentially offers a solution.
- The execution animator can be driven by calls from the scheduler. Whenever the scheduler performs an action, such as initiating a new task, a message is sent to the animator describing the state change. The animator then determines, on the basis of current display options, what changes to the screen are necessary, and then effects the changes. Techniques for scrolling program text in the task rectangles can be taken from existing single process debugging systems [HLDB83].
- If the program animator is requested to highlight all accesses to global (shared) variables, including alias references, then the debugger can adopt a software-implemented tagged memory architecture. This strategy for dealing with alias references has been proposed by Johnson [John79] in a system for debugging single process programs. Direct extension to a multi-task model appears to present little difficulty.
- The host system must provide software simulators corresponding to each external input/interrupt to the target machine. External interrupts to an Ada program appear as entry calls. Thus one simulator task is required for each potential source of hardware interrupt to the target. Sources of external inputs to the target could be modeled the same way. (Recall that the *rate* of interrupt requests/inputs is completely specified in a concurrency history. Finding the proper "speed" is thus a part of finding the desired concurrency history. Note that this requires all points of external interaction to appear in the individual concurrency states.)

4. Conclusion

By some estimates [DoD80] debugging of target machine executions accounts for 25% of total embedded-system development costs. This high cost can be attributed, at least in part, to a lack of effective tools. This paper has presented an entirely new approach to host-target debugging in which debugging of target executions can be carried out on a host supplying many automated tools. The basic solution proposed is potentially very inefficient, but it makes only nominal demands on knowledge of the target's activities. As additional information about the target's execution is supplied, the efficiency of the process increases dramatically (because the search space is further pruned). Some program animation techniques have also been sketched. We believe the use of animation promotes rapid understanding of the actions of a program. Furthermore a graphic display is an effective device for controlling a debugging execution.

Design and implementation of the approach described is under way. Clearly we need to carry out many experimental studies to determine the practical utility of the various techniques. In particular we need to investigate additional ways of capturing information about a target's execution (using both hardware and software technology) to guarantee rapid reconstruction of it on the host. Finally, this study has emphasized the importance of building extensible, composable programming environments such that a variety of tools can be applied in an integrated fashion.

Acknowledgements

Ralph London, Tim Standish, Anne Brindle, Dave Martin, Jeff Greenburg, and Carol LeDoux all contributed to the formulation of these ideas. 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 author 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. The support of The Aerospace Corporation is also gratefully acknowledged.

References

[Ada83]
Military Standard Ada Programming Language (ANSI/MIL-STD-1815A-1983). American National Standards Institute, 22 January 1983.

[DoD80]
Department of Defense. Requirements for Ada programming support environments. "Stoneman". February 1980.

[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.

[HLDB83]
Proceedings of the ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-

Level Debugging, Asilomar, CA. (March 1983). Appeared as *Software Engineering Notes, 8* 4, (August 1983).

[Hoar78]
Hoare, C.A.R. Communicating sequential processes. *Communications of the ACM, 17,* 10, 666-677 (1978).

[IRTB81]
Industrial Real-Time Basic. Draft Standard, European Workshop on Industrial Computer Systems (EWICS TC2), 1981.

[John79]
Johnson, Mark Scott. Translator design to support run-time debugging. *Software — Practice and Experience, 9,* 1035-1041 (1979).

[Mart77]
Martin, F. HAL/S - The avionics programming system for shuttle. Proc. AIAA Conference on Computers in Aerospace, Los Angeles, CA, pp. 308-318, 1977.

[Mori83]
Moriconi, Mark. PegaSys: an environment for displaying, animating, and reasoning about graphical descriptions of systems. Proceedings of the Symposium on Software Validation, Darmstadt, FRG, (September 1983) North-Holland.

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

[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).