SOFTWARE ENGINEERING

debugging
host-target development
concurrency
real-time software
Ada
environments

GÉNIE LOGICIEL

déverminage
développement croisé
parallélisme
temps réel
Ada
environnements

Debugging Real-Time Software in a Host-Target Environment

Techniques de localisation d'erreurs pour logiciels d'applications en Temps réel

Richard N. TAYLOR

Richard Taylor received his Ph. D in Computer Science from the University of Colorado in 1980. In 1981-82 he was an Assistant Professor of Computer Science at the University of Victoria, Canada. Since 1982, he has been an Assistant Professor of Information and Computer Science at the University of California, Irvine. His research interests are in programming environments and techniques for analyzing, testing and debugging concurrent software.

CSNet address : taylor @ uci
Arpanet : taylor.uci @ csnet-relay

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

RÉSUMÉ *Une stratégie très répandue pour le développement de logiciel de contrôle de processus consiste à effectuer l'essentiel de la mise au point et des tests sur un gros ordinateur dit « hôte » puis à transférer le code sur la machine de destination pour les contrôles finals et l'exécution en production. La machine « hôte » est en général de taille importante et offre un grand choix d'outils pour le développement de logiciel, alors que la machine pour laquelle le code est produit est petite et simple. Un des problèmes que présente une telle stratégie se manifeste lorsque le logiciel doit se plier à des contraintes en temps réel et est composé de plusieurs processus communicants. Si un test échoue sur la machine de destination, il peut être extrêmement difficile de déterminer la cause de l'échec. Les « debuggers » de la machine hôte ne sont pas utilisables parce que les mêmes données causent souvent des comportements différents sur leur machine. Ces divergences sont dues à des différences de vitesse, aux algorithmes de gestion, etc. Cet article propose une solution partielle à ce problème : il s'agit de reproduire l'exécution menant à l'échec de façon à rendre le « debugging » au niveau du langage source possible sur la machine hôte. Cette solution comprend l'utilisation intégrée d'un analyseur statique de parallélisme, d'un interpréteur interactif et d'outils de visualisation graphique des programmes. Bien qu'elle soit d'application générale, la solution est décrite ici dans le contexte de programmes en Ada.*

TABLE DES MATIÈRES

1. Introduction
1.1 Objective

2. Solution Scheme
2.1 Static analysis of concurrent programs
2.2 Path finding strategy
2.3 Speed-up through dynamic analysis
2.4 Execution visualization and intra-task debugging

3. Implementation
3.1 Some implementation issues

4. Conclusion

Acknowledgments

References

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

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 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 rendez-vous), 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 multitasked systems [Ada 83]. The technique has broad applicability, however, and could be used in debugging, for example, CSP [Hoare 78], HAL/S [Martin 77], or Industrial Real-Time Basic [IRTB 81] programs.

Several other research groups are investigating the problems of debugging concurrent and distributed systems. A variety of promising work is described in [HLDB 83]. 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 machi-

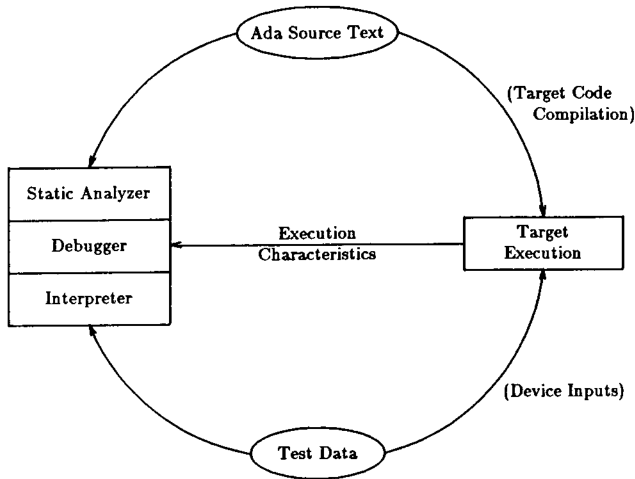

Figure 1. — Overview of Host-Target Debugging.

ne'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.
- The process of reconstructing the execution may be terribly slow, though it certainly will be more efficient than having a person attempt the same. (After presentation of the basic technique we will describe a series of optimizations that may make the process tractable — at the expense of requiring more information about the target's execution than just the data and the final state.)

A technology key to the entire process is static analysis of concurrent Ada programs. This technology is described in detail in [Taylor 83a]. The following subsection summarizes the key points.

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 indicates 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 sub-paths taken internally to a task are irrelevant, as long as they do not affect the synchronization activities.) 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.

The concepts will be illustrated with an example. Figure 2 presents an Ada program designed to solve a version of the familiar 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. There are only five forks on the table, one between each of the five plates. The program of Figure 2 simply models the system. Here each philosopher is a separate task, as is each fork. The philosopher tasks request the fork ressources by issuing entry calls. The program presented is a poor one in the sense that it is possible for deadlock to occur : if all five philosophers are able to simultaneously acquire the fork to their 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 statements 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. $Fork_0$ is shown at " Down' " signifying it is now *awaiting* a call on Down — in the model this signifies that it is currently held. Since Kierkegaard was on the $Fork_1$ queue first, it is possible for the system to progress to

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'

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

```
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 1.0;                    --eating time
         Forks(N).Down;                --put down left fork
         Forks((N+1) mod 5).Down;      --put down right fork
         delay 1.0;                    --thinking time
      end loop;
   end T;
end Philosopher;

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

begin
    null;
end Dining_Philosophers;
```

Figure 2. — Dining Philosophers, Reserved Seating.

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 makes no assumptions concerning relative processor speed, scheduler algorithm, or the like — all possibilities are explored and reported.

Unfortunately static analysis also has several significant limitations. First, it 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 most real programs. A second, much more significant limitation, is that static

analysis is accurate only when individual program objets (especially tasks or entries) can be precisely identified statically. Program features that cause dynamic identification, such as access values and subscripts, may be inadequately handled. In the current formulation of the analysis algorithm a family of entries is treated as a single entry. Acess types are not handled at all in the current version. 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 computed all possible concurrency states, using all combinations of 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 [Taylor 83b]. One suspects that normally a very large number of states will be generated, but we currently do not have any experimental verification of this. Certainly some program organizations will lead to fewer concurrency states than others.

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. Once again, all possibilities are considered. 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 and refinements 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. Furthermore the quality of the reconstruction is improved.

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, though they must be in order, and could be captured by 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 $h_1 = s_1\, s_2\, s_3 \dots s_n\, s_{n+1} \dots C$ 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 s' that leads eventually to C, then that history $h_2 = s_1\, s_2 \dots s_n\, s' \dots C$ is pursued, again until reaching C in the host execution or until the process can continue no further. If the process stalls yet again, 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 concurrency 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-