COMPUTER SYSTEMS LABORATORY

DEPARTMENTS OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
STANFORD UNIVERSITY · STANFORD, CA 94305-2192

Debugging Ada Tasking Programs

David Helmbold & David Luckham

Program Analysis and Verification Group Report No. 25
Technical Report No. 84-262

July 1984

This work was supported by the Air Force Office of Scientific Research under Grant AFOSR 83-0355 and the Advanced Research Projects Agency, Department of Defense, under Contract N00039-84-C-0211.

Debugging Ada Tasking Programs

by

David Helmbold & David Luckham

Program Analysis and Verification Group
Computer Systems Laboratory
Stanford University

Abstract

A new c/ass of errors, not found in sequential languages, can result when the tasking constructs of Ada are used. These errors are called deadness errors and arise when task communication fails. Since deadness errors often occur intermittently, they are particularly hard to detect and diagnose. Previous papers, [2] and [4], describe the theory and implementation of runtime monitors to detect deadness errors in tasking programs. The problems of detection and description of errors are different. Even when a dead state is detected, giving adequate diagnostics that enable the programmer to locate its cause in the Ada text is difficult. This paper discusses the use of simple diagnostic descriptions based on Ada tasking concepts. These diagnostics are implemented in an experimental runtime monitor. Similar facilities could be implemented in task debuggers in forthcoming Ada support environments. Their usefulness and shortcomings are illustrated in an example experiment with the runtime monitor. Possible future directions in task error monitoring and diagnosis based on formal specifications are discussed.

Keywords: *Ada, tasking, deadness errors, debugging, testing, concurrent programming, concurrent specifications.*

This work was supported by the Air Force Office of Scientific Research under Grant AFOSR 83-0355 and the Advanced Research Projects Agency, Department of Defense, under Contract N00039-84-C-0211.

1. Int roduction

Deadness errors occur when part of a concurrent computation can no longer proceed due to task communication failure. Circular deadlock is one kind of deadness error. Circular deadlock arises when there is a closed loop of tasks in which each task has issued an (untimed, unconditional) entry call to the next task in the loop. Since each task will wait until its entry call is serviced, al! of the tasks will continue to wait indefinitely. Many other kinds of deadness errors are possible in Ada ([4]; see section 2). Deadness errors often occur unpredictably. Different scheduling algorithms the arbitrary choice of open select alternatives, and even the load due to other users on the host machine can affect whether or not a particular program run will exhibit a deadness error. This makes deadness errors very hard to reproduce, and hence to debug.

Previous papers have presented both compiletime [7, 8] and runtime [2, 4] techniques for detecting deadness errors. Compiletime methods are suitable for locating very simple errors, but they necessitate an exponential search. Furthermore, it is not clear that compiletime methods can be developed for situations where tasks are dynamically activated.

Runtime monitoring is capable of detecting a large class of Ada deadness errors and is therefore potentially a more practical approach to task debugging. There is, of course, a runtime overhead, but this penalty is probably not unreasonable during the debugging phase of program development. Additionally, more efficient runtime monitor designs are being developed [3] for multi-processor systems.

One of the more pressing problems with runtime monitoring is how to provide a useful description of a deadness error. Simply detecting dead states at runtime requires less information than is needed to satisfactorily describe those states. An adequate description of a dead state must enable the programmer to find the cause of the error in the original Ada text.

In this paper we describe the debugging facilities provided in an experimental runtime monitor. These facilities use Ada semantic concepts to describe the tasking state of a program. Similar facilities could be implemented, using our techniques, in task debuggers of future Ada programming support environments. An example is given to illustrate how the descriptions of deadness errors supplied by this monitor can aid debugging. The example also shows why the descriptions can be inadequate for complicated errors. Finally, we describe several possible enhancements to the monitor's debugging facilities.

2. Ada Deadness Errors

Descriptions of deadness errors are based on the concept of task status. The status of a task is what that task is doing, namely:

1. *Running;* executing its own code (including delay statements).

2. *Calling;*
   a. *Enqueued;* issuing an entry call which has not yet been answered.
   b. *In Rendezvous;* issuing an entry call which has been accepted.
   c. *Circular/y Deadlocked;* issuing an entry call which is part of a circular deadlock (see below).

3. *Accepting;* waiting for an entry call at an accept statement or simple select statement.

4. *Select-terminate;* waiting at a select statement with an open terminate alternative.

5. *Select-dependents-completed;* waiting at a select statement with an open terminate alternative and all dependents have status terminated or select-dependents-completed.

6. *Block-waiting;* waiting at the end of a block for tasks dependent on the block to terminate.

7. *Completed;* finished executing its code, and is waiting for its dependents to terminate.

8. *Terminated;* finished executing its code, and all dependents have terminated.

Most statuses have some extra information associated with them. For example, the task and entry called are associated with the status calling. For a more formal definition of these task statuses, see [4].

Any task that cannot proceed with its computation because it is waiting for another task (or tasks) is *blocked* (statuses 2 through 7). Tasks executing delays are not considered blocked, since they can proceed with their computation without the help of another task. A task is dead if it is blocked and there is no possible continuation of the computation which can cause the task to become unblocked.

The *tasking state* of a program is the the set of tasks that have been activated by the program, their statuses, and any associated information. A deadness error occurs when a program's tasking state contains a dead task.

Deadness errors can be categorized in many ways. For our purposes, deadness errors are best categorized by the statuses of the tasks involved. This categorization has the property that, in similar deadness errors, the tasks involved are doing similar things. Here are three (overlapping) kinds of deadness errors:

1. *Global blocking* errors occur when at least one task of the program is blocked, and every other task is either blocked or terminated. When global blocking occurs, no more useful work can be done by the tasking system. Most deadness errors lead to global blocking as the tasks which are still running either terminate, call a blocked task, or wait for a blocked task.

2. *Circular deadlock* arises when there is a closed loop of tasks such that each task has issued an entry call to the next task in the loop. Each task in the loop waits forever for the next task to accept its entry call.

3. *Local blocking:* Let S be the set of tasks directly or indirectly dependent on task T (or a block executed by T). All tasks of S are blocked or terminated and at least one is not terminated, none is issuing an entry call to a task outside of S U T, and task T is either block-waiting, completed, or issuing an entry call to a task in S.

These deadness errors depend only on the statuses (and associated information) of the tasks involved, so they are easy to detect given an accurate picture of the tasking state. There are numerous other forms of deadness errors which depend on the code executed by the tasks in the system. For example, *Call-wait deadlock* occurs when task T, is accepting an entry that can be called only by task T,, and T_2 is calling a different entry of T,. Many of these deadness errors can be detected if sufficient semantic information is communicated to the monitor. However, determining if a deadness error is present in the general case is as difficult as the halting problem. Some types of distributed computation errors, such as starvation, are not considered deadness errors.

3. Runtime Monitoring

Runtime monitoring involves inserting a monitor task into the source program. This monitor receives information from the program, enabling it to detect deadness errors and print debugging information. Before a program can use the monitor, the program must undergo a series of transformations. These transformations insert calls to the monitor, so the monitor can follow the task interactions of the program. The monitor itself is implemented as an Ada task, so the monitored program can be run in any Ada system.

Versions of the transformations are described in detail in both [2] and [4]. In order for the monitor to follow what is happening in the program, the program must tell the monitor what its tasks are doing. This communication must be unambiguous, so some sort of task identification (ID) is required. Access types to task objects are recommended in [1]. Unfortunately if we used this technique, the strong typing of Ada would require the monitor to use one access type for each possible task type in the program. Therefore our transformations allow for an integer ID to be associated with each task. Each task is able to access its own ID as well as the ID of any task which is visible to it. Both task IDs are visible during rendezvous. These IDs are initialized by the monitor and remain constant throughout execution of a program.

In addition to the numeric IDs, the monitor attempts to keep meaningful string identifiers for the tasks. These string identifiers are derived from names in the Ada program, e.g. the task object name, the task type name, the name of the access type designating the task type, or the name of a composite variable containing task components (such as array names). Although these names are often useful, they do not always identify tasks uniquely with the source text, i.e., they may be associated with more than one activated task.

The transformed program tells the monitor what its tasks are doing. The transformation inserts entry calls to the monitor inside each task body. These new entry calls inform the monitor when the task is about to make an entry call, execute an accept or select statement, start or end a rendezvous, leave a block, or activate a su btask.

Based on these calls, the monitor creates a *picture* of the program's tasking state. This picture is updated and checked for certain deadness errors whenever new information' is received from the program. Unfortunately, since the monitor calls and the actions that trigger them are not simultaneous, the picture may not exactly mirror the state of the tasking system. A call notifying the monitor of the execution of a tasking statement may be early or late. These early and late notifications may cause the picture to reflect a tasking state that never arises, or to miss a state which does occur. What we do know [4] is that, at any time: at least one possible continuation of the computation causes the picture and the actual tasking state to agree; and if the picture shows a deadness error, that deadness error will always exist sooner or later in the program's tasking state.

4. Debugging Facilities

To aid debugging, the monitor can print snapshots and trace task interactions. A snapshot is a textual description of the monitor's picture, and is automatically generated when a global blocking situation arises. In addition, when circular deadlock occurs, the monitor prints a few lines describing the tasks involved. The task tracing is controlled by pragma-like calls to the monitor. These calls allow tracing to be started and suspending for individual tasks or the entire program. Snapshots can also be generated by monitor pragmas.

Each snapshot includes, for each activated task: the string name, the numeric ID, the status, the entries and entry queue sizes, and the task being called (if any). For example, the following snapshot fragment shows that task number 1, source name OPERATOR, has issued an entry call to the CHANGE entry of task number 2. The operator task has three other tasks queued up at its PRE-PAY entry and no tasks waiting at the CHARGE entry. Task number 3, a customer, is at an accept statement waiting for a call to its CHANGE entry (indicated by the "*"). Both tasks are directly dependent on task number 0 (the main program) and neither task is currently being traced.

```
1 OPERATOR          is calling task number 2 at entry CHANGE
Its entries are:     PRE-PAY          (3) CHARGE          (0)
  Its father is: 0.  Its tracing is disabled.

3 CUSTOMERS         is accepting
Its entries are:    CHANGE          (0*)
  Its father is: 0.  Its tracing is disabled.
```

Other information, such as the tasks on a given entry queue, can usually be determined from a snapshot. However, since the monitor does not interact with the task scheduler (runtime supervisor) the task state (in particular the entry queues) cannot be directly manipulated.

When trace information is requested for a task, output is generated each time a monitor call involving that task is accepted. For example:

```
**TRC** call of monitor entry CALLING.
      The consumer is 3 [CUSTOMERS    ]. The server is 13 [PUMPS    ].
      The entry is [START-PUMPING ].
```

indicates that an entry call has been (or will be) issued; the consumer is a CUSTOMER task, number 3; the server is a PUMP task, number 13; and the called entry is START-PUMPING. This would be generated if the tracing for either task 3 or task 13 is enabled.

We have developed a simple runtime interface for interactive debugging. This interface allows the programmer to *sing/e-sfep* through a task's interactions and enter monitor pragmas directly from the keyboard. The commands accepted by the interface include: run the program for *n* monitor calls; print a snapshot; and turn on\off monitor tracing. After the command has completed, the interface prompts for another command. These facilities enable the programmer to tailor his debug output to the situation arising on each run.

5. An Automated Gas Station

The following example illustrates the usefulness and shortcomings of our current monitor.

An automated gas station consists of an operator with a computer, a set of pumps, and a set of customers. Each customer has a unique identification (basically a credit card number called a customer id) and each pump has a unique number (called a pump id). The operator handles payments and schedules the use of pumps with the aid of the computer. A gas station may be specified informally as follows.

Each customer arrives at the gas station wanting a random amount of gas from a random pump. The customer first goes to the operator and prepays for the pump he wants to use. Then the customer goes to the pump and starts it. When the customer is finished, he turns the pump off and collects any change from the operator.

The pumps must be activated by the operator before they can be used by customers. Each time a pump is activated, it is given a limit (the prepayment) on the amount of gas that can be pumped. When the pump is shut off, it reports the amount of gas dispensed to the operator. The pump then waits until it is activated again.

Whenever ready, the operator may either accept a prepayment from a customer, or receive a report from a pump. When a prepayment is accepted a record is entered in the computer showing the prepayment, the customer id, and the pump id. If the pump is not already in use, it is activated with the proper limit. When a pump reports a completed transaction, the current prepayment record for that pump is retrieved from the computer. The charges are computed and any overpayment is refunded to the customer. If another customer is waiting for the pump, then it is reactivated with that customer's prepayment (retrieved from the computer).

Note: Our example is a somewhat idealized version of a real life situation that does occur as a result of introducing automated bankcard facilities into some automated gas stations (we are indebted to J. Misra for bringing it to our attention). The example does.not require input data, so the problem of debugging deadness is not complicated by the necessity of selecting appropriate test data. Note also that our coding of the example (below) does not follow the safest possible rules for avoiding deadness.

5.1 **Implementation.**

An automated gas station can be modelled in Ada by a program containing tasks for each active component. Our program below consists of a task representing the station operator, a set of pump tasks, and a set of customer tasks

The declarations of the tasks can be derived from the informal specification, assuming ids. are represented by integers, as follows.

```
task OPERATOR is
     entry PRE_PAY (AMOUNT          : in INTEGER:
                    PUMP-ID         : in INTEGER;
                    CUSTOMER-ID     : in INTEGER);
     entry CHARGE( AMOUNT    : in INTEGER;
                   PUMP-ID : in INTEGER) ;
end OPERATOR:

task type PUMP is
     entry ACTIVATE(LIMIT : in INTEGER);
     entry START-PUMPING ;
     entry FINISH-PUMPING (AMOUNT-CHARGED : out INTEGER) :
end PUMP ;

task type CUSTOMER is
     entry CHANGE (AMOUNT : in INTEGER);
end CUSTOMER:
```

The gas station implementation uses a COMPUTER package, a random number package, and a standard I/O package. The computer package is simply a FIFO queue of records of prepayments for each pump; these are used by the operator. The random number generator is used by customers to decide how much gas they need and which pump to request, and also by pumps to determine the amount of gas actually pumped.

In our version of this example, the gas station is initialized so that there is an array of three pumps, called PUMPS, and an array of ten customers, CUSTOMERS. When these tasks are activated they are each given their position in their array as an ID (i.e., their customer identification or pump number); the main program calls a special entry in each pump and customer to do this before the general activity is allowed to proceed. Details of the initialization are omitted from the description below.

We may outline the corresponding task bodies as follows:

```
with TTY-IO, COMPUTER, RANDOM; use TTY-IO;

task body OPERATOR is
    AMOUNT-PAID, CUSTOMER-NUMBER : INTEGER;
begin
    loop
        select
            accept PRE-PAY (AMOUNT          : in INTEGER;
                            PUMP-ID         : in INTEGER;
                            CUSTOMER-ID     : 'in INTEGER) do
                -- if no previous customer is waiting, activate this pump.
                if COMPUTER.EMPTY_QUEUE(PUMP_ID) then
                    PUMPS(PUMP_ID) .ACTIVATE(AMOUNT):
                end if;
                -- enter a record of prepayment; records for each pump are
                -- serviced in FIFO order.
                COMPUTER.ENTER(PUMP_ID, AMOUNT, CUSTOMER-ID);
            end PRE-PAY;
        or
            accept CHARGE( AMOUNT : in INTEGER: PUMP-ID : in INTEGER) do
                -- Retrieve current record for this pump (i.e., the caller).
                COMPUTER.RETRIEVE(PUMP_ID,AMOUNT_PAID,CUSTOMER_NUMBER);
                -- Give change to the customer.
                CUSTOMERS(CUSTOMER_NUMBER).CHANGE(AMOUNT_PAID - AMOUNT);
                -- If another customer is waiting for this pump, activate it
                if not COMPUTER. EMPTY-QUEUE( PUMP-ID) then
                    PUMPS(PUMP_ID).ACTIVATE(COMPUTER.TOP_AMOUNT(PUMP_ID));
                end if;
            end CHARGE ;
        end select;
    end loop;
end OPERATOR;
```

```
task body PUMP is
     MY-NUMBER          : INTEGER;
     CURRENT-CHARGES : INTEGER;
     AMOUNT-LIMIT       : INTEGER:
begin
     . . .   -- MY-NUMBER is initialized to a unique pump id.
     loop
          accept ACTIVATE(LIMIT : in INTEGER) do
               AMOUNT-LIMIT := LIMIT;
          end ACTIVATE:
          accept START-PUMPING;
          accept FINISH-PUMPING( AMOUNT-CHARGED : out INTEGER) do
               - - Compute actual charge as a random number less than the limit.
               CURRENT-CHARGES := RANDOM.GET_CHARGES(AMOUNT_LIMIT);
               AMOUNT-CHARGED  := CURRENT-CHARGES;
               OPERATOR.CHARGE (CURRENT-CHARGES, MY-NUMBER);
          end FINISH-PUMPING;
     end loop;
end PUMP;

  - - Each customer task will model a sequence of customers entering
  -- and leaving the gas station;

task body CUSTOMER is
     MY-NUMBER        : INTEGER;
     MY-MONEY         : INTEGER;
     PUMP-NUMBER      : INTEGER;
     AMOUNT-PUMPED : INTEGER;
begin
     . . .   -- MY-NUMBER is initialized to a unique customer id.
     loop
          - - Randomly decide amount of gas needed and choice of pump.
          RANDOM.DRIVE_IN(MY_MONEY, PUMP-NUMBER);
          OPERATOR.PRE_PAY(MY_MONEY, PUMP-NUMBER, MY-NUMBER);
          PUMPS(PUMP_NUMBER).START_PUMPING;
          PUMPS(PUMP_NUMBER).FINISH_PUMPING(AMOUNT_PUMPED);
          accept CHANGE(AMOUNT : in INTEGER) do
          . . .   -- compare AMOUNT with MY-MONEY less the AMOUNT-PUMPED;
                  -- output a complaint to the programmer's console if not equal.
          end CHANGE;
     end loop:
end CUSTOMER;
```

Note: The program variables for the customer identifications and pump numbers are not related to the unique task IDs assigned by the monitor.

5.2 The First Debugging Session.

The first run of the monitored version of the gas station resulted in the following output from the monitor to the programmer's console:

```
*** Interactive Task Tracing  after 0 monitor calls.  Type '?' for help.
=> r
```

This is the interface's prompt. The "r" command tells the interface to run the program for a number of task interactions.

```
*** Number of monitor calls until interacting (follow with a <CR>)? 1000
```

Response tells the monitor to let the program run until 7000 tasking statements have been executed before entering interactive mode again.

```
**MON** A circular deadlock has occurred.
```

A description of the circular deadlock follows.

```
Task OPERATOR     is calling entry CHANGE of task CUSTOMERS
Task CUSTOMERS    is calling entry FINISH-PUMPING of task PUMPS
Task PUMPS        is calling entry CHARGE of task OPERATOR
**MON** thus forming a closed loop of tasks.
```

This description is given using the source text strings associated with each task when it is activated. There are actually 10 customer task components associated with the array name, CUSTOMERS, and three pump task components associated with the array name, PUMPS. So we don't know from this which customer and pump are in the deadlock, nor what the other tasks are doing.

However, before we have time to think, everything grinds to a halt (as would be expected since the operator is called by all of the other tasks):

```
**MON** GLOBAL DEADLOCK HAS BEEN DETECTED
```

Now follows a complete picture of the dead state. Each task status description is preceded by its unique task id; each entry is followed by it's queue count and a '' if the task is accepting that entry. Entries named GET_..._ID are used to initialize the gas station and are omitted from discussion, although the monitor will monitor them.*

```
**MON**    TASK INFORMATION
0 MAIN      is completed
Its entries are:  <NONE>
   Its father is: -1.  Its tracing is disabled.

1 OPERATOR   is deadlocked, calling task number 2 at entry CHANGE
Its entries are:  PRE_PAY   (5) CHARGE    (1)
   Its father is: 0.  Its tracing is disabled.

2 CUSTOMERS is deadlocked, calling task number 13 at entry FINISH-PUMPING
Its entries are:  CHANGE    (1) GET-CUSTOMER-ID (0)
   Its father is: 0.  Its tracing is disabled.

3 CUSTOMERS is calling task number 13 at entry START-PUMPING
Its entries are:  GET-CUSTOMER-ID (0)
   Its father is: 0.  Its tracing is disabled.
```

Since the monitor does not list the entry CHANGE for this customer task, we know that no call to that entry has been rnade during this execution.

```
4 CUSTOMERS is calling task number 13 at entry START-PUMPING
Its entries are:  GET-CUSTOMER-ID (0)
   Its father is: 0.  Its tracing is disabled.
```

```
5 CUSTOMERS is calling task number 13 at entry START-PUMPING
Its entries are:  GET-CUSTOMER-ID (0)
  Its father is: 0.  Its tracing is disabled.

6 CUSTOMERS is calling task number 14 at entry FINISH-PUMPING
Its entries are:  GET-CUSTOMER-ID (0)
  Its father is: 0.  Its tracing is disabled.

7 CUSTOMERS is calling task number 1 at entry PRE-PAY
Its entries are:  GET-CUSTOMER-ID (0)
  Its father is: 0.  Its tracing is disabled.

8 CUSTOMERS is calling task number 1 at entry PRE-PAY
Its entries are:  GET-CUSTOMER-ID (0)
  Its father is: 0.  Its tracing is disabled.

9 CUSTOMERS is calling task number 1 at entry PRE-PAY
Its entries are:  GET-CUSTOMER-ID (0)
  Its father is: 0.  Its tracing is disabled.

10 CUSTOMERS is calling task number 1 at entry PRE-PAY
Its entries are:  GET-CUSTOMER-ID (0)
  Its father is: 0.  Its tracing is disabled.

11 CUSTOMERS is calling task number 1 at entry PRE-PAY
Its entries are:  GET-CUSTOMER-ID (0)
  Its father is: 0.  Its tracing is disabled.

12 PUMPS     is accepting
Its entries are:  ACTIVATE   (0*) GET PUMP ID    (0)
  Its father is: 0.  Its tracing is disabled,

13 PUMPS     is deadlocked, calling task number 1 at entry CHARGE
Its entries are:
FINISH-PUMPING (0) START-PUMPING (3) ACTIVATE  (0) GET-PUMP-ID (0)
  Its father is: 0.  Its tracing is disabled.

14 PUMPS     is calling task number 1 at entry CHARGE
Its entries are:
FINISH-PUMPING (0) START-PUMPING  (0) ACTIVATE  (0) GET-PUMP-ID (0)
  Its father is: 0.  Its tracing is disabled.

**MON**  end of task information.
Nobody ready to run - going to sleep
```
this final comment is from the runtirne task supervisor package; the monitor is now blocked in status accepting, waiting for further messages.

Summary:
The dead state description may be summarized as follows:

- The operator (task 1), customer task 2, and pump task 13 are in a circular deadlock. The operator is calling the customer to give change, the customer is in a rendezvous with the pump at finish pumping, and the pump is in a rendezvous with the operator at the charge entry.

- The remaining state picture describes the global blocking of all tasks that resulted from the initial deadlock. Two customers (tasks 3 and 4) are also waiting for pump task 13. Customer task 6 is in rendezvous with pump task 14, while pump task 14 is blocked trying to signal the operator. The other customers are all blocked calling the operator at prepay, and pump task 12 is waiting to be activated by the **operator.**

Analysis:
The global blocking resulted from a circular deadlock containing the operator. The static description of the circular deadlock identifies the actors and their statuses with the source text adaquatly. It is therefore possible in this case to decide how to alter the source text so that this particular deadlock cannot occur again.

The first line of attack is to simply reorder the rendezvous statements of a single task. For example, we may try changing the order of two entry calls, an entry call and an accept statement, or two accept statements (this includes unnesting of accepts or entry calls within accept bodies). Replacing an entry call by a timed entry call may also be considered as part of this first level syntactic approach.

In this example, there are two obvious possibilities, both involving the unnesting of nested rendezvous.

1. In the pump body: move the entry call to the operator out of the accept body of Finish Pumping. This implies that the customer completes its call to the pump before the pump calls the operator.

2. In the operator body: move the entry call to the customer out of the accept body of Charge. This implies that the pump completes its call to the operator before the operator calls the customer.

The second transformation was chosen since it releases the pump earlier and therefore tends to maximize the concurrent use of pumps. If the **if** statement activating the pump is not also moved out of that accept body, the next monitored run reveals a circular deadlock between the operator and a pump on any run in which another customer is waiting for that pump.

The accept alternative for CHARGE in the operator body now reads as follows (new variables, CHANGE-AMOUNT, IDLE-PUMP are declared in the OPERATOR body):

```
select
    ...
or
    accept CHARGE( AMOUNT  : in INTEGER; PUMP-ID : in INTEGER) do
        -- Retrieve current record for this pump (i.e., the caller).
        COMPUTER.RETRIEVE(PUMP_ID, AMOUNT-PAID, CUSTOMER-NUMBER) ;
        -- Use new local variables declared in the operator body.
        CHANGE-AMOUNT := AMOUNT-PAID - AMOUNT;
        IDLE-PUMP      : = PUMP-ID ;
    end CHARGE;
        -- Give change to the customer.
        CUSTOMERS(CUSTOMER_NUMBER).CHANGE(CHANGE_AMOUNT);
        -- If another customer is waiting for this pump, activate it.
        if not COMPUTER. EMPTY-QUEUE( IDLE-PUMP) then
            PUMPS(IDLE_PUMP).ACTIVATE(COMPUTER.TOP_AMOUNT(IDLE_PUMP));
        end if:
```

```
end select ;
```

The corrected text was recompiled and monitored again. This version of the gas station does not exhibit any further deadness errors when monitored using the standard runtime task scheduler of the Adam system [5] (this scheduler preempts a running task after it has used up its time-slice). Furthermore, the example will now run without deadness errors on any scheduler that has a FIFO runqueue and no preemptive time-slicing.

5.3 Monitoring with Scheduler Changes.

The remaining potential deadness errors in this example will occur only if the underlying runtime task scheduler is changed sufficiently.

One problem is how to enumerate a set of changes to the scheduler so as to guarantee the promotion of all potential deadness errors. In general this involves changes both to the runtime queue management and to the size of the preemptive timeslice. This problem is analagous to test vector generation for the promotion of single "stuck at" faults in integrated circuits. Although we may expect the general problem to be exponential in the number of possible rendezvous, there is probably much that can be done using the tasking structure and semantics of the subject program to reduce the number of changes that must be considered. In a sophisticated APSE one may expect a user interface to the scheduler for definition of scheduling algorithms. Perhaps one choice should be a nondeterministic scheduler, where each scheduling decision is made randomly.

Changes in the runtime scheduler may be simulated to some extent by source text changes: changes of task priorities and insertion of delay statements. In our gas station example, a delay statement with an expression having a random value was placed in the customer task type body immediately after prepaying the operator:

```
        delay RANDOM.GET_DELAY;
```

While a task is executing a delay it is taken off the scheduler's runqueue. Insertion of random delays has theeffectof randomly reordering the runqueue.

One of the subsequent monitored runs detects a global blocking:

```
**MON** GLOBAL BLOCKING HAS BEEN DETECTED
**MON**    TASK INFORMATION
```
Now follows a picture of the global blocked state.

```
0 MAIN                is completed
Its entries are:        <NONE>
  Its father is: -1.  Its tracing is disabled.

1 OPERATOR            is calling task number 2 at entry CHANGE
Its entries are:   PRE_PAY     (6)  CHARGE      (2)
  Its father is:  0.  Its tracing is disabled.

2 CUSTOMERS           is calling task number 13 at entry START-PUMPING
Its entries are:   CHANGE      (1)  GET-CUSTOMER-ID  (0)
  Its father is:  0.  Its tracing is disabled.
```

```
3  CUSTOMERS        is  accepting
Its entries are:  CHANGE      ( 0*) GET-CUSTOMER-ID (0)
  Its father is: 0.  Its tracing is disabled.

4  CUSTOMERS        is calling task number 12 at entry FINISH-PUMPING
Its entries are:  GET-CUSTOMER-ID  (0)
  Its father is: 0.  Its tracing is disabled.

5  CUSTOMERS        is calling task number 14 at entry FINISH-PUMPING
Its entries are:  GET-CUSTOMER-ID  (0)
  Its father is: 0.  Its tracing is disabled.

6  CUSTOMERS        is calling task number 1 at entry PRE-PAY
Its entries are:  GET-CUSTOMER-ID (0)
  Its father is: 0.  Its tracing is disabled.

...
```

CUSTOMERS 7 - 11 are similarly blocked calling PRE-PAY.

```
12 PUMPS            is calling task number 1 at entry CHARGE
Its entries are:
FINISH-PUMPING (0) START-PUMPING (0)  ACTIVATE (0) GET-PUMP-ID (0)
  Its father is: 0.  Its tracing is disabled.

13 PUMPS            is  accepting
Its entries are:
FINISH-PUMPING (0) START-PUMPING  (1) ACTIVATE (0*) GET-PUMP-ID (0)
  Its father is: 0.  Its tracing is disabled.

14 PUMPS            is calling task number 1 at entry CHARGE
Its entries are:
FINISH-PUMPING (0) START-PUMPING (0) ACTIVATE (0) GET-PUMP-ID  (0)
  Its father is: 0.  Its tracing is disabled.
**MON**  end of task information.
** Nobody ready to run - going to sleep
```

Final comment from supervisor.

Summary:
The dead state is a global blocking and does not contain any circular deadlock. The operator (task 1) is calling customer task 2 to give change, but task 2 is calling pump task 13 to start pumping. Task 13, however, has not yet been activated by the operator, and is blocked in status accepting activate. Customer task 3 is blocked in status accepting, waiting for change. Customer tasks 4 and 5 are both blocked calling the finish pumping entries of pump tasks 12 and 14; these pumps are both blocked calling the Charge entry of the operator. The remaining six customers are blocked waiting to prepay the operator.

Analysis:
This static picture still provides enough information relating the state to the source text to suggest a syntactic change which will ensure that this state cannot occur again.

In the operator body (see corrected version, section 5.2): change the order of the entry calls giving the customer change and activating the pump, so that the pump is activated first. Now customer 2 and pump 13 will be able to proceed.

```
select

or
    accept CHARGE( AMOUNT  : in INTEGER; PUMP-ID : in INTEGER) do
        . . .
    end CHARGE;
        -- Reordered entry calls in the operator body:
        -- if another customer is waiting for this pump, activate it.
        if not COMPUTER. EMPTY-QUEUE( IDLE-PUMP) then
            PUMPS(IDLE_PUMP).ACTIVATE(COMPUTER.TOP_AMOUNT(IDLE_PUMP));
        -- Give change to the customer.
        CUSTOMERS(CUSTOMER_NUMBER).CHANGE(CHANGE_AMOUNT);

end select;
```

After this repair, another monitor run (again with random delays), shows that a circular deadlock occurs. Customer task 2 is calls FINISH-PUMPING; during this rendezvous, the pump calls the operator at the charge entry; meanwhile the operator is calling customer task 2 to return change. This is easily fixed by moving the call to the operator in the pump body out of the accept FINISH-PUMP I NG statement.

Pump body:

```
            . . .
        accept FINISH_PUMPING(AMOUNT_CHARGED : out INTEGER) do
            -- Compute actual charge as a random number less than the limit.
            CURRENT-CHARGES  := RANDOM.GET_CHARGES(AMOUNT_LIMIT);
            AMOUNT-CHARGED := CURRENT-CHARGES;
        end FINISH-PUMPING;
            OPERATOR.CHARGE(CURRENT_CHARGES, MY-NUMBER);
    end loop:
end PUMP:
```

After this final "fix", the gas station functions without any further dead states occurring under any runtime scheduling. However, customers complain from time to time about getting'the wrong change.

5.4 Inadequacy of Dead State Descriptions.

Obviously there is something "fishy" that is not expressed in the dead state description (5.3) since customer task 3 is blocked accepting change when the operator is trying to give change to customer task 2 who is trying to start pumping. By the time the dead state occurs, the "real" error has already happened and is overwritten in the state by subsequent events.

The sequence of events leading to the deadness error holds the clues. Since the prior sequence of events is not kept by the monitor, the programmer must essentially step through the task interactions, either by hand or using the stepping feature of the monitor interface, to discover what actually goes wrong. Our example is simple enough so that the race condition between customer tasks 2 and 3 can be easily discovered. They arrive at the operator in one order, request the same pump, and then may arrive at the pump in a different order depending on the task scheduling (or external events that affect the amount of computation in a given time slice). But the operator's computer keeps their prepayment records in the order in which they prepay. Even if there are no deadness errors, a customer may get change intended for someone else.

We note also that it is possible to fix the source text so that dead states will not occur on the basis of state descriptions without ever considering the plight of customer task 3. This might be considered as a case of "a little learning being a bad thing". More to the point, deadness should not be considered separately from other kinds of tasking errors such as incorrect synchronization.

The question is: what can be provided beyond descriptions of tasking states? Already, dead state descriptions can become unmanageable and contain much that is irrelevant to describing the real error. Histories of events will be even more irrelevant in most cases, and very costly to monitor. Simple tracing and stepping facilities may be of some use. But manual stepping towards an error can be a tedious chore if the program is anything more than a toy example. Manual stepping may be combined with automatic monitoring, the idea being to allow the user to manually step the tasking interactions when the state sequence is close to the error. However, even the interactions between the user and the monitor may affect the scheduling that promotes the error.

6. Future Debugging Facilities:

The facilities currently provided by the monitor are helpful, but in general, inadequate. However, the concept of task state, and the data structures and algorithms implemented in the present monitor, provide a basis for more sophisticated task error 'monitoring.

There are essentially three kinds of possible extensions. First are simple extensions to provide more details about a dead state. These are easy to implement and would help in some situations. The second concern the monitoring of histories. Because history is often important in determining the cause of a deadness error, the programmer should be able to query the past tasking states for relevent information; the main problems are how to monitor histories efficiently and separate out relevent events. The third extension is to allow the programmer to specify constraints on sequences of task states. These specifications would be supplied either with the program or interactively during a monitor session. They would be monitored just like deadness errors are checked for now.

6.1 Additional State Information

The current monitoring of task states and associated information can be enhanced in many ways: more accurate task naming, more detailed information associated with task statuses, entry parameter tracing, and monitoring flow of control could all be implemented.

The current monitor assigns the same source text name to all tasks in the same array or of the same access type. If the programmer was allowed to chose his own mnemonic names for these (and other) tasks, then the monitor's output could be more meaningful. The status calling currently stands for two statuses, enqueued and in rendezvous. By distinguishing these, we give the programmer more information about the state of race conditions and nested rendezvous.

Additional information could be monitored. Obvious candidates are entry parameter values and control information. Once monitoring of parameters is implemented, it should be little more trouble to keep track of critical program variables as well. One of the current monitor's most important defects is the lack of control or program counter information. In our simple example each entry has only one accept statement and is called from only one place. In more complicated examples it becomes important to know at which accept or call statement a task is waiting.

6.2 State Histories

Often by the time a deadness error is detected, those tasks “at fault” are obscured by the large number of other blocked tasks. Ideally, the programmer would like to see the first ‘fishy’ state, where things start to go awry.

Some simple ways of storing and accessing the sequence of monitor pictures can be implemented. If the monitor’s trace output is stored in a file, it is possible to replay the sequence of task interactions. During replay, the exact same sequence of monitor pictures will occur. This allows the programmer to trap intermittent deadness errors and examine them without having to wait until they resurface.

Once we have all the inputs to the monitor recorded, we might also incorporate facilities like fast-forward and rewind. Ideally, the task states would be displayed graphically so the programmer could speed through much of the execution and view in detail only the critical portions. Perhaps some sort of search facility could be incorporated where, for example, the programmer could go directly to the next state where task T1 **calls** T2.

6.3 Event Specifications

For the purpose of this discussion, an event is either a task state or a finite sequence of task states.

The kind of facilities in section 6.2 give the programmer some simple ways to search out relevent events in a task state history. The next step is to provide an ability to describe events. This requires definition of a language in which the programmer can write checkable specifications for events. The language should be powerful enough to describe events related to other kinds of tasking errors beyond deadness, such as race conditions and synchronization problems. It should be simple enough so that event specifications can easily be monitored.

We suggest two kinds of specifications for which runtime checking appears implementable using structures and algorithms similar to those in our current monitor.

1. **Constraints on tasking states.**

   An example of a constraint on a single task state is (informally), “tasks A, B, and C are never all calling task D at the same time”. A constraint of this kind can be expressed in a very simple formal language. It would translate into a check ‘on the monitor’s internal picture. If a state arises which contradicts the constraint a warning would be issued, possibly accompanied by debugging information. .

2. **Constraints on sequences of tasking states.**

   An example of a constraint on a sequence of task states is (informally), **“after** a customer C prepays, **until after** customer C next starts pumping, the following sequence of events does **not** happen: a customer C’ prepays **and then** starts pumping at pump P, **and then** customer C starts pumping at the same pump P."

The ability to monitor this assertion would have been most helpful in debugging the gas station example. These kinds of constraints have the form of temporal templates. They are instantiated by events (e.g.

CUSTOMER 8 prepays the OPERATOR); the instantiated constraint then remains in effect on all subsequent states until a second event occurs (CUSTOMER 8 starts pumping at PUMP 13).

Some features of a formal language to express these kinds of specifications might be sketched asfollows. The basic expressions of the language are events. Events may contain variables over task ids. Statements of the language consist of three parts (some of them optional):

starting event: When an instance occurs, checking of the statement starts,
termination event: When an instance occurs, checking of the statement stops,
sequence of events (or negated sequence of events): An instance of the sequence must (or must not) occur during checking.

Example:

```
when OPERATOR.PRE_PAY by C =>
    not [OPERATOR.PRE_PAY by C'and then
         P.START_PUMPING by C'and then
         P.START_PUMPING by C ]:
until P' .START_PUMPING by C;
```

In this example, C, C ', P, P ' are variables over task ids. Its intended meaning and runtime monitoring may be described very informally as follows. When an event happens which is **an** instance of “customer C calls OPERATOR . PRE__PAY", all occurrences of C are instantiated to the calling customer’s task id. The following events (in square brackets) now constrain the execution of the program. Essentially this constraint requires that the subsequence of three events does not happen until after (the instance of) C calls some pump, P ' , at **START-PUMPING.**

Design of an event language for task debugging raises three issues. First, it must be capable of expressing useful events for task debugging, and runtime monitoring must be practicable. Second its syntax should be very simple and intuitively understandable. Finally, its formal semantics must be translatable into runtime checks.

7. Conclusions

It is likely that deadness errors will be a problem in Ada tasking systems for some time. Ideally, one would like a comprehensive set of programming techniques for ensuring the absence of deadness. Such techniques which are compatible with runtime efficiency requirements, will probably not be developed in the near future. Meanwhile, the development of tasking debuggers for Ada programming environments remains an important problem.

We have implemented a runtime monitor which detects and describes certain kinds of deadness errors in Ada programs. Its descriptive facilities are based on the concept of tasking state which is itself based on the semantic concepts of Ada tasking. Our conclusion from experiments with this prototype monitor is that it is a useful debugging tool in many situations.

Both the monitor itself and the preprocessor which links the monitor to the Ada program are implemented in Ada [6]. The monitor (extended to include the additional features outlined in sections 6.1 and 6.2) and preprocessor could be reimplemented to production quality standards forming a highly portable Ada task debugger.

Our deadness monitor is not a comprehensive tool, it is a first step in the formation of an environment for task debugging. At the moment, only deadness errors are detected. But, perhaps of more concern is that the detection of dead states uses only the semantics of Ada tasking and does not take into account knowledge about the program itself. As a result, many types of errors go undetected. Also, by the time a deadness error occurs, those tasks initially at fault may be obscured by the resulting deterioration of the tasking system.

One promising direction for further research is runtime monitoring for user specified tasking events (i.e. assertions about tasking interactions as outlined in section 6.3). Quite simple formal languages appear adequate to describe important events, including not only deadness errors, but also race conditions and synchronization. Runtime monitoring for such events could result in a tasking snapshot being generated as soon as a specified event occurs. It seems likely that runtime monitoring for event languages can be implemented using techniques similar to those we have used for deadness detection.

References

[1] *The Ada Programming Language Reference Manual*
US Department of Defense, US Government Printing Office, 1983.
ANSI/MILSTD 1815A Document.

[2] German, SM., Helmbold, D.P., and Luckham, D.C.
Monitoring for Deadlocks in Ada Tasking.
In Gargaro, A.B. (editor), *Proceedings of the AdaTec Conference on Ada,* pages 10-25. ACM, Arlington, Virginia, October, 1982.

[3] Helmbold, D.
Distributed Deadness Monitoring in Ada.
1984.
Forthcoming report.

[4] Helmbold, D., and Luckham, D.C.
Techniques for Runtime Detection of Deadness Errors in Ada Tasking.
CSL Technical Report 83-249, Stanford University, November, 1983.
Program Analysis and Verification Group Report 22.

[5] Luckham, D.C., von Henke, F.W., Larsen, H.J., and Stevenson, D.R.
ADAM -An Ada-based Language for Multi-processing.
Technical Report CSL-TR-83-240, Stanford University, May, 1983.
To appear in Software Practice and Experience.

[6] Rosenblum, D.
A Method of Designing Ada Tools using DIANA Trees as an Internal Form.
1984.
Submitted to IEEE Computer Society 1984 Conference on Ada Applications and Environments, October 15-18, St. Paul, Minnesota.

[7] Taylor, R.N.
Complexity of Analyzing the Synchronization Structure of Concurrent Programs.
Information and Computer Sciences Report, University of California, Irvine, August, 1982.

[8] Taylor, R.N.
Analysis on Concurrent Software by Cooperative Application of Static and Dynamic Techniques.
In Hans-Ludwig Hausen (editor), *Proceedings of the Symposium on Software Validation.* North-Holland Publishing, Darmstadt, F.R.G., September 26 - 30, 1983.