A TWO-PHASE REPRODUCTION METHOD FOR Ada TASKING PROGRAMS

Mamdouh M. Najjar and Tzilla Elrad

Computer Science Department
Illinois Institute of Technology
Chicago, Illinois 60616

ABSTRACT

Different results are produced when an Ada tasking program is re-executed with the same input due to two types of nondeterminism. This problem exists in cyclic debugging of Ada tasking programs. Nondeterminism reproduction is difficult in Ada due to some Ada characteristics. Our approach uses a preprocessor to extend an Ada tasking program P using a path specification S into P' such that P' is a deterministic version of P. P' can then be re-executed as many times as required by the debugger to locate the source of an error. Each phase handles a different type of nondeterminism. Phase One creates one Ada task controller per task. Each controller handles the arrival sequence of entry calls to its assigned task. Phase Two handles nondeterministic selections by controlling the selection of alternatives within selective wait statements. One advantage of this approach is that it uses more than one Ada task controller for the reproduction process. This eliminates the need for a master controller which can be a bottleneck to a solution. The two phases are easy to understand and to implement.

1. INTRODUCTION

There are basically two types of nondeterminism that cause Ada tasking programs to produce different results for the same input every time they are executed. Global nondeterminism arises as a result of the relative progress of tasks within a program, and local nondeterminism arises as a result of an explicit choices of a nondeterministic control structure [10, 15]. Reproducing the same results from the same input in a language that supports one or both of these types is called the reproduction testing problem [4, 21]. Reproducing concurrent programs normally requires the reproduction of the two types of nondeterminism. Global nondeterminism is usually more difficult to reproduce than local nondeterminism. This is due to the fact that global nondeterminism is difficult to record. Recording an execution sequence for independent events in different tasks is an example of recording global nondeterminism. On the other hand, local nondeterminism is local to each task and can easily be recorded and replayed. The problem exists in cyclic debugging of concurrent programs.

Cyclic debugging is a well known process for debugging sequential programs. It is used to locate and remove errors after they have been uncovered by a test case. This process is well understood for sequential programs but not as well understood for concurrent programs. The same process has been adopted for concurrent programs [17]. Cyclic debugging of Ada tasking programs cannot be achieved without being able to reproduce the same results from the same input. Locating and removing an error usually requires more than one execution. This requires finding ways for reproducing the two types of nondeterminism mentioned earlier. Solutions for nondeterminism reproduction are different for different languages because of the characteristics of the interprocess communications and to the nondeterministic control constructs supported by a language.

Reproduction of global nondeterminism for an Ada program is reduced in this paper to the reproduction of rendezvous (no reproduction is done

March 1989, Atlantic City, NJ

for shared variables). The reproduction of local nondeterminism is reduced into the reproduction of nondeterministic selections within an Ada task due to a selective wait statement.

Reproduction of a rendezvous requires that the two partners of a rendezvous to match. In languages that support the symmetric naming convention, i.e., the called task knows the names of its callers and vise versa, a construct for matching the two partners of a rendezvous is usually built into these languages or done automatically [14]. Reproduction of a rendezvous in such languages is obviously easier than in those languages that adopt the asymmetric naming convention (the called task does not know the names of its callers). We expect that rendezvous reproduction in Ada will be difficult for a number of reasons: Ada adopts the asymmetric naming convention, Ada handles entry queues in strictly first-in first-out order, and Ada does not have a mutual control construct, i.e., accepting an entry call according to some value of its passed parameters.

A solution to the reproduction problem for Ada transforms an Ada program P into P' such that the reproduction of the same results of P requires one execution of P' with an additional input of a rendezvous sequence which represents the previous execution of P [21]. The solution is based on the reproduction of a rendezvous sequence using a controller that controls the arrival of entry calls to the called task. Each entry call must first call a controller and identify its source and destination; then the controller returns the call when the source is the other partner of the next rendezvous in the destination task. The next entry call to the next rendezvous is released by the controller when the previous rendezvous has started. One drawback of this method is that a centralized controller, which can be a bottleneck to the program, is used.

Some approaches for debugging concurrent programs avoid the problem by building a debugger that has the ability to discover and locate errors or to record the program's state at each stage of the execution [1, 9].

Another approach suggests using a new programming construct called preference control to control the race conditions within Ada tasks [8]. This method handles only local nondeterminism and does not force a selection; rather it suggests one. Other related non-Ada work is presented in several references used for this paper [1,17,18,20,22].

This approach basically reproduces a program's rendezvous sequence by reproducing all task's local rendezvous sequences. A local rendezvous sequence is associated with a task in an Ada program. This approach is partially based on the theoretical work given in the following references: [2,6,7,15,16]. The approach suggests reproducing each local rendezvous sequence independent of the other local rendezvous sequences to reproduce an original behavior of a program.

In this approach, a controller is used to simulate a communication environment for each task in the original execution. The selection of a recorded sequence of nondeterministic decisions a task has taken is enforced. As a result, a task behaves in the same way it did in the original execution.

The approach is divided into two major phases. Phase One, which handles global nondeterminism, uses an Ada Task Controller (ATC) per task to control an arrival sequence of entry calls to a called task. It insures that the order of entry calls at each entry's queue is in a predetermined order. The second phase, which handles local nondeterminism, controls nondeterministic selections within individual tasks. This is done by using a set of conditions to disable or enable rendezvous in a selective wait statement. Using these conditions one can enable the next rendezvous of a rendezvous sequence. These two phases distinguish between the two types of nondeterminism mentioned earlier and handle each type separately. Note that each of these two phases requires some extensions to the Ada source program, i.e., the addition of some special Ada code to

the original program. This extension process is referred to as extending a program.

One advantage of this approach is that it uses one or more ATCs. One ATC is assigned to control entry calls to one or more tasks. This simplifies the implementation of ATCs and eliminates the need for a master controller, which can be a bottleneck to a solution.

The design of an approach that can be divided into two smaller phases is another advantage of this approach. Each phase deals with a problem in the reproduction process, but the two phases work together to achieve the goal. When a problem is spotted at the reproduction process, the nature of the problem guides us to the phase in fault. Other advantages include better understanding of the reproduction process and ease of implementation.

This approach limits handling of local nondeterminism constructs by using the selective wait statement (the other types of select statement are not handled) and by assuming that no rendezvous nesting occurs in it. The COUNT attribute and shared variables are also not handled. An approach for handling these constructs is given in reference [21].

A discussion of problem of reproducing an Ada tasking program and what special Ada characteristics may influence the solution is provided in Section 2. Outlines of the reproduction process are given in Section 3. Explanations of the two phases of this approach are presented in Section 4. A complete example is given in Section 5, and conclusions are presented in Section 6. Appendices I and II list the Ada code for the example given in Section 5.

2. REPRODUCING ADA TASKING PROGRAMS

In languages that adopt a symmetric naming convention, rendezvous can be reproduced easily; the two partners of a rendezvous automatically match in Communicating Sequential Processes (CSP) [14]. In CSP, rendezvous match through an Input and Output commands: a caller issues an Output command specifying its destination task, and the called task issues an Input command specifying its source task [14]. In Ada, rendezvous are more difficult to reproduce because an entry in a destination task (a called task) should rendezvous with the first call at its queue regardless of who is calling it, and each entry queue is handled in strictly first-in first-out order. This implies that a destination task cannot determine where each call originates. If the name of the calling task (source task) is included as an entry call parameter, the destination task does not find out the name of its caller until the rendezvous has begun. Because Ada does not have a mutual control construct, this restriction cannot be avoided.

A mutual control construct can be useful for solving the problem of mismatching the two partners of a rendezvous in Ada. Such a solution requires either simulating mutual control by the existing Ada constructs or adding such a construct to the language. However, a mutual control construct is not needed if we are able to duplicate every entry queue order in the reproduction execution. This can be done by duplicating (replying) the original arrival sequence of entry calls to a destination task, and as a result, every entry queue sequence in a the destination task is duplicated. This approach diminishes the need for a mutual control construct for the purpose of reproduction of rendezvous in Ada. Such a construct will still be useful for explicit scheduling of Ada tasks. This approach adopts the approach of duplicating the arrival sequence of entry calls to a destination task. This is done by using an ATC per destination task to control its arrival sequence of entry calls.

3. REPRODUCTION PROCESS OUTLINE

Figure 1 presents the general outline of the approach. A path specification is a set of local rendezvous sequences that were recorded in a previous execution, usually one local rendezvous sequence per task. A path specification file is a file that contains a local rendezvous sequence for each non-

calling task (a task with at least one accept statement that produced a rendezvous in the original execution). A preprocessor command file is used to specify the number of ATCs that will be used in a program and to specify which task is controlled by which controller. An Ada source program, a path specification file, and a preprocessor command file are used as inputs to a preprocessor. A preprocessor is a program that extends an Ada source program to accommodate new Ada code for reproduction. The new added code includes the number of ATCs specified in the preprocessor command file (ATCs are explained later in this section).

The output of a preprocessor is an extended Ada program that has the same semantics of the original Ada program. The differences are the following: the extended Ada program is deterministic, it always follows the same path (specified in the path specification file), and it gives the same results for the same input every time it is executed. When the extended Ada program is executed, the results of its execution will be compared to either the expected results or the results of the original execution. Modifications can then be made to the original Ada program, if desired, and the whole process can be repeated until the desired results are achieved.

Definition 1:

A local rendezvous sequence for a task T is a totally ordered sequence of rendezvous accepted by task T (T is the destination task), where each rendezvous is represented as a three-entity tuple:

< Rendezvous sequence number,
Calling task identity, Entry name >

A rendezvous sequence number is unique within a local rendezvous sequence of task T. Note that from this definition we can conclude that a task with one or more accept statements which produces no rendezvous in the original execution, or one with no accept statements at all has no local rendezvous sequence. We call such a task a demanding task.

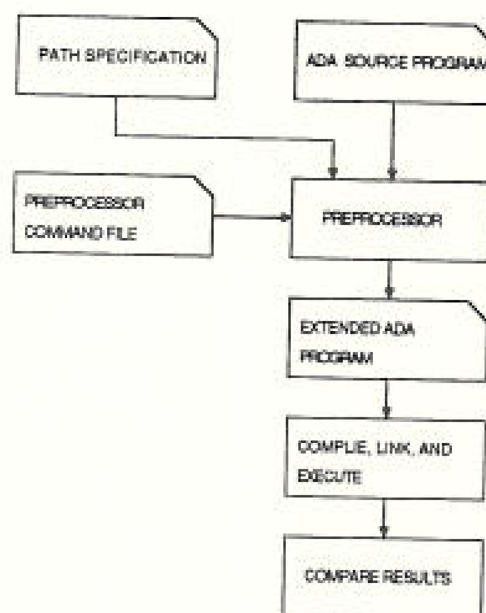

Figure 1. An outline of this approach.

4. TWO-PHASE REPRODUCTION FOR ADA TASKING PROGRAMS

Two phases of program reproduction- an entry calls control phase and a nondeterministic selections control phase - are discussed in this section. The extension procedure for each phase is also presented.

The first phase is to use one Ada task controller per task (or a group of tasks) to control the arrival sequence of entry calls for that task. An ATC assigns a unique number, called entry call sequence number, to entry calls that are calling the same entry. Entry calls can then be accepted in sequence using an entry family approach, provided by Ada, and a loop with an entry family index. A loop is used only if no loop already exists. This phase is explained in more detail in subsection 4.1. It is assumed here that each task in the program can be uniquely identified. The advantage of having more than one ATC is significant in multiprocessor systems. For example, having one ATC for each group of tasks that run on the same processor reduces the amount of communications between processors.

The second phase is to control nondeterministic selections inside

Ada tasks. This is done by extending each task T into T' such that the semantics of T and T' are the same. Basically, each task keeps track of its local rendezvous sequence and enables only the rendezvous at the top of its local rendezvous sequence and disables all other alternative rendezvous. The index to the local rendezvous sequence will then be updated, and the process will be repeated again until all rendezvous occur. The entry call for the enabled rendezvous is assumed to be available from the first phase. If it is not, the task will be blocked until the appropriate entry call arrives. More details are given in subsection 4.2. These two phases work together to achieve the reproduction of Ada tasking programs.

The separation between the two phases of the approach is necessary because each phase solves a separate problem in the reproduction process. The advantages are to simplify the implementation of the two phases and to make each phase easier to understand. This is especially true at the stage where we compare the results of the program with the results of its reproduction. A failure in the reproduction of interprocess communications most probably points to a failure in the entry calls control phase.

4.1 Entry calls control

We will first explain Phase One in general and apply it to one entry. Later in this section, we will explain how to expand it to include more than one entry. An example of the two phases is given in section five.

Figure 2 shows a task K that contains an entry WRITE. It also shows that there are three entry calls to the WRITE entry: x, y, and z (where x is at the top of the entry's queue). Note that in this figure, we are using symbols that are similar to the one given in reference [3]. We are also temporarily using the calling's task identity as the entry's call identity, i.e., we are using the letter x as a task name and as an entry call identification (similarly y and z).

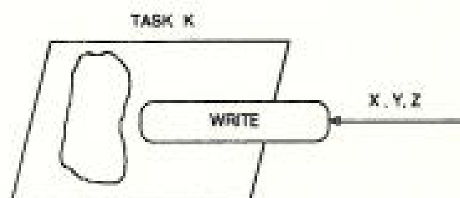

Figure 2. A task with one entry.

The purpose of phase one, in this example, is to insure that the WRITE's entry queue is in the predetermined order at the reproduction execution, which is (x,y,z). Figure 3 depicts how Phase One handles this problem. Each entry call to the WRITE entry is extended into two calls.

The first call is to an ATC that handles task K. This first call is called the sign-in call. The purpose of this call is to get an entry call sequence number, which corresponds to the order of the call at the WRITE's entry queue. Tasks x, y, and z can call the Ada task controller in any order, but they will always get the same entry call sequence number, e.g., the call issued by task z to the ATC will return an entry call sequence number of three.

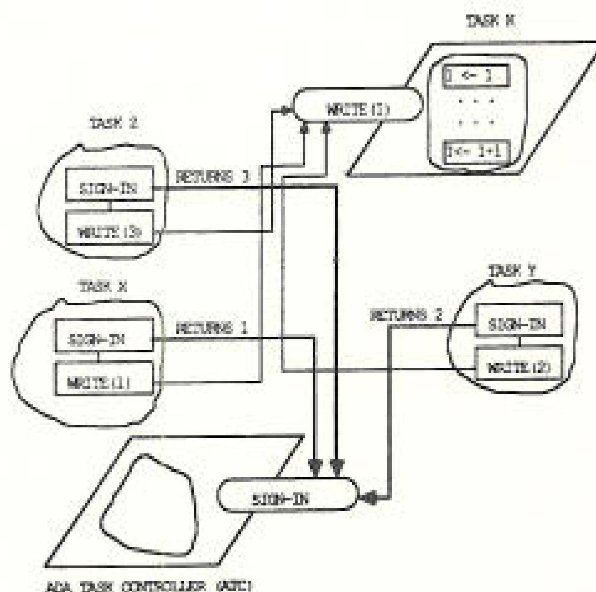

Figure 3. Sign-in calls in phase one.

381

The second call uses the entry call sequence number to call the original entry using an entry family approach. Task z issues the entry call WRITE(3). These two entry calls are similar to the two-stage sign-in process suggested for explicit scheduling in reference [12].

To preserve the order of calls, the WRITE entry accepts one call at a time using an entry family and a loop with an entry family index (I), which is initialized to one and incremented by one every time the entry is involved in a rendezvous. Note, the calls to WRITE(I) can only be accepted in the following order: WRITE(1), WRITE(2), and WRITE(3). This preserves the original (x,y,z) entry calls sequence. A loop is required if no loop already exists, and an entry family index is required as shown in Figure 3. The loop stops when no more calls are issued to the entry. In summary, the number of iterations an entry goes through is equal to the number of calls (rendezvous) in which the entry is involved.

The ATC that handles task K should have an access to the predetermined entry calls sequence of the WRITE's entry. This enables it to assign an entry call sequence number to each call it receives. The ATC recognizes a call by the identity of the calling task, which should be passed to the ATC as an input parameter by the sign-in call. In the approach, it is part of the preprocessor to build an entry calls sequence for each entry within a task from the task's local rendezvous sequence. It is also part of the preprocessor to make these sequences accessible to the appropriate ATC.

Let us now assume that there are two entries in task K, a READ and a WRITE. In this case, two entry call sequences are built from a task's local rendezvous sequence, one for each entry. We will also use two entry family indices. An ATC in this case treats each sequence independently of the other. The ATC assigns an entry call sequence number according to the calling's task name and the called's entry name. These two names are passed as parameters by the sign-in call (refer to the READ procedure in Appendix II). This means that the above discussion regarding entry extension applies to multiple entries regardless of the number of entries within a task. The only difference is that an ATC has to control one more entry calls sequence for each additional entry. The conclusion is that each entry is extended into an entry family with an initial entry family index of one and a limit of the number of entry calls to an entry.

It is clear from the above discussion that the original Ada source program should be extended to accommodate new Ada code for the purpose of reproduction. A summary of the extension procedure used in this phase is as follows:

1. Each entry call is extended into two calls, a sign-in call and a call to the original entry using an entry family approach.

2. Each accept is extended to an entry family, a loop (if required), and an entry family index.

3. An ATC is created for each task (or a group of tasks) to handle entry calls sequences, which are provided by the next step of this list.

4. A sequence of entry calls is built from a local rendezvous sequence of a task for each entry it contains.

These extensions are problem independent. The approach assumes that these extensions are part of the preprocessor. Refer to example one for more about these extensions.

4.2 Nondeterministic Selections Control

In Phase One, we tried to insure that entry calls to every task arrive in a predetermined order. This phase insures that a task selects a predetermined sequence of selections from a set of different alternatives available to it. We mentioned earlier that the selective wait statement is what makes an Ada task selects a different selection from the same set of alternatives every time it is executed (local nondeterminism). The approach handles local nondeterminism

by handling the selective wait statement. We assume that no rendezvous nesting exists within the selective wait statement. The other types of select statement, namely the conditional entry call and the timed entry call, are not handled by the approach. Although an approach similar to the one given in reference [21] can be adopted. The approach also does not handle the COUNT attribute and shared variables. An approach for handling these two attributes is given in reference [21].

In this phase, each task follows its own local rendezvous sequence. This is done in two steps: Step One is to extend each task in an Ada source program to include a list of its own local rendezvous sequence; Step Two is to include a condition in each entry's When-clause that matches its own name with the entry name at the top of the task's local rendezvous sequence to which it belongs. The top of a local rendezvous sequence is determined by using a rendezvous index which is initialized to one and incremented by one after every rendezvous that occurs within the same loop (refer to RW task in Appendix II).

Conditions serve as guards to entries [14]. In the set of conditions, only one condition is always true. The true condition allows the rendezvous at the top of the local rendezvous sequence to occur. The false conditions prevent any other rendezvous to occur. The When-clause always signals the entry that the task should be involved in next. The other partner of the rendezvous, which is viewed as an entry call by the destination task, is provided by Phase One of the approach. After each rendezvous, an index that points to the next rendezvous in the local rendezvous sequence is incremented by one.

To see how the two phases work together, assume that a task is used with two entries, READ and WRITE. A local rendezvous sequence of this task is (<1,x,READ>, <2,y,WRITE> ,<3,z,READ>). This three-rendezvous list contains three sub-lists. Each sublist represents a rendezvous. Each sublist contains three entities: a rendezvous sequence number, a name of a calling task, and a called entry name. Assume also that the entry calls coming from tasks x, y, and z arrive in a predetermined order by Phase One. The task has a loop that iterate three times and involves in three rendezvous then terminates (see Figure 4).

When executing task G (Figure 4) and during the first iteration, the READ and WRITE alternatives are available. The READ's When-clause becomes true because it was involved in the first rendezvous. The WRITE's When-clause becomes false. So the READ entry is selected for the first rendezvous. The other partner of the rendezvous is task x. By assumption, the entry call from task x to the READ entry is at the top of the READ's entry queue. The two partners of the first rendezvous now match and the rendezvous occur. The two other rendezvous are reproduced in the same way.

This phase requires some extensions to an Ada source program. The first extension is to make each task in a program access its own local rendezvous sequence. A local rendezvous sequence is represented as an array of entry names. A rendezvous index is needed to point to the next rendezvous in the sequence. The last rendezvous occurs at the last iteration of the selective wait statement.

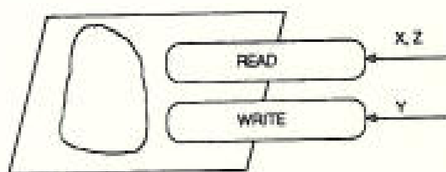

Figure 4. A task with two entries.

The second extension is to include in each entry's When-clause a condition that matches the entry's own name with the name of the entry at the top of the task's local rendezvous sequence. The approach assumes that these two extensions are part of the preprocessor. The next

section explains the reproduction process of an extended Ada program.

5. A COMPLETE EXAMPLE

An Ada program is listed in Appendix I [12]. The program is a controller for a shared resource that allows multiple readers at the same time and only one writer at a time. In this example, we plan an execution scenario of the program and then determine how this scenario is specified. We also explain the needed extensions to the original program.

EXAMPLE 1:

Using the Ada code in Appendix I, assume that there are four demanding tasks C1, C2, C3, and C4 that use the RESOURCE package. Further assume that C1 and C2 called RW for reading the resource where C1 called before C2. Task C3 called for writing while C1 and C2 were still in the process of reading. Task C4 called for reading after C3 had finished writing. Because task RW has an infinite loop, assume here that the number of readers and writers are finite and it will terminate.

To specify the above scenario, we need to determine a local rendezvous sequence that represents the above path. One possible local rendezvous sequence is:

```
LRS1 := ( <1,C1,START_READ>, <2,C1,END_READ>, <3,C2,START_READ>,
<4,C2,END_READ>, <5,C3,START_WRITE>, <6,C3,END_WRITE>,
<7,C4,START_READ>, <8,C4,END_READ> )
```

Recall that each rendezvous is represented as:

<Rendezvous sequence number, Calling task identity, Entry name>.

Rendezvous 2 and 3 can be exchanged to get another local rendezvous sequence that still represents the same path. The above local rendezvous sequence is a path specification for the program in Appendix I with the execution scenario explained above. LRS1 is the content of the path specification file (see Figure 1). Note that tasks C1, C2, C3, and C4 are assumed to be demanding tasks (refer to the end of Section 3 for the definition of a demanding task), and as a result, no local rendezvous sequences are specified for them.

When this path specification is read by the preprocessor, it builds a sequence of entry calls for each entry in the RW task. The preprocessor builds the following entry calls sequences.

```
START_READ_SEQ :=  (C1, C2, C4)
END_READ_SEQ := (C1, C2, C4)
START_WRITE_SEQ := (C3)
END_WRITE_SEQ := (C3)
```

This set of entry calls sequences are then built into an ATC for task RW (refer to RW_C task in Appendix II). The preprocessor creates the RW_C as an Ada task controller for RW task and adds it to the original Ada source program. Note that it should be specified that an ATC be built for the RW task in the preprocessor command file. The preprocessor also extends an Ada program according to the extension procedures of the two phases given in subsections 4.1 and 4.2. Appendix II lists the program after it has been extended by the preprocessor.

There are two packages in Appendix II: the RESOURCE package and a CONTROLLER package. The RESOURCE package is extended to accommodate new Ada code. A symbol at the end of a line indicates how much a line is extended. The symbol "--@" indicates that the line is added completely to the original program. The symbol "--&" indicates that some extension occurred in the line. Note that the CONTROLLER package is completely added so there is no need for using any symbols.

Note how each call to the RW task is extended in the READ and WRITE procedures as a result of the entry calls control phase. The sign-in call to the controller (RW_C) includes the caller identification, the requested entry name, and a dummy parameter to return an entry sequence number. The original call is extended to include the entry call sequence number (ENTRY_CALL_SEQ_NO). Note also how each entry is extended in the RW task. Each entry is extended to a family of entries and has its own

family of entry index. These extension are part of Phase One; the rest of the extensions in the RW task are part of Phase two. However, the CONTROLLER package is a result of the extensions in Phase One.

Note also how each entry's When-clause was extended to include an additional condition and how the while loop keep track of the number of rendezvous for reproduction using the rendezvous index (NEXT_R). Note also how the local rendezvous sequence (LOCAL_REND_SEQ) is represented. These extensions are a result of the nondeterministic selections control phase.

As a result of these extensions, the specification of the RESOURCE package, and the RW task were extended. The program given in Appendix II is a deterministic version of the original program in Appendix I, and it will always follows the same path (specified by LRS1) every time it is executed.

6. CONCLUSIONS

Reproduction of Ada tasking programs is a problem that must be dealt with in cyclic debugging of Ada programs. The method of avoiding the problem by building a debugger that has the ability to discover and locate errors is inadequate because this method mixes the testing and debugging phases, is complex, and is expensive. The method of extending a nondeterministic program into a deterministic one is adequate because it is easy to understand and to implement; however, the problem is difficult in Ada because of the asymmetric naming convention to the way Ada handles entry queues. The approach distinguishes between two types of nondeterminism: global nondeterminism and local nondeterminism. It handles each type separately. The approach extends a nondeterministic Ada tasking program into a deterministic one. This is done in two phases: Phase One handles global nondeterminism, and Phase Two handles local nondeterminism. Global nondeterminism is handled by using one Ada task controller per task to control the arrival sequence of calls to a destination task. Having more than one controller eliminates the need for a master controller that can be a bottleneck to the reproduction process. Local nondeterminism is handled by restricting the number of open alternatives in a selective wait statement using a When-clause as a guard to each alternative. The method is easy to understand and to implement.

APPENDIX I

```
package body RESOURCE is
S : SHARED_DATA :=  -- The shared data

task RW is
  entry START_READ;
  entry END_READ;
  entry START_WRITE;
  entry END_WRITE;
end RW;

task body RW is
  NO_READERS: NATURAL := 0;
  WRITER_PRESENT: BOOLEAN := FALSE;
begin
  loop
    select
      when not WRITER_PRESENT =>
                accept START_READ;
      NO_READERS := NO_READERS + 1;
    or
      accept END_READ;
      NO_READERS := NO_READERS - 1;
    or
      when not WRITER_PRESENT AND
      NO_READERS = 0 =>
                accept START_WRITE;
      WRITER_PRESENT := TRUE;
    or
      accept END_WRITE;
      WRITER_PRESENT := FALSE;
    end select;
  end loop;
end RW;

procedure READ (X:out SHARED_DATA) is
  begin
    RW.START_READ;
    X := S;
    RW.END_READ;
  end READ;

procedure WRITE(X : in SHARED_DATA) is
  begin
    RW.START_WRITE;
    S := X;
    RW.END_WRITE;
  end WRITE;
end RESOURCE;
```

APPENDIX II

```
with CONTROLLER;
package RESOURCE is                                    --@
use CONTROLLER;
  type SHARED_DATA is ...;                             --@
  procedure READ(CALLER_ID:in
                 CALLER_NAME;
           X : out SHARED_DATA);                       --&

  procedure WRITE(CALLER_ID : in
           CALLER_NAME;
           X : in SHARED_DATA);                        --&
end RESOURCE;

package body RESOURCE is
S : SHARED_DATA :=  -- The shared data

task RW is
 entry START_READ(REND_INDEX);
 entry END_READ(REND_INDEX);                           --&
 entry START_WRITE(REND_INDEX);                        --&
 entry END_WRITE(REND_INDEX);                          --&
end RW;                                                --&

task body RW is
 LOCAL_REND_SEQ:RENDEZVOUS_LIST;                       --@
 NEXT_R : REND_INDEX :=1;                              --@
 SR,ER,SW,EW:REND_INDEX := 1;                          --@
 LOCAL_REND_SEQ   (1   ..   8)   :=                    --@
(START_READ, END_READ,
 START_READ, END_READ,                                 --@
 START_WRITE, END_WRITE,                               --@
 START_READ, END_READ);                                --@
 NO_READERS: NATURAL := 0;                             --@
 WRITER_PRESENT: BOOLEAN := FALSE;

begin
while NEXT_R <=
             RW_REND_INDEX_LIMIT  --@
 loop
  select
  when not WRITER_PRESENT and
     LOCAL_REND_SEQ(NEXT_R) =
      START_READ  =>                                   --@
       accept START_READ(SR);                          --&
 NO_READERS := NO_READERS + 1;
 SR   :=   SR + 1;
 or                                                    --@
 when LOCAL_REND_SEQ(NEXT_R) =
  END_READ=>accept END_READ(ER);  --&
 NO_READERS := NO_READERS - 1;
 ER   :=   ER + 1;
 or                                                    --@
 when not WRITER_PRESENT and
          NO_READERS = 0 and
    LOCAL_REND_SEQ(NEXT_R) =
    START_WRITE =>                                     --@
      accept START_WRITE(SW);                          --&
WRITER_PRESENT := TRUE;
SW   :=   SW  + 1;
                                                       --@
   or
   when LOCAL_REND_SEQ(NEXT_R)=
             END_WRITE    =>                           --@
          accept END_WRITE(EW);                        --&
  WRITER_PRESENT := FALSE;
  EW   :=   EW  + 1;
  end select;                                          --@
  NEXT_R := NEXT_R + 1;
  end loop;                                            --@
end RW;

Procedure READ(CALLER_ID :in
  CALLER_NAME; X :out SHARED_DATA) is
 begin
  RW_C.SIGN_IN(CALLER_ID,
    START_READ,ENTRY_CALL_SEQ_NO); --@
  RW.START_READ(
             ENTRY_CALL_SEQ_NO); --&
  X := S;
  RW_C.SIGN_IN(CALLER_ID, END_READ,
             ENTRY_CALL_SEQ_NO);  --@
  RW.END_READ(ENTRY_CALL_SEQ_NO);  --&
end READ;

procedure       WRITE(CALLER_ID      :in
CALLER_NAME; X :in SHARED_DATA) is
 begin
  RW_C.SIGN_IN(CALLER_ID, START_WRITE,
             ENTRY_CALL_SEQ_NO); --@
  RW.START_WRITE(
             ENTRY_CALL_SEQ_NO); --&
  S := X;
  RW_C.SIGN_IN(CALLER_ID, END_WRITE,
             ENTRY_CALL_SEQ_NO); --@
  RW.END_WRITE(ENTRY_CALL_SEQ_NO); --&
  end WRITE;
end RESOURCE;

package CONTROLLER is

type ENTRY_NAME  is (START_READ,
 END_READ,START_WRITE, END_WRITE);

RW_REND_INDEX_LIMIT : constant := 8;

type REND_INDEX is POSITIVE;

type RENDEZVOUS_LIST  is array
    REND_INDEX) of ENTRY_NAME;

type CALLER_NAME is (C1, C2, C3, C4);

type ENTRY_QUEUE is array(REND_INDEX)
                      of CALLER_NAME;

procedure SEARCH( SEQUENCE : in out
                           ENTRY_QUEUE;
              ID    : in CALLER_NAME;
          OUT_NO    : out REND_INDEX);
end CONTROLLER;
```

```
package body CONTROLLER is

task RW_C is
entry SIGN_IN  (CALLER_ID : in
                  CALLER_NAME;
   ENTRY_REQ : in ENTRY_NAME;
   CALL_SEQ_NO : out REND_INDEX);
end RW_C;

task body RW_C is

CALL_INDEX : INTEGER := 0;
START_READ_SEQ, END_READ_SEQ   :
                        ENTRY_QUEUE;
START_WRITE_SEQ, END_WRITE_SEQ :
                        ENTRY_QUEUE;
START_READ_SEQ (1 .. 3):=(C1,C2,C4);
END_READ_SEQ (1 .. 3) := (C1,C2,C4);
START_WRITE_SEQ (1) := (C3);
END_WRITE_SEQ (1) := (C3);
begin
  loop
  when   CALL_INDEX<=
RW_REND_INDEX_LIMIT =>
    accept SIGN_IN (CALLER_ID : in
                         CALLER_NAME;
         ENTRY_REQ : in ENTRY_NAME;
      CALL_SEQ_NO : out REND_INDEX) do
    case ENTRY_REQ is
    when START_READ =>
     SEARCH(START_READ_SEQ;
                  CALLER_ID;
              CALL_SEQ_NO);
    when END_READ =>
     SEARCH(END_READ_SEQ;
                CALLER_ID;
            CALL_SEQ_NO);
    when START_WRITE =>
     SEARCH(START_WRITE_SEQ;
                   CALLER_ID;
               CALL_SEQ_NO);
    when END_WRITE =>
     SEARCH(END_WRITE_SEQ;
                 CALLER_ID;
             CALL_SEQ_NO);
    when others   => null;
   end case;
  end SIGN_IN;
  CALL_INDEX := CALL_INDEX + 1;
 end loop;
end RW_C;

procedure SEARCH(  SEQUENCE : in out
                        ENTRY_QUEUE;
              ID : in CALLER_NAME;
         OUT_NO : out REND_INDEX) is
begin
  INDEX := 1;
  while INDEX <= SEQUENCE'LENGTH
    loop
     if SEQUENCE(INDEX) = ID then
        OUT_NO := INDEX;
        SEQUENCE(INDEX) := null;
        exit;
     else
        INDEX := INDEX + 1;
     end if;
   end loop;
end SEARCH;
end CONTROLLER;
```

REFERENCES

[1] F. Baiardi, N.D. Francesco , G. Vaglini, "Development of a Debugger for a Concurrent Language," IEEE Trans. on Software Engineering, VOL. SE-12, NO. 4, April 1986, pp. 547-553.

[2] H. Barringer, I. Mearns, "A Proof System for Ada Tasks," The Computer Journal, Vol. 29, NO. 5, 1986, pp. 404-415.

[3] G. Booch, "Software Engineering With Ada," The Benjamin/ Cummings Company, California, 1983.

[4] Per Brinch Hansen, "Reproducible Testing of Monitors," Software-Practice and Exper., Vol. 8, 1978, pp. 721-729.

[5] T. Elrad, "A Practical Software Development for Dynamic Testing of Distributed Programs," Proceedings of the 1984 International Conference on Parallel Processing, August 1984.

[6] T. Elrad, "Data Dependencies Within Distributed Programs," Proceedings of the Hawaii International Conference on System Sciences, January 2, 1985.

[7] T. Elrad and N. Francez, "Decomposition of Distributed Programs Into Communication-closed Layers," Science of Computer Programming 2, North-Holland, 1982, pp. 155-173.

[8] T. Elrad, F. Maymir-Ducharme, "Race Control for the Validation and Verification of Ada Multitasking Programs," Proceedings of the Sixth Annual National Conference on Ada Technology, May 14-17, 1988.

[9] R. G. Fainter and T.E. Lindquist,"Debugging Tasked Ada

381

Programs," Proceedings of the Ada Applications for the NASA Space Station Conference, in R.L. Bown(ed.), June 1986, pp. B.1.1.1-23.

[10] N. Francez, C.A.R. Hoare, D. J. Lehmann, W. P. DE Roever, "Semantics of Nondeterminism, Concurrency, and Communication," Journal of Computer and System Sciences 19, 1979, pp. 290-308.

[11] H. Garcia-Molina, F. Germano, W. Kohler, "Debugging a Distributed Computing System," IEEE Trans. on Software Engineering, Vol. SE-10, No. 2, March 1984, pp. 210-219.

[12] N. Gehani, "Ada Concurrent programming," Prentice-Hall, New Jersey, 1984.

[13] D. Helmbold, D. Luckham, "Debugging Ada Tasking Programs" IEEE Software, March 1985, pp. 47-57.

[14] C.A.R. Hoare, "Communicating Sequential Processes," CACM, VOl. 21, NO. 8, August 1978, pp. 666-677.

[15] S. Katz, D. Peled, "Interleaving Set Temporal Logic,"Proc. of the Sixth Annual ACM Symposium on Principles of Distributed Computing, August 1987, pp. 178-190.

[16] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Comm. of ACM, VOL. 21, NO. 7, July 1978, pp. 558-565.

[17] T.J. Leblanc, J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," IEEE Trans. on Computers, VOL. C-36, NO. 4, April 1987, pp. 471-482.

[18] B.P. Miller, "A Mechanism for Efficient Debugging of Parallel Programs," SIGPLAN NOTICES, VOL. 23, NO. 7, July 1988, pp. 135-144.

[19] A. Pnueli, "The Temporal Semantics of Concurrent Programs," Theoretical Computer Science, Vol. 13, 1981, pp. 45-60.

[20] J.M. Stone, "Debugging Concurrent Processes: A Case Study," SIGPLAN NOTICES, VOL. 23, NO. 7, July 1988, pp.145-153.

[21] K.C. Tai, E.E. Obaid, "Reproducible Testing of Ada Tasking Programs," Proc. IEEE-CS Second Inter. Conf. on Ada Applications and Environments (1986), pp. 69-79.

[22] K.C. Tai, S. Ahuja, "Reproducible Testing of Communication Software," Proc. of IEEE COMPSAC 87, Oct. 1987, pp. 331-337.

Mamdouh Najjar received the B.S. and M.S. degrees in computer science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, in 1982 and 1986, respectively. He is currently a full-time Ph.D. student at Illinois Institute of Technology, Chicago, Illinois. His research interests include, concurrent programming, distributed systems, and distributed testing and debugging.

Tzilla Elrad received an M.S. degree in computer science from Syracuse University, N.Y. and a Ph.D. in computer science from the Technion in Israel, in 1978 and 1981, respectively. She is an assistant professor of computer science at Illinois Institute of Technology. Her main interests are in concurrent and distributed languages, concurrent programming for real-time applications and the use of Ada for such systems. She is the chair of chicago SIGAda. Her BITNET address is: CSELRAD@IITVAX.