Reproducible Execution of SR Programs

Ronald A. Olsson
Department of Computer Science
University of California, Davis
Davis, CA 95616-8562 U.S.A.

530-752-7004
530-752-4767 (fax)
olsson@cs.ucdavis.edu

April 8, 1999

Abstract

Reproducing the execution of a concurrent program is important in debugging and testing. It requires that, regardless of the actual order in which processes may execute, the reproduced execution is identical, with respect to the order in which certain activities occur, to a previously recorded execution. This paper presents a solution to the reproducibility problem for programs written in the SR concurrent programming language. Our solution transforms an arbitrary SR program into one for recording an event sequence and one for replaying from an event sequence. SR provides a rich collection of synchronization mechanisms, including rendezvous, asynchronous message passing, remote procedure call, and dynamic process creation. SR language features allow: flexible invocation servicing (e.g., use of invocation parameters in selecting an invocation to service in message passing or rendezvous); dynamically created processes and resource (module) instances; dynamic communication paths between processes; and dynamic distribution of programs across multiple machines. Because of these features, adaptations of previous solutions to the reproducibility problem for other languages and notations do not work for SR. Our solution handles all the above features. It results in a naturally distributed control algorithm for programs that are distributed. This paper also describes the implementations of our transformation tools.

Keywords: concurrent programming, reproducibility, replay, concurrent programming languages, debugging, testing, program transformation, synchronization.

1 Introduction

Earlier work [1, 2, 3, 4, 5, 6, 7] has presented solutions to the problem of reproducing the execution of concurrent programs. (A nice summary and classification of earlier work appears in [8].) Essentially, the problem is to be able to reproduce the execution, with respect to the order in which certain activities occur, of a given concurrent program on a given input. Solutions assume the availability of information representing the program's execution on a given input. The extent to which the original program's behavior can be replayed depends on the extent to which the program's activities were recorded. As described in [3], these activities usually include synchronization, but not input/output, so the output from a replayed program might be interleaved differently from the output from the original program. The reproducibility problem is important in the debugging of concurrent programs so that a particular execution history can be replayed, for example, to extract additional information.

Solutions to the reproducibility problem for programs written in a given language should, for reasons of portability, be independent of the implementations of that language [1, 3]. Thus, desirable solutions do not rely on the internals of the language's compiler or run-time system, or on the operating system on which the language's programs run. Instead, they employ source level transformations, transforming the original program into a program in the same language whose execution is guaranteed to reproduce the execution of the original program. Earlier work has presented solutions to the reproducibility problem for various synchronization mechanisms such as semaphores and monitors [6, 1], message passing [2], and rendezvous as found in Ada [3].

This paper presents a transformational solution to the reproducibility problem for programs written in the SR concurrent programming language [9, 10]. SR provides a rich collection of synchronization mechanisms, which makes the reproducibility problem more difficult and for which adaptations of other solutions [1, 2, 3, 4, 5, 6, 7] will not work. The specific SR language features that cause difficulties include

- flexible invocation servicing (e.g., use of invocation parameters in selecting an invocation to service in message passing or rendezvous)
- dynamically created processes and resource (module) instances
- dynamic communication paths between processes
- dynamic distribution of programs across multiple machines

These same features also lead to a clean solution to the reproducibility problem. Our solution handles all the above features. It results in a naturally distributed control algorithm for programs that are distributed.

The work on Instant Replay [7] provides the conceptual framework on which most work in reproducible execution is based. It shows that only the relative execution order of events needs to be recorded, and from that sequence the program can be replayed. We too apply these general ideas in employing an event sequence, defining events, etc. However, we also define new transformations and algorithms for dealing with the entire SR language and the specific language features enumerated earlier. Accordingly, this paper reports on our experience with and techniques for applying the general conceptual framework to the replay problem for programs written in a specific, real concurrent programming language. Our work is potentially applicable to other languages and notations that employ such features; e.g., invocation selection in MPI [11] is functionally similar to that in SR.

The rest of this paper is organized as follows. Section 2 introduces relevant SR background and terminology. It also introduces some basic transformations defined by the language itself; these transformations are employed subsequently as part of our main transformations. Section 3 presents our transformations; each subsection presents transformations for the language mechanisms in the corresponding subsection of Section 2. Section 4 presents the additional processes used in controlling the recording and replaying. Section 5 discusses experience building and using tools that implement our transformations. Section 6 makes some general observations about our approach and compares our work with related work. Finally, Section 7 contains some concluding remarks. Additionally, Appendix A describes additional details for one transformation and Appendix B discusses assumptions and shortcomings of our transformations.

2 SR Background

The SR concurrent programming language [9, 10] provides a variety of mechanisms for writing parallel and distributed programs. Processes in an SR program can interact via operations and shared variables.

2.1 Invoking and Servicing Operations

An operation can be considered a generalization of a procedure: It has a name and can take parameters and return a result. An operation can be invoked in two ways: synchronously by means of a call statement or asynchronously by means of a send statement. An operation can also be serviced in two ways: by a procedure-like object called a proc or by input statements. (SR's input statement combines and generalizes aspects of Ada's accept and select statements.) This yields the following four combinations:

Invocation	*Service*	*Effect*
call	**proc**	procedure call
call	**in**	rendezvous
send	**proc**	dynamic process creation
send	**in**	message passing

A guard on an input statement can also contain a *synchronization* expression and a *scheduling* expression. The former specifies which invocations are acceptable; the latter specifies the order in which to service acceptable invocations. To illustrate, consider the simple solution to the Dining Philosophers Problem shown in Figure 1.[1] It defines one server process and `N` philosopher processes within a single resource, SR's

[1]Note to editor and referees: As per CPE formatting guidelines, figures appear starting on page 28.

main modularization component (described in more detail in Section 2.5). The first arm of the server's input statement uses a synchronization expression, introduced by `st` (such-that), to accept an invocation of `getforks` from philosopher `id` only when its neighbors are not eating. Note how the synchronization expression references the invocation's parameter, `id`.

As another example, consider the following code for a server process that implements a shortest-job-next request allocator.

```
process server
  var free := true
  do true ->
    in request(size) st free by size -> free := false
    [] release() -> free := true
    ni
  od
end
```

Each iteration of the server's loop will service an invocation of `request`, but only if `free` is true, or an invocation of `release`. The invocation serviced by the first arm will be the one that minimizes the scheduling expression (i.e., the expression following `by`). Note how the scheduling expression references the invocation's parameter, `size`.

SR allows arrays of operations to be declared, e.g.,

```
op h[1:4](int)
```

A particular element of an array of operations can be serviced, e.g.,

```
in h[2](x) -> ... ni
```

In addition, an arm of an input statement can contain a quantifier, indicating that any one element from the specified range within an array of operations can be serviced. For example, each execution of the following input statement services one element of the array of operations `h[1:4]`.

```
in (i := 1 to 4) h[i](x) -> ... ni
```

The final arm of an input statement can be an else arm. The statements associated with this arm will be executed if no pending invocation satisfies a guard in the other arms. One use of an else arm is:

```
# service all invocations of a for which x = t
do true ->
  in a(x) st x = t -> ... [] else -> exit ni
od
```

The overall effect of the loop is to service all pending invocations of `a` whose parameter `x` is equal to `t`. On each iteration of the loop, the input statement services one such invocation, if there is one, or exits the loop.

SR allows operations to be shared by processes or to be local to a specific process. Shared operations are useful for implementing shared work queues and semaphores (see Section 2.2). Local operations are useful for implementing private communication channels between two processes. SR also allows *capabilities* for operations. A capability for an operation is essentially a pointer to an operation, and can be assigned to variables and passed as parameters, thus permitting dynamic communication paths. Consider, for example, the following code.

```
op h[1:4](x: int)
var c: cap (x: int)
 ...
if a > b -> c := h[1]
[] else ->  c := h[2]
fi
 ... # other code not changing c
call c(20)
```

Depending on the values of `a` and `b`, the capability `c` will be assigned either `h[1]` or `h[2]`. Execution of `c(20)` will invoke either `h[1]` or `h[2]`, passing `20` as the parameter.

2.2 Abbreviations

The example in Figure 1 uses the process abbreviation to specify processes that are created automatically when the enclosing resource is created. The process keyword is defined as an abbreviation for an operation declaration, a proc, and one or more sends to that proc by the resource's initial code, which is executed by the resource's "initial" process when the program begins execution (see Section 2.5). For example, the `phil` processes can be expressed as

```
op phil(id: int)
proc phil(id)
  # same code as in the body of phil previously
end
# this loop creates N instances of process phil
fa id := 1 to N -> send phil(id) af
```

We will employ this general process transformation in Section 3; we refer to it as *T-process*.

SR provides a receive statement as an abbreviation for a common form of input statement. For example, it defines the receive statement

```
receive f(y)
```

to be equivalent to the input statement

```
in f(x) -> y := x ni
```

We refer to this general receive transformation as *T-receive*.

SR provides semaphore declarations and P and V operations. However, these are actually just abbreviations for operations and send and receive statements. Specifically, a semaphore is a parameterless shared operation. A V on a semaphore corresponds to a send to the operation; a P corresponds to a receive on the operation. Initialization of the semaphore corresponds to a for-all loop that sends to the operation the given number of times. We refer to this general semaphore transformation as *T-sem*.

2.3 Concurrent Invocation

SR's concurrent invocation statement — a *CI-stmt* for short — provides a way to start several invocations at the same time. As a simple example, the concurrent invocation statement in Figure 2. consists of three invocations: one each of `p`, `q`, and `r`. The final invocation assigns the value that `r` returns to `a`. The statement terminates when all its invocations have completed.

The CI-stmt allows each arm to have a post-processing block of code (PPB) associated with it; a PPB is executed as the associated invocation completes. For example, the output of the program fragment in Figure 3 is 1 and 3, in unknown order. PPBs are executed one at a time by the process executing the CI-stmt; they are *not* executed concurrently. If a PPB executes an exit statement, then the executing process does not wait for further invocations to complete. For example, the code in Figure 4 uses a quantifier to initiate four invocations. When any of the invocations terminates, the PPB records, in `which_one`, the index for that invocation and then exits the `co` without waiting for the other invocations to complete.

2.4 Reply and Forward

SR's reply statement allows a servicing process to send an "early reply" to its invoker. After execution of a reply statement, the invoking and servicing processes can both continue their executions.

SR's forward statement defers replying to an invocation and instead passes on this responsibility to another operation. It passes this responsibility to the target operation as part of a new invocation.

2.5 Resources

So far, the example programs have contained just one resource, SR's primary modular component. SR allows instances of resources to be dynamically created. Resource instances are created via the create statement. To illustrate this mechanism, we present, in Figures 5 and 6, another version of Dining Philosophers. Like the version in Figure 1, it employs centralized control, but it places philosophers and servants in different resource instances. The main resource consists entirely of "initial" (top-level) code. This code is executed when an instance of the resource is created; in the case of `Main`, its initial code is executed when the program begins execution. The value returned by a create statement is a resource capability for the newly created resource instance. A resource capability contains operation capabilities for all operations defined in the resource's specification part; their values are assigned to those operations just created in the newly created resource instance. `Main`'s initial code creates one instance of the `Servant` resource and `n` instances of the `Philosopher` resource, passing to each philosopher its integer identity, `i`, and a capability for the servant, `s`. Each philosopher requests forks from the servant, eats, and later release forks. The servant handles these invocations in the same way as the servant in Figure 1 did.

2.6 Virtual Machines

SR defines the notion of a *virtual machine* (VM) as its language mechanism for distributing a program onto physical machines. Each VM resides on one physical machine. Like resource instances, VMs are created dynamically via the create statement.

The program in Figures 5 and 6 executes on a single virtual machine and, therefore, on a single physical machine. It can be easily modified, though, so that each philosopher executes on a different virtual machine, each on a different physical machine. Only `Main`'s loop needs to be changed, for example, as shown in Figure 7. The `on` clause in the first create specifies the physical machine on which the VM is to be created; the value returned by this create statement is a capability for the newly created virtual machine. The `on` clause in the second create specifies the virtual machine on which the resource instance is to be created. The array `host` contains (as strings) physical machine names.

2.7 Global Components

The other SR modularization construct is the global component. A global component is structurally similar to a resource. However, only one instance of a global component can exist on a given virtual machine. Global components are convenient for declaring variables, operations, and library procedures shared by processes executing within a virtual machine. A global component is created implicitly when the first import statement naming the component is encountered during execution rather than explicitly via a create statement.

3 Transformations

There are two sets of transformations: recording and replaying. The *recording transformations* transform the original program into one that records a sequence of events that later can be replayed. The program that applies the transformations is called the *recorder* and the transformed program is called the *recording program*. Similarly, the *replaying transformations* transform the original program into one that replays from a recorded sequence of events. The program that applies the transformations is called the *replayer* and the transformed program is called the *replaying program*. Both sets of transformations include the abbreviation-expanding transformations T-process, T-receive, and T-sem, as given in Section 2.2. Both recording programs and replaying programs employ a *controller process*, an extra process that coordinates the recording or replaying activities. The controller in the recording program is called the *recording controller*; the controller in the replaying program is called the *replaying controller*.

The recording transformations are generally quite simple. As one example, after a process services an invocation via an input statement, it informs the recording controller which invocation it serviced by invoking an operation in the controller. The code for that operation writes the event to the event file. Section 4 gives more details regarding the recording transformations and controller. Except as otherwise noted, the transformations given in the rest of this section are replaying transformations.

The subsections that follow present the transformations for language mechanisms described in corresponding subsections of Section 2. Section 3.1 presents transformations for single resource programs (on one VM) that use only basic SR synchronization constructs (call, send, input, proc), establishing the basic idea behind our approach. Sections 3.3 and 3.4 extend the transformations to handle concurrent invocation, reply, and forward statements. Sections 3.5, 3.6, and 3.7 further extend the transformations to handle multiple resource instances, multiple VM's, and global components.

3.1 Invoking and Servicing Operations

3.1.1 Events and Process IDs

The basic idea is to exploit the fact that in SR operations are used for all communication between processes. (Processes can also communicate via shared variables; see Section 6.) We therefore assign a unique invocation number to each invocation. Each process is given a unique process number. Since processes are created via invoking operations (Sections 2.1 and 2.2), the process number assigned to a process is simply the number of the invocation that created the process.

The act of invoking or servicing an operation constitutes an *event* in the execution of a program. More specifically, each invocation of an operation is an event, and each service of an invocation is also an event. Thus, a process number is just an event number.

Each event is represented by a pair of numbers:

(process number, event number of invocation being serviced)

For an invoke event, the first number is the process number of the invoking process; the second number is not used. For a service event, the first number is the process number of the servicing process; the second number is the number of the event of the invocation being serviced. Consider, for example, the partial event stream:

event number	*event pair*	*kind of event*
...		
30	(19, N/A)	invoke
...		
34	(8, 30)	service
...		

These two events involve processes numbered 19 and 8, which were created due to the invocations that occurred at events numbered 19 and 8 (not shown above). The two events indicate that process 19 invoked an operation at event count 30. That invocation is serviced by process 8 at event count 34.

3.1.2 Transformations

Our recording transformations, given the above form of an event, assign each invocation an invocation number. Our replaying transformations, then, ensure that invocations are serviced in the correct order. In both transformed programs, processes coordinate with a controller process before actually invoking or servicing an operation. The controller needs to uniquely identify each invocation. Thus, each operation is given an extra parameter to hold the invocation number of each invocation. As mentioned earlier, for an operation serviced by a proc, this parameter becomes the newly created process's process number.

The replaying transformations for invoking and servicing operations are:

- *T-invoke*: Immediately before invoking an operation, the invoking process must receive from the controller the event number for the invocation, which it then includes as a parameter in the invocation.
- *T-input*: Immediately before executing an input statement, the servicing process must receive from the controller the invocation number of the invocation to service. The input statement is then transformed to service only that invocation as further illustrated and described below.
- *T-proc*: When a process begins executing the body of a proc, it interacts with the controller in the same way as in T-input, even though it already has its event number (i.e., its process number) from a parameter to the proc. This interaction allows the controller to treat consistently all servicing of invocations.

Recall that the abbreviation-expanding transformations — T-process, T-receive, and T-sem — described in Section 2.2 also involve operations and map abbreviated code into more basic code.

The transformations also handle invocations of operations that return values, which are typically invoked within expressions. For example, T-invoke will transform the invocations of `f` and `g` in the expression `x:=y+f(45)*(g(34)-8)` so that each includes its event number as an additional parameter.

3.1.3 Example

Figure 8 illustrates these transformations by showing the transformed version of Figure 1. The figure omits or slightly modifies some details. For example, it omits the declaration of `r_event`, which is the type of event numbers. As indicated above, event numbers are integers; later (Section 3.6), however, they will be pairs of integers. The transformations shown here are independent of the actual definition of `r_event`.

- Lines 10-11 and 24 show the applications of a T-process transformation and then a T-invoke transformation that adds the event number as an extra parameter. Lines 26-27 and 37-39 are similar; the loop corresponds to the quantifier in the original process heading.

- Lines 31 and 33 illustrate applications of the T-invoke transformation. Again, note the extra parameter.
- Lines 16-21 illustrate an application of the T-input transformation. Note how the original synchronization expression is no longer needed, but that, to ensure the correct replay order, each arm now contains a synchronization expression that specifies to accept an invocation only if its event number matches the one that was recorded in the event stream. (See Section 3.1.4 for more details.)
- Lines 12-13 and lines 28-29 illustrate two applications of the T-proc transformation. Lines 3-4 are similar, but they are for the initial process. Note that `r_mypid` on line 4 refers to an extra parameter (not shown here) to the resource; see Section 3.5.

3.1.4 T-input — more details

It is important to note how T-input handles the potential complexities of SR's input statement (Section 2.1) in a very simple way. Whether a guard in the original input statement contains a synchronization or scheduling expression is irrelevant in the transformed program since those expressions are — as seen in Figure 8 — both replaced by a simple synchronization expression, using only event numbers, that enforces the desired order of servicing invocation. For the same reason, an input statement containing multiple arms that service the same operation or an input statement in which the arms contains quantifiers is also handled correctly.

Extending T-input to handle the else arm on an input statement is straightforward. Such an input statement is transformed into an equivalent if statement to handle the else and an else-less input statement. The event sequence records whether the else arm or a regular arm of the input statement should be executed. Consider, for example, the input statement

```
in f(x) -> S1
[] else -> S2
ni
```

It is transformed into

```
r_myevn := r_service(r_mypid)
if r_myevn = r_ELSE ->
  S2
[] else ->
  in f(x,r_evn) st r_myevn = r_evn -> S1 ni
fi
```

Without this extension to T-input, a process executing such an input statement might find that no invocations are pending and execute the else arm, when in fact the invocation it is supposed to service has not yet arrived.

3.1.5 Basic replaying controller

To complete the picture of the basic transformations for invoking and servicing operations, Figure 9 shows the essence of the code for the replaying controller. (Section 4 shows the full controller code.) Each iteration of the controller's loop handles a request by a process to invoke or to service an operation. The such-that clause on each arm is used to handle only the request from the process whose turn it is. For example, suppose that several philosophers in Figure 8 request forks (line 31) at about the same time. Each invokes `r_invoke`. The controller will allow only the philosopher whose turn it is, as given in the event sequence, to proceed once the controller reaches that event in the sequence.

3.2 Abbreviations

Recall that the T-process, T-receive, and T-sem transformations from abbreviated forms to unabbreviated forms were given in Section 2.2. They were already illustrated in the example in Section 3.1.3.

3.3 Concurrent Invocation

The transformation for a CI-stmt (concurrent invocation statement) that contains no post-processing block (PPB) is straightforward. First, invocation numbers for all the invocations are recorded by the recording program or retrieved by the replaying program. Then, a transformed CI-stmt is used to perform the invocations. This statement differs from the original one in that each of its invocations includes as a parameter the appropriate invocation number, just as for the invocations in Section 3.1. Hence, no change to the controller is necessary. To illustrate, consider the CI-stmt

```
co p(3) // q() oc
```

The transformed version is roughly:

```
begin
  var r_arm_1: r_event; r_arm_1:= r_invoke(r_mypid)
  var r_arm_2: r_event; r_arm_2:= r_invoke(r_mypid)
  co p(3,r_arm_1) // q(r_arm_2) oc
end
```

Because the order in which invocations are performed within a CI-stmt is nondeterministic, the transformed code first computes invocation numbers to ensure that the individual invocations within the CI-stmt will be numbered consistently between the recording and replaying programs. Otherwise (i.e., if `r_invoke(r_mypid)` were included directly in the invocations), the invocations may occur in different orders and be assigned different numbers. (Also see Appendix B.1.)

We treat a CI-stmt that contains an invocation that assigns to a variable — such as in Figure 2 — as having a PPB. The assignment is conceptually part of an implicit PPB because it occurs after the invocation completes.

The transformation for a CI-stmt that contains one or more PPBs is more complicated. It is described in detail in Appendix A, but its key requirements are as follows. Recall from Section 2.3 that the order in which PPBs are executed depends on the order in which invocations complete. The transformations must ensure that that order in the replaying program is identical to that in the recording program. As a specific example, consider the CI-stmt in Figure 3. Suppose that, in the recording program, the invocation of `r` completes before the invocation of `p`. In the replaying program, the two invocations might complete in the opposite order; the replaying program, though, must not execute the PPB for `p` (i.e., `write(1)`) until after the PPB for `r` (i.e., `write(3)`), even though `p` completes first. Similarly, the value assigned by the program fragment in Figure 4 to `which_one` in the replaying program must be the same value as that in the recording program, even if other invocations complete earlier in the replaying program.

3.4 Reply and Forward

Our scheme easily handles SR's reply and forward statements. A reply statement does not need to be transformed at all because it does not generate or service an invocation. On the other hand, a forward statement does generate an invocation. We, therefore, treat a forward statement just as another kind of invocation statement — i.e., as a call or send statement — and so we just apply T-invoke.

3.5 Resources

Extending our scheme to handle multiple resource instances is fairly straightforward. The creation of a new resource instance, accomplished by a create statement, is treated the same as invoking an operation. Specifically, the controller is invoked (via `r_invoke`) immediately before a resource instance is created to ensure that the creation occurs in the right order and to obtain the event number for the process that executes the resource's initial code. The process number (returned by `r_invoke`) is then passed as an additional resource parameter, similar to how operations are passed an extra parameter (Section 3.1). That number then becomes the process number for the resource's initial process. For example, the create statements and the loop in Figure 5 are transformed to

```
s:= create Servant(n,r_invoke(r_mypid))
fa i := 1 to n ->
  create Philosopher(s,i,t,r_invoke(r_mypid))
af
```

Accordingly, the declarations of the servant and philosopher resources are transformed to include the extra parameter, e.g.,

```
body Servant(id: int; r_mypid: r_event)
```

Because SR requires that a program's main resource be parameterless, the transformations introduce a new main resource. The new main resource creates an instance of the original main resource, passing to it as a parameter the event number to be used as the process number for its initial process.[2]

3.6 Virtual Machines

Allowing multiple virtual machines, which might execute on multiple physical machines, is conceptually fairly simple, but it does slightly complicate the transformations. Each VM is assigned a unique VM number, which is included in the representation of an event. Specifically, the process number part of an event number is augmented with the number of the VM on which the event occurred. That is, the event representation defined in Section 3.1.1 is extended to:

((VM number, process number), event number of invocation being serviced)

The event sequence is split into multiple sequences, one for each VM in the program. Also, each VM has its own local controller process, used in recording or replaying the events caused by processes that execute on that VM. Processes on a given VM do not interact with controllers on other VMs, but they continue to interact with processes on other VMs. (See Section 6 for related discussion.)

The recorder must, therefore, assign a unique number to each VM. The transformation of each `create vm()` includes interaction with a centralized process (one per entire program) that doles out VM numbers and interacts with the controller on the local VM to record the creation event or to replay the creation event in the recorded order. (Since VM creation occurs relatively infrequently, such centralization is not a bottleneck.)

The VM number needs to be passed to the VM when it is created. Unfortunately, that cannot be accomplished directly because `create vm()` does not take parameters. Hence, we use a slightly roundabout method. To illustrate, the code in Figure 7 is transformed in the replaying program to

```
fa i := 1 to n ->
  var vmcap: cap vm
  begin
    var r_id := r_vcreate(r_mypid)
    vmcap := create vm() on host[i]
    import r_phelper
    create r_phelper (r_id) on vmcap
  end
af
```

When the above code invokes the local controller (via `r_vcreate`), the controller interacts with the centralized process to obtain the VM number for the new VM; the local controller returns that VM number, which is assigned to `r_id`. The above code then creates the actual VM. Finally, it creates, on the new

[2] The main resource could be treated specially and not passed a parameter. However, that is problematic because a program can (although rare) create multiple instances of its main resource.

VM, an instance of a helper resource (`r_phelper`), to which it passes the VM number. In turn, the helper resource (not shown) passes the VM number to an operation in the controller on the new VM, so that the new controller knows its VM number.

Note that when a process on one VM services an invocation from a process on another VM, the recording controller on the first VM will record in its event sequence the process number of the invoking process. The VM part of that number differs from the VM number of the recording controller.

3.7 Global Components

The transformations for a global component are similar to those for an instance of a resource. The key difference results from how a global is created — implicitly via an import statement rather than explicitly via a create statement. As for an instance of a resource, a global's initial process is assigned a process number. The recorder transforms each global component so that it first invokes an operation in the controller to obtain a process number. This operation records in the event sequence the usual event number along with the name of the global component. Similarly, the replayer transforms each global component so that when it is created via an import statement, it will first retrieve from the controller the initial process's process number. This transformation differs from the one for resource instances in how the initial process gets its process number. Since resource instances are distinguishable (e.g., via their resource parameters, and how and by whom their services can be referenced), they are passed distinguishing initial process numbers from their creators. On the other hand, there is only one instance of a given global component on a given VM and its services are available to all its importers, so instances of global components do not need to be similarly distinguished. Hence, the creation of a global component appears to be spontaneous, i.e., as the result of some import statement being executed. So, the initial process obtains its process number when it begins to execute.

4 Controllers

As part of the transformations, the recorder and replayer place a controller process into the transformed program. The controller process appears as a process within a new global component. The controllers are independent of the original program.

The controller processes behave as outlined in Section 3. In more detail, the recording controller assigns event numbers for operation invocation, resource creation, virtual machine creation, and global component creation as those events occur during execution; it also writes those events to a file. Its code is outlined in Figure 10. The record type `r_event` is used to construct the event number from `myvid`, the assigned number for the virtual machine on which the controller runs, and the event counter `ev_num`. Each arm corresponds to one of the kinds of events and writes an event consisting of five fields to the sequence file. Some of those fields are constants: `r_NA` for "not applicable", `r_GLOB` for "global", and `r_VM` for "virtual machine". For an event that creates a global component, the last field records the global's name; for other events, the string that is written identifies the kind of event and is used only to debug the recorder and replayer.

Each controller writes its own event file, containing all the events that occurred on that VM. The creation of a virtual machine, as discussed in Section 3.6, is reflected only as a comment in Figure 10.

The essence of the replaying program's controller appears in Figure 11. This expanded version of Figure 9 handles all replaying transformations, including those for multiple VMs. On each iteration of its loop, the controller reads one event from the event file for its VM and services the invocation that corresponds to that event. Each kind of event is handled by its own arm of the input statement, as it was in the recording controller. The desired correspondence is enforced by the such-that expression on the guard of each arm, which forces the controller to wait for the invocation from the right process.

An option to the recorder causes it to generate random delays in the recording program. This randomness is useful for testing with different orders of execution. The same option to the replayer also generates random delays in the replaying program. Given that the replaying program is supposed to reproduce the execution of the recording program, this randomness is useful only for debugging the replayer. (It would also be useful for testing for unprotected shared variables; see Appendix B.4).

5 Experience Building and Using the Transformers

5.1 Implementation

We have fully implemented the transformations described in Section 3. Our recorder and replayer each takes as input a valid SR program and outputs an appropriately transformed SR program, which is then compiled, linked, and executed using the standard SR implementation. Each of the transformers performs its transformations by parsing (using lex and yacc) the SR source input, building a parse tree, and then outputting the transformed program from the parse tree. We found it easier to print the additional transformed code during a traversal of the parse tree rather than to add that code as new nodes in the parse tree itself and print the parse tree during a separate pass.

5.2 Experience and Use

We have used the recorder and replayer on numerous programs, including truly distributed programs. These tools have also been used as a debugging aid by students learning about concurrent programming. The experiences using the tools have generally been positive in that they provide reproducible execution, even when the programs include random delays within critical sections, as many of the students' programs do to simulate other activities. Below, we describe several typical uses of the transformation tools in debugging students' concurrent programs. One negative aspect observed in these experiences is that the tools do not currently handle shared variables, a common source of programming errors. (See Appendix B.)

The transformation tools were used in debugging application code that employs a barrier to synchronize worker processes in a grid computation using Jacobi iteration to approximate a solution to a PDE. The code was observed to intermittently produce erroneous results. The recorder tool was used to record a particular sequence of events that resulted in the erroneous results. Due to the intermittency of the bug, the user needed to have the recorder run the program a few times before finding an erroneous execution. Then, additional output statements to obtain further information about program execution were added to the program and the replayer was used to reproduce the erroneous execution. The output was helpful enough to lead to identifying the error: use of a one-time barrier rather than a reusable barrier, which did not properly delay all processes at the barrier. Note that the program containing the additional output statements, which was run by the replayer, is different from the original program, which was run by the recorder. However, its event sequence is unchanged by output statements and so the replayer can reproduce (with respect to the event sequence) the recorded erroneous execution. This technique has been useful in other scenarios too.

The transformation tools were also used in debugging three independent versions of application code that implements the algorithm from [12] for mutual exclusion in a distributed environment. Each of these program versions intermittently exhibited the erroneous behavior of not providing mutual exclusion. The first version had an "off by one" error (a $<$ instead of a $<=$) in the loop that decides when a process has received "go ahead" replies from the other processes to its request for obtaining mutual exclusion. The second version had a logical error: wrongly responding to requests from other processes while the process at a node was currently in its critical section. The third version had an error in updating each node's sequence number, which is used to resolve which process should be given permission to enter its critical section when more than one are trying at about the same time. The transformation tools were used in ways similar to those in the above barrier scenario to identify these bugs, i.e., use the recorder to record an event sequence on which the program exhibits the erroneous results, add additional output statements to the program to obtain more information, and run that program under the replayer. The bugs in the first and third versions were not immediately obvious to the programmer, so the transformation tools were very helpful in these cases. However, the bug in the second version indicated a significant misunderstanding and could have more easily been spotted by the student re-examining the basic algorithm instead of by using the transformation tools. In this case, though, the tools were still useful to the student to narrow down the cause of the bug.

One interesting behavior observed in the use of the transformation tools is that the correct behaviors of the recording and replaying programs depend on them running on the same input. In one debugging session on a version of Dining Philosophers similar to that in Figures 5 and 6, different values for the number of philosophers were accidentally input to the recording program and the replaying program, 5 and 6 respectively. Interestingly, the replaying program reproduced the execution and output of the recording program. The only difference was that the replaying program ended with a blocked process when it should have terminated normally. The main process in the program was the one reported as being blocked. The controller, correctly, followed the given event sequence and did not allow the main process to create philosopher number 6, so the main process blocked — in particular, waiting for the invocation `r_invoke(r_mypid)` to complete. The replaying program got that far in its execution because all five philosophers needed to replay the program's execution — as given by the event sequence — had been created and were interacting as they should with the controller; also, the main process had no further events in the event sequence that it needed to execute. On the other hand, with an incorrect input of 4, the replaying program "hangs" sooner because the controller will not receive an `r_invoke` from the main process for the event in the sequence that creates the missing philosopher.

5.3 Overhead

The transformed programs require additional overhead for the processes to communicate and synchronize with the controller and for the controller to write or read the event file. The additional execution time varies according to the frequency in the original program of activities such as generating and servicing invocations. For a collection of test programs, the transformed programs ran slower by 25-60%. For one contrived test program, the replaying program actually ran *faster* by about 28% (although the recording program still ran slower by about 27%). In that program, a number of invocations were sent to an operation serviced within a loop by an input statement with a complicated scheduling expression. On each iteration of the loop, the code generated by the SR implementation evaluates the scheduling expression for *all* pending invocations so that it can pick the invocation that minimizes the value of the scheduling expression. T-input (Section 3.1) eliminates the scheduling expression and replaces it with a simpler synchronization expression. The code generated by the SR implementation for the transformed program evaluates the synchronization expression until it finds an invocation for which the synchronization expression is true. For this particular program, that almost always occurs before it reaches the final pending invocation, thus resulting in faster execution.

5.4 Helpful Language Features

In building the transformers, three features of SR made it easier to generate the transformed code. First, SR allows input statements to select an invocation based on the invocation's parameters. This feature is fundamental in the controllers and transformed input statements. (Of course, this feature also makes reproducing execution more difficult in the first place.) Second, SR allows declarations, including import statements, to be intermixed with code, and allows a given component to be imported into a given block multiple times. The transformers can, therefore, generate declarations or import statements as needed rather than needing to ensure that these statements are placed at the beginning of a block and that a given component is imported just once. The transformers similarly can generate code between declarations, e.g., for the `r_service` in T-proc or for the initialization in T-sem. Third, SR operations can return values. The transformers exploit this feature in several places by generating code inline within expressions rather than generating code using a temporary variable before the expression. For example, the value-returning operation `r_invoke` is used within (basic) call and send invocations, as on line 24 of Figure 8:

```
24   send server(r_invoke(r_mypid))
```

Without using a value-returning operation here, a different `r_invoke` operation, using a result parameter, would need to be invoked before the invocation. That by itself would complicate the transformers. Further complexity would also result from the need to preserve short-circuit evaluation when multiple invocations appear within a logical expression.

6 Discussion

A comprehensive approach to reproducible execution for the debugging and testing of Ada programs is described in [3]. It is difficult to directly compare that approach and our approach because they are aimed at two different languages, whose respective semantics impose different demands with respect to the reproducibility problems. Nonetheless, the two approaches have a number of noteworthy similarities and differences.

Several possible approaches to collecting and replaying events are given in [3]. Among the possibilities are to use the total ordering of events or the partial ordering derived from (i.e., implied by) the total ordering. The partial ordering is derived from the perspective of accepting tasks. That is, associated with each task, T, in a program is the sub-sequence of the original sequence of all events for which T is the accepting task. The transformations given in [3] for replaying run off the total ordering, but they only guarantee that the replayed events follow the partial ordering. The exact form of events employed varies, but in the totally ordered sequence, each event contains the identifiers of the calling and accepting tasks, and other information. Before a rendezvous is allowed to be performed the accepting task and calling tasks must coordinate (via rendezvous) with a control task. The control task allows these tasks to proceed — i.e., be permitted to begin execution of the call and accept statements — only when the rendezvous is allowed by an event in the sequence. Before these two tasks actually perform the permitted rendezvous corresponding to the recorded event, the control task may have interacted with another pair of tasks, which may have completed their rendezvous.

Tools can implement different variants of these orderings in a number of ways, as described in [3]. One issue is how many controllers should be present in the transformed program and where they should be located. The choices range from one controller for the entire program, to one controller for each physical machine in a program, to one controller for each accepting task in a program. The transformations and tools given in [3, 4, 5] work with the first choice.

Our approach defines events differently. We record *separate* events for the invoking and servicing of an operation, rather than one for those actions combined. The service event does not necessarily immediately follow the corresponding invoke event in the sequence. (In fact, if the invocation is serviced from a different

VM, the service event is in a different sequence, as described in Section 3.6.) These separate events and separate recordings are necessary to support SR's send statement, which is asynchronous. In particular, a separate invoke event is necessary so that the sender does not block until the receiver is ready to service the message, which could easily result in deadlock. It also simplifies the handling of invocations that cross VM boundaries, which otherwise would need to be coordinated for recording a single event.

Our approach records a separate event sequence for each VM in the program capturing the events that occur on that VM. Together, these sequences form a partial ordering of the program's overall behavior. Within each VM, these sequences form a total order, but for the same reason as noted above for [3] and further because we have separate invoke and service events, the replay of each follows only the derived partial ordering; each of these partial ordering is from the view of the servicing process. This approach, unlike that in [3, 4, 5], allows a distributed algorithm with processes on a given VM controlled by and interacting with the controller only on that VM, as described in Section 3.6. Note that whenever a new VM is created, our transformations automatically place a new controller on that VM. Thus, our approach becomes distributed as the program itself becomes distributed. Having multiple controllers avoids additional messages between machines and potential bottlenecks.

Our work handles dynamic process creation because it is a fundamental part of SR and hence essential. Earlier work does not; e.g., [3] does not address dynamic task creation. Our work also handles dynamic paths of communication between processes (capabilities in SR terminology) and shared operations, whose invocations can be serviced by multiple processes (Section 2). Both dynamic processes and dynamic paths are handled by using invocation identifiers that are tuples of numbers (Section 3.6) in the event sequence. Our approach also allowed fairly straightforward extensions to handle reply, forward, and concurrent invocation statements. Using names of processes or operations, as in [3, 4, 5], would not work. For example, the name of the operation actually invoked via an invocation of an operation capability is not known until run-time and is often not accessible to the invoking process. Thus, our approach requires that each process be given its identifier at the start of its execution, which is also the approach taken in the transformations given for monitoring deadlock in Ada programs in [13]. As a result, our approach and the approach in [13] have minor problems in elaborating declarations. For example, our approach fails for some resource specifications; see Appendix B.5 for details.

The transformation for a selective wait statement in [3] includes a busy-waiting loop that waits to ensure the desired entry call has arrived before the selective wait statement is executed. This loop is needed to prevent the erroneous execution of the else part in case the desired entry has not yet been called, the same problem discussed in Section 3.1.4. Our solution to this problem, transforming an input statement with an else arm to an if statement and an else-less input statement, could also be used in the framework of the [3] work to eliminate the busy waiting and simplify the transformation.

In their most general form (to handle two or more alternatives for the same entry), the transformations in [3] record the number of the alternative taken, and replay that same alternative by adding to each guard a clause (or, if no guard existed, by introducing a new clause) that checks the alternative number. In contrast, our replaying transformation of the input statement always removes any existing synchronization and scheduling expressions and adds a new synchronization expression that ensures only the right invocation is serviced. Evaluation of this synchronization expression takes little extra time for the invocation that is serviced, but the expression also needs to be evaluated for any other pending invocation, which represents an additional cost.

SR's '?' operator (not previously discussed) returns the number of invocations currently pending for an operation. Thus, it is similar to Ada's COUNT attribute. The transformations for uses of '?' are essentially the same as for those for the COUNT attribute presented in [3]. In both, the recording controller saves the current value in the event sequence and the replaying controller retrieves the value from the event sequence.

Our approach cannot be applied directly to the problem of reproducing Ada programs because Ada does not permit parameters of an entry call to be used in deciding which entry call to service — that functionality is the basis of the SR approach in selecting invocations. That functionality can be simulated in Ada — for example, using the technique described in [14] or using a technique that employs the requeue statement in Ada 95 [15] — but the resulting simulation would be cumbersome and costly.

A scheme for reproducing execution of programs that use send and receive primitives for synchronization is given in [2]. That scheme is not powerful enough to handle, or easy to generalize, to SR programs. For example, it cannot handle synchronization expressions and scheduling expressions on SR's input statement or most of SR's other synchronization mechanisms.

Transformations for reproducing the execution of programs that employ semaphores for synchronization are described in [1, 16]. In SR, semaphores are defined as a special kind of operation. The transformations for SR's semaphores are therefore a special case of the more general transformations for operations. Specifically, T-sem transforms declarations and uses of SR semaphore into declarations of operations and statements that invoke and service operations. In turn, T-input and T-invoke transform these further. The applications of these transformations yields transformed programs that are more complicated and less efficient than programs transformed using the semaphore-specific techniques given in [1, 16].

As noted in Section 5, we have fully implemented our transformations. They work on all SR language mechanisms, but they make some assumptions or have some shortcomings — depending on one's perspective — with respect to the following items:

item	*arises in practice?*	*have workaround?*
Consistent Order of Expression Evaluation	no	yes
Sequential Determinism	no	yes
Invoking User-defined Operations in Guards	rarely	no
Shared Variables	yes	yes
Executable Code in Resource Specs	rarely	no

As indicated, for most of these items, we can define additional transformations; a few of these items are not common in practice. See Appendix B for further discussion, examples, and extensions of our transformations.

7 Conclusion

In this paper, we have presented a solution to the problem of reproducing the execution of programs written in the SR concurrent programming language. We gave program transformations that work for the full SR language — including SR's synchronization mechanisms, dynamic processes, dynamic communication paths, shared operations, and truly distributed programs. We also described our implementation of the program transformations, both for recording an event sequence and replaying from a previously recorded event sequence. Our approach is not without shortcomings, but it is nonetheless quite successful.

Acknowledgements

Mandy Chung and Michele Fulton provided useful comments on an earlier draft of this paper.

Appendix A Transformation of General CI-stmts

Section 3.3 presented the transformation for concurrent invocation statements (CI-stmts) without post-processing blocks of code (PPBs). That basic transformation would not necessarily preserve the required ordering of PPB execution, as demonstrated by the examples in Section 3.3. However, it can be adapted to ensure that PPBs execute in order. Two changes are needed:

- The recording program must record the order in which PPBs execute.
- The replaying program must ensure that a PPB is allowed to start execution only if it is the PPB's turn.

The first change is simple: the recording program records a new "start of PPB" event when a PPB begins execution; this event records the number of the invocation for which the PPB is being executed. The second change is more complicated. The replaying program knows, from the recorded "start of PPB" events, the order in which the PPBs are to be executed. However, the replaying program must prevent the process executing a CI-stmt from executing the PPB code unless it is that PPB's turn.

To provide this kind of control, the replaying program needs to check, when each invocation completes, whether it is that invocation's turn to execute its PPB. If not, that invocation needs to be delayed from completing. Unfortunately, such delaying cannot be expressed directly as part of the CI-stmt. The effect, however, can be simulated via additional transformations that use a separate process for each invocation. For example, the transformed versions, for both recording and replaying, of Figure 3 will be roughly[3]:

```
co co_helper_p(3,r_invoke(r_mypid)) ->
   # interact with controller -- start of PPB.
   write(1)
// co_helper_q(r_invoke(r_mypid)) ->
   # interact with controller -- start of PPB.
// a := co_helper_r(x,y,r_invoke(r_mypid)) ->
   # interact with controller -- start of PPB.
   write(3)
oc
```

The `co_helper` procs do the actual invocations and provide the opportunity to delay execution of a PPB. For this example, they are:

[3] For brevity of exposition, this code computes the invocation numbers inline (i.e., `r_invoke(r_mypid)`) rather than before the CI-stmt as it should and as seen earlier.

```
proc co_helper_p(x,r_mypid)
   p(x,r_invoke(r_mypid))
   # interact with controller -- co_inv_done.
end

proc co_helper_q(r_mypid)
   q(r_invoke(r_mypid))
   # interact with controller -- co_inv_done.
end

proc co_helper_r(x,y,r_mypid) returns z
   z := r(x,y,r_invoke(r_mypid))
   # interact with controller -- co_inv_done.
end
```

The recording controller simply records each start of PPB event (`start_PPB`); it does nothing at the end of each invocation (`co_inv_done`). The replaying controller enforces the ordering based on the recorded event sequence. It does so by allowing a process to move from completing an invocation to executing its PPB only when a recorded event indicates it should. Specifically, the replaying controller accepts a `co_inv_done` that corresponds to the recorded PPB event and then waits until the process begins the PPB and so informs it. This wait ensures that the process does not begin a PPB for another invocation that may have just completed.

These transformations handle all the complexities of CI-stmts: multiple executions, nesting, and early exit (as seen in Figure 4). The key point for handling each is that the execution of a PPB is uniquely identified by the associated invocation. For example, suppose the PPB in Figure 4 did not contain an `exit`; then it would be executed four times. The different executions of the PPB are recorded by their unique invocation numbers, so they can be replayed in the same order.

One problem with early exit is that the `co_helper` processes for invocations completing after the one that executes the PPB that exits will be left around waiting for the controller to accept them, but the controller will never do so. These processes can be cleaned up by having two new events that are used to tell the controller when a new CI-stmt begins and when a CI-stmt terminates. Then, the controller can accept (but do nothing with) the `co_inv_done` for CI-stmts that have terminated.

The `co_helper` procs, as shown earlier, assume that the name of the actual operation to invoke is within scope and that the name of the proc does not collide with other `co_helper` names. For example, `co_helper_p` assumes that the operation `p` is within scope and no other `co_helper_p` proc is needed. These assumptions might not hold. However, these problems can be solved by passing as an extra parameter to each `co_helper` proc the actual operation to be invoked and by uniquely naming each `co_helper` proc.

Appendix B Assumptions and Shortcomings

As noted in Section 6, our transformational approach makes some assumptions or has some actual and potential shortcomings. This appendix discusses those items further.

B.1 Consistent Order of Expression Evaluation

Our scheme requires that the SR compiler be consistent in the code it generates for expression evaluation. Specifically, the code to evaluate a given expression must evaluate the expression in the same order from one compilation to another. This requirement is needed for general expression evaluation, where evaluations of operands might have side effects, and it is particularly important for our handling of invocations. For example, SR semantics allows the two operands of '+' in an expression such as `f(x)+g(x)` to be evaluated in either order, so an SR compiler may generate code to evaluate the operands in either order. If the order of evaluation of the above expression differs between the recording program and the replaying program, then the replaying program will likely fail due to invocation numbers being different. This potential shortcoming can be averted by transforming each expression into a sequence of simpler expressions, employing additional variables and control statements as needed; the sequence then defines a particular order of evaluation. In practice, this potential shortcoming is not a problem because the code generated by the SR compiler is consistent in its expression evaluation.

B.2 Sequential Determinism

A similar shortcoming is that our transformations do not take into account the nondeterministic nature of SR's if and do statements. If multiple guards are true, any of the arms associated with a true guard can be chosen to be executed. If the recording program and replaying program differ in how they choose arms, then the replaying program will most likely execute incorrectly. This potential shortcoming can be averted by transforming these statements into statements whose executions are deterministic. An if statement can be transformed into a series of nested if statements. For example, the if statement

```
if B1 -> S1
[] B2 -> S2
fi
```

can be transformed into

```
if B1 -> S1
[] else ->
  if B2 -> S2
  fi
fi
```

A do statement with multiple arms can be transformed into a do statement with a single arm containing a nested if statement, similar to the one above.

Alternatively, and more in line with our general approach, new recording transformations for if and do statements could record, as a new kind of event, the arm number of the arm that was executed. New replaying transformations would transform an if or do statement so that first it retrieves from the controller the arm number to execute and then it executes that arm; for example, the if statement above would be transformed to

```
arm := r_arm(r_mypid)
if arm = 1 -> S1
[] arm = 2 -> S2
fi
```

The original guards would no longer be needed. This scheme is similar to that used in [3] for numbering arms in a selective wait statement to distinguish between multiple alternatives for the same entry; however, that scheme still uses the original guards (although it seems that they could be eliminated because the arm number provides sufficient information for replay).

It is tempting to apply a similar scheme to handle input statements. However, that would not work because not only is the arm number needed but so is some identification of the specific invocation that is being serviced.

In practice, this potential shortcoming is not a problem because the SR compiler is consistent in deciding which arm to execute.

B.3 Invoking User-defined Operations in Guards

Another shortcoming is that our transformations do not work for all invocations of user-defined operations within synchronization and scheduling expressions, e.g.,

```
in f(x) st g(x) < 20 -> ...  ni
```

The problem is that the function invocation, `g(x)` above, is transformed away by T-input, i.e.,

```
in f(x,r_evn) st r_myevn = r_evn -> ...  ni
```

This transformed code will behave identically to the original provided that `g` has no side effects and makes no other invocations of operations. However, if one of these conditions does not hold, those side effects or invocations will not occur in the replaying program, which is therefore likely to fail. A related point is that our transformations assume that synchronization and scheduling expressions have no side effects. SR does allow such side effects, but they are rarely used and are considered bad programming style. One reason is that, in general, the number of times that synchronization and scheduling expressions are evaluated is nondeterministic [10]. In practice, such function invocations (and side effects) rarely appear.

B.4 Shared Variables

Our transformations make no guarantees about the order in which processes access shared variables or perform input/output. Our scheme could be extended by defining events that correspond to these activities, as in earlier replay work (e.g., [7]). Specifically, each read of a shared variable could be treated as another kind of event for which a process would need to interact with the controller. The recording program would record the value in the event sequence and the replaying program would retrieve the value. (The values written to a shared variable would not need to be recorded or retrieved.) Of course, the transformed program would execute slower due to additional interaction with the controller, and the lengths of the event sequences would increase. However, given that misuse of shared variables is a common programming error, then the additional cost would be justified in many debugging situations. Whether the recorder and replayer should handle shared variables could easily be made a user-selectable option; to reduce costs, only specific variables could be selected.

B.5 Executable Code in Resource Specs

The specification part of a resource can create a new resource instance, as illustrated in the following code fragment.

```
resource r2
   import r1
   const c := create r1()
body r2()
    ...
```

Our transformations do not work for such cases. Recall from Section 3.5 that creation of a resource instance requires that a resource's initial process obtains its process number via an extra parameter to the resource and that the process number of the process executing the create needs to be passed as a parameter to the controller. In the above, however, the creating process — `r2`'s initial process — cannot refer to its resource parameter until after it is supposed to have created the instance of `r1`. This problem has so far been observed only in a contrived test program. Similar problems cannot arise within processes since processes in SR have no separate specification part.

References

[1] R.H. Carver and K. Tai. Replay and testing for concurrent programs. *IEEE Software*, pages 66–74, March 1991.

[2] K. Tai and S. Ahuja. Reproducible testing of communication software. In *Proceedings of 1987 IEEE COMPSAC*, pages 331–337, October 1987.

[3] K. Tai, R.H. Carver, and E.E. Obaid. Debugging concurrent Ada programs by deterministic execution. *IEEE Trans. on Software Engineering*, SE-17(1):45–63, January 1991.

[4] H. Sohn, D. Kung, P. Hsia, Y. Toyoshima, and C. Chen. Reproducible testing of distributed and concurrent programs. In *Proceedings of 12th International Society for Computers and Their Applications*, pages 288–291, San Francisco, CA, March 1996. also available via http://pepe.uta.edu/sohn/pubs/.

[5] H. Sohn, D. Kung, P. Hsia, Y. Toyoshima, and C. Chen. Reproducible testing for distributed programs. In *Proceedings of 4th International Conference on Telecommunication Systems, Modeling and Analysis*, Nashville, TN, March 1996. also available via http://pepe.uta.edu/sohn/pubs/.

[6] P. Brinch Hansen. Reproducible testing of monitors. *SOFTWARE–Practice and Experience*, 8:721–729, August 1978.

[7] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Trans. on Computers*, C-36(4):471–482, April 1987.

[8] C. Dionne, M. Feeley, and Jocelyn Desbiens. A taxonomy of distributed debuggers based on execution replay. In *Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 203–214, Sunnyvale, CA, August 1996. also available via http://www.iro.umontreal.ca/~feeley/.

[9] G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[10] G.R. Andrews and R.A. Olsson. *The SR Programming Language: Concurrency in Practice.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993. http://www.cs.arizona.edu/sr/.

[11] *MPI: A Message-Passing Interface Standard (Version 1.1).* Message Passing Interface Forum, June 1995. http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html.

[12] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion. *Communications of the ACM*, 24(1):9–17, 1981.

[13] S.M. German. Monitoring for deadlock and blocking in Ada tasking. *IEEE Trans. on Software Engineering*, SE-10(6):764–777, November 1984.

[14] N. Gehani and W.D. Roome. Rendezvous facilities: Concurrent C and the Ada language. *IEEE Trans. on Software Engineering*, SE-14(11):1546–1553, November 1988.

[15] S. Tucker Taft. Ada 9X: A technical summary. *CACM*, 35(11):77–82, November 1992.

[16] R. A. Olsson. An improved solution to the reproducibility problem for semaphores. Technical Report CSE-94-6, University of California, Davis, Department of Computer Science, 1994.

```
resource Main()
  const N := 5
  op getforks(id: int)
  op relforks(id: int)
  process server
    var eating[1:N] := ([N] false)
    do true ->
      in getforks(id) st not eating[(id mod N) + 1]
          and not eating[((id-2) mod N) + 1] ->
              eating[id] := true
      [] relforks(id) ->
              eating[id] := false
      ni
    od
  end

  # N instances of process phil, each with its own id.
  process phil(id := 1 to N)
    do true ->
      call getforks(id)  # get both forks
      write("Philosopher", id, "is eating")
      send relforks(id)
      write("Philosopher", id, "is thinking")
    od
  end
end
```

Figure 1: Dining Philosophers

```
co p(3) // q() // a := r(x,y) oc
```

Figure 2: A simple concurrent invocation statement.

```
co p(3) -> write(1) // q() // a := r(x,y) -> write(3) oc
```

Figure 3: A concurrent invocation statement with PPBs.

```
# read one copy of a replicated file,
# recording which response was received first
co (i := 1 to 4) fd[i].read(arguments) ->
    which_one := i; exit
oc
# communicate further with fd[which_one]
```

Figure 4: A concurrent invocation statement with an “early” exit.

```
resource Main()
  import Philosopher, Servant
  var n, t: int
  read(n); read(t)
  # create the Servant and Philosophers
  var s: cap Servant
  s := create Servant(n)
  fa i := 1 to n ->
    create Philosopher(s, i, t)
  af
end
```

Figure 5: Main resource in multiple-resource Dining Philosophers

```
resource Philosopher
  import Servant
body Philosopher(s: cap Servant; id, t: int)
  process phil
    fa i := 1 to t ->
      s.getforks(id)
      write("Philosopher", id, "is eating")   # eat
      s.relforks(id)
      write("Philosopher", id, "is thinking") # think
    af
  end
end

resource Servant
  op getforks(id: int)  # called by Philosophers
  op relforks(id: int)
body Servant(n: int)
  process server
    var eating[1:n] := ([n] false)
    do true ->
      in getforks(id) st not eating[(id mod n) + 1]
          and not eating[((id-2) mod n) + 1] ->
              eating[id] := true
      [] relforks(id) ->
              eating[id] := false
      ni
    od
  end
end
```

Figure 6: Philosopher and servant resources in multiple-resource Dining Philosophers

```
fa i := 1 to n ->
  var vmcap: cap vm
  vmcap := create vm() on host[i]
  create Philosopher(s, i, t) on vmcap
af
```

Figure 7: Modified `Main` loop for multiple-VM Dining Philosophers

```
resource Main(...)
  ...
  var r_myevn: r_event
  r_myevn := r_service(r_mypid)

  const N := 5
  op getforks(id: int; r_evn: r_event)
  op relforks(id: int; r_evn: r_event)

  op server(r_evn: r_event)
  proc server(r_mypid)
    var r_myevn: r_event
    r_myevn := r_service(r_mypid)
    var eating[1:N]:=([N] false)
    do true ->
      r_myevn := r_service(r_mypid)
      in getforks(id,r_evn) st r_myevn = r_evn ->
         eating[id]:= true
      [] relforks(id,r_evn) st r_myevn = r_evn ->
         eating[id]:= false
      ni
    od
  end server
  send server(r_invoke(r_mypid))

  op phil(val id: int ;r_evn: r_event)
  proc phil(id,r_mypid)
    var r_myevn: r_event
    r_myevn := r_service(r_mypid)
    do true ->
      call getforks(id,r_invoke(r_mypid))
      write("Philosopher",id,"is eating")
      send relforks(id,r_invoke(r_mypid))
      write("Philosopher",id,"is thinking")
    od
  end phil
  fa id := 1 to N ->
    send phil(id,r_invoke(r_mypid))
  af
end Main
```

Figure 8: Dining Philosophers (Figure 1) Transformed for Replay

```
op r_invoke(pid: r_event) returns inv: r_event
op r_service(pid: r_event) returns inv: r_event
  ...
process controller
  var ev := 0
  var order_pid, order_ev_num: r_event
  do ``read order_pid and order_ev_num from file'' ->
    ev++
    in r_invoke(pid) returns ev_num st order_pid = pid ->
      ev_num := ev
    [] r_service(pid) returns ev_num st order_pid = pid ->
      ev_num := order_ev_num
    ni
  od
end
```

Figure 9: Basic replaying controller

```
process controller
  var ev_num := 0
  var frec := open(``file for writing sequence of events'')
  do true ->
    ev_num++
    in r_invoke(pid) returns evn ->
      evn := r_event(myvid, ev_num)
      write(frec, pid.vid, pid.ev, r_NA.vid, r_NA.ev, "invoked")
    [] r_serviced(pid, evn) ->
      write(frec, pid.vid, pid.ev, evn.vid, evn.ev, "serviced")
    [] r_gcreated(name) returns pid ->
      write(frec, r_GLOB.vid, r_GLOB.ev, myvid, ev_num, name)
      pid := r_event(myvid, ev_num)
    [] r_vcreate(pid) returns vmid ->
      # set vmid.myvid to the vm number for new vm
      write(frec, pid.vid, pid.ev, vmid.myvid, r_VM, "create-vm")
    ni
  od
end
```

Figure 10: Recording Controller

```
process controller
  var ev := 0
  var frec := open(``file for reading sequence of events'')
  var order_pid, order_ev_num: r_event; var order_name: string[100]
  do read(frec, order_pid.vid, order_pid.ev,
              order_ev_num.vid, order_ev_num.ev, order_name) = 5 ->
    ev++
    in r_invoke(pid) returns ev_num
       st order_pid.vid = pid.vid and order_pid.ev = pid.ev ->
      ev_num := r_event(myvid,ev)
    [] r_service(pid) returns ev_num
       st order_pid.vid = pid.vid and order_pid.ev = pid.ev ->
      ev_num := order_ev_num
    [] r_gcreate(name) returns pid
       st order_pid.ev = r_GLOB.ev & order_name = name ->
      pid := r_event(myvid, order_ev_num.ev)
    [] r_vcreate(pid) returns vmid
       st order_pid.vid = pid.vid and order_pid.ev = pid.ev
          and order_ev_num.ev = r_VM ->
      # return vm number for new vm
      vmid.myvid := order_ev_num.vid
    ni
  od
end
```

Figure 11: Replaying Controller