

## 5.5 Tracing, Testing, and Replay for Message Passing Programs

Outline:

- SYN-sequences for channel-based programs
- Tracing, replaying, and checking the feasibility of SYN-sequences.
- Reachability testing for programs that use the *channel* classes.

### 5.5.1 SR-sequences

Object-based SYN-sequence: there is one SYN-sequence for each synchronization object in the program.

Thread-based SYN-sequence: one SYN-sequence for each thread.

#### 5.5.1.1 Object-Based SR-sequences.

Let CP be a concurrent program that uses *channels*. Assume for now that CP has no selective wait statements.

The synchronization objects in CP are its channels and the synchronization events in CP are executions of *send()* and *receive()* operations on these channels.

A SYN-sequence for a channel is a sequence of send and receive events (or “SR-events”) of the following types:

- SRsynchronization: a synchronization between send and receive operations on a synchronous channel
- asynchronousSendArrival: an arrival of an asynchronous send operation whose message is eventually received
- asynchronousReceive: a receive operation that eventually receives a message on an asynchronous channel

- unacceptedSend: a synchronous send operation whose message is never received
- unacceptedAsynchSend: an asynchronous send operation whose message is never received
- unacceptedReceive: a receive on an asynchronous or synchronous channel that never receives a message
- sendException: a send operation on an asynchronous or synchronous channel that causes an exception to be thrown
- receiveException: a receive operation on an asynchronous or synchronous channel that causes an exception to be thrown
  - startEntry: the start of a rendezvous on an entry
  - endEntry: the end of a rendezvous on an entry

Notice that for asynchronous messages, there are *arrival* events but no *send* events. This is because it is the order in which messages arrive that determines the result of an execution, not the order in which messages are sent.

An object-based SR-event for channel C is denoted by

$(S, R, N_S, N_R, \text{eventType})$

where:

- S is the sending thread of a send operation or the calling thread of an entry call.
- R is the receiving thread of a receive operation or the accepting thread of an entry call.
- $N_S$  is the sender’s order number, which gives the relative order of this event among all of the sending/calling thread’s events.
- $N_R$  is the receiver’s order number, which gives the relative order of this event among all of the receiving/accepting thread’s events.
- eventType is one of the send-receive event types, which are listed above.

In addition:

- The order number of an event executed by thread  $T$  gives the relative order of this event among all the events exercised by  $T$ . For example, an event  $T$  with order number 2 is the second event exercised by  $T$ .
- For *asynchronousSendArrival*, *unacceptedSend* and *sendException* events,  $R$  and  $N_R$  are not applicable (NA) since no receiving thread is involved with these events.
- For *unacceptedReceive* and *receiveException* events,  $S$  and  $N_S$  are not applicable since there is no sending thread.
- The order number of an *asynchronousSendArrival* event is for the send operation of the thread that generated it.

The program in Listing 5.15 shows why order numbers are needed in SR-events.

```
mailbox C1, C2; // synchronous mailboxes
  Thread1   Thread2   Thread3
  C1.send(); C1.receive(); C2.receive();
  C2.send();
```

Listing 5.15 Demonstrating the need for order numbers.

Assume that order numbers are not specified.

- The only feasible SR-sequence for channel  $C1$  (without order numbers) contains the single event: (Thread1, Thread2, SRsynchronization).
- The only feasible SR-sequence for  $C2$  also contains a single event:  
(Thread1, Thread3, SRsynchronization).

Suppose we reverse the order of the send operations in *Thread1*:

```
Thread1   Thread2   Thread3
C2.send(); C1.receive(); C2.receive();
C1.send();
```

Then the feasible SR-sequences (without order numbers) for channels  $C1$  and  $C2$  in the new program are the same as in the original program, even though the two programs are different in an obvious and important way.

Adding order numbers to the SR-events distinguishes between these two different programs. The SR-sequence for  $C1$  in the first program becomes

```
// 1st event of Thread1 and 1st event of Thread2
(Thread1, Thread2, 1, 1, SRsynchronization).
```

and the SR-sequence for  $C1$  in the second program becomes

```
// 2nd event of Thread1 and 1st event of Thread2
(Thread1, Thread2, 2, 1, SRsynchronization).
```

When order numbers are specified, an SR-sequence of  $C1$  that is feasible for the first program will be infeasible for the second, and vice versa.

Given the above format for an object-based SR-event, an SR-sequence of channel  $C$  is denoted as

$C: ((S_1, R_1, N_{S1}, N_{R1}, eventType_1), (S_2, R_2, N_{S2}, N_{R2}, eventType_2), \dots)$

where  $(S_i, R_i, N_{S_i}, N_{R_i}, eventType_i)$  denotes the  $i$ th,  $i > 0$ , SR-event in the SR-sequence of  $C$ .

An object-based SR-sequence of a program contains an SR-sequence for each channel in the program.

### 5.5.1.2 Thread-Based SR-sequences.

Each thread in the program has its own SR-sequence.

There are two possible formats for the SR-events in a thread-based SR-sequence.

*Format 1:* A thread-based SR-event for thread  $T$  is denoted by

$(C, N_C, \text{eventType})$

where:

- $C$  is the channel name.
- $N_C$  is the channel order number.
- $\text{eventType}$  is the type of the event.

The channel order number  $N_C$  gives the relative order of this event among all the events involving channel  $C$ .

The list of event types is the same as that above for object-based events, except that we remove type *SRsynchronization* and add separate event types for synchronous send and receive operations:

- send: a synchronous send operation executed by thread  $T$  where the message is eventually accepted
- receive: a synchronous receive operation executed by thread  $T$  that eventually receives a message

(Using separate synchronous-send and synchronous-recv events for the events in object-based sequences is also possible, but combining them into a single send-recv synchronization event makes object-based sequences easier to read.)

The second format for thread-based SR-sequences is used for programs that have selective wait statements.

Format 2 below has the same amount of information as Format 1, but the two formats are best suited for different tasks:

- Format 2 better matches the solution that we have developed for determining feasibility and replaying SR-sequences, while
- Format 1 is a better format for visualizing the trace of an execution.

In Format 2, each channel has one thread that is the “owner” of the channel.

- For a *link*, *port*, or *entry*, (or actually their “selectable” versions) the owner is the thread that executes *receive()* operations on the channel.
- For these types of channels, there is only one thread that can be the receiver.
- A *mailbox* can have multiple receiving threads, so one of the receivers must be arbitrarily chosen to be the owner.

If a thread  $T$  contains a selective wait:

- the selective wait will choose among one or more channels, all of which we require to be owned by thread  $T$ .
- the channels of a selective wait in  $T$  must be *links*, *ports*, or *entries* that are owned by  $T$ . This means that *mailboxes* cannot be used in selective waits.

Format 2: A thread-based SR-event for thread  $T$ , where  $T$  is the owner of the channel associated with the event, is denoted by

$(S, N_S, C, \text{eventType})$

where:

- $S$  is the sending thread ( $T$  is always the receiver)
- $N_S$  is the sender’s order number
- $C$  is the channel name
- $\text{eventType}$  is the type of the event.

The list of thread-based SR-event types is the same as that above for object-based events, except that we add an *elseDelay* event type for selective wait statements:

- *elseDelay*: selection of an else or delay alternative.

In addition:

- An *SRsynchronization* event representing a synchronization between send and receive operations on a synchronous channel appears in the SR-sequence for the owning thread (i.e. the receiving thread). There is no corresponding send event in the sequence for the sending thread.
- In Format 2, no order number is required for channel *C* since the SR-sequence of the owning thread *T* contains all the events for channel *C* and thus the order numbers for events on *C* are given implicitly (i.e., the *i*th event for *C* implicitly has order number *i*.)

For Format 1, an SR-sequence of thread *T* is denoted as:  $T: ((C_1, V_{C_1}, \text{eventType}_1), (C_2, V_{C_2}, \text{eventType}_2), \dots)$ .

For Format 2, an SR-sequence of thread *T* is denoted as:  $T: (S_1, N_{S_1}, C_1, \text{eventType}_1), (S_2, N_{S_2}, C_2, \text{eventType}_2), \dots)$ .

A thread-based SR-sequence of a program contains a thread-based SR-sequence for each thread in the program.

We can easily translate a thread-based SR-sequence into an object-based SR-sequence, and vice versa.

Example: Consider again the simple program in Listing 5.15.

```
mailbox C1, C2; // synchronous mailboxes
Thread1 Thread2 Thread3
C1.send(); C1.receive(); C2.receive();
C2.send();
```

Listing 5.15 Demonstrating the need for order numbers.

The object-based SR-sequence of this program is

$C_1: (\text{Thread1}, \text{Thread2}, 1, 1, \text{SRsynchronization})$

$C_2: (\text{Thread1}, \text{Thread3}, 2, 1, \text{SRsynchronization})$ .

The corresponding thread-based SR-sequence using Format 1 is:

Thread<sub>1</sub>: (C1, 1, send), (C2, 1, send)

Thread<sub>2</sub>: (C1, 1, receive)

Thread<sub>3</sub>: (C2, 1, receive).

The thread-based SR-sequence using Format 2 is:

Thread1: (empty) // send events do not appear in the SR-sequence of the sender

Thread2: (Thread1, 1, C1, SRsynchronization)

// 1 is Thread1's order number for its first *send* operation

Thread3: (Thread1, 2, C2, SRsynchronization).

// 2 is Thread1's order number for its second *send* operation

Thread1's sequence is empty under Format 2, since Thread1 only executes send events and these send events appear as *SRsynchronization* events in Thread2 and Thread3.

### 5.5.1.3 Totally-Ordered SR-sequences.

The object- and thread-based SR-sequences defined above are all partially-ordered -- there is a separate sequence for each channel or thread in the program.

In a totally-ordered SR-sequence there is only one SR-sequence for the program and this sequence includes events for all the channels and threads.

An SR-event in a totally-ordered SR-sequence is denoted by

(S, R, C, eventType)

where:

- S is the sending thread of a send operation or the calling thread of an entry call.
- R is the receiving thread of a receive operation or the accepting thread of an entry call.
- C is the channel name.
- eventType is one of the types listed above for Format 2.

A totally-ordered SR-sequence contains no order numbers for channels or threads, since this information is specified implicitly by the ordering of events in the sequence.

A totally-ordered SR-sequence of program CP is denoted as: ((S<sub>1</sub>, R<sub>1</sub>, C<sub>1</sub>, eventType<sub>1</sub>), (S<sub>1</sub>, R<sub>2</sub>, C<sub>2</sub>, eventType<sub>2</sub>), ...).

For example, a totally-ordered SR-sequence for the program in Listing 5.15 is

( (Thread<sub>1</sub>, Thread<sub>2</sub>, C<sub>1</sub>, SRsynchronization),  
(Thread<sub>1</sub>, Thread<sub>3</sub>, C<sub>2</sub>, SRsynchronization) ).

A totally-ordered SR-sequence can be translated into a partially-ordered sequence, and vice versa

- A totally-ordered sequence is usually easier to understand than a partially-ordered one.
- If a program is running on a single CPU, a single controller can easily record a totally-ordered sequence.
- However, using a single controller creates a bottleneck during execution.
- Since threads and channels are user-level objects in the program, they can be instrumented to collect either thread-based or object-based sequences.

In a distributed program (which involves multiple machines):

- a channel is typically not a user-level object, so object-based sequences may be difficult to collect.
- If multiple controllers (e.g., one controller per machine) are used to collect partially-ordered SR-sequences, then a bottleneck is avoided, but the trace files will be created on separate machines across the network.
- If a single controller is used to collect a totally-ordered SR-sequence from a distributed program, then certain event-ordering issues must be addressed so that the SR-sequence is accurately captured (see Section 6.3).

### 5.5.2 Simple SR-sequences

The format of an Object-based simple send-receive event depends on the type of channel:

- *Mailbox*: a simple send-receive event for *mailbox*  $M$  is denoted by  $(S, R)$ , where  $S$  is the ID of the sending thread and  $R$  is the ID of receiving thread for mailbox  $M$ .
- *Port/Entry*: a simple send-receive event for *port*  $P$  or *entry*  $E$  is denoted by  $(S)$ , where  $S$  is the ID of the sending thread for *port*  $P$  or the calling thread for *entry*  $E$ . (The receiving thread is not recorded since the receiving thread for a *port* or *entry* is always the same thread, which is the owner of the *port* or *entry*.)
- *Link*: A simple send-receive event for a *link* requires no information about the sender and receiver. This is because the sender and receiver are always the same thread.

Note: If program  $CP$  uses only *links* for communication and no selective wait statements, then there is no non-determinism that needs to be controlled during replay. (This assumes that message passing is the only possible source of non-determinism in  $CP$ .)

To simplify the recording and replaying of simple SR-sequences, we use the common format  $(S, R)$  for all channel types, where  $S$  is the ID of the sending thread and  $R$  is the ID of the receiving thread for the *link*, *port*, *entry*, or *mailbox*.

A simple object-based SR-sequence of channel  $C$  is denoted as

$C: ((S_1, R_1), (S_2, R_2) \dots)$ .

An object-based simple SR-sequence of a program contains a simple SR-sequence for each channel in the program.

A thread-based simple SR-event has two possible formats, corresponding to Formats 1 and 2 defined above:

Format 1:

- A simple SR-event is denoted as  $(N_C)$ , where  $N_C$  is the order number of channel  $C$ , which is the *mailbox*, *port*, *entry* or *link* that is involved in this event.
- A simple SR-sequence of a thread  $T$  using Format 1 is denoted as:  $T: ((N_{C_1}), (N_{C_2}), \dots)$

Format 2:

- A simple SR-event is denoted as  $(S)$ , where  $S$  is the sending thread for this event.
- A simple SR-sequence of a thread  $T$  using Format 2 is denoted as:  $T: ((S_1), (S_2), \dots)$

A totally-ordered simple SR-sequence of program  $CP$  is denoted as

$((S_1, R_1) (S_2, R_2), \dots)$

where:

- $S$  is the sending thread of a send operation or the calling thread of an entry call.
- $R$  is the receiving thread of a receive operation or the accepting thread of an entry call.

```
mailbox C1, C2; // synchronous mailboxes
Thread1 Thread2 Thread3
C1.send(); C1.receive(); C2.receive();
C2.send();
```

Listing 5.15 Demonstrating the need for order numbers.

For the program in Listing 5.15, the object-based, simple SR-sequence of this program is

```
C1: (Thread1, Thread2)
C2: (Thread2, Thread3)
```

The thread-based, simple SR-sequence using Format 1 is

```
Thread1: (1), (1)
Thread2: (1)
Thread3: (1)
```

Let Thread2 be the owner of channel *C1* and Thread3 be the owner of channel *C2*, then the thread-based simple SR-sequence using Format 2 is

```
Thread2: (Thread1)
Thread3: (Thread1).
```

The thread-based, totally-ordered simple SR-sequence is

```
(Thread1, Thread2), (Thread1, Thread3).
```

### 5.5.3 Determining the Feasibility of an SR-sequence

Before a thread can perform a send or receive operation, it must request permission from a control module. The controller is responsible for reading an SR-sequence and forcing the execution to proceed according to this sequence.

The *send()* and *receive()* methods in the *channel* classes are modified by adding one or more calls to the controller. Listing 5.16 shows a sketch of Java class *link* with the modifications highlighted.

In trace mode:

- Methods *send()* and *receive()* notify the controller of any exceptions that occur.
- Operation *receive()* notifies the controller when a message is received, which causes an *SRsynchronization* event to be recorded. The arguments on the calls refer to the fields of an SR-event, which were defined in the previous section. (sender (S), receiver (R), order number of the sender ( $N_S$ ), order number of the receiver ( $N_R$ ), channel name (C), order number of the channel ( $N_C$ ), and the various event types).

In test mode,

- Methods *send()* and *receive()* issue a call to the controller to request permission to exercise send and receive events.
- The call to *control.msgReceived()* notifies the controller that the event has occurred, so that permission can be given for the next event.

The controller:

- Inputs an SR-sequence and attempts to force the SR-sequence to be exercised.
- If the next event in the SR-sequence indicates that T is expected to request a certain permit, the controller will wait for T to issue an entry call to the appropriate channel.
- If no call arrives before a timeout expires, then the sequence is assumed to be infeasible.

```

public class link extends channel {
...
public final void send(Object sentMsg) {
    if (mode == TEST)
        control.requestSendPermit(S,Ns,C);
    synchronized(sending) {
        // save first Thread to call send
        if (sender == null) sender = Thread.currentThread();
        if (Thread.currentThread() != sender) {
            if (mode == TRACE)
                control.traceMsg(S, Ns, C, Vc, sendException);
            if (mode == TEST) {
                control.requestSendExceptionPermit(S,Ns, C);
                control.msgReceived();
            }
            throw new InvalidLinkUsage
                ("Attempted to use link with multiple senders");
        }
        if (sentMsg == null) {
            if (mode == TRACE)
                control.traceMsg(S, Ns, C, Vc, sendException);
            if (mode == TEST) {
                control.requestSendExceptionPermit(S, Ns, C);
                control.msgReceived();
            }
            throw new NullPointerException("Null message passed to send()");
        }
        message = sentMsg;
        sent.V();
        received.P();
    }
}
}

```

Listing 5.16 Class *link* modified for tracing and feasibility.

```

public final Object receive() {
    Object receivedMessage = null;
    if (mode == TEST) control.requestReceivePermit(R,Nr,C);
    synchronized (receiving) {
        // save first Thread to call receive
        if (receiver == null) receiver = Thread.currentThread();
        f (Thread.currentThread() != receiver) {
            if (mode == TRACE)
                control.traceMsg(R,Nr,C,Vc,receiveException);
            if (mode == TEST) {
                control.requestReceiveExceptionPermit(R, Nr, C);
                control.msgReceived();
            }
            throw new InvalidLinkUsage
                ("Attempted to use link with multiple receivers");
        }
        sent.P();
        receivedMessage = message;
        if (mode == TRACE) control.traceMsg
            (S, R, Ns, Nr, C, Vc, SR_SYNCHRONIZATION);
        else if (mode == TEST) control.msgReceived();
        received.V();
        return receivedMessage;
    }
}
}

```

Listing 5.16 (continued) Class *link* modified for tracing and feasibility.

### 5.5.4 Deterministic Testing

Deterministic testing of a concurrent program CP involves the following steps:

1. Select a set of tests, each of the form  $(X, S)$ , where X and S are an input and a complete SYN-sequence of CP respectively.
2. For each selected test  $(X, S)$ , force a deterministic execution of CP with input X according to S. This forced execution determines whether S is feasible for CP with input X. (Since S is a complete SYN-sequence of CP, the result of such an execution is deterministic.)
3. Compare the expected and actual results (including the output, feasibility of S, and termination condition) of the forced execution. If the expected and actual results are different, a fault is detected. The replay tool can be used to locate the fault.

Note that for deterministic testing, a test for CP is not just an input of CP. A test consists of an input and a SYN-sequence, and is referred to as an IN-SYN test.

The selection of IN-SYN tests for CP can be done in different ways:

- Select inputs and then select a set of SYN-sequences for each input.
- Select SYN-sequences and then select a set of inputs for each SYN-sequence.
  - Select inputs and SYN-sequences separately and then combine them.
  - Select pairs of inputs and SYN-sequences together.

To use an IN-SYN test  $(X, S)$  for deterministic testing of CP, we need to specify the expected output, the expected feasibility of SYN-sequence S (feasible or infeasible), and the expected termination condition (normal or abnormal).

### 5.5.5 Reachability Testing for Message-Passing Programs

Reachability testing can be used to automatically derive and exercise every partially-ordered SR-sequence of a message-passing program.

Reachability testing identifies race conditions in an execution trace and uses the race conditions to generate race variants.

Recall from Chapters 3 and 4 that a race variant represents an alternative execution behavior that could have happened, but didn't, due to the way race conditions were arbitrarily resolved during execution.

Replaying a race variant ensures that a different behavior is observed during the next execution.

Fig. 5.18a shows an execution trace for three threads that use asynchronous ports for message passing. Thread1 and Thread3 each send a single message to Thread2.

Thread2 receives the message from Thread1 first, then Thread3.

Fig 5-18b shows a race variant of this trace. In this variant, Thread2 receives the message from Thread3 first.

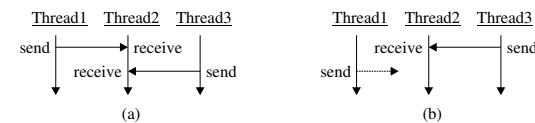


Figure 5.18 Execution trace and a race variant.

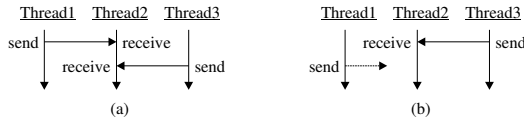


Figure 5.18 Execution trace and a race variant.

Note that the race variant in Fig. 5.18b contains only one receive event for Thread2. We cannot include a second receive event for Thread2 since we cannot be sure that Thread2 will execute another receive statement after the variant is replayed.

Suppose that Thread2 executes the following statements:

```
x = p2.receive();
if (x)
    p2.receive();
```

In the execution trace in Fig. 5.18a, Thread2 executes both receive events. Thus, we can say with certainty that the value  $x$  received from Thread2 caused the condition in Thread2's if-statement to be evaluated to *true*.

In the race variant in Fig. 5.18b:

- We changed the sending thread for Thread2's first receive to Thread3.
- The value Thread2 receives from Thread3 may or may not cause the condition in Thread2's if-statement to be evaluated to *true*.
- Since we generate variants without examining the source code of the program, we can no longer be sure what will happen after Thread2's first receive.
- To be safe, we remove from the variant all events that happen after the changed receive event.

The variants of a trace depend on the type of synchronization provided by the channels.

Fig. 5.19a shows a program that uses asynchronous ports. An execution trace of this program is given in Fig. 5.19b.

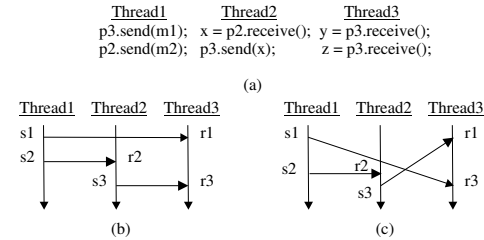


Figure 5.19 Execution trace and a variant for asynchronous ports with FIFO synchronization.

Assume that the ports used in the program of Fig. 5.19a are the asynchronous ports implemented in Section 5.1:

- Thread1's send to Thread3 definitely happens before Thread2's send to Thread3 since Thread1 and Thread2 synchronize in between sending their messages to Thread3.
- Based on this "happened before" relation, the port implementation used in this chapter guarantees that Thread3 will receive the message sent by Thread1 before it receives the message sent by Thread2.

In general: if a message  $m_1$  is sent to thread T before another message  $m_2$  is sent to T, then message  $m_1$  is received by T before  $m_2$  is received.

This type of synchronization is called *causal synchronization*. If causal synchronization is used, no race variants can be generated for the trace in Fig. 5.19b.

Formal definitions of the "happens before" relation and causality are given in Chapter 6.

Causal synchronization is not always guaranteed. *FIFO synchronization* is commonly used in distributed programs, which send message across a communications network.

FIFO synchronization guarantees that messages sent from one thread to another thread are received in the order that they are sent.

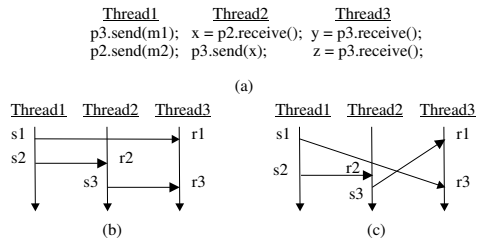


Figure 5.19 Execution trace and a variant for asynchronous ports with FIFO synchronization.

Fig. 5.19c shows a variant of the trace in Fig. 5.19b.

- This variant assumes that FIFO synchronization is used, which allows Thread3 to receive messages from Thread1 and Thread2 in the reverse order that they were sent.
- FIFO synchronization guarantees that Thread3 receives Thread1's messages in the order that Thread1 sends them, and that Thread3 receives Thread2's messages in the order that Thread2 sends them, but the first message Thread3 receives can be from Thread1 or from Thread2.

Distributed programming is discussed in Chapter 6. To prepare for Chapter 6, we have implemented a special asynchronous channel port with FIFO synchronization and support for reachability testing.

- These FIFO ports make it possible to write programs that behave like distributed programs but run on a single computer and so are easier to write and easier to debug
- Writing such a program is a good first step towards writing a real distributed program.

Table 5.1 shows the results of applying reachability testing to programs that use message passing with selective waits.

- Program BB is the bounded buffer program in Listing 5.10.
- Program RW is the readers and writers program in Listing 5.11.

Program	Config.	#Seqs.	Program	Config.	#Seqs.
BB-Select	3P+3C+2S	144	RW-Select	3R+2W	768

Table 5-1 Reachability testing results for message-passing programs

These programs, which use selective waits, generate far fewer sequences than the corresponding bounded buffer and reader and writer programs in Chapters 3 and 4 that used semaphores and monitors.

## 5.5.6 Putting it all Together

### 5.5.6.1 Using the Java *channel* Classes.

An execution is traced by setting the mode to *trace*:

```
-Dmode=trace
```

and then specifying whether the tracing strategy should be object-based or thread-based:

```
-Dstrategy=object // collect object-based SR-sequences
```

```
-Dstrategy=thread // collect thread-based SR-sequences
```

and whether there should be a single Controller or multiple Controllers:

```
-Dcontrollers=single // single Controller thread
```

```
-Dcontrollers=multiple // multiple Controller threads
```

Using a single Controller results in a totally-ordered (object-based or thread-based) SR-sequence.

If multiple Controllers are used, the collected SR-sequence will be partially-ordered (one per thread or object).

If the program contains a *selectiveWait*, then the tracing strategy must be thread-based and mailboxes cannot be used in the *selectiveWait*.

When a thread-based strategy is used, the owner of each *link*, *port*, *mailbox*, or *entry* P must be indicated by calling `P.setOwner(T)`, where thread T is the owner of port P. An example of this is shown in the *main()* method in listing 5.20.

```
public final class Buffer {
    public static void main (String args[]) {
        selectableEntry deposit = new selectableEntry();
        selectableEntry withdraw = new selectableEntry();
        boundedBuffer b = new boundedBuffer(deposit,withdraw,3);
        Producer pl = new Producer (deposit, 1);
        Consumer cl = new Consumer (withdraw, 1);
        deposit.setOwner(b); withdraw.setOwner(b);
        b.setDaemon(true);
        b.start(); cl.start(); pl.start();
        try {cl.join(); pl.join();}
        catch (InterruptedException e) {System.exit(1);}
    }
}
```

Part of Figure 5.20.

The command:

```
java -Dmode=trace -Dstrategy=thread -Dcontrollers=single Buffer
```

creates several trace files for program *Buffer*:

- channel-replay.txt contains a totally-ordered simple SR-sequence of the execution
- file channel-test.txt, contains a totally-ordered complete SR-sequence.

The command

```
java -Dmode=trace -Dstrategy=thread -Dcontrollers=multiple Buffer
```

creates files that contain a separate simple SR-sequence and complete SR-sequence for each of the threads.

Random delays are executed during trace mode by setting `-DrandomDelay=on`. When *on* is specified, random delays are executed before each *send()* and *receive()*. To turn random delays off, use *off*; this is also the default value. The value of property *randomDelay* is ignored during replay and test modes.

To turn deadlock detection on during tracing, specify `-DdeadlockDetection=on`. To turn deadlock detection off, use *off*; this is also the default value.

A simple SR-sequence is replayed by specifying *replay* mode and using the same values for the *strategy* and *controllers* that were used during tracing:

```
java -Dmode=replay -Dstrategy=thread -Dcontrollers=single Buffer
```

As in Chapters 3 and 4, reachability testing is performed on *Buffer* by setting the *mode* property to *rt* and executing a driver process named *RTDriver* with “Buffer” as the command-line argument:

```
java -Dmode=rt RTDriver Buffer
```

The feasibility of an SR-sequence is determined in *test* mode:

```
java -Dmode=test -Dstrategy=thread -Dcontrollers=single Buffer
```

If the mode is not specified, the default value for the *mode* property will turn tracing, replay, and testing off.

The default value for the strategy property is *object* and the default value for the *controllers* property is *single*.

### 5.5.6.2 Using the C++ *channel* Classes.

In Windows/DOS, an execution of a C++ version of program *Buffer* in Listing 5.20 is traced by executing the command:

```
set MODE=TRACE // Unix: setenv MODE TRACE
```

and then specifying whether the tracing strategy should be object-based or thread-based:

```
set STRATEGY=OBJECT // collect object-based SR-sequences; Unix: setenv  
STRATEGY OBJECT
```

```
set STRATEGY=THREAD // collect thread-based SR-sequences; Unix: setenv  
STRATEGY THREAD
```

and whether there should be a single Controller or multiple Controllers:

```
set CONTROLLERS=SINGLE // single Controller thread; Unix: setenv  
CONTROLLERS SINGLE
```

```
set CONTROLLERS=MULTIPLE // multiple Controller threads; Unix: setenv  
CONTROLLERS MULTIPLE
```

Using a single Controller results in a totally-ordered (object-based or thread-based) SR-sequence.

If multiple Controllers are used, the collected SR-sequence will be partially-ordered (one per thread or object).

If the program contains a *selectiveWait*, then the tracing strategy must be thread-based.

When a thread-based strategy is used, the owner of each link, port, or entry P must be indicated by calling P.setOwner(T), where thread T is the owner of port P. An example of this is shown in the *main()* method in listing 5.20.

```

public final class Buffer {
    public static void main (String args[]) {
        selectableEntry deposit = new selectableEntry();
        selectableEntry withdraw = new selectableEntry();
        boundedBuffer b = new boundedBuffer(deposit,withdraw,3);
        Producer pl = new Producer (deposit, 1);
        Consumer cl = new Consumer (withdraw, 1);
        deposit.setOwner(b); withdraw.setOwner(b);
        b.setDaemon(true);
        b.start(); cl.start(); pl.start();
        try {cl.join(); pl.join();}
        catch (InterruptedException e) {System.exit(1);}
    }
}

```

Part of Figure 5.20.

Tracing an execution with a THREAD strategy and a SINGLE controller creates several trace files for *Buffer*:

- file channel-replay.txt contains a totally-ordered simple SR-sequence of the execution
- file channel-test.txt, contains a totally-ordered complete SR-sequence.

Using MULTIPLE controllers creates files that contain a separate simple SR-sequence and complete SR-sequence for each of the threads.

Random delays can be executed during trace mode using “set RANDOMDELAY=ON.” When ON is specified, random delays are executed before each *send()* and *receive()*. To turn random delays off, use OFF; this is also the default value. The value of variable RANDOMDELAY is ignored during replay and test modes.

To turn deadlock detection on during tracing or reachability testing, execute” set DEADLOCKDETECTION=ON.” To turn deadlock detection off, use OFF; this is also the default value.

A simple SR-sequence is replayed by executing “set MODE=REPLAY”, and using the same values for the STRATEGY and the CONTROLLERS variables that were used during tracing.

As in Chapter 3, reachability testing is performed on *Buffer* by setting the MODE to RT and customizing the driver process in file *RTDriver.cpp*. This file is part of the synchronization library. Directions for customizing the driver process are in the file.

The feasibility of an SR-sequence is determined in TEST mode: set MODE TEST. If the mode is not specified, the default value for the MODE variable will turn tracing, replay, and testing off.

The default value for the STRATEGY variable is OBJECT and the default value for the CONTROLLERS variable is SINGLE.