4.10 Tracing and Replay for Monitors

Outline:

- simple M-sequences of monitor-based programs
- tracing and replaying simple M-sequences during debugging
- modifying the SC and SU monitor toolboxes to support tracing and replay
- complete M-sequences and a technique for using complete M-sequences to test monitor-based programs
- applying reachability testing to monitor-based programs

4.10.1 Simple M-sequences

Let M be a monitor that is implemented using one of the monitor toolboxes.

An execution of a program that uses M can be viewed as a sequence of *P()* and *V()* operations on the semaphores in the implementation of M:

- a simple SYN-sequence for M is the collection of simple PV-sequences for the semaphores in the implementation of M.
- we can replay an execution of a program that uses M by applying the replay method that was presented in Chapter 3 for replaying PV-sequences of semaphore-based programs.

A simpler method is based on the property of *entry-based execution*:

- The execution of threads inside an SU monitor is completely determined by the order in which the threads enter the monitor via monitor calls, and the values of the parameters on these calls.
- The execution of threads inside an SC monitor is completely determined by the order in which the threads enter the monitor via monitor calls, the values of the parameters on these calls, and the order in which signaled threads reenter the monitor.

The execution of threads in a monitor having entry-based execution is completely determined by the order in which the threads (re)enter the monitor and the values of the parameters on the calls to the monitor.

Let CP be a concurrent program that uses monitors. The synchronization objects in CP are its monitors. The synchronization events in a simple SYN-sequence of CP depend on the type of monitors being used:

SC monitors: A simple SYN-sequence for an SC monitor is a sequence of events of the following types:

- entry into the monitor by a calling thread
- reentry into the monitor by a signaled thread

SU monitors: A simple SYN-sequence for an SU monitor is a sequence of events of the following type:

- entry into the monitor by a calling thread

Simple SYN-sequences of monitors are called simple M-sequences:

- An event in a simple M-sequence is denoted by the identifier (ID) of the thread that executed the event.
- The simple M-sequence of a program CP is a collection of simple M-sequences for the monitors in CP.

We can replay an execution of CP by replaying the simple M-sequence for each monitor.

Consider the SC bounded buffer monitor in Listing 4.3. A possible simple M-sequence of *boundedBuffer* for a single *Producer* thread (with ID 1) and a single *Consumer* thread (with ID 2) is: (2, 1, 2, 1, 1, 2, 2).

This sequence is generated when:

- The *Consumer* enters the monitor first and executes *notEmpty.wait()*.
- The *Producer* then enters the monitor, deposits an item, and signals the *Consumer*.
- The *Consumer* then reenters the monitor and consumes an item.

Note: This sequence begins with 2, 1, 2 since the *Consumer* generates one event when it enters the monitor (the first 2) and one event when it reenters the monitor (the second 2).

If the same scenario were to occur when *boundedBuffer* was an SU monitor, then the simple M-sequence of *boundedBuffer* would be: (2, 1, 1, 1, 2, 2).

In this sequence, the *Consumer* does not generate an event when it reenters the monitor, since *boundedBuffer* is an SU monitor. Thus, the sequence begins with 2, 1 and is followed by a second entry event for the *Producer* (the second 1) and not a reentry event for the *Consumer*.

We are making several important assumptions in our definition of simple M-sequences:

1. All shared variables are accessed inside a monitor and monitor (re)entry is the only source of non-determinism in the program.
2. Semaphores with FCFS notifications are used in the implementation of condition variables. If notifications wake up waiting threads in an unpredictable order, then non-determinism is introduced in the implementation of the monitor.
3. *VP*() operations are used in the semaphore implementation of a monitor. Without *VP*() or some other mechanism, context switches can occur between *V()* and *P()* operations, which introduces non-determinism in the implementation of the monitor.

Assumptions 2 and 3 are necessary for entry-based execution. If assumption 1 does not hold, then replaying the simple M-sequences of an execution may not replay the execution.

```
class boundedBuffer extends monitor {
    private  int fullSlots = 0;  // number of full slots in the buffer
    private int capacity = 0;   // capacity of the buffer
    private int[] buffer = null;    // circular buffer of ints
    private int in = 0, out = 0;
    private  conditionVariable notFull = new conditionVariable();
    private  conditionVariable notEmpty = new conditionVariable();

   public boundedBuffer(int bufferCapacity ) {
      capacity = bufferCapacity;buffer = new int[bufferCapacity];}
      public void deposit(int value) {
      while (fullSlots == capacity)
          notFull.wait();
      buffer[in] = value;
      in = (in + 1) % capacity;   ++fullSlots;
      notEmpty.signal(); //alternatively:if (fullSlots == 1) notEmpty.signal();
   }
   public int withdraw() {
      int value;
      while (fullSlots == 0)
          notEmpty.wait();
      value = buffer[out];
      out = (out + 1) % capacity; --fullSlots;
      notFull.signal(); //alternatively:if (fullSlots == capacity–1) notFull.signal();
      return value;
   }
}
```

Listing 4.3 Monitor class *boundedBuffer*.

4.10.2 Tracing and Replaying Simple M-sequences

SU toolbox: semaphore *mutex* controls entry into the monitor. During execution, the identifier of a thread that completes a *mutex.P()* operation is saved to a trace file.

SC toolbox: semaphore *mutex* controls entry into the monitor for threads calling the monitor, and reentry into the monitor for signaled threads. The identifier of the thread that completes a *mutex.P()* operation is recorded and saved to a trace file.

To replay a simple M-sequence:

- Before each *mutex.P()* operation, a thread calls *control.requestEntryPermit(ID)* to request permission to enter the monitor.
- After completing *mutex.P()*, the thread calls *control.releaseEntryPermit*() to allow the next thread in the simple M-sequence to enter the monitor.

Java monitor class *monitorTracingAndReplay* in Listing 4.28 is very similar to the control object in Listing 3.32 that was used for replaying simple PV-sequences.

Method *requestEntryPermit()*: Threads that try to enter the monitor out of turn are delayed in method *requestEntryPermit()* on a separate *conditionVariable* in the *threads* array.
A thread uses its *ID* to determine which *conditionVariable* in the *threads* array to wait on.

Method *releaseEntryPermit()*: increments *index* to allow the next (re)entry operation in the *simpleMSequence* to occur. If the thread that is to perform the next entry is blocked in *requestEntryPermit()*, it is awakened.

```
class monitorTracingAndReplay extends monitorSU {
   monitorTracingAndReplay () {
   /* input sequence of Integer IDs into vector simpleMSequence
       and initialize the threads array */
   }
   public requestEntryPermit(int ID) {
      enterMonitor();
      if (ID != ((Integer)simpleMSequence.elementAt(index)).intValue())
         threads[ID].waitC();    // thread with identifier ID is delayed on
      exitMonitor();         //   condition variable threads[ID]
   }
   public releaseEntryPermit() {
      enterMonitor();
      if (index < simpleMSequence.size()-1) {
         ++index;
         // if the next thread in the simpleMSequence was delayed in
         // requestEntryPermit(), wake it up.
         threads[((Integer)simpleMSequence.elementAt(index)).intValue()].
            signalC_and_exitMonitor();
         return;
      }
      exitMonitor();
   }
   // record integer ID of entering thread in trace file
   public void traceMonitorEntry(int ID) { … }
   // record integer ID of reentering thread in trace file
   public void traceMonitorReEntry(int ID) { … }
   // simple M-sequence traced during a previous execution.
   private vector simpleMSequence;
   // out of order threads are delayed on condition variables.
   private conditionVariable[] threads;
   // index of the next event in simpleMSequence to be replayed.
   private int index = 1;
}
```

Listing 4.28 Java monitor *monitorTracingAndReplay* for tracing and replaying simple M-sequences.

4.10.3 Other Approaches to Program Replay

Another approach to replaying an execution of a multithreaded Java program is to trace and replay the scheduling events of the operating system.

A thread schedule is a sequence of time slices, where a time slice is the interval of time between two context switches. A thread schedule of an execution can be replayed by forcing context switches to occur at the exact same points during the execution.

Since detailed scheduling information is not always available from the operating system, a logical thread schedule can be used instead.

A logical thread schedule of an execution is a record of the number of critical operations that the threads executed during their time slices. Critical operations are:

- read and write operations on shared variables,
- entering and exiting synchronized methods and blocks,
- wait and notify operations.

A logical thread schedule can be replayed my modifying the Java Virtual Machine (JVM) to count critical operations.

For example, assume Thread 1 executes fifty critical operations during its time quantum. This can be represented by

(1, 20, 69) // Thread 1 executed the (fifty) operations numbered 20 through 69.

The DejaVu system uses logical thread schedules to replay multithreaded Java programs. Java applications can be traced and replayed in DejaVu without any modifications to the source code.

4.11 Testing Monitor-Based Programs

Outline:

- complete M-sequences
- determining whether a particular complete M-sequence is allowed by a program.

4.11.1 M-sequences

Debugging process:

- Observe a failure
- Replay the simple M-sequence of the execution, perhaps several times, to locate the fault that caused the failure.
- Modify the program to fix the bug
- Test the modification to make sure that the fault has been corrected and that no new faults were introduced by the modification.

The last step is commonly known as “regression testing”.

Regression testing requires that we determine whether or not a particular SYN-sequence is feasible:

- A feasible SYN-sequence is a sequence that can be exercised by the program.
- The SYN-sequence may represent an illegal (or “invalid”) behavior that was observed when the program failed.
- A SYN-sequence may represent a legal (or “valid”) behavior that was exhibited previously, then this valid sequence should remain feasible after the modifications.

We may also want to use test sequences to detect failures.

- If a test sequence is expected to be feasible but is not feasible, or is expected to be infeasible but is not infeasible, then we can start the debugging process.
- Selecting sequences and checking their feasibility is part of a general testing technique called *deterministic testing*.

Determining the feasibility of a SYN-sequence is a problem that is subtly different from the replay problem:

- Replaying a SYN-sequence involves repeating a sequence that is known to be feasible (Assuming the sequence was just exercised and the program has not been modified.)
- During regression testing, we want to know whether or not a program that is executed with a given input can exercise a particular sequence and produce a particular output.

For regression testing, we need more information than is contained in a simple M-sequence.

Consider the following simple M-sequence for a 2-slot bounded buffer with a single *Producer* (with ID 1) and a single *Consumer* (with ID 2): (1, 1, 1, 2, 2).

This gives the order in which threads entered the monitor, but which methods did they call? During testing, we must check that threads call the expected monitor methods.

Suppose now that we also specify the name of the method: (1,deposit),(1,deposit), (1,deposit), (2,withdraw),(2,withdraw)):

- If the third item was deposited into a full buffer, then the first item was lost.
- If the *Producer* executed *notFull.wait()* then a third deposit was not made until after a withdrawl, and a failure was avoided.

We cannot determine from the information in this sequence what happened.

Even after checking the output, we may still be unsure about what happened. If the items deposited were 'C', 'A', and 'C', then the items withdrawn are also 'C', 'A', and 'C', even if the first item is overwritten by the third item.

To more completely characterize the execution of a monitor-based program, we will consider the following types of monitor events:

(a) the entry of a monitor method and, for SC monitors, the reentry of a monitor method
(b) the exit of a monitor method
(c) the start of execution of a *wait* operation
(d) the start of execution of a *signal*, or *signalAndExit*, or *signalAll* operation.

What about other events such as the end of execution of a *wait* or *signal* operation, or the resumption of a monitor method after a *wait* or an SU *signal* operation?

- Their occurrence can be inferred by the occurrence of events of types (a) through (d).
- It may still be desirable to trace these events to aid in understanding executions.

A sequence of events of types (a) through (d) is called a complete M-sequence, as opposed to "simple M-sequences" which are used for replay.

The events of an M-sequence have the format:(Type,Thread,Method,ConditionVariable)

- Type: the type of this event
- Thread: the ID of the thread executing this event
- Method: the monitor method of this event, qualified with the monitor name if there are multiple monitors
- ConditionVariable: the name of the condition variable if this event is a *wait*, *signal*, *signalAndExit*, or *signalAll* event; and "NA" otherwise.

Example: a possible M-sequence of the SC bounded buffer monitor in Listing 4.3 is:

```
((enter,    consumer,   withdraw,  NA  ),
 (wait,     consumer,   withdraw,  notEmpty   ),
 (enter,    producer,   deposit,   NA  ),
 (signal,   producer,   deposit,   notEmpty ),
 (exit,     producer,   deposit,   NA  ),
 (reenter,  consumer,   withdraw,  NA  ),
 (signal,   consumer,   withdraw,  notFull ),
 (exit,     consumer,   withdraw,  NA  )).
```

Another way to make an M-sequence clearer is to allow user-level events to appear in the sequence. These events label the (atomic) communication actions that occur inside the monitor.

A call to method *exerciseEvent("event")* generates a communication event labeled "event" and records it in the M-sequence.

For example, the following SC bounded buffer monitor has calls to *exerciseEvent()* to generate communication events labeled "deposit" and "withdraw":

```
public void deposit(int value) {
    enterMonitor();
    while (fullSlots == capacity)
        notFull.waitC();
    buffer[in] = value;
    exerciseEvent("deposit");   // generate a "deposit" event
    in = (in + 1) % capacity; ++fullSlots;
    notEmpty.signalC();
    exitMonitor();
}
public int withdraw() {
    enterMonitor();
    int value;
    while (fullSlots == 0)
        notEmpty.waitC();
    value = buffer[out];
    exerciseEvent("withdraw"); // generate a "withdraw" event
    out = (out + 1) % capacity; --fullSlots;
    notFull.signalC();
    exitMonitor();
    return value;
}
```

The M-sequence shown above with communication events "deposit" and "withdraw" added is:

```
((enter,    consumer,   withdraw,    NA        ),
(wait,      consumer,   withdraw,    notEmpty  ),
(enter,     producer,   deposit,     NA        ),
(comm,      producer,   deposit,     NA        ),  // communication event
(signal,    producer,   deposit,     notEmpty  ),
(exit,      producer,   deposit,     NA        ),
(reenter,   consumer,   withdraw,    NA        ),
(comm,      consumer,   withdraw,    NA        ),  // communication event
(signal,    consumer,   withdraw,    notFull   ),
(exit,      consumer,   withdraw,    NA        )).
```

For each communication event, we record its label and the ID of the thread that executed the event.

A sequence of communication events is useful for understanding *what* happened, i.e., the communication that occurred among the threads, without any details about *how* the threads were synchronized to achieve that communication:

```
(comm,    producer,   deposit,   NA  ),
(comm,    consumer,   withdraw,  NA  ).
```

When we modify a program to correct a fault, we want an answer to the following question:

> If we execute the program with the same the input that detected the fault, can the same path be exercised and the same output be produced?

An M-sequence partially identifies the path exercised by a program and helps us answer this question.

Consider the following monitor method:

```
public void F() {
   enterMonitor();
   if (…) {
      condition.waitC();
   }
   if (… ) {/* true-part*/  }
   else {/* false-part */ }
   condition.signalC();
   exitMonitor();
}
```

An M-sequence for an execution of a program containing method *F()* indicates the order in which threads entered *F()* and whether or not the *wait()* operation in *F()* was executed.

- This tells us a lot about the path through *F()* that each thread executed.
- But, an M-sequence, by itself, does not indicate whether the threads executed the true-part or the false-part of the second if-statement in *F()*.
- So a path consists of more than just an M-sequence, and more than just a sequence of statements executed by the thread(s).

Note: The output is likely to indicate whether the true-part or the false-part was executed. Thus, the M-sequence and the output of an execution, together, describe the execution very well and together are often are enough to determine if a failure has occurred

A formal definition of a "path" for a concurrent program is given in Chapter 7.

4.11.2 Determining the Feasibility of an M-sequence

The feasibility of an M-sequence is determined using the same technique that was used for replay:

- Before a thread can perform a monitor operation, it requests permission from a control module.
- The control module is responsible for reading an M-sequence and forcing the execution to proceed according to this sequence.
- If the M-sequence is determined to be infeasible, the control module displays a message and terminates the program.

4.11.2.1 Modifying the monitor operations. Each monitor operation is modified by adding one or more calls to the controller. Below, we describe the modifications made to the SC monitor operations.

Method *enterMonitor*():

```
public final void enterMonitor(String methodName) {
   control.requestMPermit(ENTRY,threadID,methodName,”NA”);
   mutex.P();
   control.releaseMPermit();
}
```

For the method name to be available inside the toolbox, it must be passed by the user as an argument to *enterMonitor()*

```
public void deposit(char value) {
   enterMonitor(”deposit”);
   ...
}
```

Method *waitC*():

```
public void waitC() {
  // wait your turn  to enter
  control.requestMPermit(WAIT,threadID,methodName,conditionName);
  numWaitingThreads++;
  threadQueue.VP(mutex);
  // wait your turn to reenter
  control.requestMPermit(REENTRY,threadID,methodName,"NA");
  mutex.P();
  control.releaseMPermit(); // allows next event in the M-sequence to occur
}
```

A *conditionVariable's* name is automatically generated when the *conditionVariable* is constructed. Alternately, a more descriptive name can be specified when the *conditionVariable* is constructed, but each name must be unique:

```
notFull = new conditionVariable("notFull");
```

signalC():

```
public void signalC() {
   control.requestMPermit(SIGNAL,threadID,methodName,conditionName);
   if (numWaitingThreads > 0) {
      numWaitingThreads--;
      threadQueue.V();
   }
}
```

exitMonitor():

```
public void exitMonitor() {
   control.requestMPermit(EXIT,threadID,methodName,"NA");
   mutex.V();
}
```

If a monitor contains calls to *exerciseEvent()*, then Threads executing *exerciseEvent()* must request permission before executing a communication (COMM) event:

```
public void exerciseEvent(String eventLabel) {
      control.requestMPermit(COMM,threadID,eventlabel,"NA");
}
```

4.11.2.2 The Controller. The controller attempts to force a deterministic execution of the program according to the M-sequence that it inputs. The M-sequence is feasible if and only if the M-sequence is completely exercised.

Ideal: the controller would display a message if it finds that an M-sequence definitely cannot be exercised. However, the problem of determining whether a concurrent program terminates for a given input and SYN-sequence is, in general, undecidable.

Practical: the controller specifies a maximum time interval that is allowed between two consecutive events:

- If a timeout occurs, then the sequence is *assumed* to be infeasible.
- It is always possible that some larger timeout value would give the next event enough time to occur and allow the sequence to complete.

The controller is implemented in two parts. The first part inputs an M-sequence and then forces the execution to proceed according to the order of events in the M-sequence.

- This is implemented just like the replay controller.
- The controller also checks that the attributes of each event match the attributes in the M-sequence, e.g., the method names match, the condition variable names match.
- If a mismatch is detected, then a diagnostic is issued and the program is terminated.

At any point in the sequence, if an expected request does not arrive within the timeout interval, a diagnostic is issued and the program is terminated.

This timeout function of the controller is handled by a "watch dog" thread:

- Monitors the value of the variable *index*, which indicates which event is being exercised.
- If the value of *index* does not change within the timeout interval, then the M-sequence is assumed to be infeasible.

```
final class watchDog extends Thread {
  public void run() {
     // index == i indicates that execution has proceeded to the ith event
     while (index < MSequence.size()) {
        int saveIndex = index;
        try { Thread.sleep(2000); } // two second timeout interval
        catch (InterruptedException e) {}
        if (saveIndex == index) {
           /* issue diagnostic and exit program */
        }
     }
  }
}
```

4.11.3 Determining the Feasibility of a Communication-sequence

A Communication-sequence is a sequence of communication events.

A set of Communication-sequences can be generated to test a monitor without knowing the details about the synchronization events in the monitor, e.g., the names of the condition variables and monitor methods.

(comm, producer, **deposit**, NA),
(comm, consumer, **withdraw**, NA),
(comm, producer, **deposit**, NA),
(comm, consumer, **withdraw**, NA).

The feasibility of a Communication-sequence is determined just like an M-sequence, except that only the thread IDs and event labels of communication-events are checked.

Threads that wish to (re)enter a monitor must request permission from the controller first:

- The controller grants permission for $Thread_i$ to enter the monitor only if $Thread_i$ is the thread expected to execute the next communication event in the sequence.
- In other words, the order in which threads execute communication events must match the order in which they (re)enter the monitor.
- If this is not possible, then the Communication-sequence is not feasible.

Example: Consider a *boundedBuffer* monitor with a capacity of three.

- Even if we never see a trace with four consecutive *deposit* operations during non-deterministic testing it is possible that *boundedBuffer* contains a fault that allows a *deposit* into a full buffer.
- Likewise, it is possible that a *withdrawal* is allowed from an empty buffer.

To test these cases, we can specify two Communication-sequences and check their feasibility. A sequence that checks for an invalid *withdrawal* is:

(comm, consumer,**withdraw**, NA)   // withdraw from an empty buffer

The following invalid sequence contains four consecutive *deposits*:

(comm, producer, **deposit**, NA),
(comm, producer, **deposit**, NA),
(comm, producer, **deposit**, NA),
(comm, producer, **deposit**, NA).   // deposit into a full buffer

If either of these Communication-sequences is feasible, then a fault is detected in *boundedBuffer.*

Alternatively: use reachability testing to indirectly check the feasibility of these sequences. That is, if neither of these sequences are exercised during reachability testing, then they are infeasible.

4.11.4 Reachability Testing for Monitors

Reachability testing can also be used to automatically derive and exercise every (partially-ordered) M-sequence of a monitor-based program.

- Reachability testing identifies race conditions in an execution trace and uses the race conditions to generate race variants.
- Recall from Chapter 3 that a race variant represents an alternate execution behavior that *definitely could have happened*, but didn't, due to the way race conditions were arbitrarily resolved during execution.
- Replaying a variant ensures that a different behavior occurs during the next execution.

Fig. 4.30a shows an execution trace of a bounded buffer program with two producers (P1 and P2), two consumers (C1 and C2), and an SC monitor M.

- A solid arrow from a producer or consumer thread to monitor M indicates that the thread called and entered monitor M.
- In this execution, the producer and consumer threads enter the monitor in the order (C1, P1, C1, P2, C2).
- The 2nd entry by C1 occurs when C1 reenters the monitor after being signaled by P1.

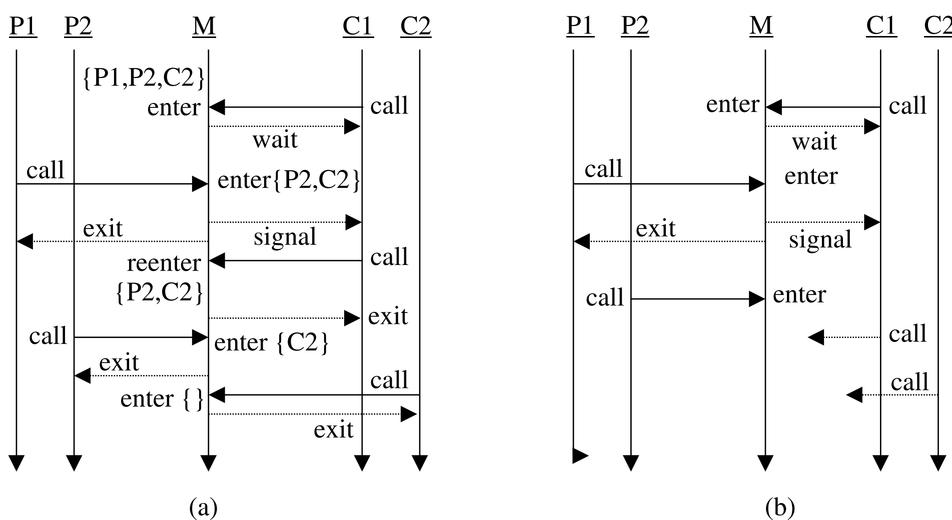

Figure 4.30 Execution trace and a race variant

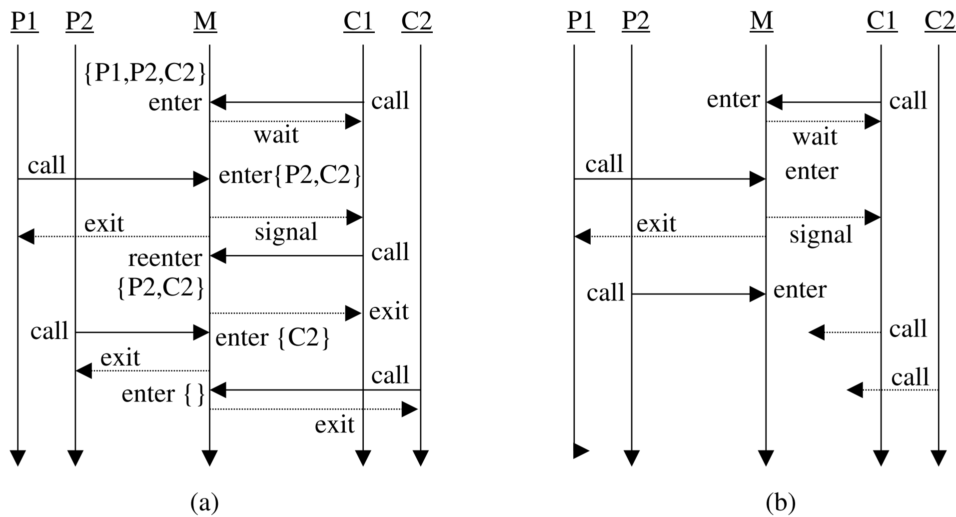

Figure 4.30 Execution trace and a race variant

Fig. 4.30a identifies the racing threads, called the “race set”, for each entry event:

- The race set for C1’s reentry event is shown beside the reentry event as {P2, C2}.
- This indicates that P2 or C2 could have entered the monitor instead of C1.
- Note that the monitor calls made by P2 and C2 were concurrent with C1’s call to reenter the monitor. (This is how the racing monitor calls were identified.)

A race variant is created by changing the calling thread for a monitor (re)entry event. The new calling thread must be a member of the race set for the (re)entry event.

Fig. 4.30b shows one of the eight race variants for the trace in Fig. 4.30a.

- In this variant, P2 enters the monitor before C1 reenters the monitor.
- When this race variant is replayed, the two producers will both deposit their items before either consumer withdraws one.
- The order in which the two consumers will (re)enter the monitor and withdraw their items after the variant is replayed is non-deterministic.

The complete trace of whichever sequence occurs can be analyzed to derive more variants, which can be replayed to generate more traces, and so on, so that all possible M-sequences are exercised during reachability testing.

Table 4.2 shows the results of applying reachability testing to several monitor programs:

- BB is the bounded buffer program in Listing 4.3.
- DP2 is the second solution to the dining philosophers program in Listing 4.8.
- RW is the readers and writers program in Listing 4.9.

For BB and RW, we show results for SU and SC monitors

Program	Config.	#Seqs.	Program	Config.	#Seqs.	Program	Config.	#Seqs.
BB-MonitorSU	**3P+3C+2S**	**720**	**RW-MonitorSU**	**3R+2W**	**13320**	**DP2-MonitorSU**	**3**	**30**
BB-MonitorSC	**3P+3C+2S**	**12096**	**RW-MonitorSC**	**3R+2W**	**70020**	**DP2-MonitorSU**	**4**	**624**
						DP2-MonitorSU	**5**	**19330**

Table 4.2 Reachability testing results for the example programs in Section 4.2.

These results help to quantify the difference between SU and SC monitors.

- SC monitors generate more M-sequences than SU monitors since SC monitors have races between signaled threads that are trying to reenter the monitor and calling threads that are trying to enter the monitor for the first time.
- SU monitors avoid these races by giving signaled threads priority over calling threads.

As we discussed in Chapter 3, the number of sequences exercised during reachability testing can be reduced by considering the symmetry of the threads in the programs.

- For the bounded buffer program, reachability testing considers all the possible orders in which a group of producers can perform their deposit operations, or a group of consumers can perform their withdraw operations.
- Producer threads may enter the monitor in the order ($Producer_1$, $Producer_2$, $Producer_3$), or ($Prodocer_2$, $Producer_1$, $Producer_3$), or ($Producer_3$, $Producer_2$, $Producer_1$), etc.
- Using thread symmetry, reduces the number of sequences exercised for the two bounded buffer programs from 720 and 12096 to 20 and 60, respectively.

4.11.5 Putting it All Together

4.11.5.1 Using the Java Toolboxes.

Extend class *TDThread,* instead of class *Thread*. This ensures that each thread receives a unique ID.

To trace an execution of program Buffer use: java –Dmode=trace Buffer
This creates three files:

- file monitor-replay.txt contains the simple M-sequence of the execution
- file monitor-test.txt contains the M-sequence of the execution
- file monitor-comm.txt contains the Communication-sequence of the execution.

Random delays can be executed during trace mode by setting property *randomDelay* to the value *on*: java –Dmode=trace –DrandomDelay=on Buffer

Deadlock detection is turned on by specifying –DdeadlockDetection=on. To turn deadlock detection off, set *deadlockDetection* to *off*; this is also the default value.

If separate sequences are desired for each monitor in the program, the *controllers* property can be used to create one controller per monitor. Valid values for this property are *single*, which is the default value, and *multiple*.

java –Dmode=trace –DrandomDelay=on –Dcontrollers=multiple Buffer

The simple M-sequence in file monitor-replay.txt is replayed using:

java –Dmode=replay Buffer

Reachability testing is performed on *Buffer* by setting the *mode* property to *rt* and executing a driver process named *RTDriver*:

java –Dmode=rt –DdeadlockDetection=0n RTDriver Buffer

The feasibility of the M-sequence in file monitor-test.txt is determined using:

java –Dmode=test Buffer

The feasibility of the Communication-sequence in file monitor-comm.txt is determined using: java –Dmode=commTest Buffer

The value used for property *controllers* during replay and testing must match the value that was used during tracing.

If the mode is not specified, the default value for the mode property turns tracing, replay, and testing off. Specifying "-Dmode=none" has the same effect.

The M-sequence recorded during a program's execution can be used to replay the execution. If the program is not modified after the M-sequence is recorded, then the M-sequence remains feasible, and checking the feasibility of this feasible M-sequence (using –Dmode=test) amounts to replaying the execution

4.11.5.2 Using the C++/Win32/Pthreads Toolboxes.

The mode of execution is specified using the environment variable MODE.
The possible values for MODE are TRACE, REPLAY, TEST, RT (for *R*eachability *T*esting), COMMTEST (to determine the feasibility of a Communication-sequence), and NONE.

To trace an execution of program Buffer in Windows/DOS execute:

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

and then executing program *Buffer*.

Random delays are controlled by variable RANDOMDELAY:

```
            set RANDOMDELAY=ON
// Unix:    setenv RANDOMDELAY ON
```

Deadlock detection is controlled by variable DEADLOCKDETECTION:

```
            set DEADLOCKDETECTION=ON
// Unix:    setenv DEADLOCKDETECTION ON
```

An execution in TRACE mode creates three trace files:

- file monitor-replay.txt contains the simple M-sequence of the execution
- file monitor-test.txt contains the M-sequence of the execution
- file monitor-comm.txt contains the Communication-sequence of the execution.

The value of environment variable CONTROLLERS determines how many sequence files are created:

```
   set CONTROLLERS=SINGLE   // one simple M-sequence and
                            // M-sequence per program
// Unix: setenv CONTROLLERS SINGLE
```

```
set CONTROLLERS=MULTIPLE  // one simple M-sequence and
                          // M-sequence per monitor
// Unix: setenv CONTROLLERS MULTIPLE
```

The default value for CONTROLLERS is SINGLE.

Reachability testing is performed setting the MODE to RT and customizing the driver process in file *RTDriver*.cpp, which is part of the synchronization library. Directions for customizing the driver process are in the file.

Some development environments have powerful debuggers that make it possible to stop threads, examine call stacks and variables of threads, suspend and resume threads, etc.

In replay mode, you can use such a debugger to set a breakpoint inside the control monitor at the point where a thread receives permission to execute the next event:

```
if (ID != nextEvent.getThreadID()) {
   // nextEvent does not involve the requesting thread
   threads[ID].waitC();       // wait for permission
   //get nextEvent again since it may have changed during the wait
   nextEvent = (monitorEvent)MSequence.elementAt(index);
}
// set breakpoint here; a thread has received permission to execute the next event.
```

With this breakpoint in place, it is possible to step through the execution and replay monitor events one at a time. Additional breakpoints can be set in the *run()* methods of the threads to watch what each thread does between events, without disturbing the replay.

The Win32 *OutputDebugString()* function can be used to send text messages to the debugger while the program is running. (In the Visual C++ environment, these messages appear in the tabbed window labeled “Debug”.)