The Modern Multithreading Library

This document describes how to use the testing and debugging features of the Java synchronization library that is described in the textbook "Modern Multithreading" by Richard Carver and Kuo-Chung Tai. Details about the library classes and the testing and debugging features they provide are given in the book. Here we describe how to write multithreaded Java programs that use the library and how to turn the testing and debugging features on and off.

Several example programs and instructions for how to run them have been included in this distribution. The files for each program are in a separate directory:

- BoundedBufferMonitorSC: An SC monitor solution to the bounded buffer problem.
- BoundedBufferMonitorSU: An SU monitor solution to the bounded buffer problem.
- BoundedBufferSelect: A solution to the bounded buffer problem that uses a selective wait.
- BoundedBufferSemaphores: A semaphore solution to the bounded buffer problem.
- CyclicScheduler: An implementation of Milner's Cyclic Scheduler.
- DiningPhilosophers: A semaphore solution to the dining philosophers problem that contains a deadlock.
- DistributedDiningPhilosophers: A solution to the distributed dining philosophers problem.
- DistributedMutualExclusion: A distributed mutual exclusion algorithm using message passing and monitors.
- DistributedMutualExclusion2: Same distributed mutual exclusion algorithm using message passing only.
- DistributedReachabilityTesting: Illustrates the use of running reachability testing with multiple processes, on a multi-core machine or on a compute cluster.
- DistributedSpanningTree: An implementation of a distributed spanning tree algorithm.
- ReadersWritersMonitorSC: An SC monitor solution to the readers and writers problem.
- ReadersWritersMonitorSU: An SU monitor solution to the readers and writers problem.
- ReadersWritersSelect: An solution to the readers and writers problem that uses a selective wait.
- ReadersWritersSemaphores: A semaphore solution to the readers and writers problem.
- SharedVariables: Illustrates the use of the shared object classes for detecting data races during reachability testing.
- StatefulReachabilityTesting: A demonstration of stateful reachability testing applied to the same distributed mutual exclusion algorithm as above.

1. Class *TDThread*

User threads should extend class *TDThread*.

```
class Producer extends TDThread { … }
```

Class *TDThead* automatically generates names and unique integer identifiers for each thread. The identifiers are used in the execution. Only *TDThread* threads can use the testing and debugging features in the library, which means that main thread should not call any synchronization functions. Instead, it should create *TDThread* threads that do all the work.

The *TDThread* identifiers that are generated for testing and debugging are saved in file ThreadID.txt whenever the testing and debugging functions are used. Here is the TDThread.txt file created for the BoundedBufferMonitorSU example program that is included with the library:

```
main_innerThread1 1
main_innerThread2 2
main_innerThread3 3
main_innerThread4 4
main_innerThread5 5
main_innerThread6 6
```

Each line gives the name of a thread and its ID. The thread names are generated internally. Users can supply more meaningful names by invoking the appropriate constructor in class *TDThread*:

```
class Producer extends TDThread {
   public Producer(int ID) {
     super("Producer"+ID);
   }
   …
}
```

```
class Consumer extends TDThread {
   public Consumer (int ID) {
      super("Consumer"+ID);
   }
   …
}
```

In the program's main() method we would write:

```
Producer p1 = new Producer(1);
Producer p2 = new Producer(2);
Consumer c1 = new Consumer(1);
Consumer c2 = new Consumer(2);
```

The ThreadID.txt file that is created from these user-supplied names is:

```
Producer1 1
Producer2 2
Producer3 3
Consumer1 4
Consumer2 5
Consumer3 6
```

Note that the IDs supplied by the user are used to generate unique names, but class *TDThread* generates its own internal IDs. User-supplied thread names must be unique or they will be rejected.

If you are planning to use reachability testing, the *main()* method must not return before the program's *TDTheads* have completed. To enforce this rule, use a try-catch block at the end of your *main()* method to *join()* all the threads:

```
public static void main (String args[])
   Producer p1 = new Producer(1);
   Producer p2 = new Producer(2);
   Consumer c1 = new Consumer (1);
   Consumer c2 = new Consumer (2);
   p1.start(); p2.start(); c1.start(); c2.start();
   try {
     p1.join(); p2.join(); c1.join(); c2.join();
```

```
    }
    catch(Exception e) {}
  }
```

In general, the *main()* methods in your programs should look as follows:

```
  public static void main (String args[])
    create and start TDThreads
    try {
      join all the threads
    }
    catch (Exception e) {}
  }
```

2. Tracing executions

Executions are traced by running them in "trace" mode. Here is the command for tracing an execution of example program BoundedBufferSemaphores:

```
java -Dmode=trace BoundedBufferSemaphores
```

This will produce 3 files:
- ThreadID.txt: Thread IDs that were generated
- semaphoreID.txt: semaphore names that were generated
- semaphore-replay.txt: simple PV-sequence that can be used to replay the program

(Eclipse Users: In your Run configuration for a Java Application:
- under Main, the main class is BoundedBufferSemaphores
- the VM argument is -Dmode=trace
)

Here are the contents of these three files for an execution of BoundedBufferSemaphores.

ThreadID.txt:

```
Producer1 1
Producer2 2
Producer3 3
Consumer1 4
Consumer2 5
Consumer3 6
```

semaphoreID.txt:

```
notFull
notEmpty
mutexD
mutexW
```

As with thread names, semaphore names can be supplied by the user. Here is how we specified the names generated in semaphoreID.txt:

```
notFull = new countingSemaphore(capacity,"notFull");
notEmpty = new countingSemaphore(0,"notEmpty");
mutexD = new countingSemaphore(1,"mutexD");
mutexW = new countingSemaphore(1,"mutexW");
```

User-supplied names must be unique or they will be rejected.

semaphore-replay.txt:

```
1
1
1
1
4
4
4
4
5
2
…
```

File semaphore-replay.txt contains a sequence of thread IDs, called a simple PV-sequence. A PV-sequence represents the sequence in which threads completed (not started) P and V operations during the execution. PV-sequences are described in Chapter 3 of the text.

To execute random delays during trace mode, set property –DrandomDelay=on. Random delays will be executed at the beginning of each thread's execution, and at the start of P, V, lock, unlock, send, and monitor entry operations.

3. Deadlock Detection

Deadlocks can be detected by setting property deadlockDetection=on in trace mode:

```
java -Dmode=trace -DdeadlockDetection=on BoundedBufferSemaphores
```

Here is the output generated for a version of program *BoundedBufferSemaphores* that contains a deadlock:

```
Monitoring for Deadlock.
Deadlock detected:
- Consumer2 blocked on P operation of mutexW
- Consumer1 blocked on P operation of notEmpty
- Consumer3 blocked on P operation of mutexW
- Producer3 blocked on P operation of notFull
```

Execution trace:

Thread Producer2 completed mutexD.P()
Thread Producer2 completed notFull.P()
Thread Producer2 completed mutexD.V()
Thread Consumer1 completed mutexW.P()
Thread Consumer1 blocking on notEmpty.P()
Thread Producer1 completed mutexD.P()
Thread Producer1 completed notFull.P()
Thread Producer1 completed mutexD.V()
Thread Consumer2 blocking on mutexW.P()
Thread Consumer3 blocking on mutexW.P()
Thread Producer3 completed mutexD.P()
Thread Producer3 blocking on notFull.P()

To use the deadlock detection function, all the threads in your program should be expected to complete. When all the threads that are not completed are blocked, a deadlock is assumed to occur.

3. Execution Replay

Executions that have been traced can be replayed by running the program in "replay" mode:

```
java -Dmode=replay BoundedBufferSemaphores
```

Here we assume the program *BoundedBufferSemaphores* was previously executed in "trace" mode so that the PV-sequence of the execution was recorded in file semaphore-replay.txt. When the sequence in semaphore-replay.txt has been replayed, you will see the message:

```
Sequence Completed
```

If you modify the program before you replay an execution, the replay might fail. If you want to determine whether or not a particular sequence can be exercised by your program, use "test" mode instead of "replay" mode. Test mode is described next.

4. Checking Feasibility

The feasibility of user-selected sequences can be checked in "test" mode. A test sequence is feasible if it can be exercised by the program (with a specified input); otherwise, it is infeasible. Test mode is not available for semaphore-based programs, but it is available for monitor-based programs and programs that use message passing. (Monitor-based programs are easier to write and test than semaphore-based programs so the library focuses on monitors. It is not difficult to implement test mode for semaphores.)

Here we show how to use test mode on an SU monitor version of bounded buffer. Program *BoundedBufferMonitorSU* is one of the example programs. Test mode assumes that file monitor-test.txt contains an M-sequence of the program under test. Here is an M-sequence of *BoundedBufferMonitorSU*:

```
(entry,4,Buffer:withdraw,NA)
(wait,4,Buffer:withdraw,notEmpty)
```

```
(entry,1,Buffer:deposit,NA)
(signalandexit,1,Buffer:deposit,notEmpty)
(signalandexit,4,Buffer:withdraw,notFull)
(entry,5,Buffer:withdraw,NA)
(wait,5,Buffer:withdraw,notEmpty)
(entry,2,Buffer:deposit,NA)
(signalandexit,2,Buffer:deposit,notEmpty)
(signalandexit,5,Buffer:withdraw,notFull)
(entry,6,Buffer:withdraw,NA)
(wait,6,Buffer:withdraw,notEmpty)
(entry,3,Buffer:deposit,NA)
(signalandexit,3,Buffer:deposit,notEmpty)
(signalandexit,6,Buffer:withdraw,notFull)
```

Running program *BoundedBufferMonitorSU* in test mode will determine whether the sequence in monitor-test.txt is feasible:

```
java -Dmode=test  BoundedBufferMonitorSU
```

Since the sequence in monitor-test.txt is feasible, you will see:

```
Sequence Completed
```

A small change in the sequence makes it infeasible. Here is a sequence in which the consumer enters the monitor first and then exits the monitor instead of waiting for the first item to be deposited:

```
(entry,4,Buffer:withdraw,NA)
(signalandexit,4,Buffer:withdraw,notFull)
```

When we run *BoundedBufferMonitorSU* in test mode with this sequence, we'll see the following message:

```
Infeasible sequence - timeout waiting for event 2: (signalandexit,4,Buffer:withdraw,notFull).
```

This indicates that the sequence is infeasible and specifically that it was not possible for the program to exercise the second event. Since this sequence is invalid, it is expected to be infeasible. If this sequence was found to be feasible, it would mean that the program had a bug.

5. Running Programs in spectest Mode

M-sequences specify exactly what is happening inside the monitor, but they are a little awkward to use. They contain four types of events and require detailed information such as the names of the condition variables. Communication-sequences are more abstract and easier to generate. A Communication-sequence consists of events generated by calls to method *exerciseEvent()*. For example, here are methods *deposit()* and *withdraw()* from program *BoundedBufferMonitorSU*:

```
public void deposit(int value) {
  enterMonitor("deposit");
```

```
    if (fullSlots == capacity)
      notFull.waitC();
    buffer[in] = value;
    in = (in + 1) % capacity;
    ++fullSlots;
    exerciseEvent("deposit");
    notEmpty.signalC_and_exitMonitor();
  }
  public int withdraw() {
    enterMonitor("withdraw");
    int value;
    if (fullSlots == 0)
      notEmpty.waitC();
    value = buffer[out];
    out = (out + 1) % capacity;
    --fullSlots;
    exerciseEvent("withdraw");
    notFull.signalC_and_exitMonitor();
    return value;
  }
```

And here is a feasible Communication-sequence for BoundedBufferMonitorSU with 1 Producer and 1 Consumer thread:

```
(1,deposit)
(4,withdraw)
(2,deposit)
(5,withdraw)
(3,deposit)
(6,withdraw)
```

Mode spectest assumes that file monitor-spectest.txt contains a Communication-sequence of the program under test. Running program *BoundedBufferMonitorSU* in spectest mode will determine whether sequence in monitor-spectest.txt is feasible:

```
java -Dmode=spectest  BoundedBufferMonitorSU
```

Assuming that the feasible sequence above is in montor-spectest.txt, you will see:

```
Sequence Completed
```

A small change in the sequence makes it infeasible. For example, change the first event in monitor-spectest.txt to be:

```
(4,withdraw)
```

and run the program in spectest mode again. Since no sequence can start with a withdrawal by a consumer, this sequence is infeasible, and you will see:

Infeasible sequence - timeout waiting for event 1: (4, withdraw).

Be careful where you place the calls to *exerciseEvent()* as this affects the feasibility of the sequence. Also, every monitor method must have at least one call to *exerciseEvent()*.

6. Reachability Testing

Reachability testing exercises every feasible SYN-sequence of the program one time. (This requires that every execution of a program with a given input terminates, and the total number of possible SYN-sequences is finite.) To perform reachability testing (RT) you must use the RT driver. The RT driver calls the *main()* method of your program.

Reachability testing is performed in "rt" mode:

```
java -Dmode=rt RTDriver BoundedBufferMonitorSU
```

Here we specified the name of the program to be tested as a command-line parameter for *RTDriver*.

(Eclipse Users: In your Run configuration for a Java Application:
- under Main, the main class is RTDriver
- the Program Argument is BoundedBufferMonitorSU
- the VM argument is -Dmode=rt
)

The program to be tested, in this case class *BoundedBufferMonitorSU* must have a null constructor. (If class *BoundedBufferMonitorSU* has no user-defined constructor, the system will supply a null constructor.) To avoid problems, class *BoundedBufferMonitorSU* should not extend class *TDThread* or implement the *Runnable* interface.

The output of reachability testing for the *BoundedBufferMonitorSU* example program is as follows (there are 3 producers and 3 consumers in this program):

```
start:Fri Nov 18 13:29:30 GMT-05:00 2005
1/1
25/25
50/50
75/75
100/100
125/125
150/150
175/175
200/200
225/225
250/250
275/275
300/300
325/325
350/350
375/375
```

```
400/400
425/425
450/450
475/475
500/500
525/525
550/550
575/575
600/600
625/625
650/650
675/675
700/700

Reachability Testing completed.
  Executions:720 / Sequences Collected:720
  Elapsed time in minutes: 0.2
  Elapsed time in seconds: 12.0
  Elapsed time in milliseconds: 12000
```

After displaying the start time, the number of executions is displayed, every 25 executions, until RT terminates. (The hash value 25 can be changed using property –Dhash, e.g., -Dhash=1.) Then the number of exercised sequences and the elapsed time for RT is displayed. The actual elapsed time may be 4 seconds more or less than shown.

You can also run RT mode with -DdeadlockDetection=on.

During reachability testing, statistics are generated and written to file "stats.txt". One line is written for each program execution. Each line contains three numbers:

- Number of executions so far
- Number of variants groups remaining to be processed. For each sequence that is collected, zero or more variants are generated. The variants generated for a sequence are considered to be a "variant group". At any point during reachability testing, there may be 0, 1, or more variant groups remaining to be processed.
- Number of variants generated for the SYN-sequence collected during execution. This is the size of the variant group.

Here is the stats.txt file generated for the *BoundedBufferMonitorSU* program with 2 producers and 2 consumers:

```
New Test Run at Wed Jan 04 09:49:51 GMT-05:00 2006
1:NumberOfSequences/Executions 2:Remaining Groups 3:NumberOfVariants (in the group)
1 1 6
2 1 0
3 2 1
4 3 1
5 4 3
6 5 3
7 6 3
```

8 5 0
9 6 1
10 7 1
11 6 0
12 5 0
13 4 0
14 5 1
15 6 1
16 5 0
17 4 0
18 3 0
19 4 1
20 5 1
21 4 0
22 3 0
23 2 0
24 1 0

The first line "1 1 6" shows that a group of 6 variants was generated for the sequence collected during the first execution.

The first variant in this group was used to perform the second execution of the program. Line 2 "2 1 0" shows that no variants were generated for the trace collected during this execution.

The second variant in the first group was used to perform the third execution of the program. Line 3 "3 2 1" shows that a group of one variant was generated for the trace collected during this execution. There are now 2 variant groups remaining to be processed – group 1 has 4 variants remaining to be processed and group 2 has a single variant.

In all there were 23 variants generated and 24 executions of the program.

As we mentioned at the beginning of this document, if you are planning to use reachability testing, use a try-catch block at the end of your *main()* method:

```
public static void main (String args[])
   Producer p1 = new Producer(1);
   Producer p2 = new Producer(2);
   Consumer c1 = new Consumer (1);
   Consumer c2 = new Consumer (2);
   p1.start(); p2.start(); c1.start(); c2.start();
   try {
     p1.join(); p2.join(); c1.join(); c2.join();
   }
   catch(Exception e) {}
}
```

This try-catch block is required to ensure that all user threads are completed before the *main()* method returns to the RTDriver.

You can also run RT with deadlock detection (-DdeadlockDetection=on).

6. Checking the Sequences that are Exercised During Reachability Testing

User-level events can be generated and collected during execution. At the end of each execution during reachability testing, this sequence of events can be passed to a user-supplied check() routine that checks the exercised sequence for correctness.

For example, in the distributed mutual exclusion example program (*DistributedMutualExclusion3*), when each process enters its critical section, it executes:

```
ApplicationEvents.exerciseEvent("enter");
```

This generates an application event.

To check the sequences that are exercised, write the following class:

```
public class MyChecker implements Checker {
    public boolean check (ArrayList seq) {
    // returns false if any check fails
        /* your check routine here */
    }
}
```

and supply the implementation of method *check()*. Method *check()* will receive an *ArrayList* of *AppEvent*, where an *AppEvent* is:

```
public class AppEvent {
  private int tid; // thread id
  private String label; // event label
  private vectorTimeStamp stamp; // event timestamp
  public AppEvent (int tid, String label, vectorTimeStamp stamp) {
    this.tid = tid;    this.label = label;    this.stamp = stamp;
  }
  public int getThreadID () {return tid;}
  public String getLabel () {return label;}
  public vectorTimeStamp getTimeStamp () {return stamp;}
  public boolean happenBefore (AppEvent another) {
    return stamp.lessThan (another.getTimeStamp ());
  }
  public boolean isConcurrent (AppEvent another) {
    return !happenBefore(another) && !another.happenBefore(this);
  }
  public String toString () {
    StringBuffer rval = new StringBuffer ();
    rval.append("AppEvent ["); rval.append(tid); rval.append(", "); rval.append(label + ", ");
    rval.append(stamp); rval.append("]"); return rval.toString ();
  }
}
```

AppEvents are created by calls to exerciseEvent("label"). Each event contains the label, the thread ID of the calling thread, and a vector timestamp value for the event.

Compile MyChecker.java:

Windows:	javac MyChecker.java
Unix:	javac MyChecker.java

Perform RT using:

Windows:	java -Dmode=rt -DcheckTrace=on RTDriver DistributedDiningPhilosophers
Unix:	java -Dmode=rt -DcheckTrace=on RTDriver DistributedDiningPhilosophers

The *check()* method for the *DistributedDiningPhilosophers* example program checks that no neighboring philosophers eat concurrently and that all philosophers eventually eat:

```
import java.util.ArrayList;
public class MyChecker implements Checker {
    private int[] tids;         // application thread IDs
    private ArrayList seq;      // a sequence of application events
    public MyChecker () {
        tids = new int [3];
        tids[0] = 1; tids[1] = 4; tids[2] = 7;          // hardcode thread IDs
    }
    public boolean check (ArrayList seq) {
        //System.out.println("*****");
        //for (int i=0; i<seq.size();i++)
        //   System.out.println((AppEvent)seq.get(i));
        //System.out.println("*****");
        boolean rval = true;
        // check for eventual entry
        int culprit = -1;
        for (int i = 0; i < tids.length; i ++) {
            boolean entered = false;
            for (int j = 0; j < seq.size (); j ++) {
                AppEvent event = (AppEvent) seq.get(j);
                if (tids[i] == event.getThreadID ()) {
                    entered = true;
                    break;
                }
            }
            if (!entered) {
                culprit = tids[i];
                rval = false;
                break;
            }
        }
        if (!rval) {System.out.println("Failed: Thread " + culprit + " did not enter.");}
```

```
        else { // check for mutual exclusion
            for (int i = 0; i < seq.size (); i ++) {
                AppEvent event = (AppEvent) seq.get(i);
                for (int j = 0; j < seq.size (); j ++) {
                    if (i != j) {
                        AppEvent it = (AppEvent) seq.get(j);
                        if (event.isConcurrent (it)) {
                            System.out.println("Concurrent application events:" + event + "; " + it);
                            rval = false;
                            break;
                        }
                    }
                }
                if (!rval) {System.out.println("Failed: Mutual exclusion violated.");}
                else { //System.out.println("Passed");}
            }
        }
        return rval;
    }
}
```

7. Reachability Testing Reductions

There are four ways to reduce the number of sequences that are exercised during reachability testing.

For semaphore-based programs, reachability testing (RT) can be performed with the PVReduction. This reduction (safely) ignores some races between P and V operations, e.g., a race between two V operations on the same semaphore. This optimization will often cause fewer sequences to be exercised. For the *BoundedBufferSemaphores* example program, we would run:

```
Java –classpath .;.. -Dmode=rt -DPVReduction=on RTDriver BoundedBufferSemaphores
```

(Eclipse Users: In your Run configuration for a Java Application:
- under Main, the main class is RTDriver
- the Program Argument is BoundedBufferSemaphores
- the VM arguments are –classpath .;.. -Dmode=rt -DPVReduction=on
)

The output would look something like:

```
start:Mon Nov 14 15:30:23 GMT-05:00 2005
1/1
25/25
50/50

Reachability Testing completed.
  Executions:56 / Sequences Collected:56
  Elapsed time in minutes: 0.11666666666666667
```

Elapsed time in seconds: 7.0
Elapsed time in milliseconds: 7000

The number of sequences exercised may vary, depending on the races that actually occur when the program is executed.

The actual elapsed time may be up to 4 seconds more or less than shown.

The second way to reduce the number of sequences exercised during reachability testing is to use the symmetry reduction. For example, in example program *BoundedBufferSemaphores*, the *Producer* threads have identical behavior, i.e., each *Producer* executes P and V operations on the same semaphores, in the same order, independent of any inputs, and independent of the identity of the thread. We can take advantage of the symmetry of the Producer threads by ignoring the different orders that *Producer* threads execute the same operation (and the same with *Consumers*.) For example, each *Producer* thread begins by executing mutexD.P(). Normal RT will exercise the 6 possible orders in which the three *Producer* threads can execute mutexD.P(). With symmetry reduction, RT ignores the races that occur between *Producer* threads executing mutexD.P(). This causes fewer sequences to be exercised.

To use the symmetry reduction, first create a file called symmetry.txt. Each line of this file specifies the thread identifiers of one group of symmetric threads. For example, for program *BoundedBufferSemaphores* the contents of symmetry.txt should be:

5 6 7
8 9 10

The first group - 5 6 7 - represents the three *Producer* threads. The second group - 8 9 10 - represents the three *Consumer* threads. Each thread in the program should be listed in one and only one group. To see the thread identifiers, you can first run the program for a few seconds in rt mode without symmetry reduction and examine the identifiers in file ThreadID.txt. (Make sure you use the identifiers generated by rt mode, not trace mode, since the identifiers may be different. In rt mode, the synchronization objects (e.g., semaphores) receive identifiers too, but *the symmetry.txt file should only contain identifiers for threads*.)

Run reachability testing with symmetry reduction using:

```
java -Dmode=rt -DsymmetryReduction=on RTDriver BoundedBufferSemaphores
```

(Eclipse Users: In your Run configuration for a Java Application:
- under Main, the main class is RTDriver
- the Program Argument is BoundedBufferSemaphores
- the VM arguments are -Dmode=rt -DsymmetryReduction=on
)

The output should look something like:

start:Mon Jan 23 10:15:31 GMT-05:00 2006
1/1

```
Reachability Testing completed.
  Executions:9 / Sequences Collected:9
  Elapsed time in minutes: 0.06666666666666667
  Elapsed time in seconds: 4.0
  Elapsed time in milliseconds: 4000
```

You can also run *BoundedBufferSemaphores* in reachability testing mode with both the symmetry reduction and the PVReduction:

```
    java -Dmode=rt -DPVReduction=on -DsymmetryReduction=on
        RTDriver BoundedBufferSemaphores
```

The output should look something like:

```
start:Mon Jan 23 10:17:49 GMT-05:00 2006
1/1

Reachability Testing completed.
  Executions:1 / Sequences Collected:1
  Elapsed time in minutes: -0.06666666666666667
  Elapsed time in seconds: -4.0
  Elapsed time in milliseconds: -4000
```

Your output may also indicate that 3 sequences were exercised instead of 1. (The number of sequences exercised may vary, depending on the P/V races that actually occur when the program is executed.)

The symmetry reduction can be applied to all the *BoundedBuffer* and *ReaderWriter* example programs, but not to programs *CyclicScheduler*, *DiningPhilosophers*, or *DistributedMutualExclusion*. Program *DiningPhilosophers* contains some symmetry but this symmetry currently cannot be exploited by our symmetry reduction. (Different philosophers access different forks (semaphores) which violates the requirements of our symmetry reduction.)

Be careful when you write programs on which you plan to apply symmetry reduction. We use a very simple, but also very limited, technique to identify which synchronization operation is being executed by a thread. (This involves creating a *Throwable* object *e* and calling *e.getStackTrace()* to identify the point in the source code where the call is being made.) For this technique to work, only one synchronization operation can appear in a line of source code. For example, you should write:

```
    s.P();
    s.V();
```

instead of

```
    s.P(); s.V().
```

Also, the operations should not appear inside a loop.

The third way to reduce the number of sequences exercised during reachability testing is to use t-way reachability testing. T-way reachability testing exercises SYN-sequences selectively, not exhaustively. The selection of SYN-sequences is based on a combinatorial testing strategy known as *t*-way testing. Exhaustive reachability testing derives race variants to cover all possible

combinations of the race outcome changes that can be made in a SYN-sequence. In contrast, *t*-way reachability testing derives race variants to cover all possible *t*-way combinations of the race outcome changes, i.e., those involving changes to the outcomes of *t* race conditions, where *t* is usually a small number. T-way reachability testing can substantially reduce the number of SYN-sequences exercised during reachability testing. For example, for a solution to the distributed dining philosophers (DDP) problem with 3 processes, exhaustive reachability testing exercises more than 5 million sequences, while 1-way reachability testing exercises only 5,199 sequences on average.

Run t-way reachability testing using:

```
java -classpath .;.. -Dmode=rt -DtWay=on -Dt=1 RTDriver DistributedDiningPhilosophers
```

The fourth way to reduce the number of sequences exercised during reachability testing is to use random reachability testing. Like t-way reachability testing, random reachability testing exercises SYN-sequences selectively, not exhaustively. During random reachability testing, a random choice is made to either execute or discard a group of variants. The number of sequences exercised by random reachability testing appears to be similar to the number of sequences exercised by 1-way reachability testing.

Run random reachability testing using:

```
java -classpath .;.. -Dmode=rt –DrunMode=random RTDriver DistributedDiningPhilosophers
```

8. Detecting Data Races during Reachability Testing

A program that accesses shared variables outside of a critical section is said to have a data race. Data races can be detected during reachability testing by using special shared object classes that monitor Read and Write operations on the shared objects. The shared object classes are just wrappers for primitive types int, boolean, double, and long:

```
sharedInteger s = new sharedInteger(0,"s");
sharedBoolean b = new sharedBoolean(true,"b");
sharedDouble d = new sharedDouble(1.0,"d");
sharedLong l = new sharedLong(1,"l")
```

When a shared object is constructed, it must be assigned an initial value and a name. Each shared object class supplies Read and Write operations for the shared objects. For example:

```
System.out.println(s.Read());
s.Write(s.Read() + 1);
```

The Read and Write operations are traced and analyzed during reachability testing. If any Read and Write operations are involved in a data race, i.e., the shared object is accessed outside of a critical section, then reachability testing ends and the SYN-sequence that contained the data race is displayed.

Specify -DdetectDataRace=on to check for data races during RT:

Windows: java -Dmode=rt -DdetectDataRace=on …
Unix: java -Dmode=rt -DdetectDataRace=on …

Methods Read() and Write() can only be called by TDThread objects. If a normal Java Thread object calls Read() or Write(), an error will occur.

Example directory *SharedVariable* contains a program that illustrates the use of the shared object classes.

Note that if the shared object classes are not used and data races occur during reachability testing then reachability testing will usually fail and display a message indicating that a data race is likely to be what caused the failure. This message is not very helpful in determining how to fix the program. If the shared object classes are used, then the Read or Write operations involved in the data race will be displayed, helping the programmer find and fix the erroneous critical section.

9. Distributed Reachability Testing

One way to speed up reachability testing is to perform reachability testing in parallel on multiprocessor/distributed systems. Reachability testing can be performed on a multi-core machine, or on multiple workstations in a distributed system. Since test runs during reachability testing are independent, inter-process communication takes place only when variants are distributed to the workstations.

Our implementation of distributed reachability testing is a work in progress. We are tuning the implementation to achieve the best performance and ease of use.

The partitioning scheme used in MM gives each of the "worker" processes an equal number of initial variants and allows a worker process to request more variants from the "manager" process when the worker runs out. The manager can send some of its variants to a worker that requests more. (The manager process also performs reachability testing with its own set of variants.) Alternatively, the manager can steal variants from some other worker. Each worker process has one thread to execute the program under test, one thread to generate variants, and one thread to monitor the progress of reachability testing and request more variants when the worker runs out.

To perform distributed reachability testing on program *DistributedMutualExclusion*, the RTDriver program is first executed as a manager by using the following command:

```
java -Dmode=rt -DmanagerWorker=manager -DnumWorkers=10 -DserverPort=40021
RTDriver  DistributedMutualExclusion
```

The manager's properties specify the number of workers and the port number on the server machine that the workers will be using to communicate with the manager. The following command is used to start the RTDriver program as worker 1:

```
java -Dmode=rt -DmanagerWorker=worker -DworkerNumber=1 -DserverPort=40021
-DserverIP=192.168.1.1  RTDriver DistributedMutualExclusion
```

The worker's properties specify the worker's number, which identifies the worker, and the IP address and port number used to communicate with the server machine that is executing the manager program.

If the manager and worker processes are running on the same machine, you can use:
-DserverIP=LocalHost.

The value *n* for the manager's *numWorkers* property is used to partition the initial batch of variants among the *n* workers that are available when reachability testing starts. More workers can be added while reachability testing is in progress. The added workers obtain variants by stealing them from the manager or from other workers.

When the manager process and all the worker processes complete, the results for worker i are saved in file workerResultsi.txt and the file RTResults.txt contains the results for all the workers and the manager. There is a delay of several minutes at the end as workers communicate their results with the manager process.

The file workerResults1.txt produced by worker 1 above contains:

```
Reachability Testing completed.
  Executions:1592 / Sequences Collected:1592
  Elapsed time in minutes: 7.6947833333333335
  Elapsed time in seconds: 461.687
  Elapsed time in milliseconds: 461687
```

File RTResults.txt contains:

```
Reachability Testing completed for Manager
  Executions:2440 / Sequences Collected:2440
  Elapsed time in minutes: 6.500516666666667
  Elapsed time in seconds: 390.031
  Elapsed time in milliseconds: 390031

(sub)Total Executions:2440 (sub)Total Sequences Collected:2440

Reachability Testing completed for Worker 1
  Executions:1592 / Sequences Collected:1592
  Elapsed time in minutes: 8.5315
  Elapsed time in seconds: 511.89
  Elapsed time in milliseconds: 511890

(sub)Total Executions:4032 (sub)Total Sequences Collected:4032
```

The elapsed time recorded for the worker in file RTResults.txt is longer than the time recorded in workerResults1.txt due to some delays that are performed at the end of the manager's execution. The times in the workerResults.txt files are more accurate. Actually, the times recorded in workerResults.txt are also a little longer than the actual time, the actual time typically being 30 or seconds less than the recorded time.

The manager and worker in the above example were executed on the same machine, which had only 1 CPU, so there was no speedup relative to using a single process for reachability testing. When running distributed reachability testing on a compute cluster, the speedup is nearly linear.

Example directory *DistributedReachabiltyTesting* illustrates this process.

10. Stateful Reachability Testing

Stateful reachability testing extracts and stores program states to reduce the number of sequences that are executed during reachability testing. To perform stateful reachability testing you must use the stateful RT driver. Stateful reachability testing is performed in "rt" mode -Dmode=rt with the property -DstatefulRT=on.

For example, execute the following command (in Windows) to apply stateful reachability testing to a version of the distributed mutual exclusion program that uses monitors to simulate message passing:

java -Dmode=rt -Dhash=25 -DstatefulRT=on -DstatePruning=on -DmaxThreads=10
-classpath .;../ModernMultithreading.jar RTDriverStateful DMEMonitorSC

(In Unix, use "-classpath .:../ModernMultithreading.jar", which replaces ';' with ':'.)

The output of stateful reachability testing for DMEMonitorSC is as follows:

```
start:Tue Mar 12 14:07:09 EDT 2013
start input
1/1
25/25

Reachability Testing completed.
 Executions:42 / Sequences Collected:42
 Peak Stored States: 16
 Number of States: 336
 Number Variants Pruned: 32
 Elapsed time in minutes: 0.025
 Elapsed time in seconds: 1.5
 Elapsed time in milliseconds: 1500
```

After displaying the start time, the number of executions is displayed, every 25 executions, until stateful RT terminates. The information displayed after RT completes is:

- Then the number of exercised sequences,
- The peak stored states, which is the maximum number of program states that had to be stored in memory at any one time
- The number of states in the program
- The number of variants that did not have to be exercised because these variants would only visit a part of the state space that had been visited before.
- The elapsed time for RT. The actual elapsed time may be 4 seconds more or less than shown.

Regular RT exercises 96 sequences for this program. Thus, stateful RT reduces the number of executions, at the cost of more space for storing states, and additional time and effort for extracting states.