Using Instrumentation to Reproduce the Execution of Java Concurrent Programs

Márcio Eduardo Delamaro
Centro Universitário Eurípides de Marília (UNIVEM)
Av. Hygino Muzzi Fillho, 529
Marília - SP, Brazil
17525-901
delamaro@fundanet.br

Abstract

The Java language provides mechanisms to implement concurrency and synchronization based on the monitor model. Testing multi-thread Java programs many times requires to re-execute them with determined test data. The problem is that re-executing such programs with a same test datum does not guarantee the same behavior due to the non deterministic scheduling policy implemented in the JVM. In this paper we present a simple way to reproduce the behavior of a multi-thread Java program, based on program instrumentation.
Keywords: deterministic replay, Java multi-thread programs

1 Introduction

Testing is a crucial activity in the software lifecycle. It is expensive and time consuming. For this reason much effort has been spent on developing techniques and tools to support the testing activity. An important result of research is the definition of techniques and criteria to drive the generation of test sets that can suitably exercise a program.

In testing and debugging activities, many times we face the problem of executing the program under analysis and checking its results. The term "record and playback" (R&P) has been used to describe techniques and tools to register the input and output of a program execution and to use such data to automatically re-execute the same (or other) program and compare the results.

When dealing with concurrent programs one extra aspect must be considered, i.e., the non determinism. Two executions of a program on the same input may produce different – and still correct – results. If one wants to assure that the results of two executions with the same input are identical, some extra control should be applied.

When testing a sequential program one can rely on the fact that there exists only one correct output for a given test data. The statements are executed in a determined order that does not change from one execution to another, if the input is the same. This is not true for concurrent programs. The parallel or concurrent execution of several (deterministic) processes may result in different outputs, depending on the order of execution of the different processes.

In the Java programming language, concurrency is obtained by the creation of multiple threads of execution. Each thread has its own execution environment. On the other hand, they can share objects. In order to control the access to the shared objects the language

provides statements and methods that implement the monitor synchronization model. In short, each object created in the Java Virtual Machine (JVM) has associated to it a monitor that can be used to assure to a thread, exclusive access to the object.

In this paper we describe an approach to R&P of Java concurrent programs based on program instrumentation. A given program P that is subject to analysis (testing or debugging) is instrumented in such a way that, when executed with a test case t, will produce, in addition to its ordinary output, a sequence of events called "synchronization sequence" intended to identify the order of access of each thread to the shared objects. This is the "record" phase. In the "playback" phase the same P is instrumented in such a way that the recorded "synchronization sequence" is used to guide the program re-execution to reproduce its original behavior.

In the next section the basics of the Java synchronization mechanisms are discussed. Next we present our model of instrumentation to R&P Java concurrent programs and the classes we created to implement it. In Section 4 it is discussed how our technique can also be used to try to exercise the program more completely by generating new synchronization sequences to it. Section 5 discuss the limitations and drawbacks of our approach. Section 6 comments on related work and Section 7 presents our final observations.

2 Overview of Java Synchronization Mechanisms

The unit for parallelism in Java are "threads". Threads have their own execution environment and may execute independently. They can also share objects, and thus, a mechanism to synchronize the access to the shared objects is needed.

The basic mechanisms for thread synchronization in Java are synchronized methods and synchronized blocks. Every object in Java has a **monitor** associated to it. This monitor is used to guarantee that only one thread at time has access to an object. Program 1 shows an example of synchronized method. In a program with two or more threads sharing an object X of type `MyClass`, only one at a time can enter `myMethod` using that object. The access to `myUnsyncMethod` does not have such restriction. Hence, a thread T_1 can execute, for instance, `myMethod` on object X concurrently with thread T_2, executing `myUnsyncMethod` on the same object X but cannot execute `myMethod` concurrently with thread T_2, executing `myMethod` or `myOtherMethod`.

If a given thread T_1 is executing a synchronized method on an object X and another thread T_2 tries to enter a synchronized method on the same object the system blocks T_2 until T_1 terminates executing the synchronized method. Then T_2 can execute, or more precisely, it can compete to get access to the object, since other threads may also be trying to access the same object within a synchronized method.

When a thread T_1 gains access to a synchronized method of an object X we say that T_1 owns or has locked X's monitor. A static method can be synchronized. In this case, entering the method locks the monitor associated to the `Class` object associated to the class where the method is defined.

Synchronized blocks are similar to synchronized methods but the protected code is restricted to a piece of a method and the object on which the lock is executed is explicitly declared. For example, in the code in Program 1, the thread must obtain the monitor of `myObject` before executing `doSomeOtherThing`, in method `myOtherMethod`.

Program 1 - Example of synchronized method/block

```
1  public class MyClass {
2      myOtherClass myObject = new myOtherClass();
3
4      public synchronized void myMethod() {
5          doSomething();
6      }
7
8      public void myOtherMethod() {
9          synchronized (myObject) {
10             doSomeOtherThing();
11         }
12     }
13
14     public void myUnsyncMethod() {
15         doSomeOtherThing();
16     }
17 }
```

Every object created by the Java Virtual Machine (JVM) is associated to a **wait set** which allows the thread that locked a monitor temporarily releasing that monitor until an event occurs. Class `Object` defines a `wait` method that releases the monitor of the object used in the call. The method also inserts the current thread in the object's wait set, temporarily blocking its execution. The definition of method `wait` explicitly states that the thread that calls the `wait` must be holding the monitor of the object, otherwise an exception is thrown. Since every class in Java inherits from the class `Object`, every object created in Java has a method `wait` (declared "final" in the class `Object`).

The class `Object` also declares a method `notify` which wakes up a thread waiting in a wait set. When this method is called on an object X, one of the threads is randomly removed from the wait set, becoming ready to execute. This does not mean that it will be executed immediately. First because the thread that calls `notify` must be in possession of the object monitor; thus, the thread removed from the wait set can execute only after the current thread releases the lock. Moreover, even after the current thread has released the lock, there is no guarantee that that specific thread will be the one that acquires the monitor because other threads may be in contention for the same monitor.

The thread removed from the wait set resumes from the point immediately after the call to `wait` and returns to the same state it had at that time, i.e., in possession of the monitor. The `wait` method has a variant that accepts a timeout value as argument. The semantics of this call is similar to the one described above, i.e., the thread is inserted in the wait set until a `notify` removes it. In addition, if it is not removed by a `notify` within the specified timeout, it is removed "by itself". The `notify` method has a variant `notifyAll` that, when called on object X, removes all the threads from the wait set of X.

Other methods in the Java API complete the resources of the language for concurrency. Among them, the methods in the class `Thread`, responsible for the creation and management of new threads.

3 Record and Playback

In this section we describe our approach to R&P concurrent Java programs in order to allow the reproduction of one execution of a program P, making sure that the results obtained will match the results of the original execution. Initially, we describe the two distinct types of instrumentation: one for recording an execution and another for replaying it. Then we show how the instrumentation is actually implemented, allowing both phases to be done with a single instrumentation.

It is necessary to note that we are considering programs in which non-determinism arises only from the order in which the threads are activated. In addition, in order to guarantee that any single thread isolated from the others has a deterministic behavior, any access to a shared object is done inside a synchronized block or method. It is beyond the scope of this paper but it can be shown that under these restrictions a synchronization sequence can be determined by the order in which the threads access the shared objects or, in other words, the order in which the threads execute the synchronized code. If a program P executes with an input t and produces an output O, then a second execution of P with t that follows the same synchronization sequence should also produce O.

Several approaches can be used to force the re-execution of P to follow the same synchronization sequence recorded in a previous execution. Some obvious choices are:

- The scheduler system uses the same choices when choosing a thread to execute;
- The execution of P is externally monitored in the first execution and in the re-execution the synchronization sequence is forced by this external monitor;
- The program P is instrumented in order to record the synchronization sequence in the first execution and to follow the recorded synchronization sequence in the re-execution.

Each approach has its advantages and problems. The first solution is the most complete and less restrictive, as any program could be "recorded and played back" without any change in the program. In the case of Java, the JVM is the responsible for scheduling the threads. Thus the implementation of such a solution would require modifying the JVM implementation. In addition, different implementations of the JVM would require different implementations of the R&P system. Section 6 comments about *DejaVu*, a system that uses this solution.

The second solution can be implemented by using libraries for debugging Java programs such as the Sun's Java Platform Debugger Architecture (JPDA API). The monitoring program should set break points on specific places of P in such a way that the synchronization sequence could be recorded in the first execution. In the re-execution the monitor would be able to enable/disable threads according to the order established in the synchronization sequence and in this way make the program P to follow the desired sequence. The problem in this case is that the monitored execution of a program tends to be much slower than the normal execution. In a simple test we could verify that the number of breakpoints required to record the synchronization sequence in a simple program using the JPDA API makes its execution impractical and in some cases makes the JVM crash.

The third option is the one we have adopted. It is the simplest to implement but it is more restrictive than the others. It inserts some control statements in places of the code

that deal with shared objects, for example, inside a synchronized method. To do so, it is necessary to have access to the synchronized method code, no always accessible. Let us take as an example a thread that shares an object `vec` of type `java.util.Vector`. According to the definition of the Java API, this object is implicitly synchronized, i.e., when the thread calls a method on this object it is calling a synchronized method. This fact should be registered in the synchronization sequence but, as the code of `java.util.Vector` can not be instrumented, it is not possible to do so. The best we could do is to analyze statically the program and identify the points where implicitly synchronized objects are being used.

The instrumentation of the code is done in two different moments. The first is the instrumentation for the recording phase and the second is the instrumentation for the replaying phase. In the recording phase the goal is to register which thread is accessing which object. Since we started from the assumption that every access to a shared object is done inside a synchronized code, what we want to do is to register when a synchronized block or method begins executing. Then the instrumentation consists in finding the synchronized blocks and methods and inserting a statement inside them to register which thread is blocking which object.

Let us take as example the class in Program 1. The instrumented code for recording is shown in Program 2. The call to `beginRegisterAccess` is responsible for inserting an event in the synchronization sequence, i.e., the fact that a thread has locked an object. Although not necessary for the characterization of the synchronization sequence, the call to `endRegisterAccess` inserts the opposite event, i.e., the release of an object's lock. Such an event may be useful for synchronization sequence generation, as described in Section 4.

The replaying phase requires a different instrumentation. The first point to note is that before entering a synchronized code, the thread should consult the synchronization sequence and check whether the next event is the one to be executed. For example, if thread T_1 is about to lock object O_1, there must be an statement that consults the synchronization sequence to check if the next event is "T_1 locks O_1". This is the first problem we have. For a synchronized method, "before entering" means that the instrumentation should be done before calling the method, which, for many obvious reasons is not desired. Thus, the first thing the instrumentation should do is to transform a synchronized method into a synchronized block. The object used in the synchronized block depends on the method, i.e.:

- If the method is an instance method the object is "this";
- If the method is a static method the object is the `Class` object that represents the class in which the method is defined. For example `MyClass.class`.

The instrumentation for replaying a synchronization sequence for the example in Program 1 is shown in Program 3.

Note that the call to `checkAccess` does not use the synchronizing object to consult the synchronization sequence. This is so because, considering that the thread is deterministic and followed the synchronization sequence until that point, it would not be accessing a different object. It has only to check if it is its turn to execute.

Program 2 - Example of recording instrumentation

```
1  public class MyClass {
2      myOtherClass myObject = new myOtherClass();
3
4      public synchronized void myMethod() {
5          beginRegisterAccess(Thread.currentThread(), this);
6          try {
7              doSomething();
8          }
9          finally {
10             endRegisterAccess(this);
11         }
12     }
13
14     public void myOtherMethod() {
15         Object sinc;
16         synchronized (sinc = myObject) {
17            beginRegisterAccess(Thread.currentThread(), sinc);
18            try {
19                doSomeOtherThing();
20            }
21            finally {
22                endRegisterAccess(sinc);
23            }
24         }
25     }
26 }
```

Program 3 - Example of replaying instrumentation

```
1  public class MyClass {
2      myOtherClass myObject = new myOtherClass();
3
4      public void myMethod() {
5          checkAccess(Thread.currentThread());
6          synchronized (this) {
7              nextEvent();
8              doSomething();
9          }
10     }
11
12     public void myOtherMethod() {
13         checkAccess(Thread.currentThread());
14         synchronized (myObject) {
15            nextEvent();
16            doSomeOtherThing();
17         }
18     }
19 }
```

In addition to consulting the synchronization sequence it is necessary to remove its first event, when a match is found. This cannot be done in the `checkAccess` or elsewhere before entering the synchronized block. This would allow another thread to find a match and lock the object before, not respecting the ordering in the synchronization sequence. The call to `nextEvent` inside the block assures that the object has been locked before the event is removed from the synchronization sequence, liberating other threads to follow their execution (if possible).

This is only part of the necessary instrumentation. So far we did not deal with wait sets. As stated before, it is important to register every access to shared objects (or to synchronized code). When a thread returns from a `wait`, it is re-entering a synchronized code and that event should be registered in the synchronization sequence. The registration instrumentation replaces a call like `x.wait()` by a call to another method `Wait(x)`. This new method is the responsible for:

- Calling `x.wait()`;
- Inserting an event in the synchronization sequence to state that the thread gained access to the object again, after returning from the wait.

Although not necessary for implementing the R&P, registering the call to the `wait` method may contribute to add additional information to the synchronization sequence such as the inclusion of the thread in the wait set or even to control which threads are in which wait set. The same is valid for the `notify` and `notifyAll` calls. They do not need to be replaced in the recording phase instrumentation, but doing so makes it possible to insert some control events in the synchronization sequence; for example, to check which threads are in the specific wait set at the notify call.

For the replay phase, the instrumentation of wait's becomes a little more complex. When a thread is removed from the wait set and returns from the wait, it is necessary to check whether this is the event expected in the synchronization sequence . However, if it is not, what should be done? The thread should be re-inserted in the wait set – what is not so difficult – and then to expect the execution of some other thread that matches the next event in the synchronization sequence. The problem is that such a thread may never arrive because the `notify` that should remove it from the wait set may have been wasted with the wrong thread. Hence, the program may block.

The solution we adopted is:

- replace each call `x.wait` by a call to `Wait(x)`;
- the `Wait` method calls `x.wait(timeout)`;
- when returning from the `wait`, checks whether this event matches the one in the synchronization sequence. If it does not, call `x.wait(timeout)` again.

In summary, every call to a `wait()` is replaced by a timed-wait. In this way, the thread is re-inserted in the wait set and will not be blocked forever even if no other notify removes it from the wait set. The notify calls become useless but since we have the ordering of the threads to be followed, we can expect the same behavior of the original execution.

Programs 4 and 5 show, respectively, an original class `Channel` extracted from [4] and the actual instrumentation done by our system. The methods described above are

implemented in a package `RRTools` (Figure 1 extracted from [7]). They are inside a class named `Replay` and are all static methods. The instrumentation shown in Program 5 differs from the one previously described because it intends to be used in both phases, i.e., record as well as replay.

Program 4 - The original `Channel` class

```
1  public class Channel extends Selectable{
2
3      Object chan_ = null;
4
5      public synchronized void send(Object v) throws InterruptedException {
6          chan_ = v;
7          signal();
8          while (chan_ != null) wait();
9      }
10
11     public synchronized Object receive() throws InterruptedException {
12         block();
13         clearReady();
14         Object tmp = chan_; chan_ = null;
15         notifyAll(); //should be notify()
16         return(tmp);
17     }
18 }
```

By doing so, the instrumented program follows a given input synchronization sequence and at the same time records an output synchronization sequence. Although this may sound useless since they should be the same, this approach makes it possible to provide only part of a complete synchronization sequence (a prefix) and force the program to follow it. When the prefix runs out, the program continues normally, recording the rest of the synchronization sequence. If no prefix is provided as input, the system will only record the synchronization sequence.

As shown in Program 6, methods such as `afterEnterSyncBlock` in the class `Replay` use methods in class `RR` which is the responsible by doing the synchronization sequence recording. Both classes use a system property to define from where to read the input synchronization sequence and where to write the output synchronization sequence. Thus, if one wants to execute a program `MyProg`, which has its classes instrumented, and make the program read its input synchronization sequence from a file "inSeq" and write its output synchronization sequence to a file "outSeq", the following statement should be used:

```
java -DRR=outSeq -DRW=inSeq -classpath .:RRTools-directory MyProg
```

In the next section we explain how simultaneous record and playback instrumentation can be used to improve program testing.

Program 5 - The instrumented `Channel` class

```
1  public class Channel extends Selectable {
2      Object chan_ = null ;
3      public void send (Object v) throws InterruptedException {
4          {
5              final Object OOO_0_ = this;
6              RRTools.Replay.beforeEnterSyncBlock(OOO_0_,"send-4");
7              synchronized (OOO_0_) {
8                  RRTools.Replay.afterEnterSyncBlock(OOO_0_,"send-3");
9                  try {
10                     chan_ = v;
11                     signal();
12                     while (chan_ != null)
13                         RRTools.RR.Wait(this,"send-1");
14                 } finally {
15                     RRTools.Replay.exitSyncBlock(OOO_0_,"send-2");
16                 }
17             }
18         }
19     }
20
21     public Object receive () throws InterruptedException {
22          final Object OOO_1_ = this;
23          RRTools.Replay.beforeEnterSyncBlock(OOO_1_,"receive-4");
24          synchronized (OOO_1_) {
25              RRTools.Replay.afterEnterSyncBlock(OOO_1_,"receive-3");
26              try {
27                  block();
28                  clearReady();
29                  Object tmp = chan_;
30                  chan_ = null;
31                  RRTools.RR.NotifyAll(this,"receive-1");
32                  return (tmp);
33              } finally {
34                  RRTools.Replay.exitSyncBlock(OOO_1_,"receive-2");
35              }
36          }
37     }
38 }
```

4 Behavior Generation

We commented that this model of instrumentation – in particular for the recording phase – can collect more information than that essential for R&P. This information, such as releasing of a lock, calls to notify, and others, may be used by a tool to support other activities such as: 1) a friendly visualization of a synchronization sequence by the user; and 2) automatic generation of other valid synchronization sequences for the same program.

This second characteristic together with the ability to use a prefix of a synchronization sequence to execute the program under analysis may improve software testing. Many times the tester faces the problem of trying to exercise some features in the program under test but is not able because the execution system keeps choosing the same scheduling

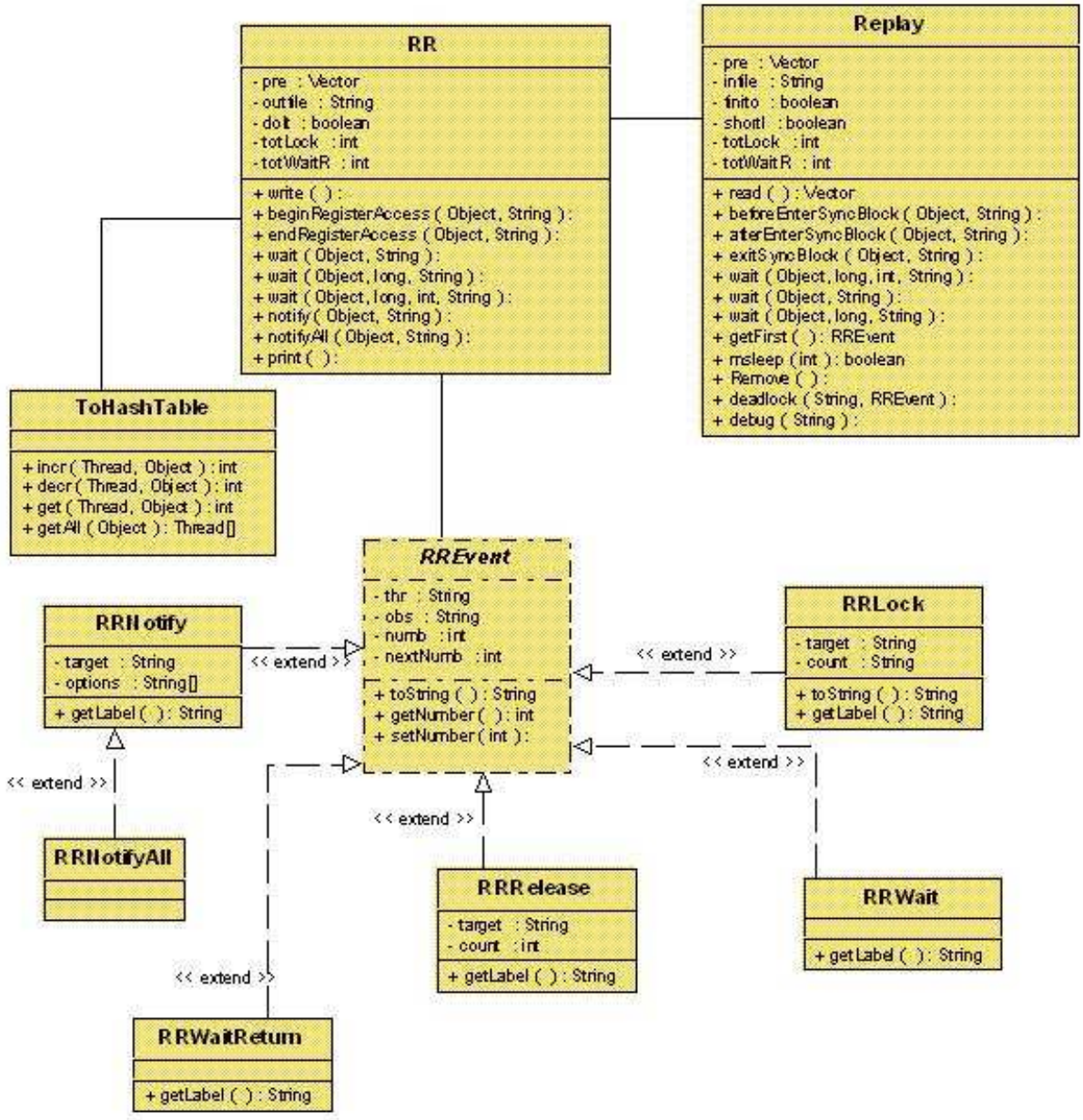

Figure 1: Package `RRTools`

sequences, that does not lead to the desired behavior. This problem is addressed in the work of Offut et al. [5] and commented in Section 6.

With our approach it is possible to implement a tool that analyzes a given synchronization sequence and creates valid alternative synchronization sequences. For example, let us take the following piece of a synchronization sequence:

- Thread T_1 locks object X_1
- Thread T_1 calls X_1.notify(). The wait set of X_1 at this point contains:

  Thread T_2 included by a non-timed wait

  Thread T_3 included by a non-timed wait

Program 6 - A summary of class `RRTools.Replay`

```
1  package RRTools;
2  public class Replay {
3  static String inFile = System.getProperty("RW");
4
5       static {
6         // load inFile
7       }
8
9       /** checks if the event is the one expected
10      deals with entering a synchronized block */
11      static public void beforeEnterSyncBlock(Object x, String obs) {
12         ...
13      }
14
15      static public void afterEnterSyncBlock(Object x, String obs) {
16          RR.beginRegisterAccess(x, obs);
17          Remove();
18      }
19
20      static public void exitSyncBlock(Object x, String obs) {
21        ...
22      }
23
24      /** replaces a wait call */
25      static public void Wait(Object x, long mili, int nano, String obs)
26                throws InterruptedException {
27        ...
28      }
29
30      /** replaces a wait call */
31      static public void Wait(Object x, long mili, String obs)
32                throws InterruptedException {
33         ...
34      }
35
36      /** replaces a wait call */
37      static public void Wait(Object x, String obs)
38                throws InterruptedException {
39         ...
40      }
41 }
```

- Thread T_1 releases X_1
- Thread T_2 locks X_1

It suggests that the call to the `notify` takes T_2 off the wait set allowing that thread to lock X_1 in the sequence. Using this synchronization sequence up to this point and replacing the last event by T_3 locking X_1 may be an interesting prefix that may cause a different behavior in the rest of the execution, leading to exercise untested features in the program.

Currently our system does not have such an analysis tool but the recording instrumentation collects much of the information required to implement this kind of analysis. It includes:

- entering a synchronized method or synchronized block;
- leaving a synchronized method or synchronized block;
- call to a `wait` method (and inclusion of the thread in the wait set);
- return from a `wait` call (and remotion from the wait set);
- call to `notify` and `notifyAll` and verification of the threads in the wait set.

5 Limitations

As stated before, the technique presented herein aims at Java multi-thread programs in which each thread is deterministic. It means that the only kind of non-determinism that may arise is from the way the threads are scheduled in the JVM.

In addition, it supposes a well-behaved program, in which every access to shared objects is done by using the constructs of the language in a determined sequence. Methods to control thread execution such as `Thread.interrupt` or `Thread.stop` may confuse the R&P system and make it fail.

Another problem, as mentioned before, is the use of synchronized methods from libraries that are not part of the program under analysis. For example, the class `java.util.Vector` defines its methods as synchronized. Since the class cannot be instrumented, it is not possible to control the order of accesses from multiple threads. The best the instrumentation system can do is to warn the programmer and to indicate the places where such classes are used. The programmer can choose to insert synchronized blocks at those points. Such blocks, although redundant, permit the instrumentation to work properly.

In the replay phase, the instrumented code has to identify which thread matches the next event in the input synchronization sequence. So, there must be a correspondence between a thread in the original run and a thread in the re-execution. Unfortunately, there is no safe way to do so, other than by the name of the thread. The Java runtime system allows the creator of a thread to name it. If it is not named, the runtime system will give it a name, based on its thread group and the sequential order it is created. Since the order of creation of the threads may not be the same, it is not guaranteed the same name is given in two different executions. In order to allow the instrumentation system to associate the threads of two different executions, it is necessary that every thread is explicitly named by the creator, so they have the same name in different runs.

6 Related Work

The use of instrumentation to control multi-process program synchronization has been used before. Carver and Thai [1] proposed the technique called "deterministic execution" where they describe how to instrument concurrent programs that use either monitors or semaphores as the synchronization model. According to the authors their technique can be extended to other synchronization models and languages.

The most commented system for applying R&P to Java found in the literature is probably DejaVu [2]. It implements a completely different approach. It was developed as an extension to the Sun's Java Virtual Machine and can be used on single processor systems, as well as on multiprocessor systems. The JVM can be started in one of two modes: 1) the record mode wherein the tool records information about logical thread scheduling; and 2) the replay mode in which the tool reproduces the execution behavior, enforcing the same logical thread schedule.

Silva-Barradas [6] uses this approach to record the synchronization sequence of Ada programs. The approach presented herein is similar to those of Silva-Barradas but requires a completely different instrumentation strategy since the concurrency and synchronization mechanisms of Java are sensibly diverse from those of Ada.

Our R&P system has been used to support the application of mutation testing for Java concurrent programs [3, 7]. It is used to record the execution of the original program and to use the synchronization sequence obtained to guide the execution of the mutants, trying to make them behave as the original program. A mutant is considered dead if it is not able to follow the original synchronization sequence or if it follows the original synchronization sequence but presents a different behavior. This approach has been proposed by Silva-Barradas [6] for Ada programs.

Offut et al. [5] have also tried to use mutation testing in concurrent (Ada) programs. The authors proposed a technique that calculates an approximated set Ω, defined as a subset of all possible results of $P(t)$. This set is calculated by executing $P(t)$ several times. The mutant M is distinguished if it produces at least one result $M(t) \notin \Omega$. One problem of their approach is the fact that in a given environment (operating system or in this case the Ada runtime system) the repeated execution of the program under test might not create several different behaviors because the system tends to take the same actions on choosing tasks to execute. With the mutation it is possible that those choices change and the mutant will produce a different (although correct) result and be erroneously considered distinguished. A possible solution would be to use our R&P system to record the first execution of $P(t)$ and then generate alternative synchronization sequences in the hope of creating alternative behaviors and so expanding the set Ω.

7 Final Remarks

Testing and debugging of concurrent programs present additional difficulties to the testing and debugging of sequential programs. One of them is the non-determinism that makes harder to reproduce the behavior of a program execution. Even if the concurrent processes are deterministic, non-determinism can arise from the possible different sequences of process scheduling.

Unfortunately, most of the operating systems and runtime systems – included the diverse versions available of the Java Virtual Machine – do not offer any support to the tester facing this problem. Thus, the tester has to find alternative approaches in order to be able to adequately exercise his/her programs. One exception is the work with DejaVu, which extends the JVM with R&P capabilities [2].

In this paper we presented an approach to reproduce the behavior of a Java multi-thread program. The program under analysis (under testing or debugging) is instrumented in such a way that the synchronization points (where the program accesses shared objects) are recorded. In a second execution this instrumented program can use this synchroniza-

tion sequence to force the same sequence of accesses to the shared objects, to produce the same behavior. In addition, the analysis of a given synchronization sequence created in a particular execution can help to create new synchronization sequences and possibly new behaviors.

The technique does not apply to concurrent Java programs in general, relying on some constraints in the program to be tested. Even so, we believe that, given its simplicity and easy of use and implementation, it may be helpful to support the testing and debugging activity for many real-world applications. In a case study [7], it was used to support the application of mutation testing for Java concurrent programs, including part of a commercial real-time monitoring system.

Acknowledgements

The author would like to thank CNPq and Fundação Araucária for partially supporting the work reported herein and Prof. Mario Jino for reviewing this text.

References

[1] R. H. Carver and K. Tai. Replay and Testing for Concurrent Programs. *IEEE Software*, 8(2):66–74, March 1991.

[2] J.D. Choi and H. Srinivasan. Deterministic replay of java multhithread applications. In *ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT)*, pages 48–59, August 1998.

[3] M. E. Delamaro, M. Pezzè, A. M. R. Vincenzi, and J. C. Maldonado. Mutant operators for testing concurrent Java programs. In *SBES'2001 – Simpósio Brasileiro de Engenharia de Software*, pages 272–285, Rio de Janeiro, RJ, October 2001.

[4] J. McGee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley and Sons, 19919.

[5] A. J. Offutt, J.M. Voas, and J. Payne. Mutation Operators for ADA. Technical Report ISSE-TR-96-09, Department of ISSE, George Mason University, Fairfax, VA, March 1996.

[6] S. Silva-Barradas. *Mutation Analysis of Concurrent Software*. Phd thesis, Department of Eletronic and Informatics, Polythecnic of Milan, Milan, Italy, 1997.

[7] E. Spreafico. *Experimental Evaluation of a Technique to Test Concurrent Java Programs (in Italian)*. Undergrad. final work, Facoltà Di Informatica - Università degli Studi di Milano, Milan, Italy, 2001.