Testing Concurrent Java Programs using Randomized Scheduling

Scott D. Stoller[1]

Computer Science Dept., State University of New York at Stony Brook, USA

Abstract

The difficulty of finding errors caused by unexpected interleavings of threads in concurrent programs is well known. Model checkers can pinpoint such errors and verify correctness but are not easily scalable to large programs. The approach discussed here is more scalable but less systematic. We transform a given Java program by inserting calls to a *scheduling function* at selected points. The scheduling function either does nothing or causes a context switch. The simplest scheduling function makes the choice blindly using a pseudo-random number generator; more sophisticated scheduling functions use heuristics to weight the choices. We try to insert as few calls as possible while still ensuring that for each reachable deadlock and assertion violation, there is a sequence of choices by the scheduling function that leads to it; thus, there is a non-zero probability of finding it by testing the transformed program, regardless of the scheduling policies of the underlying Java Virtual Machine.

1 Introduction

The difficulty of finding errors caused by unexpected interleavings of threads in concurrent programs is well known. This has prompted the development of powerful tools to help find such errors. Motivated by the increasing popularity of concurrent programming in Java, we focus here on tools that can be applied to programs written in Java, although similar tools exist for other languages.

The Java Language Specification [15] provides essentially no guarantees about the thread scheduler. Therefore, for a concurrent Java program to be robust in the face of varying load conditions (which can perturb the scheduling) and portable across all operating systems and Java run-time environments (which may have different thread scheduling policies), it must work

[1] The author gratefully acknowledges the support of NSF under Grant CCR-9876058 and the support of ONR under Grants N00014-01-1-0109 and N00014-02-1-0363. Email: stoller@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~stoller/

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

correctly despite arbitrary interleavings of the instructions executed by different threads.

Model checkers are powerful tools that can help pinpoint errors in concurrent programs and, if the program's state space is not too large, can verify correctness. Model checkers aim to control all non-determinism (including non-determinism in scheduling) and exhaustively explore the system's possible behaviors. Existing model checkers for Java include the original Java PathFinder [16], the second generation of Java PathFinder [3], which builds in part on ideas from Spin [18], and JavaChecker [24], which builds in part on ideas from VeriSoft [12]. The systematic and exhaustive nature of model checkers is attractive, but it limits their scalability.

This paper describes work-in-progress that explores a technique that is more scalable but less systematic. The approach is to insert calls to a *scheduling function* at selected points in the program under test. The scheduling function either does nothing or causes a context switch. The simplest scheduling function makes the choice blindly using a pseudo-random number generator. The pseudo-random number generator is seeded at the beginning of the execution. Normally, the seed is based on the current time and is logged. More sophisticated scheduling functions combine randomness with heuristics that weight the choices. The heuristics may consider the current state alone or the history as well. History is stored information about the current execution and possibly previous executions. The scheduling function may cause a context switch by invoking `Thread.yield` or `Thread.sleep`.

The transformed program is executed repeatedly to test it. If an error is found with a particular seed, the program is re-executed with the same seed to help reproduce the error. With this simple approach, reproducibility is likely but not guaranteed. Guaranteed reproducibility requires a capture-and-replay mechanism, such as [5].

We implemented this approach in a tool called rstest (random scheduling test). To help demonstrate its scalability, we are applying it to ArgoUML [1], an open-source UML-based graphical software design environment. ArgoUML consists of 4 MB of class files plus 7 MB of libraries (not including the standard Java run-time library). Within a fairly short time, we found what is apparently a concurrency-related error. The experiment is described in Section 7.4. More work is needed to evaluate the effectiveness of this approach at finding errors in such large systems. We also applied it to some smaller systems with known errors, which were easily revealed by rstest.

ConTest and ConTest-lite [9,8] are very similar to rstest. We designed and implemented rstest in about a man-month before learning of their work. ConTest was developed by a team at the IBM Haifa Research Laboratory during the past 2 to 3 years. Naturally, ConTest is a much more mature and comprehensive system. For example, ConTest contains a capture-and-replay mechanism and a deadlock detection component, and it uses a novel coverage metric in the scheduling function. These useful features are currently not implemented in rstest but could be added.

A distinguishing feature of our design is the method for selecting the program points at which calls to the scheduling function are inserted. Compared to ConTest, it inserts fewer calls to the scheduling function, without reducing the probability of finding a wide range of errors, and without compromising probabilistic completeness. The *probabilistic completeness* property is: for each reachable deadlock and assertion violation, there is a sequence of choices by the scheduling function that leads to it. Thus, there is a non-zero probability of finding it by testing the transformed program, even if the probability of finding it by testing the original program in the same run-time environment is zero, due to the particular thread scheduler in that run-time environment. We assume only that the thread scheduler is fair. Reducing the number of calls to the scheduling function has two benefits. First, it reduces the slowdown caused by calls to the scheduling function. Second, it reduces the average number of context switches in counterexamples (*i.e.*, executions in which errors occur) produced by the tool. This makes the counterexamples easier to understand.

Another feature of our design is the use of a loop in the scheduling function, rather than a conditional, so that a single call can produce multiple context switches. [2] Without the loop, if yield is used to cause context switches, then the technique lacks probabilistic completeness. If sleep is used to cause context switches, a similar issue arises and can be resolved by using a loop or randomizing the duration of the sleep.

Many model checkers, including Spin [18] and Java PathFinder [3], have a random testing mode that is similar in functionality to rstest, but the differences in implementation strategy have significant practical consequences: applying Spin or Java PathFinder to an application the size of ArgoUML would require significant manual effort to develop a manageable abstraction of the system, while rstest can be applied easily.

VeriSoft [12] and JavaChecker [24] are model checkers that perform systematic state-less search [12]. They are similar in implementation strategy to rstest: all three tools insert calls to a scheduler into a given program. VeriSoft has successfully been applied to industrial-size systems (*e.g.*, [13]). VeriSoft is designed to control non-determinism in multi-process systems. Typically, identifying inter-process communication statements is straightforward, and inserting calls to the scheduler at all of them is reasonable. rstest is targeted at (uni-process) multi-threaded programs. Identifying shared storage locations is non-trivial, and inserting calls to the scheduler at every access to every shared location would typically introduce significantly higher overhead than inserting those calls only at the places identified by our approach. JavaChecker is also targeted at multi-threaded systems, but it is more complicated and incurs

[2] Last-minute note: ConTest also does this (Shmuel Ur, private communication), although this is not mentioned in [9,8].

higher overhead than rstest, because it completely controls the scheduling in order to perform systematic search.

Future work abounds. The most significant tasks are to finish implementing the design described in Section 3, incorporate heuristics into the scheduling function, and perform more experiments to evaluate the effectiveness of the approach. In practice, heuristics [9,7,14] are crucial for effectively finding errors in large systems. Although our current implementation does not use any heuristics, our approach to reducing the number of calls to the scheduling function is compatible with the use of heuristics.

The rest of the paper is organized as follows. Section 2 provides background on synchronization in Java bytecode. Section 3 describes how to select the program points at which calls to the scheduling function are inserted. Section 4 discusses the scheduling function. Section 5 states the probabilistic completeness property. Section 6 describes the status of the implementation. Section 7 describes some preliminary experimental results.

2 Overview of Synchronization in Java Bytecode

The primitive synchronization operations in Java bytecode are based on the classic operations on monitors [20]. In effect, each object in Java implicitly contains a unique lock and a unique condition variable. The locks are recursive, *i.e.*, a thread can re-acquire a lock that it holds, and a lock is free iff each acquire of it has been matched by a release.

`monitorenter` and `monitorexit` are instructions that acquire and release, respectively, the lock associated with a specified object.

An invocation of a method declared as synchronized implicitly acquires the lock associated with the target object (*i.e.*, the `this` argument). Exiting from such an invocation implicitly releases the lock.

`Thread.holdsLock(`o`)` is a static native method that returns true iff the current thread holds o's lock. This is a new feature in JDK 1.4.

`wait`, `notify`, and `notifyAll` are final native methods of `Object`. They are inherited by all objects. They throw `IllegalMonitorStateException` if invoked by a thread that does not own the target object's lock; otherwise, they behave as follows. o`.wait()` adds the calling thread t to o's wait set (*i.e.*, the set of threads waiting on o), releases o's lock, and suspends t. When another thread notifies t, t contends to re-acquire o's lock. When t acquires the lock, the invocation of o.wait() returns. o`.notify()` non-deterministically selects a thread t in o's wait set, removes t from the set, and notifies t; if o's wait set is empty, o`.notify()` has no effect. o`.notifyAll()` removes all threads from o's wait set and notifies each of them. A thread t waiting on an object can also be awoken by an invocation of t`.interrupt()`, where `interrupt` is a method of `java.lang.Thread`.

There are bounded-time variants of `wait` that time-out if the waiting thread is not notified within a specified time interval.

3 Where to Call the Scheduling Function

ConTest inserts calls to the scheduling function at all *concurrent events*, which are, by definition, the events whose order determines the result of the program. Specifically, all synchronization operations (described in Section 2), all object access instructions, and all `getstatic` and `putstatic` instructions (which read and write static fields of classes) are treated as concurrent events. The *object access instructions* are `getfield` and `putfield` (which read and write instance fields of objects) and the *array access instructions* (Java bytecode has several instructions for accessing elements of arrays; different instructions are used for different types of elements). Furthermore, ConTest "executes sleep()s before and after concurrent events" [8, section 5.1].

Static analysis can be used in straightforward ways to reduce the number of inserted calls to the scheduling function. For example, we use an escape analysis and do not insert calls to the scheduling function at accesses to objects that (according to the analysis) have not yet escaped from the thread that created them. This does not reduce the probability of detecting assertion violations and deadlocks. We designed and implemented a simple escape analysis for this purpose; more sophisticated escape analyses, such as [4,25], could be used instead.

The rest of this section describes a more powerful approach to reducing the number of inserted calls to the scheduling function. Storage locations are classified as unshared, protected, or unprotected (defined below). This provides a basis for classifying operations as visible or invisible. The visible operations (except class initialization) are a subset of ConTest's concurrent events. Calls to the scheduling function are inserted immediately before visible operations. This suffices for probabilistic completeness, discussed in Section 5. Fewer calls to the scheduling function are inserted, because the calls are: (1) not inserted before `monitorexit`, returns from synchronized methods, or `notifyAll`, (2) inserted before only *some* occurrences of the other instructions that ConTest treats as concurrent events, and (3) not inserted *after* any of the instructions that ConTest treats as concurrent events.

This approach is only partially implemented. Our current implementation is described in Section 6.

Classification of Storage Locations.

A storage location is *unshared* if it is accessed by at most one thread. In Java, all variables (locals and parameters) are unshared. Heap locations (*i.e.*, instance fields of objects) and static locations (*i.e.*, static fields of classes) may be shared or unshared.

An (heap or static) location x is *protected* (by a lock) if either (1) every access to x after initialization is a read, or (2) there exists a lock ℓ such that, for every access to x after initialization, ℓ is held by the accessing thread when the access occurs. The definition of "initialization" is based on the principle that initialization of a location x must end before concurrent access to x by multiple threads is possible. Specifically, we define initialization as follows. Initialization of a heap location x ends when a reference to x escapes from the thread that allocated x. Initialization of a static location x in class C (*i.e.*, x is a static field of C) ends when the class initializer for C terminates. Recall that the JVM automatically provides synchronization for class initialization to ensure that other threads cannot access x until the class initializer for C terminates [15, Section 12.4].

This definition of "protected" is based closely on the Eraser locking discipline [21], except that we use a different definition of initialization, because Eraser's definition would not ensure probabilistic completeness.

The *unprotected* category is intended for storage locations that are shared and not protected. However, any location can safely be classified as unprotected, *i.e.*, the probabilistic completeness results in Section 5 still hold. As will become evident, to minimize the number of calls to the scheduling function, locations should be classified as unshared or protected whenever possible.

Java's primitive synchronization operations are treated specially in the following classification of operations, so the above classification of storage locations is not applied to storage locations accessed by those operations (*e.g.*, the location that indicates which thread holds an object's lock).

Classification of Operations.

An operation is *visible* if it (1) accesses an unprotected location, or (2) is a potentially blocking synchronization operation or a call to `Thread.interrupt`, or (3) is non-deterministic.

Operations that possibly access unprotected locations are object access instructions, `getstatic`, and `putstatic`.

The potentially blocking synchronization operations are `monitorenter`, invocations of synchronized methods, two operations in `wait` (the initial blocking operation, and the operation that re-acquires the lock after the thread has been notified), and operations that may implicitly cause a class to be initialized (specifically, in states in which class C is not fully initialized, all operations that access class C are visible, namely, accesses to static fields of C, invocations of static methods of C, creation of instances of C, and invocations of some methods in `java.lang.Class` and `java.lang.reflect`).

Informally, class initialization is visible because it involves acquiring a lock, namely, the lock associated with the class. That lock protects a shared storage location containing the initialization status of the class (*e.g.*, "initialization in progress by thread t" or "fully initialized").

The only operation that we regard as non-deterministic is `notify`.

It might seem surprising that `notifyAll` is invisible and `Thread.interrupt` is visible, since both operations are deterministic and have similar effects on a waiting thread. The root of the difference is that o`.notifyAll` is effective only if the calling thread holds o's lock, while `Thread.interrupt` has no such constraint.

`Thread.holdsLock`, the new synchronization primitive in JDK 1.4, is invisible. Intuitively, this is because execution of other threads cannot affect its return value.

3.1 The Transformation

The transformation is parameterized by a classification of objects into the three above categories. Objects are classified by their type. Specifically, the transformation is parameterized by a list of unshared classes (*i.e.*, all instances of these classes are classified as unshared) and a list of protected classes (*i.e.*, all instances of these classes are classified as protected). Other classes are treated as unprotected by default. Section 3.2 describes how these lists are obtained.

Based on this classification of objects, a call to the scheduling function is inserted before each visible operation. For most bytecode instructions, it is easy to determine statically whether the instruction performs a visible operation. However, due to dynamic method dispatch, it is impossible (in general) to determine statically whether an invocation instruction invokes a synchronized or unsynchronized method. [3] This is not a big problem, because inserting a call to the scheduling function before an invisible operation is harmless, except for the overhead. So, a call to the scheduling function is inserted before each "`invokevirtual` $C.m$" instruction such that $C.m$ or any method that overrides it is synchronized, where m denotes the name of a method and its type signature. Similarly, a call to the scheduling function is inserted before each "`invokeinterface` $I.m$" instruction such that some class C implements I and has a synchronized method $C.m$.

3.2 Obtaining the Classification of Objects

The classification of objects can be obtained through static analysis, run-time monitoring, or a combination of the two.

Static Analysis.
Escape analysis can be used to conservatively determine which objects are unshared. Type systems for race-free programs [10,2] are designed to show that locks specified in type annotations protect certain objects. The soundness theorem for the type system states, roughly, that in a well-typed program, all objects are unshared or protected. It is not difficult to generalize this result to state that, if parts of the program are well-typed, then (all instances of) certain classes are unshared or protected.

[3] This difficulty could be avoided by inserting a call to the scheduling function at the beginning of each synchronized method, rather than at call sites. But then the call to the scheduling function occurs after, not before, the visible operation. This does not satisfy the hypotheses of the probabilistic completeness result in Section 5.

In general, the user supplies type annotations, which are automatically checked. By a combination of well chosen defaults and trial-and-error type inference [11], even with few or no manually supplied type annotations, most of a program will typically pass the type checker, allowing many of the classes that are unshared or protected to be classified as such.

Run-Time Monitoring.
Run-time monitoring can also be used to obtain, by iterative refinement, a probabilistically correct classification. (Correctness is discussed in Section 5.) The initial classification may be arbitrary; a simple choice is to initially classify all classes as unshared. The program is transformed to check for violations of the classification, as described below, in addition to the insertion of calls to the scheduling function. If a violation of the classification occurs in any execution, the classification is automatically modified to eliminate it. A violation involving an unshared class C causes C to be re-classified as protected. A violation involving a protected class C causes C to be re-classified as unprotected. Then the program is re-transformed, and testing resumes.

To check for violations of the classification of class C as unshared, each object access instruction that accesses an object o of class C is instrumented with a call to a static method `checkUnshared(`o`)`, which maintains a hash map `firstAccessed` that maps each object reference o to the first thread that accessed o. An entry that maps o to `Thread.currentThread()` is created in h when o is first accessed. Subsequent accesses to o check whether `Thread.currentThread()` equals `firstAccessed.get(`o`)`; if not, a violation is reported. `firstAccessed` should be a weak hash map (see `java.util.WeakHashMap`); this means that an entry for o in the hash map does not prevent o from being garbage collected. It should also be an *identity* hash map (see `java.util.IdentityHashMap` in JDK 1.4); this means that during look-ups, equality tests are done with `==`, not `equals`. Relying on `equals` would be inappropriate and dangerous, because application programmers can override it with arbitrary code.

To check for violations of the classification of a class as protected, the lockset algorithm [21] may be used. It was designed to detect data races. It maintains, for each object o, the set of locks that have been held at all accesses to o so far during execution, excluding accesses during initialization. To accommodate our definition of initialization, escape analysis is used to determine when a reference to a newly created object can escape from the creating thread, and code is inserted at appropriate points to change the state maintained by the lockset algorithm for the object from initialization to post-initialization. Note that the lockset algorithm is run only for instances of classes that are classified as protected.

Combining Static Analysis and Run-Time Monitoring.
Type systems for race-free programs are attractive, because they incur no run-time overhead in the transformed program. However, the type systems are incomplete (*i.e.*, programs may contain protected classes that cannot be verified as such by the type checker), especially in the absence of manually supplied type annotations. This suggests the following hybrid approach. Use a type system (or other static analysis) to identify some classes that are definitely unshared or definitely protected; run-time monitoring is unnecessary for these classes. Use iterative refinement based on run-time monitoring to obtain a probabilistically correct classification for the remaining classes.

3.3 Discussion

Use of static analysis to reduce the number of inserted calls to the scheduling function appears to have no disadvantages. For run-time monitoring, it is unclear whether the benefits outweigh the overhead. The answer is more likely to be positive for monitoring whether a class is unshared, because it has significantly less overhead than the lockset algorithm. Running the lockset algorithm for the sole purpose of reducing the number of inserted calls to the scheduling function is unlikely to be worthwhile, but the lockset algorithm is often run for other reasons, in which case we may exploit it for this purpose, too. For example, ConTest includes a race detection component [8, Section 3] and tries to force operations involved in races to execute in a different order in subsequent executions.

4 Design of the Scheduling Function

The semantics of Java does not constrain which thread runs next after a context switch. Therefore, probabilistic completeness of the approach requires that a call to the scheduling function has a non-zero probability of transferring control to each runnable thread. Depending on the JVM used during testing, a single call to yield might not achieve this.

For example, consider a program with three threads. Let a1, b1, a2, and a3 denote accesses to a shared storage location. Let sched1a, sched2, *etc.*, denote calls to the scheduling function (the suffixes “1a”, *etc.*, are not part of the actual call; they are included only to provide a distinct name for each call site, for use in the figure below). The main thread, t1, starts the other two threads. Their actions are:

Fig. 1. Possible schedules of the sample program, under the stated assumptions about the JVM's scheduler. Values of the ready list are shown at selected points.

thread t1	thread t2	thread t3
t2.start	sched2	sched3
sched1a	a2	a3
a1		
t3.start		
sched1b		
b1		

Suppose each invocation of the scheduling function randomly performs zero or one calls to yield. Suppose the scheduler of underlying JVM is round-robin; thus, it maintains a list of threads that are ready to run (the "ready list"), and when the current thread blocks or uses up its current quantum of CPU time, the scheduler puts the current thread at the tail of the list and runs the thread at the head of the list. Suppose further that the scheduler puts newly started threads at the head of the ready list, and that its scheduling quantum is sufficiently large that it does not cause context switches during execution of this short program. Figure 1 shows all of the possible schedules for this program under these assumptions. There is a schedule that is possible according to the semantics of the language but cannot occur under these assumptions, namely, the schedule in which accesses to the shared location occur in the order a1, a2, a3, b1.

If yield is used to cause context switches, an easy way to avoid this problem is to use a scheduling function in which the call to yield is inside a loop, *e.g.*,

```
static java.util.Random prng = ...;
static float contextSwitchProb = ...;

public static void schedFn() {
  while (prng.nextFloat() < contextSwitchProb) Thread.yield();
}
```

where `prng` is a pseudo-random number generator, and `contextSwitchProb` is a parameter that determines the probability of performing a context switch. It is easy to see that, after adding these loops, each runnable thread has a non-zero probability[4] of being the next one to exit from the loop, assuming the thread scheduler is fair. Continuing the above example, after a1 and a2 execute, t1 might execute the loop in the scheduling function a second time, thereby performing an additional context switch that allows a3 to occur before b1.

If sleep is used to cause context switches, a similar issue arises and can be resolved by using a loop or randomizing the duration of the sleep.

5 Probabilistic Completeness

The following results are corollaries of the reduction theorems in [24].

Suppose the classification of objects is correct (*e.g.*, it is ensured by static analysis). Consider a program transformed as described in Section 3.1. For each reachable deadlock and assertion violation, there is a sequence of choices by the scheduling function that leads to it. This implies that there is a non-zero probability of finding it by testing the transformed program,

Suppose correctness of the classification of objects is not ensured by static analysis. Consider a program transformed as described in Section 3.1 to contain calls to the scheduling function and to monitor for violations of the classification. Probabilistic completeness holds for finding mis-classifications as well as deadlocks and assertion violations. Specifically, if there is a reachable state in which the classification has been violated, then there is a sequence of choices by the scheduling function that leads to a state in which the classification has been violated. When a mis-classification is found, it should be corrected, because there may be other mis-classifications that will not be discovered until that is done and the program is re-transformed.

These results rely only on context switches caused by the scheduling function. The thread scheduler of the underlying run-time system may cause additional context switches at any time. We assume only that the thread scheduler is fair.

These results do not apply to programs that (before transformation) use real-time primitives, such as `Thread.sleep`, bounded-time versions of `Thread.wait`, and `System.currentTimeMillis`. In practice, there is no obstacle to applying the tool to such programs. This is likely to be useful if the overhead of the instrumentation does not perturb the timing too much. Our design reduces the number of calls to the scheduling function and hence the overhead, making the approach more useful for such systems.

[4] With this simple change to the scheduling function, these probabilities are not equal. This can be remedied, at the cost of some complication.

6 Implementation

rstest is implemented as bytecode transformations using the Byte Code Engineering Library [6]. In retrospect, the Soot compiler infrastructure [22] would probably have been a better choice. Soot's intermediate representation is easier to analyze and manipulate than bytecode. For example, with BCEL, it is a hassle to insert calls to a scheduling or monitoring function that takes the object being accessed as an argument, because a reference to that object is not necessarily on top of the operand stack immediately before the access (*e.g.*, for an invocation of a synchronized method, the object whose lock is being acquired is on the operand stack below the other arguments to the method). With Soot, adding such invocations should be easy.

In the current implementation, every object must be treated as unshared or unprotected. Our tool inserts calls to the scheduling function at visible operations (except at operations that may cause class initialization; this remains to be implemented) and monitors for violations of the classification of objects as unshared. Based on a simple escape analysis, it usually avoids inserting calls to the scheduling function at accesses to the `this` argument in constructors.

Support for treating objects as protected is future work. We plan to integrate a type system for race-free programs [2] and perhaps the lockset algorithm (the easiest way would be to adapt the implementation in Java PathExplorer [17]).

7 Experimental Results

For each application, we started with every class treated as unshared. If rstest found a violation of this, the offending class was re-classified as unprotected. Only application classes were transformed, not classes in the Java run-time library. Unless otherwise noted, the scheduling function contains a loop like the one shown in Section 4, except with sleep instead of yield, with a sleep time of 1 millisecond and a context switch probability of 1/8. Most of the experiments, including all those with reported running times, were performed with Sun JDK 1.3 on a Sun Ultra10 with 440MHz UltraSPARC CPU; a few were performed with Sun JDK 1.3 on Windows XP. Reported running times are user+system time, unless otherwise noted. Counts of "lines of code" exclude blank lines and comments.

7.1 Clean

The Clean example is based on code in NASA's Remote Agent and involves two threads that use bounded counters, synchronized methods, `wait`, and `notifyAll`. The code is 57 lines and is roughly the same as in [3, Figure 1]. The threads run in infinite loops, repeatedly interacting, but a context switch at an inopportune time leads to deadlock. Without rstest, no deadlock occurred in 10 minutes of execution. With rstest, the deadlock occurred after 0.5 seconds (this is the average for 10 runs).

7.2 Fund Managers

The fund mangers example is from [19]. It involves multiple "fund manager" threads that repeatedly transfer money between accounts. These transfers should not change the total amount of money. However, if a context switch occurs at an inopportune moment during a transfer, money can disappear or be created. The code is 123 lines. We made two small changes to the code: we reduced the number of fund managers from 150 to 2, because it is desirable to test and debug with a small configuration before doing stress tests, and we removed the call to `Thread.setPriority`, which was originally included only as an artificial device to make the error more likely to manifest itself. Each execution of the program performs a few thousand transfers. Without rstest, the bug never manifested itself in 10 executions. With rstest, the bug manifested itself many times in each execution.

7.3 Xtango Animation Library

The Xtango animation library [23] provides methods for drawing geometric objects and text and moving them around. We used S. Hartley's implementation, which is about 1300 lines of code. It comes with a demo program that, among other things, creates two circles $c1$ and $c2$ and calls `exchangePosAsync(`$c1$, $c2$`)` and then `exchangePos(`$c1$, $c2$`)`. Both methods cause the two circles to exchange position by sliding across the screen. `exchangePosAsync` creates a separate thread to slide them while the original thread proceeds to the next command. `exchangePos` lets the calling thread do the work.

Exchanging positions is conceptually a symmetric operation, so a naive user might just as well have written `exchangePos(`$c2$, $c1$`)` as the second command. This can cause deadlock, because `exchangePos` and `exchangePosAsync` both lock the first argument and then the second argument. We made this change to the demo program. Without rstest, deadlock did not occur in 200 executions. With rstest, deadlock occurred about once per 17 executions. The same result was obtained with sleep times of 1 millisec and 2 millisec.

We also applied rstest to the animations of quicksort and dining philosophers that come with Xtango. They contain about 230 and 340 additional lines of code, respectively. The perturbation to the scheduling caused some transient peculiarities in the animation of quicksort, which draws a sequence of boxes, normally erasing each box before drawing the next one. With rstest, erasure of one edge of a box would sometimes be delayed until after the next box had been drawn. The dining philosophers behaved normally.

The inserted code had a noticeable effect on the speed of the animation. For quicksort, elapsed (*i.e.*, wall clock) time for one execution increased from about 7 seconds for the original program to 31 and 37 seconds with sleep times of 1 and 2 millisec, respectively. These elapsed times are for programs transformed to call the scheduling function but not to monitor for violations of the classification of unshared objects.

Code inserted to monitor for violations of the classification of unshared objects caused a slowdown of about 15% in both Xtango and ArgoUML.

7.4 ArgoUML

ArgoUML [1] is an open-source UML-based graphical software design environment. The core of ArgoUML 0.10 is 4 MB of class files, which use the GEF Graph Editing Framework (0.8 MB of class files) and libraries for parsing and data interchange (6 MB of class files). We do not currently have access to appropriate software for capturing and replaying user input, so we tested the transformed program with semi-random manual inputs. We used `yield` in the scheduling function, because `yield` caused significantly less slowdown than `sleep`.

First, we transformed only the core classes. About 30 minutes of input produced only some warning messages that also occurred in the original program. Next, we transformed GEF as well. Less than 5 minutes of input led to

```
java.lang.ClassCastException: org.tigris.gef.presentation.FigRect
at org.argouml.uml.diagram.static_structure.ui.FigClass.createFeatur
eIn(FigClass.java:716)
[ several lines omitted ]
at java.awt.EventDispatchThread.run(EventDispatchThread.java:85)
```

ArgoUML caught the exception and continued, although mouse clicks in the graph editing window did not work correctly for a while. The error did not occur in about 30 minutes of similar input to the original program. This suggests (but is far from conclusive) that the error is due to rstest's perturbation to the scheduling. A capture-and-replay mechanism for user input is needed for more systematic experiments.

Acknowledgments.
I thank Han Li for implementing most of rstest, Leena Unnikrishnan for implementing the escape analysis, and Shmuel Ur for answering questions about ConTest.

References

[1] ArgoUML. http://argouml.tigris.org.

[2] Chandrasekar Boyapati and Martin C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM Press, November 2001.

[3] Guillaume Brat, Klaus Havelund, Seung-Joon Park, and Willem Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, pages 3–12, September 2000.

[4] J.-D. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, October 1999. Appeared in *ACM SIGPLAN Notices* 34(10).

[5] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of Java multithreaded applications. In *Proc. ACM Symposium on Parallel and Distributed Tools (SPDT)*, pages 48–59. ACM Press, 1998.

[6] Markus Dahm. Byte Code Engineering Library (BCEL). http://bcel.sourceforge.net.

[7] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In *Proc. 8th Int'l. SPIN Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag, May 2001.

[8] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs, 2002. Available from ur@il.ibm.com. This is an extended version of: ConTest — A User's Perspective. In *Proc. 5th International Conference on Achieving Quality In Software*, 2002.

[9] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1), 2002.

[10] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.

[11] Cormac Flanagan and Stephen Freund. Detecting race conditions in large programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 90–96. ACM Press, June 2001.

[12] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–186. ACM Press, 1997.

[13] Patrice Godefroid, Robert S. Hanmer, and Lalita Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft. In *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 124–133. ACM Press, 1998.

[14] Patrice Godefroid and Sarfraz Khurshid. Exploring very large state spaces using genetic algorithms. In *Proc. 8th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2280 of *Lecture Notes in Computer Science*, pages 266–280. Springer-Verlag, April 2002.

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, 2nd edition, 2000.

[16] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[17] Klaus Havelund and Grigore Rosu. Monitoring Java programs with Java PathExplorer. In *Proc. First Workshop on Runtime Verification (RV'01)*, volume 55(2) of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2001.

[18] Gerard J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[19] Java Developer Connection Tech Tips of March 28, 2000. http://developer.java.sun.com/developer/TechTips/2000/tt0328.html.

[20] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):106–117, February 1980.

[21] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[22] Soot. http://www.sable.mcgill.ca/soot/.

[23] John T. Stasko. Animating algorithms with XTANGO. *SIGACT News*, 23(2):67–71, 1992. S. Hartley's Java version is available at http://www.mcs.drexel.edu/~shartley/.

[24] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 2002. Available at http://www.cs.sunysb.edu/~stoller/STTT.html.

[25] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 187–206. ACM Press, October 1999. Appeared in *ACM SIGPLAN Notices* 34(10).