Multithreaded Java program test generation

by O. Edelstein
E. Farchi
Y. Nir
G. Ratsaby
S. Ur

We describe ConTest, a tool for detecting synchronization faults in multithreaded Java™ programs. The program under test is seeded with a sleep( ), yield( ), or priority( ) primitive at shared memory accesses and synchronization events. At run time, ConTest makes random or coverage-based decisions as to whether the seeded primitive is to be executed. Thus, the probability of finding concurrent faults is increased. A replay algorithm facilitates debugging by saving the order of shared memory accesses and synchronization events.

The increasing popularity of concurrent Java** programming in Web applications on the Internet, as well as other distributed, mostly client/server, applications, has brought to the forefront the analysis of defects in concurrent programs. Such concurrent defects—for example unintentional race conditions and deadlocks—are difficult to uncover and analyze, and often remain undetected past the product deployment.

There are a number of reasons for this difficulty:

- For a given functional test, the size of the set of possible interleavings is exponential in the length of the program, and it is not practical to test them all (except for small programs[1,2]). Only a few of the interleavings actually produce the concurrent fault, and thus the probability of producing the concurrent fault is very low.
- Since the scheduler is deterministic, the execution of short tests that are independent of I/O delays and network load will usually create the same interleaving regardless of the environment. This is also the case when long tests that are not too dependent on I/O delays and network load are executed in a similar environment.
- Tests that reveal a concurrent fault in a production environment or in stress test are usually long and run under various environmental conditions (such as system load or software version). As a result, such tests are not necessarily repeatable, and when a fault is found, much effort is invested in recreating the conditions in which it occurred.

Research on finding concurrent faults focuses on detecting actual data races (e.g., References 3, 4, and 5 among others), using algorithms for efficient identification of race conditions in the current run. As mentioned above, the chance that a race condition will occur is low, and the race detection tool does nothing to increase it. Moreover, if the race is intentional, false alarms will result. Furthermore, some concurrent defects are not captured by the formal definition of a race condition. For example, one could write a faulty program that depends on the scheduling algorithm.[6]

Researchers have also looked at the problem of replay in distributed and concurrent settings. In a distributed setting, processes might synchronize by sending and receiving messages. For a given run, an order is determined over the message sending and message receiving events. A replay for a distributed

setting is a re-execution of the program in which all these events occur in the same order as in the original run. In a concurrent setting, a replay algorithm preserves the order of shared memory access and synchronization events. (This problem was solved for the Java language in Reference 7.) Replay support facilitates debugging as it removes nondeterminism from the execution of a distributed or concurrent program.

Deterministic testing or debugging of distributed or concurrent programs is discussed in References 8 and 9. It is a weak form of replay in which some of the dependencies between processes are captured at run time. Next, during subsequent runs, the debugging tool attempts to force these dependencies. In contrast to replay, no guarantee is provided that the same timing will be obtained in each run.

Model checking has been applied to testing of multithreaded Java programs in Reference 10. In model checking each process is modeled by a state machine. State machines might communicate by sending and receiving messages. An additional state machine models the predicate to be checked. Together the process state machines and the predicate state machine form the Cartesian product state machine. State space exploration is applied to the Cartesian product state machine. At state space exploration time, the predicate state machine is used to determine if the predicate is violated. Systematic state space exploration has inherent scalability issues.[11]

In this paper we study the problem of generating different interleavings for the purpose of revealing concurrent faults. Since the size of the search space is exponential in the program length, we take a heuristic approach. We study whether seeding the program with sleep( ), yield( ), and priority( ) primitives increases fault detection capability. We seed the program at shared memory access events and synchronization events. At run time, we make random or coverage-based decisions as to whether seeded primitives are to be executed. Using the seeding technique, we were able to dramatically increase the probability of finding typical concurrent faults[6] injected in Java programs. The probability of observing the concurrent faults without the seeded delays was very low. In addition, we found that the sleep( ) primitive was almost always better at finding the injected defects than the yield( ) and priority( ) primitives. Finally, we defined and implemented an architecture, ConTest, that combines the replay algorithm introduced in Reference 7, which is essential for debugging, with our seeding technique.

Our experiments were conducted on a single processor. Any concurrent defect found by the seeding technique on a single processor is also a defect on a multiprocessor.

Although the seeding technique gives good results in practice, it has inherent limitations. First, the interleaving space is exponential in the length of the program. In addition, different interleavings occur with different probability. We discuss how coverage-directed generation of interleavings can be used to overcome these limitations.

We utilize the test suites employed in functional test and system test to detect concurrent faults. A test suite definition includes the expected results, which are used to indicate if a fault occurred. We rerun each test many times. The seeding technique causes different interleavings to occur. The final results are examined to determine if a fault was observed. This approach integrates seamlessly into standard testing practices. The automated tests are simply re-executed. Our approach can be combined with race detection tools to improve the detection of concurrent faults.

The rest of the paper is organized as follows. In the next section, we cover race detection and replay. In the section "Use scenario," we describe a typical use of ConTest for testing multithreaded Java applications. The section "Architecture" explains ConTest's architecture and discusses some design issues. Next, we describe the seeding technique, and then we explain how coverage can be used to improve it. The section "Experiments" details six experiments in which some commonly found concurrent defects are detected. We present our conclusions in the last section.

Race detectors and replay

In this section we discuss existing race detection and replay algorithms in the context of debugging synchronization faults. For the purposes of our work, we define an interleaving as a complete order over the operations of the program, such that on a system with a single processor there is a possible run in which this was the temporal order of the execution. In multiprocessor systems the definition of temporal order is more complicated, but for our

purposes any reasonable definition will do (see Reference 12, for example).

Detecting race conditions. Many concurrent defects result from data-race conditions. A data-race condition is defined by Savage[4] as two accesses to a shared variable, at least one of which is a write, with no mechanism used by either to prevent simultaneous access.

A race condition is a possible source for a defect, since the value of the variable at the time of reading depends on the scheduling. However, not all race conditions are defects. For example, the following code swaps two integers. There is a race condition, but no defect, as the swapping occurs regardless of the interleaving.

```
class Change{
   static int x = 4, y = 5;
   //Used to implement a busy wait.
   static int z1 = −1, z2 = −1;
   //Swap the value of x and y concurrently
   public static void main(String args[ ]){
      (new Thread(new ChangeA( ))).start( );
      (new Thread(new ChangeB( ))).start( );}
}
class ChangeA implements Runnable{
   public void run( ){
      Change.z1 = Change.x;
      while(Change.z2 == −1)
         System.out.println("A is waiting");
      Change.x = Change.z2;}}
class ChangeB implements Runnable{
   public void run( ){
      Change.z2 = Change.y;
      while(Change.z1 == −1)
         System.out.println("B is waiting");
      Change.y = Change.z1;}}
```

It should be noted that race conditions are execution-dependent: a program might be in a race condition in one execution and not in another. Therefore, tools that detect races at run time (or by analyzing the trace of a given run) are likely to miss some potential data races.

In addition, the use of locks might prevent race detection tools from exposing a concurrent defect which is not a race condition. For example, suppose there are two accesses to a variable, and suppose there is a lock associated with that variable. If the lock is obtained before each access and released just after it, then the order of the accesses is arbitrary. If this is unintentional, it might be a defect; however, formally, there is no race condition, and race detection tools will not alert us to the problem.

In conclusion, race detection tools, having no knowledge of the function of the program, would miss defects on the one hand and result in false alarms on the other. The "benign" data race cited above is an example of code that results in a false alarm.

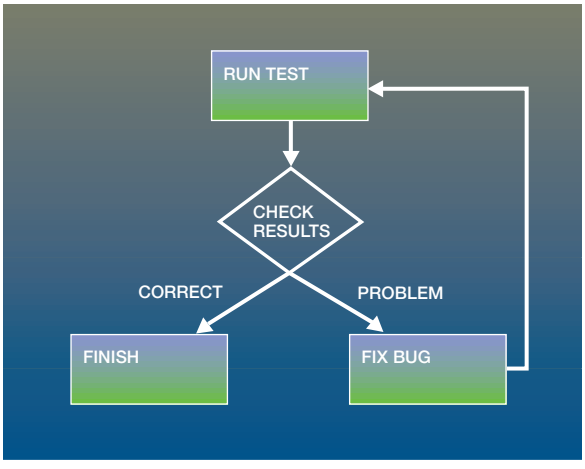
Replay. Deterministic programs are debugged by repeatedly executing the program using the test that produced the fault. Since multithreaded programs are nondeterministic, re-executing the program may not reproduce the fault. As a result, there is a need for a record and replay algorithm that will force the same order of shared variable access and synchronization events within the different threads.

The purpose of a replay algorithm is to reproduce a concurrent fault. But because the replay algorithm also introduces delays, it changes the interleavings of a specific run whenever scheduling decisions are recorded. This, in turn, may increase or decrease the chance that a concurrent fault will occur. Since the replay algorithm is not designed to find concurrent faults, it does not systematically cover the space of possible interleavings. Concurrent faults usually occur in the field when the software is running and the replay facility is turned off. When the replay facility is turned on in order to record the concurrent fault and its environment, there is a chance the fault disappears! This is due to the delays introduced by the replay algorithm itself. It follows that in practice many faults cannot be reproduced in this way.

An algorithm that defines "logical time" for a distributed send/receive environment is discussed in Reference 13 (pages 591–598). This algorithm was originally designed by Lamport.[12] Because logical time defines a linear order of send and receive operations, the algorithm can be used for replay purposes. It employs local process clocks and synchronizes them on receive operations. In Reference 7, this idea was adapted to multithreaded, shared memory Java programs. In order to enable replay, only the following events, denoted *concurrent events*, must be captured:

- Shared variable accesses. Some accesses to a shared variable may occur in the wrong order and cause a faulty value.
- Synchronization events, e.g., synchronized, wait( )

Figure 1 Conventional testing



and notify( ) primitives. Such events may change the order of access to shared variables.

The Java replay algorithm in Reference 7 produces sufficient trace information to determine a linear order on concurrent events. A global variable, global_clock, is updated atomically with each execution of a concurrent event (with the exception of the monitorenter bytecode instruction and wait( ) synchronization primitive). After the execution of a concurrent event, the global variable, global_clock, is compared with a thread scoped local clock variable to determine if a context switch occurred. In order that the linear order be consistent with each thread execution order and with the execution order of synchronization events, accessing the global variable, global_clock, and executing the concurrent event must be done atomically. Since global variable access and concurrent event execution are handled atomically, the replay algorithm adds synchronization to the program under test. For example, shared variables that were not synchronized in the original program are synchronized in the modified program. A naive implementation of a race detection algorithm might cause race conditions to be masked as a result of the added synchronization. We discuss this effect further in the architecture section.

Since we are able to reproduce the correct linear order of concurrent events, we can also reproduce concurrent faults when they occur.

This algorithm was implemented in Reference 7 by changing a Java virtual machine implementation. Working at the Java virtual machine level makes the implementation easier because the scheduler can be directly modified. However, an implementation that requires a specific Java virtual machine is limiting. Furthermore, for every new version of the Java virtual machine, a new version of the modified scheduler is needed. For these reasons we decided to implement the replay algorithm above the Java virtual machine level.

Use scenario

In a typical ConTest user scenario, a given functional test *t* is run (without human intervention) against *P*, the program under test. This is done repeatedly until a coverage target is achieved. The common practice in unit, function, and system tests (but not in load testing), is to execute the test once, unless a fault is found. The process is depicted in Figure 1. Load testing, i.e., testing the application under a real or simulated workload, increases the likelihood of some interleavings that are unlikely under light load. However, load testing is not systematic, is expensive, and can only be performed at the very end of the testing cycle.

The process of re-executing tests using ConTest is depicted in Figure 2. Each time the functional test *t* is run, ConTest produces, as a result of the seeding technique, a potentially different interleaving. During the execution of the functional test *t*, coverage and replay information is produced and collected. For example, we might collect, as coverage information, data about whether or not a context switch occurred while executing a program method. Test results are checked at the end of the run. If a fault is observed, replay information is saved for later use in the debugging process.

In order to carry out the scheme above it is required to have a test whose result can be checked automatically. This is the case when a regression test suite is developed. During manual testing or even during debugging, ConTest may still be used if the test result is captured by some capture-and-replay tool. In addition, when a fault is detected, the debugging process is greatly facilitated by ConTest's replay support. The seeding technique causes the fault to occur while the replay algorithm is executing. As a result, no additional work is needed to reproduce the fault.

Architecture

ConTest has several components. The first of these, the replay component, is implemented by adapting the replay algorithm introduced in Reference 7 to source level instrumentation. When *t* is executing against *P*, the replay component is called every time a concurrent event occurs, both before and after the event. Thus, the replay component wraps each concurrent event. The next component is the (heuristic) seeding component. Sleep( ), yield( ), and priority( ) primitives are seeded into the program, in order to increase the probability of producing different interleavings on each run. The seeding component wraps the replay component (see Figure 3). Finally, the fault detector component (not shown) is used to determine whether or not a fault occurred. The fault detector checks that the program behaved correctly on a given test. A race detection tool may be used to augment this component. A race detection tool can spot potential problems even if the program terminated correctly (there was a race but the final result is correct) or can be used in the absence of a fault detection component. Then, the race detection tool should disregard synchronization events introduced by the replay algorithm.

Under the current implementation, the schedule is changed using standard methods of the Java Thread class. For example, delays are introduced using sleep( ). Thus, ConTest is not part of the scheduler and has no special system privilege. This is one possible implementation of ConTest. A different approach would be to implement ConTest as part of the Java virtual machine. Although similar interleaving generation techniques are then possible, the architecture is less portable.

We collect coverage data, which represent the extent to which the interleaving space is covered when *t* is repeatedly executed against *P*. Then, we use heuristics to determine the probability and type of delays introduced.

We next discuss some ConTest design issues. The replay algorithm presented in Reference 7 was implemented as part of the Java virtual machine. ConTest, on the other hand, implements the replay algorithm by instrumentation of the Java source code under study. This approach is less efficient in terms of required execution time, but it is more portable (for example to C++).

The replay algorithm wraps every concurrent event: synchronization events and shared variable accesses.

Figure 2 Re-executing tests using ConTest

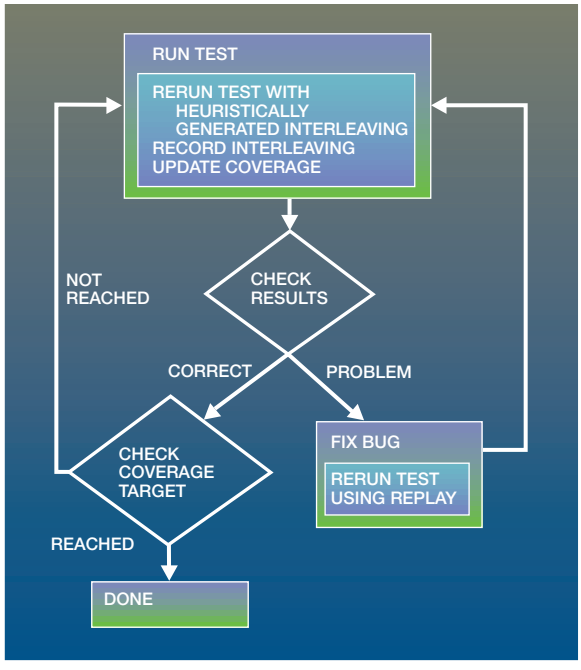
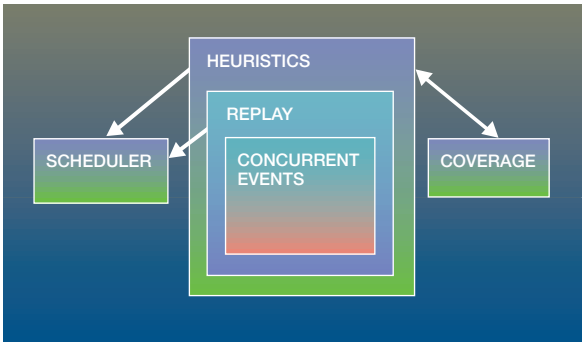


Figure 3 Layering for concurrent event instrumentation



The seeding technique is based on adding the synchronization events, sleep( ), yield( ), and priority( ), to *P*, the program under test. Each new synchronization event introduced by the generation algorithm is wrapped by the replay algorithm.

We observe that this is not really necessary. The methods we use, (sleep( ), yield( ), and priority( )), only impact the time slice allocated to a thread by the

scheduler. Thus, if the replay algorithm only captures the linear order of shared memory access and synchronization events of the original program, this linear order is a legal linear order of the original program under some (possibly other) scheduler. Therefore, replay can be ensured by simply capturing the concurrent events of the original program. As a result, seeded sleep( ), yield( ), and priority( ) synchronization primitives are not treated as concurrent events for the purpose of replay, and the global clock variable is not updated when they are executed.

A Java scheduler is not required to be fair.[13] The seeding technique is applicable even if the scheduler is fair. In Appendix A we discuss the validity of the seeding technique under a fair scheduler.

Source code level instrumentation introduces some unique design issues. A Java expression might include reads and writes in arbitrary subexpression locations. ConTest is required to instrument a Java expression so that its semantics is preserved and control is passed to ConTest before and after a read or a write occur. In addition, ConTest should be able to perform its calculation and the read or the write atomically. This is done by defining *before* and *after* concurrent event methods that accept as input the value read (or written). Overloading of after_write( ) and before_write( ) is used to identify the variable type. For example, the subexpression

```
x = someExpression;
```

is changed to

```
(type of x) after_write(
   (x = (type of x)
        before_write(someExpression))).
```

As described in the beginning of this section (see also Figure 3), control is passed to ConTest by the instrumentation before and after each concurrent event. This example details how this is done at the source code level. In the above instrumentation example, the before_write( ) method passes control to ConTest before the write operation to variable *x*, the concurrent event in this case, is executed. The after_write( ) method passes control to ConTest after the write operation to variable *x* is executed. This order of control is achieved as the Java virtual machine first calculates someExpression. Next, the before_write( ) method is executed. Control is passed to ConTest, at which time ConTest executes its heuristics and replay algorithms. In addition, the protocol of the before_write( ) method is that someExpression is passed unharmed as the return value of the before_write( ) method. In the next stage of execution the return value of before_write( ), someExpression, is assigned to *x*. Finally, after_write( ) is executed. Again, control is passed to ConTest, as required, right after the assignment to variable *x* has been performed.

The seeding technique

In this section we describe and compare the seeding techniques used to obtain new interleavings and reveal concurrent faults. The following three seeding techniques were implemented at concurrent events:

- A sleep( ) statement is executed with some probability, which will cause the thread to be blocked for a randomly selected period of time.
- A yield( ) statement is executed with some probability, which will pass the control to another thread. If only one thread is active, the execution of the yield( ) statement has no effect.
- A priority( ) primitive, which causes a change in thread priority, is executed with some probability. Depending on the way the scheduler interprets the change of priority, this might impact thread scheduling.

For a single processor, we found that when there are many concurrent threads, yield( ) and sleep( ) gave similar results. A thread starts executing when its start( ) method is invoked. It is observed, in the subsection "First experiment," later, that if yield( ) was performed after a new thread instance was created and before it started executing, control was not likely to be transferred to the new thread. On the other hand, if sleep( ) was performed after a new thread instance was created and before it started executing, control was likely to be transferred to the new thread. As a result, seeding the program with the yield( ) primitive has less effect if the implementation of the Java virtual machine causes the current thread to continue its run. Seeding the program with priority( ) statements revealed the injected defect in many cases, but priority( ) statements did not perform as well as either sleep( ) or yield( ) statements (see the section "Experiments in the detection of concurrent faults," later, for details).

As a result, sleep( ) was chosen as the preferred seeding primitive. Actual defects identified by ConTest

were found using the heuristic seeding of the program with the sleep( ) synchronization primitive.

Using coverage to improve the seeding technique

The seeding technique described in the previous section produces good results in practice, but it also has inherent limitations. The interleaving space is exponential in the program length and the probability of occurrence varies for different interleavings. Thus, low probability faulty interleavings may be masked. In this section we discuss how to overcome the limitations of the seeding technique through coverage-directed generating of interleavings.

Coverage analysis formally defines the concept of a "good" set of tests. The purpose of coverage analysis is to direct the test sampling process to effectively detect defects. In addition, coverage is used to handle exponentially large spaces by covering them with polynomial-sized partitions. A list of coverage tasks (the testing requirements) is defined, with each task corresponding to one element of the partition. Each task should be exercised by some test in the test set.

The result of a concurrent program depends on the order in which different concurrent events are performed. Thus, a test of a concurrent program is determined by a program input and an interleaving.[14] As described in the earlier section "Use scenario," ConTest is based on the assumption that the user created a good test suite that covers the program inputs. The seeding technique is used to obtain additional tests by generating different interleavings for a given input.

In the literature there are a number of coverage models for concurrent and distributed programs. Taylor et al. generalize serial models by applying them to the "execution graph" of the concurrent program.[15] Yang et al. apply the define-use coverage criterion to a generalized control graph of the parallel program.[16] Marick's GCT (Generic Coverage Tool) "race coverage" requires that if a method can be executed by two threads simultaneously, this simultaneous execution will in fact occur.[17]

We introduce two new coverage models to control the way interleavings are generated. These models satisfy several important requirements:

- The sampling is directly defined on the interleaving space and is not a generalization of a sequential coverage criterion.
- The size of the model is polynomial in the number of seeded primitives. We would like to create a hierarchy of models of increasing size so that the testing effort can be controlled.
- The coverage model is derived from a general defect pattern (as in mutation coverage[18]).
- The information derived from the coverage model can be used to improve the seeding technique.

A synchronized method is typically obtained in Java by adding the synchronized keyword to a method's prototype. At run time, a lock is captured before a synchronized method is executed, and released afterwards. A typical concurrent defect occurs when the user does not notice that the method prototype needs to be synchronized or drops the synchronized keyword to improve performance. We attempt to capture this defect pattern with the following two coverage models.

Our first coverage model determines whether the execution of each method was interrupted by a context switch.[19] There are two tasks for every method, one that requires that the method be interrupted and another that requires that it not be. This model conforms to the above list of requirements. As in statement coverage, the number of tasks in this model is linear in the length of the program and not proportional to the number of interleavings, which is exponential. It is therefore feasible to attempt to cover all the tasks in this model.

The test generation scheme that creates good coverage for the first model is trivial; we use the seeding technique to cause a context switch whenever we are executing a method in which no context switch occurred in past executions. If a context switch did occur, we do not cause a thread switch. Thus, we increase the probability of covering both tasks for each method.

A second model determines if a method's execution was interrupted by any other method. A coverage task of the form (method A, method B) is covered if method A performed a context switch while executing under thread D, and before control returned to thread D method B was executed in a different thread, F.

The size of the second coverage model is of order $O(n * n)$, where n is the number of methods in the

program under test. As a result, this coverage model is larger than the first. Nevertheless, it is still far from exponential in the program length. This model is also stronger than the first model. For example, in the first model, we only check whether there was a context switch that interfered with method execution, whereas in the second and stronger model, we also check the source of the context switch interference.

In most experiments, seeding with sleep( ) uncovered defects more often than seeding with yield( ) or priority( ) primitives.

The test generation scheme that creates good coverage for the second model is more complex. First we create the task list (all possible pairs of methods). At run time, assume that the program is currently executing method M, that other threads are executing methods $N, K, \ldots, L$, and that we are about to decide whether or not to force a context switch. We force a context switch if either $\langle M, N\rangle$ *or* $\langle M, K\rangle, \ldots, \langle M, L\rangle$ satisfy new coverage tasks or there are uncovered coverage tasks starting with N *or* $K, \ldots, L$. This way every context switch will either cover new tasks from the set $\{\langle M, N\rangle, \langle M, K\rangle, \ldots, \langle M, L\rangle\}$ or prepare the way for new tasks that start with the methods $\{N$ *or* $K, \ldots, L\}$ to be covered. The decision to force a context switch is nondeterministic; if there is a possibility of satisfying a new coverage task, it is done with higher probability.

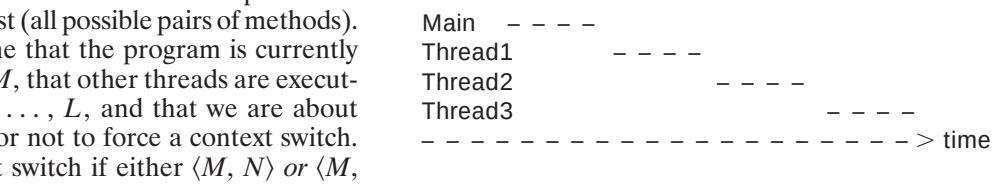
A method in main( ), executing before additional threads are invoked, cannot be interrupted. As a result, coverage tasks that capture such a method interference are uncoverable. Because the heuristics for detecting such tasks are not directly related to the main focus of this paper, they are omitted.

Experiments in the detection of concurrent faults

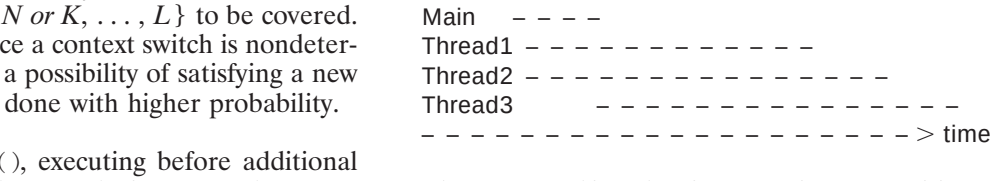
In this section we show how our seeding technique revealed concurrent faults in fault-injected and real-life multithreaded Java programs. All of the experiments, unless otherwise specified, were run under a stable environment, a single processor, and IBM Java version and run-time environment 1.3.0, with the IBM Just-in-Time Compiler enabled. The operating system used was AIX*4.3 and the hardware platform was RS6000*. In most cases, seeding the sleep( ) primitive found defects more often than the other seeding options.

First experiment. The program under test creates three threads. Each thread checks a global parameter, denoted First. If First is true, it changes it to false and prints "race." The fault occurs when a context switch happens between the time that First is checked and the time it is set to false. More than one thread then determines that it is the first, and more than one thread prints "race."

This fault was *never* found in 1000 executions of the unseeded program under test. Indeed, when the program is not seeded, and there is no environmental interference, each thread is completed *before* the next thread starts. A timing diagram of the execution might look like this:



The seeding technique will not only cause different interleavings, such as thread 2 before 1, but also different execution times for the threads. For example, the following interleaving may occur.



The generated interleaving space is very sensitive to whether sleep( ) or yield( ) are invoked at some concurrent event. A thread starts executing when its start( ) method is invoked. It was observed that if yield( ) was performed after a new thread instance was created and before it started executing, control was not likely to be transferred to the new thread. On the other hand, if sleep( ) was performed after a new thread instance was created and before it started executing, control was likely to be transferred to the new thread. ConTest (using sleep( )) found faults in 200 out of 1000 runs. Using yield( ), three faults were found in 1000 runs. This is explained by the fact that threads are longer in the seeded pro-

gram. In the original program, 1000 executions never revealed the fault.

Second experiment. The program under test was created by Sun to demonstrate a possible concurrent defect.[20] A description of the program follows.

The main( ) routine starts many threads of type move_money( ). These threads are assigned high priority. move_money( ) repeatedly chooses two global array locations, representing accounts, adds a random amount to one location (account) and subtracts the same amount from the other. main( ) then calls check_sum( ), which checks the total amount of money each user has. check_sum( ) is called a number of times.

A fault occurs if there is a thread switch in move_money( ), and check_sum( ) is executed after adding to one global array location (account) and before subtracting from another.

We ran the tests 1000 times under three different Java virtual machines. The results reported below were independent of the particular machine being used.

In the unseeded program we found the fault in about 40 percent of the runs. When running the program seeded with sleep( )s, we found the faults in 85 percent of the runs. The probability of revealing the fault increased dramatically in this case.

In addition, we observe that check_sum( ) reports an error if a context switch at a specific program location occurred in at least one move_money( ) invocation. The probability of a correct run depends on all the threads *not* switching at a particular program location. If we assume there is a probability p that a single thread does not switch, and that the threads behave independently, then the probability of a correct run for n threads is p^n for a single execution of the check_sum( ) routine. The probability of a correct execution is p^{nm} for m such executions. The probability decreases rapidly as the number of threads and the number of executions of check_sum( ) increase.

To better emulate real-life conditions, we modified the original program so that move_money( ) contained busywork (a loop of empty writes). This greatly reduced the probability that a thread switch would occur at the "correct" location, that is, the one that reveals the fault. Nonetheless, after this modification, faults were found 80 percent of the time using ConTest. Faults were not found with the unseeded program, where the changes had also vastly reduced the probability of a thread switch at the precise location that reveals the fault. On the other hand, the probability of revealing the fault did not change much in the seeded version of the modified program. This was due to the correct placement of the seeding primitives at concurrent events.

Third experiment. In this experiment there are a number of threads that execute the following code:

```
class ChangeNotification implements Runnable{
static boolean notified = false;
public void run( ){
   while(notified = = false)
     {Thread.currentThread( ).yield( );};
   System.out.println(
     "subject current value is:" +
      subject.currentValue( ));
}

public void changeNotification(Subject subject){
   notified = true;
   this.subject = subject;
}}
```

These threads wait until notified is true and print subject.currentValue( ). However, if in the main thread there is a thread switch after notified = true, then subject is not initialized (i.e., this.subject = subject is not executed) and an exception occurs. In the original code the fault was never observed (500 executions with 2000 threads altogether). In the code seeded with yield( ) or sleep( ) the fault was observed about 700 times in 2000 tests. The reason this fault does not occur in the uninstrumented version is that the main thread is short (relative to the scheduler time slice); therefore, there is no thread switch after notified = true. Here, however, yield( ) operates as well as sleep( ), as all the threads are alive at this time.

Fourth experiment. In this experiment the program being run creates n threads recursively. If control shifts to a new thread immediately after it is created, then the new thread looks for the received parameter in a hash table before the hash table entry is set by the creating thread. This causes an exception which is handled by the program. Interestingly, in AIX version 4.3, when the main thread (i.e., the thread that executes main( )) creates a new thread, the probability that control will immediately switch to this thread is about one-third. When any thread other

Table 1 Number of concurrent threads reached

	Faults	Depth				
		0	1	2	3	4
NT	370	10000	10000	0	0	0
yield	12	2764	6280	6068	3457	1098
sleep	0	5259	9037	4640	956	100
priority	6	7569	9223	2099	634	266

than the main thread creates a new thread, control does not immediately shift to it. In Windows NT**, the probability that the control will immediately shift is very low (we have never seen it) for any thread, including the main thread.

If the control shifts to the new thread as the threads are created, then the program generates concurrent Thread_Number threads. If control does not shift to new threads, then only one thread at a time is created. We find that we will rarely have more than one thread in normal execution. We almost always have one executing thread and one that is waiting for its turn. The exact interleaving depends somewhat on the hardware/software combination on which we execute, but tends to be consistent for any such combination. There are many possible behaviors, all satisfying the Java thread behavior requirements.

If the seeded program executes a sleep( ) before passing the parameter to the new thread, control will always shift and an exception (the manifestation of a defect in this program) will always be observed. If a yield( ) is used by the seeded program, then control will shift most of the time. The reason for the difference is that performing a yield( ) by the only running thread has no effect, while performing a sleep( ) by the only running thread introduces a delay and might result in a context switch.

Each test was executed with main( ) starting ten threads. We ran 1000 tests. Table 1 shows the number of times each depth (number of concurrent threads) was reached. In the unseeded program there was never more than one concurrent thread. The faults in the unmodified program, when run on AIX version 4.3, were always found in the transition between the main thread and the first created thread. These faults were not found on NT using Sun's Java virtual machine.

This experiment shows that behavior of multithreaded programs greatly depends on the system in which they are executed. If a ConTest-like tool is used, the dependency can be reduced. The ramifications of this will be discussed in the conclusions.

Dining philosophers and deadlock detection. ConTest is also useful for deadlock detection. To demonstrate this, we have implemented the classical symmetric dining philosophers algorithm. The symmetry of the algorithm may cause it to run into a deadlock. While the deadlock did not occur when running the symmetric dining philosophers algorithm without ConTest for a quarter of an hour, it occurred almost immediately every time it was run using the seeding technique (seeding the program with sleep( )s). We next describe the symmetric dining philosophers algorithm and an interleaving that causes the deadlock. We explain why the seeding technique increased the probability of deadlock detection.

The dining philosopher problem is described informally. There are n philosophers who sit around a table and think. Between each pair of philosophers there is a single fork. From time to time a philosopher gets hungry. In order to eat, the philosopher requires exclusive use of two forks, the one to the immediate right and the one to the immediate left. After eating, the philosopher relinquishes the two forks.

Each philosopher executes the same symmetric algorithm. For example, each philosopher might attempt to pick up the left fork. If successful, the philosopher then picks up the right fork and eats. A deadlock occurs if all of the philosophers pick up their left fork and then attempt to pick up their right fork. Because picking up a fork contains concurrent events, the probability of executing a context switch after the left fork is picked up is increased when executing the seeded program. As a result, the probability of observing the deadlock increases.

A real-life race condition defect. A crawler algorithm is embedded in an IBM product. The crawler algo-

rithm is implemented using a worker thread design pattern (see Reference 21, pages 290–296). Thus, the implementation uses synchronization primitives, such as synchronized block, wait( ), and interrupt( ), for worker and manager coordination. The worker's objective is to search for relevant information.

A skeleton of the crawler algorithm was tested. The skeleton has 19 classes and 1200 lines of code. Although the code skeleton has a small number of lines, it is complicated by the worker and manager communication protocol. In fact, worker threads wait for connection, and the connection manager waits for live connections in order to be released. (In both cases the wait( ) primitive is performed.) A queue of connections is handled by the manager. If the queue is either full or empty, threads execute the wait( ) primitive. This causes threads to wait, until they are interrupted by the manager. In addition, idle workers are retrieved by the manager. Finally, access to shared data structures by different workers is synchronized so as to merge retrieved information in a consistent way.

ConTest was able to find an unknown race condition defect in the algorithm within one hour of its execution. The fault was a null pointer exception (java.lang.NullPointerException). A description of the defect follows. The finish( ) method of the Worker class has the following line:

```
if(connection ! = null) connection.setStopFlag( );.
```

If the connection variable is not null and then a context switch occurs, the connection variable might be set to null by another thread. If this happens before connection.setStopFlag( ) is executed, a null pointer exception (java.lang.NullPointerException) is taken. To fix this defect, the above statement should be executed within an appropriate synchronized block.

Conclusions

The main contribution of this work is an effective method for finding concurrent defects. The major advantages of this method are, first, it does not require additional user involvement and, second, it does not give false alarms if a correct functional test is used.

This technology has great potential because it enables the early detection of concurrent defects. In addition, the technology can be adapted to work in the distributed setting, using the replay algorithm defined in Reference 22.

The technology embodied in ConTest is currently applicable to functional and system testing. The main requirement is that the test results can be verified automatically. This usually applies to regression testing and not to unit testing. Coupling the technology with some tool that captures results and then replays the test will enable the use of ConTest in unit testing as well.

The seeding technique introduces additional context switching, which degrades performance. Future work should address this issue in system testing. In testing small components this issue is less important.

In the future ConTest will be used to simulate execution in various Java environments. Different environments interpret the Java thread semantics in different ways (all of which are correct). If the testing is done on one operating system, there is no guarantee that the application will work on another. Using ConTest, we can cause any program to behave as if it is running on any given environment.

The seeding technique can be further improved by creating interleavings that provide high coverage for a multithreaded coverage model. We have implemented, but not yet tested, a number of methods to achieve this aim. One way to create interleavings that provide high coverage is to use previously collected coverage information[15,23–25] in the interruption decision.

The seeding technique can be further improved by considering the history of previous interruption decisions when arriving at the next one. Otherwise, if the probability of interruption is high, the probability of a long, uninterrupted run is very low. Consider a program in which two threads are created, each having 100 concurrent events in which an interruption decision is made. If the original program is run in a stable environment, thread A will always run from start to finish, after which thread B will run. In the seeded program, if the probability of an interruption is kept reasonably high regardless of previous interruption decisions, then the likelihood of the original interleaving is practically nil. Yet the original interleaving might be the one containing the fault. Thus, not interrupting the program for a lengthy stretch of execution may be the appropriate course of action.

Appendix A: Validity of the seeding technique

In the section "Architecture" we claimed that an interleaving obtained by the seeding technique is a possible interleaving of the original program. This claim is trivial when the scheduler is not fair. Indeed, a Java scheduler is not required to be fair. In this section we prove that the interleaving obtained by the seeding technique is a possible interleaving of the uninstrumented program under some fair scheduler. We prove this claim in a theoretical setting. We choose sleep( ) as the primitive to be used by the seeding technique because it was found to be most effective in detecting faults.

We start with some definitions. Informally, an asynchronous shared memory system (Reference 13, pages 237–241) consists of a finite collection of processes that interact using a finite set of shared variables. More precisely, an asynchronous shared memory system is a simple type of state machine, or automation, in which the operations are associated with named actions. The actions are classified as either input, output, or internal. Input and output actions are used for communication with the automaton's environment. An I/O automaton is composed of processes, each with a number of operations. Each operation has a precondition on inputs and internal variables, as well as an *effect*. Each effect may be executed whenever the precondition of that operation is satisfied. Only one operation in a process is executed at a time.

For a given automaton A, a trace is a list of A's operations. The list can be either finite or infinite. An execution of a trace $(\text{operation}_1, \text{operation}_2, \ldots)$ under an automaton A is a sequence of triples,

$$((\text{beforeState}_1, \text{operation}_1, \text{afterState}_1), (\text{beforeState}_2, \text{operation}_2, \text{afterState}_2), \ldots)$$

where operation_i, $i = 1, 2, 3, \ldots$, is an operation, and beforeState_i, afterState_i, $i = 1, 2, 3, \ldots$, are states. In addition, the precondition of operation_i is met in beforeState_i and the state that results from applying operation_i to beforeState_i is afterState_i. Finally, afterState_i is the same state as beforeState_{i+1}.

For a given execution

$$((\text{beforeState}_1, \text{operation}_1, \text{afterState}_1), (\text{beforeState}_2, \text{operation}_2, \text{afterState}_2), \ldots)$$

under automaton A, its corresponding trace is a listing of the operations in the order of their executions, i.e., $(\text{operation}_1, \text{operation}_2, \ldots)$.

A trace α of an automaton A is said to be fair if the following conditions hold for each process (Reference 13, pages 212–215):

- α represents an execution of A.
- If $\alpha = ((\text{beforeState}_1, \text{operation}_1, \text{afterState}_1), \ldots, (\text{beforeState}_k, \text{operation}_k, \text{afterState}_k))$ is finite, then no operation is enabled in the final state of α (afterState_k).
- If α is infinite, α's execution contains, for a process i, infinitely many operations or infinitely many occurrences of states in which no operation is enabled.

According to this definition, a scheduler provides a process with infinite opportunities to execute. Whenever there is such an opportunity, the process executes if it is enabled.

Given an automaton A with n processes, we model the seeding technique by defining an automaton A' as follows. For each process i, $1 \leq i \leq n$, we define an internal Boolean variable, sleep_i, initialized to false. For each of process i's operations, j, we replace its precondition, $\text{pre}_{i,j}$, with the precondition $(\neg\text{sleep}_i \wedge \text{pre}_{i,j})$. We call such an operation obtained from A an original A operation.

For each process i, we add the operations Sleep and Awake. Sleep and Awake model the intervention caused by the seeding technique. The Sleep operation has the precondition

$$\{(\text{sleep}_i = \text{false}) \wedge \bigvee(\text{on all preconditions of operations in i})\}$$

and the effect $\text{sleep}_i = \text{true}$. Awake has the precondition $\text{sleep}_i = \text{true}$ and the following effect:

- $\text{sleep}_i = \text{false}$
- Execute one of the original A operations that are currently enabled.

A Sleep operation of process i disables all of the original A operations in i. However, since Awake sets sleep_i to false, sleep_i's value does not disable original A operations. Awake then ensures that at least one of the original A operations is executed if there is an original A operation enabled.

To state our claim, we use the following construction. Given a trace α' of a fair execution of A', we construct a new trace α as follows:

- We remove all the Sleep operations from α'.
- We remove from α' all the Awake operations that did not perform an original A operation during (α')'s execution.
- For every Awake operation in α' that performed an original A operation during (α')'s execution, we replace the Awake operation with that original A operation.

Claim A.1 *α is a trace of a fair execution of A.*

By considering a fair trace α' of A', we model that the instrumented program A' is executed under a fair scheduler. If the above claim is true, then α is a fair trace of A. Thus, α could have been produced by executing α on A using a fair scheduler, and a defect identified by α' is a defect of the original automaton A.

Proof: Let α' be a fair trace of A'. Consider the trace α obtained from α' by removing all the Sleep operations and replacing every Awake operation with the original automaton operation performed during its execution, if one existed. Clearly, α is a legal trace of A, and if sleep_i is disregarded, the execution of α by A results in the same sequence of memory states as the execution of α' by A'.

We first assume that α' is finite. A Sleep operation that belongs to process i cannot be the last operation in α', as it will set sleep_i to true. As a result, the Awake operation that belongs to process i becomes enabled. This contradicts the fact that α' is a fair execution of A'.

Assume that the last operation of α' is one of the operations obtained from A, *op*, or Awake. From the fairness of α', we have that sleep_i is false at the end of the execution of α'; otherwise, Awake is enabled. From the fairness of α' we also have that no original A operation is enabled at the end of the execution of α'. As the memory state at the end of the executions of α' by A' and α by A are the same when sleep_i is disregarded, process i's operations in A are also disabled at the end of α's execution. Thus, α is fair.

Consider next the case of an infinite trace. If the Sleep operation of process i is executed, then Sleep_i is set to true. As a result, the Awake operation of process i becomes the only enabled operation of this process. Since α' is fair, process i will eventually get a chance to execute an enabled operation. As the only enabled operation of process i at this stage is its Awake operation, this operation will be executed. As a result, every Sleep operation of process i that appears in α' has a matching Awake operation of process i that appears in α', with no operation of process i appearing between the Sleep and Awake operations. We call such a pair of Sleep and Awake operations *a matching pair*.

If the number of matching pairs for process i is finite, then there is an infinite suffix of α', β', that contains no matching pairs of process i. We also denote the matching suffix of α by β. Since a suffix of an infinite fair trace is fair, β' is fair. As β' has no matching pairs, it is always the case that sleep_i is false. Since β' is fair, either no operation from i was enabled infinitely many times in β', or an operation of i was executed infinitely many times in β'. Since sleep_i is false in β' and the sequence of state changes is the same in both β' and β, either i is not enabled an infinite number of times in β or an i operation is executed an infinite number of times in β. Thus, as process i was arbitrary, β is fair. We deduce that α is fair.

Assume that the number of matching pairs for process i is infinite. There are two cases to consider in a matching pair's Awake operation: either an original A operation was executed by the Awake operation or it was not. First consider an Awake operation that did not execute an original A operation. Since an Awake operation sets sleep_i to false before checking if an original A operation of process i was enabled, no original operation will be enabled at the current stage of execution in α. Next, consider an Awake operation that executed an original A operation. As Awake is replaced by the original A operation and α' is a legal trace, the original A operation is still enabled at the current stage of α's execution. Thus, each matching pair of process i will either contribute an enabled operation in α or indicate a stage of the execution in which process i was not enabled in α. An infinite number of such cases exist. Thus, as process i was arbitrary, we deduce the fairness of α from the fairness of α'.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., or Microsoft Corporation.

Cited references and notes

1. S. K. Damodaran-Kamal and J. M. Francioni, "Nondeterminacy: Testing and Debugging in Message Passing Parallel Programs," *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, ACM, New York (1993), pp. 118–128.
2. S. K. Damodaran-Kamal and J. M. Francioni, "Testing Races in Parallel Programs with an OtOt Strategy," *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, ACM, New York (1994), pp. 216–227.
3. B. Richards and J. R. Larus, "Protocol-Based Data-Race Detection," *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, ACM, New York (1998), pp. 40–47.
4. S. Savage, "Eraser: A Dynamic Race Detector for Multithreaded Programs," *ACM Transactions on Computer Systems* **15**, No. 4, 391–411 (1997).
5. E. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai, "Towards Integration of Data-Race Detection in DSM Systems," *Journal of Parallel and Distributed Computing* **59**, No. 2, 180–203 (1999).
6. See Appendix E in B. Lewis and D. J. Berg, *Threads Primer*, Prentice Hall, Englewood Cliffs, NJ (1996).
7. J.-D. Choi and H. Srinivasan, "Deterministic Replay of Java Multithreaded Applications," *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, ACM, New York (1998).
8. J. M. Stone, "Debugging Concurrent Processes: A Case Study," *Proceedings of the ACM SIGPLAN Conference on Programming Language and Implementation (PLDI)*, ACM, New York (1988), pp. 145–153.
9. E. Farchi, M. Factor, and Y. Talmor, "Testing for Timing-Dependent and Concurrency Faults," *Proceedings of Software Testing Analysis and Review*, Software Quality Engineering (1998).
10. S. D. Stoller, "Model-Checking Multi-Threaded Distributed Java Programs," *Proceedings of the 7th International SPIN Workshop on Model Checking of Software*, Springer-Verlag, New York (2000).
11. G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, Englewood Cliffs, NJ (1991). See section 11.3.
12. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM* **21**, No. 7, 558–565 (1978).
13. N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, San Francisco, CA (1996).
14. A. Y. H. Zomaya, K. C. Tai, and R. H. Carver, "Testing of Distributed Programs," *Parallel and Distributed Computing Handbook*, A. Y. H. Zomaya, Editor, McGraw-Hill, Inc., New York (1996).
15. R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural Testing of Concurrent Programs," *IEEE Transactions on Software Engineering* **18**, No. 3, 206–215 (1992).
16. C.-S. Yang, A. Souter, and L. Pollock, "All-du-path Coverage for Parallel Programs," *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'98)*, ACM, New York (1998), pp. 153–162.
17. B. Marick, *A Tutorial Introduction to GCT*, http://www.testingcraft.com/gct-tutorial.pdf.
18. J. Offutt and R. Untch, "Mutation 2000: Uniting the Orthogonal," *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, IEEE, New York (2000), pp. 45–55.
19. We determine whether a method is interrupted by keeping a global static variable of the last executing thread. When a new concurrent event is executed, we check if the current thread is the last executing thread. When the current thread and the last executing thread do not match, a context switch has occurred.
20. See the Java Developer Connection Technical Tips, dated March 28, 2000, at http://developer.java.sun.com/developer/TechTips/2000/tt0328.html.
21. D. Lea, *Concurrent Programming in Java, Second Edition*, Addison-Wesley Publishing Co., Reading, MA (2000).
22. R. Konuru, H. Srinivasan, and J.-D. Choi, "Deterministic Replay of Distributed Java Applications," *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)* (2000). See http://www.ipdps.org/.
23. R. N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs," *Communications of the ACM* **26**, 362–376 (May 1983).
24. S. Weiss, "A Formal Framework for the Study of Concurrent Program Testing," *Proceedings of the 2nd Workshop on Software Testing, Analysis and Verification*, IEEE, New York (1988), pp. 106–113.
25. S. Morasca and M. Pezze, "Using High Level Petri Nets for Testing Concurrent and Real Time Systems," in *Real-Time Systems: Theory and Applications*, H. Zedan, Editor, Elsevier Science Publishers, Amsterdam, Holland (1989), pp. 119–131.

Accepted for publication September 23, 2001.

Orit Edelstein *IBM Research Division, Haifa Research Laboratory, MATAM, Haifa 31905, Israel (electronic mail: edelstein@il.ibm.com).* Mrs. Edelstein holds B.Sc. and M.Sc. degrees in computer science from the Technion, Israel Institute of Technology. After working for a few years in software development with another company, she joined the IBM Haifa Research Laboratory where she is active in the area of programming languages and software development environments, and where she is currently managing a project in software testing and verification. Her areas of interest also include compilers, code optimization, distributed environments, and object-oriented programming.

Eitan Farchi *IBM Research Division, Haifa Research Laboratory, MATAM, Haifa 31905, Israel (electronic mail: farchi@il.ibm.com).* Dr. Farchi received his Ph.D. degree in mathematics from the University of Haifa, Israel, in 1999. He also holds a B.Sc. degree in mathematics and computer science and an M.Sc. degree in mathematics from the Hebrew University, Jerusalem, Israel. Since 1992 he has been with the IBM Haifa Research Laboratory, where he led an effort toward improving the performance of operating systems. He is currently involved in software testing and in developing coverage-directed tools for testing concurrent and distributed programs. Dr. Farchi is a frequent speaker at software testing conferences, is the author of a tutorial on the testing of distributed components, and teaches software engineering at the University of Haifa.

Yarden Nir *IBM Research Division, Haifa Research Laboratory, MATAM, Haifa 31905, Israel (electronic mail: nir@il.ibm.com).* Mr. Nir holds a B.Sc. degree in computer science from the Technion, Israel Institute of Technology. For the past two years he has been working (part time) at the IBM Haifa Research Laboratory, where he participates in the development of software test-

ing tools. He has recently begun his studies for an M.A. degree in philosophy at the University of Haifa.

Gil Ratsaby *IBM Research Division, Haifa Research Laboratory, MATAM, Haifa 31905, Israel (electronic mail: ratsaby@il.ibm.com).* Mr. Ratsaby holds a B.A. degree in computer science from the Technion, Israel Institute of Technology. Since 1999 he has been with the IBM Haifa Research Laboratory where he has worked on the development of software testing tools. Mr. Ratsaby has recently started work toward an advanced degree in computer science, specializing in quantum computing.

Shmuel Ur *IBM Research Division, Haifa Research Laboratory, MATAM, Haifa 31905, Israel (electronic mail: ur@il.ibm.com).* Dr. Ur holds a Ph.D. degree in algorithm optimization and combinatorics from Carnegie Mellon University, Pittsburgh, PA. He also holds B.Sc. and M.Sc. degrees in computer science from the Technion, Israel Institute of Technology. He is currently with the IBM Haifa Research Laboratory in Haifa, Israel where he works in the field of software testing. His main interest is coverage analysis in software testing, a field in which he has published several papers on such topics as functional coverage, minimizing regression suite size, coverage-directed generation, visual code coverage techniques, and coverability. Dr. Ur has also published papers in the fields of hardware testing, artificial intelligence, and algorithms. He holds a number of patents, and he teaches at the Technion and at the University of Haifa.