Automating the Testing of Multi-threaded Java Programs

Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur

IBM Research Laboratory in Haifa

Abstract. Finding bugs due to race conditions in multi-threaded programs is difficult, mainly because there are many possible interleavings, any of which may contain a fault.
In this work we present a methodology for testing multi-threaded programs which has minimal impact on the user and is likely to find interleaving bugs. Our method reruns existing tests in order to detect synchronization faults. We found that a single test executed a number of times in a controlled environment, may be as effective in finding synchronization faults as many different tests. A great deal of resources are saved since tests are very expensive to write and maintain. We observed that simply re-running tests, without ensuring in some way that the interleaving will change, yields almost no benefits.
We implemented the methodology in our test generation tool — ConTest. ConTest combines the replay algorithm, which is essential for debugging, with our interleaving test generation heuristics. ConTest also contains an instrumentation engine, a coverage analyzer, and a race detector (not finished yet) that enhance bug detection capabilities. The greatest advantage of ConTest, besides finding bugs of course, is its minimal effect on the user. When ConTest is combined into the test harness, the user may not even be aware that ConTest is being used.

1 Introduction

The increasing popularity of concurrent Java programming — on the Internet as well as on the server side — has brought the issue of concurrent defect analysis to the forefront. Concurrent defects such as unintentional race conditions or deadlocks are difficult and expensive to uncover and analyze, and such faults often escape to the field.

One reason for this difficulty is that the set of possible interleavings is huge, and it is not practical to try all of them [1]. Only a few of the interleavings actually produce concurrent faults. Thus, the probability of producing a concurrent fault is very low. Another problem is that since the scheduler is deterministic, executing the same tests many times will not help, because the same interleaving is usally created. This is true for simple tests, regardless of the environment, and

[1] Trying all of the possible interleavings has in fact been done for small programs in [2] and [3].

for tests of average complexity reexecuted in a similar environment. The problem of testing multi-threaded programs is compounded by the fact that tests that reveal a concurrent fault in the field or in stress test are usually long and run under different environmental conditions. As a result, such tests are not necessarily repeatable, and when a fault is detected, much effort must be invested in recreating the conditions under which it occurred.

Research on finding concurrent faults focuses on detecting actual data races (e.g., [10], [11], [5] and many others), using algorithms for efficient identification of race conditions that actually occurred in the current run. As mentioned above, the chance that a race condition will occur is low, and an actual race detection tool does nothing to improve it. Moreover, if the race is intentional, false alarms will result. 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 order [7]. Data race reports generated by most existing methods can potentially include many artifacts, which can overwhelm the programmer with irrelevant information [8]. A false alarm or artifact is a detected race condition that does not represent a program defect. False alarms may result from numerous causes, for example, infeasible data races, implicit synchronization, data that does not affect program outcome and/or limitations and shortcomings of the particular race detection method used.

Research has also looked at the problem of replay in several distributed and concurrent contexts. This problem was solved for the Java language in [1]. Model checking has been applied to testing of multi-threaded Java programs in [12], but systematic state space exploration has inherent scalability issues.

In previous work [4] we studied 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 seed the program with conditional sleep statements at shared memory access and synchronization events. At run time, we make random, biased random, or coverage-based decisions as to whether seeded primitives are to be executed. Using the seeding technique, we dramatically increased the probability of finding typical concurrent faults [7] injected in Java programs.

In this work, we report on a multi-threaded bug detection architecture that combines a replay algorithm, which is essential for debugging, with our seeding technique, as well as other smaller components such as coverage and race detection. We call this architecture ConTest.

We utilize the test suites employed in function 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 then examined to determine if a fault was observed and therefore false warnings are never issued. This approach integrates seamlessly into standard testing practices. The automated tests are simply reexecuted.

Section 2 starts with an explanation, from the user's point of view, of how our technology is used by testers of multi-threaded Java applications. We show

that the tester does not have to do much, and has to know even less. Section 3 details an insider's view of what really happens when ConTest is used and includes a brief description of ConTest components. Section 4 sheds some light on the implementation of ConTest and is of special interest to tool makers. It may be skipped by people less familiar with the Java language. Section 5 presents the experiments in which typical concurrent defects are revealed using ConTest, as well as some real bugs found. We present conclusions in Section 6.

2 User Perspective

In this section we explain how ConTest fits into the general testing scheme.

When testing a program, a set of tests is selected and each is executed. For every test, if a defect is revealed, the program is debugged and the test is executed again. In the normal course of things the test will be executed only once (Figure 1.A).

When ConTest is used, the difference (see Figure 1.B) is that after the test is executed, ConTest is used to decide whether the same test should be executed again. The test may be executed many times until ConTest does not require a re-execution. The reason for re-execution is discussed in the next section; however, from the user's perspective, it is a fully automated process.

The only requirement from the user is to have a test whose result can be checked automatically, when the tests are executed multiple times. This is usually the case if the test is part of a regression bucket. If the testing is manual, ConTest may still be used if we know that the test should produce the same results every time it runs and the test result is captured with some capture-and-replay tool. If this is done the test may be executed many times and the result compared to the results of the first run.

In addition to finding more bugs with no additional human intervention, ConTest supplies additional benefits. Concurrent Coverage is measured and can be reported to the user, and when a fault is observed, the debugging process is greatly facilitated by ConTest's replay and debugging support.

One of the most important design goals of ConTest was to be as unobtrusive as possible. The only additional work entailed is instrumenting of the Java byte-code and interfacing ConTest with the test harness to determine if a test should be re-executed. If a ConTest enabled test harness is used, then the user may be totally oblivious to it's existence.

3 Insider View

This section explains what really happens when ConTest is used. We start by explaining the different components of ConTest. We show how they mesh with each other, and with the test harness, to produce the required results. There is a lot hidden "under the cover". The reason we can do so much without being a burden on the user is that the instrumented application is executed, as opposed to the original application, and therefore ConTest has a high degree of control.

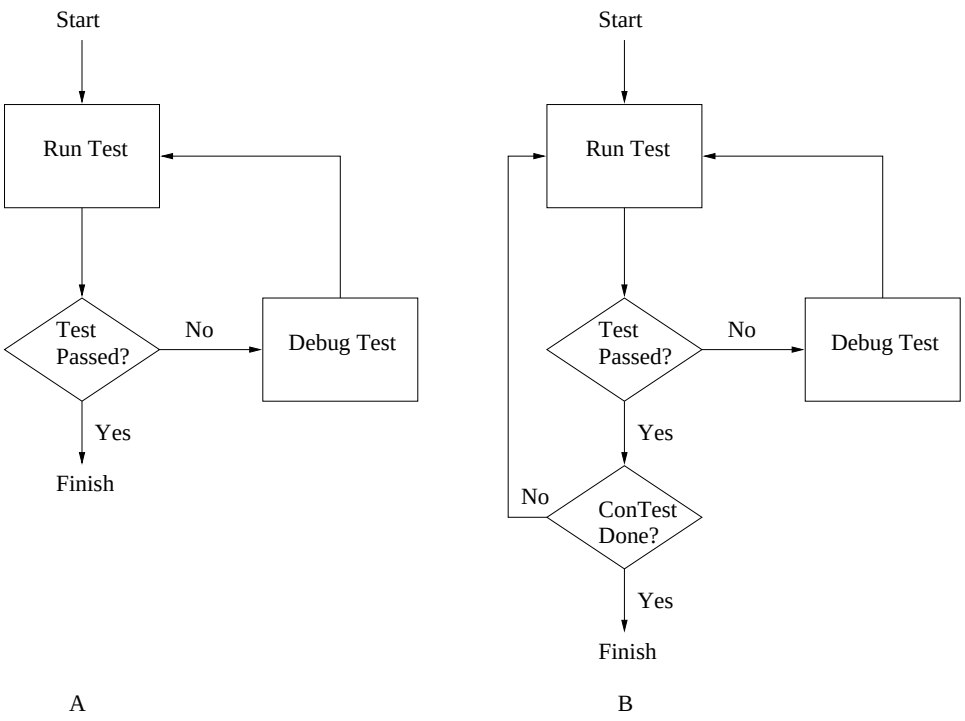

Fig. 1. Modification in Testing Practice using ConTest.

ConTest is composed of the following components:

- *An instrumentation engine.* This component adds irritator calls to the application's bytecode. The irritator in turn calls the coverage component and the replay component. The instrumentation engine is used once on the application. The instrumented application remains functionally unchanged.
- *A coverage enabled irritator.* The irritator function is to cause interesting interleavings to occur. Interleavings are caused by adding conditional sleep() statement in "concurrent" events, that is synchronization events and shared memory accesses (see also section 4). A number of heuristics are used to decide when to cause a thread switch via the sleep() statement. The heuristics range from being very simple ones such as sleep() with a probability of .5, to complex ones using multi-threaded coverage models as a base for the decision. We observed that without the irritator, running the program under test multiple times cause only a small number of interleavings to occur, while with the irritator many interleavings occur.
- *Capture and replay component.* When a test is executed, the generated interleaving is captured and the information required for replay is written to a file. If it is later decided to execute the same test with the same interleaving,

the replay component may be used. Our implementation is environment, JVM, JDK and operating system independent so that a test whose behavior was captured in one environment may be replayed in another.

- *Coverage.* Coverage is used for three different functions in ConTest. The first, which is invisible to the user but very important internally, is that it enables coverage-based heuristics used by the irritator. The heuristics used by the irritator can use coverage information to decide if a given sleep() will be executed and how long it will last. The second function is to report to the user which concurrent event has been reached so far, similar to the way coverage is used by other coverage tools. The third function is the decision of how many times each test needs to be executed before we start working on the next test.
- *Race Detection.* A race is usually defined as two accesses to the same memory, at least one of which is a write, done by two different threads with no synchronization between the accesses. Unlike all other race detection tools, in ConTest the race detection component *never* reports on races to the user. When this component finds a race, it communicates with the replay and irritator components to ensure that the test is re-executed; this time the race will be forced to occur in the opposite order. If the race results in a bug, the user can view any execution that caused the bug with a debugger, and to stop at a breakpoint just before the race occur.

The interplay of the different components and the test harness is shown in Figure 2. For clarity, the scheme is shown for a single test case. The first step in using ConTest is to instrument the application. The test harness then uses the instrumented application instead of the original application.

The solid line denotes the main path that is followed when neither bugs nor races are found. In this path the test is executed using the irritator which creates interesting interleaving using one of the ConTest interleaving generation heuristics. As the test is executed, coverage is measured, races are searched for and the interleaving is recorded. If the test contains no race and the test passed, this specific execution (combination of test and interleaving) of the test is completed. A stopping criteria is used to decide if the same test will be executed again. In ConTest this stopping criteria is based on coverage but it could have been as simple as "execute the test at least 10 times."

If, on the other hand, a race is found, we execute the test again. This explanation follows the broken line. This time around we use replay to ensure that the exact same interleaving will occur until we reach the location in which the race was detected. We use the irritator to force the other interleaving for the two concurrent events that formed the race (this is possible due to the definition of a race condition). If both executions of the test passed we revert to the main path and the location where it is decided if the test should be re-executed. Although we found a race we do not report it to the user since we could not show that this race causes a bug. Our approach, based on past experience with testing tools, is that it is better not to report on potential bugs when there is a likelihood of the

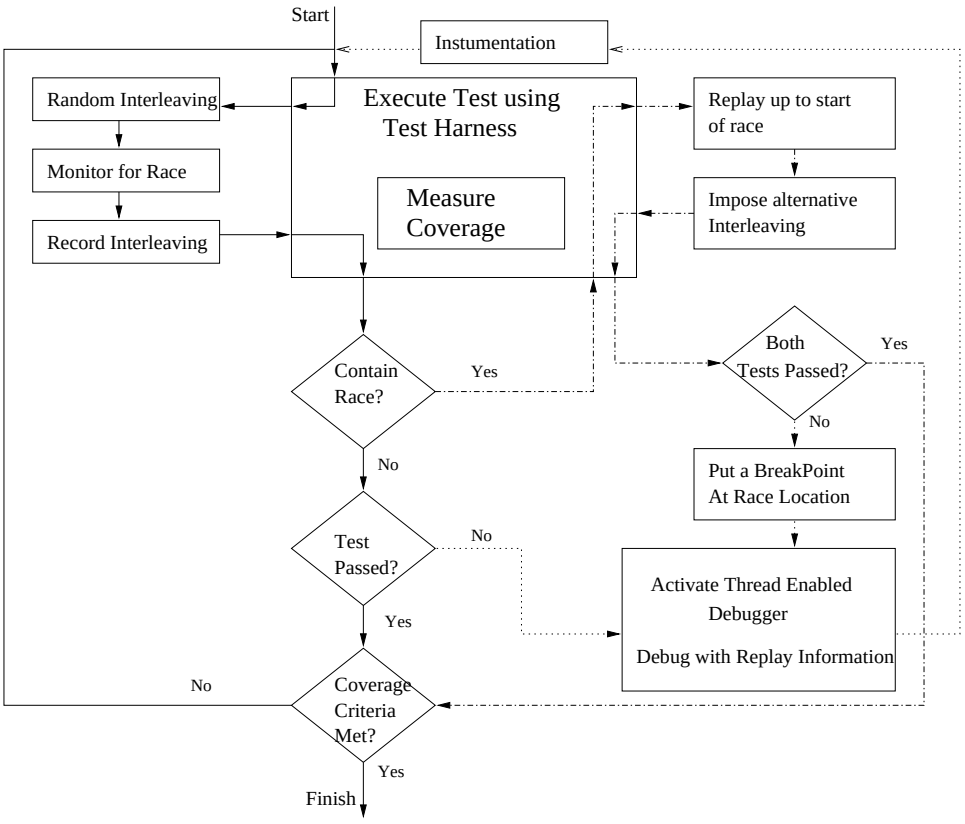

Fig. 2. Modification when using ConTest.

report being wrong. The reason is that after a few wrong reports, the user will stop using the tool.

If a bug is found (the dotted line), the test in which the bug was found is debugged. The replay information is used to ensure that as it is being debugged it will follow the original interleaving in which the bug was found. Further, if the bug occurs in a test that contains a race, a break point is installed at the location of the race to facilitate debugging. After the bug is fixed the modified application is instrumented and the process is repeated on this test, and possibly also on the other tests to ensure that no new bugs were introduced.

Ideally, Figure 2 would illustrate a set of tests and coverage would be added only after the test is found to yield correct results but this small inaccuracies were glossed over to make the figure and the explanation simpler.

4 Tips from Developers

This section discusses implementation details of the various components which are Java specific. Here our developers present their ingenious solutions to interesting problems.

4.1 Instrumentation

Replay must be supported to enable the debugging of the bugs found by ConTest in multi-threaded programs. All replay algorithms require that appropriate code is inserted before and after any concurrent event. In [1], a replay algorithm was designed and implemented inside a JVM. Capturing concurrent events inside the JVM is relatively easy, however, the implementation becomes JVM and platform specific. Since ConTest is a platform independent tool we had to perform the instrumentation either at the bytecode level or at the source code level. We have created instrumentations at both the bytecode and the source code level. In the source code, we used our own instrumentor, and in the bytecode we used an instrumentor called CFParse available from alphaWorks (see http::/www.alphaWorks.ibm.com). Source code level implementation is more portable, however bytecode implementation has the advantages of being simple, fast and very safe.

There are a number of difficulties inherent in source code instrumentation. Partly due to these difficulties, and after considerable experimentation, we preferred the bytecode instrumentation. The first difficulty is that a Java expression is not atomic, and naive instrumentation may not capture all concurrent events. Therefore, expressions require complex instrumentations. Every read of variable v is replaced by: (type of v)after_read(before_read(id), v, id). This enables the passing of control to ConTest just before and after the read of the variable v, and still meets the semantics of the expression. Any part of the expression which represent a write, v = SomeExpression, is replaced by: (type of v)after_write((v = (type of v)before_write(SomeExpression, id)), id), which also maintain the expression semantics, and enables passing the control to ConTest just before and after the write. Another difficulty is that different concurrent events are implicit at the source level. For example, if an $i++$ appears in a Java expression, the addition to i should occur before the next access to i in the expression. During this time, a context switch might occur and i might be accessed by a different thread. Simulating such an interleaving at the source code level is difficult. Yet another problem is that it is not easy to distinguish between global variables (which should be instrumented) and local variables, since Java allows multiple references to the same object.

A decision had to be made where to use the irritator. If a context switch is forced before or after a concurrent event, and the replay algorithm is correctly redesigned, we get a legal interleaving of the original program. But the other direction is also true. Any legal interleaving may be obtained by forcing context switches at the appropriate concurrent events. Thus, it is enough that the irri-

tator executes sleep()s before and after concurrent events and not at any other program location. Indeed this is how ConTest is implemented.

4.2 Replay

The irritator component introduces sleep()s before and after concurrent events. The replay algorithm introduced in [1] is broken if we introduce the sleep() primitive before and after concurrent events in a naive way. Every Java thread has a related interrupt bit. Any thread can set this bit if it has the right security permission. In addition, this bit can be inspected by any thread that has the right security permission. The sleep() primitive can take an interruptedException if this bit is turned on, after which the bit is turned off. It follows, that combining the replay algorithm in [1] with a naive irritator which uses the sleep() primitive, might introduce side effects, and the user will receive reports for faults that can never occur in the un-instrumented program. To overcome this problem we redesigned the replay algorithm in [1]. We implemented a program invariant: regular Java interrupts can only be executed when we are not executing the irritator's sleep(). When the irritator's sleep() is being executed, Java interrupts are handled through ConTest's internal interrupt mechanism which keeps the semantics of Java interrupts, but eliminates the possible side-effects.

During the implementation of ConTest, we modified the replay algorithm in [1] in various other ways. For a given run, a concurrent segment ([1]) is a maximal list of concurrent events that are executed in that order and belong to the same thread. In [1], the end of a concurrent segment is identified through the use of global and thread scoped counters. We simplified the capture of replay information so that the end of a concurrent segment is identified by a thread change. As a result we eliminated the need to maintain a complex data structure. The replay algorithm was also re-implemented to avoid any kind of busy waits. Finally, we implemented a special facility that consistently saves the replay information if the program blocks due to a deadlock or an exit operation, or if a user requests it. This is implemented by an auxiliary thread, which, upon request, loops over all the living threads and prints their replay information. Caution should be taken to print this information consistently, as there may be concurrent events that did not finish at exit or deadlock time.

4.3 Coverage Directed Test Generation

ConTest has heuristics that use coverage information to direct the irritator and determine whether a sleep() operation is executed and for how long. The coverage models are derived from general defect pattern.

A synchronized method is typically obtained in Java by adding the synchronized keyword to a method's prototype. At runtime, a lock is captured before a synchronized method is executed and released afterwards. A typical concurrent defect occurs when a method that should have been synchronized is not. This is usually due to an oversight, but the synchronized keyword is sometimes intentionally dropped to improve performance. For a race to happen we need two

accesses to the same memory location. For this reason it is most common when two methods that belong to the same class, usually two instances of the same method, interact when a *synchronized* should have been in place. We attempt to capture this defect pattern by the following coverage model.

A coverage model determines whether a method's execution was interrupted by any other method in that class. A coverage task of the form (method A, method B) where method A and B are in the same class is covered if in method A a context switch occurred, and before control returned, method B was executed by a different thread.

To generate interleavings for this model, first we create the task list that includes all possible pairs of methods in the same class. At run time, assume that the program is currently executing method M, that other threads are executing methods $N, K, ..., L$, and that we are about to decide whether or not to force a context switch. We force a context switch if either $< M, N >$ *or* $< M, K >, ..., < M, L >$ satisfy new coverage tasks or there are uncovered coverage tasks starting with N *or* $K, ..., L$. In this way, every context switch either covers new tasks from the set $\{< M, N >, < M, K >, ..., < M, L >\}$ or prepares the way for new tasks that start with the methods $\{N$ *or* $K, ..., L\}$ to be covered. The decision to force a context switch is non-deterministic; if there is a possibility of satisfying a new coverage task, it is done with high probability.

The coverage enabled irritator uses the sleep() primitive to force a context switch. In our experiments we found out that the sleep() primitive worked better as an irritator than the yield() primitive, and of the changing of thread priority. Thus, we only use the sleep() primitive in our heuristics.

4.4 Distributive Java Programs

ConTest is currently implemented for concurrent Java programs. In the case of distributive Java programs, ConTest can be used similarly and effectively for any part of the distributed program, but without a deterministic replay of the entire distributed environment. Building on the replay algorithm introduced in [9], ConTest can be generalized to replay distributed Java programs. In distributed environments test heuristics can be implemented to mutate the order of events arriving at the server.

5 Bugs Finding with ConTest

In this section we show how ConTest reveals concurrent faults in fault-injected and real-life multi-threaded Java programs. Our goal is to describe common bugs and explain how these bugs can be found and debugged using ConTest. We also explain why these bugs are very hard to find in a normal testing environment. In [4] we give further details on the experiments executed in the development of the different heuristics used to create interesting interleaving, and show a larger selection of programs and other interesting bugs.

5.1 Race and Setup Problems

We use the example in Figure 3 to show how races can occur. In this simple example, five threads are created. Each is assigned a local variable with values 1,10,100,1000 and 10000 respectively. A global variable, *Global* is created and initialized to zero, then all the threads are run. When a thread is run, it adds the local variable to *Global* and terminates. The main thread waits for a long time and then prints *Global*. When this program is executed it *always* prints the expected result of 11111 as output [2].

However, there is a bug hiding. Method *add()* containing the command *example.Global += Local* is not atomic. Looking at the bytecode at the end of Figure 3 we can see that first *Global* is put into the method local variable, then *local* is put into the method local variable. The two are added and only then the result is put back into *Global*. If a thread switch occurs between the time that *Global* is copied to the method's local variable and the time the result is written back into *Global*, any change that occurs in *Global* during the thread switch will be erased once the method regains control and writes the result to *Global*. This bug is never found when the program is executed stand-alone since every thread is very short. In fact every thread is much shorter than the time slice — the time it has for execution if no change of control occurs — and therefore it runs until completion without being interrupted. Such a bug may be found in the field if, for example, a higher priority application is running and it *steals* the control at the right time.

We wrote this program with this specific bug in mind and expected to see every combination of five digit numbers composed of zero's and ones written (with at least one 1 present) as a final result when run under ConTest. For example the number 00100 will be the final result if the third thread mask all the other threads. Once we have run it 1000 times we have seen every combination with some combinations being more common than others. However, we were surprised to see the value 00000, which is not explained by this bug. After some thought, we found that we inadvertently introduced another bug, which is not uncommon, with our "long" sleep (see Figure 3). We assumed that all the threads will be done before we print, which is not necessarily the case in practice. For example, it could have been the case that main() went to sleep, another application altogether got the control and when control returned, it was returned to main(). One of the heuristics implemented in ConTest reduces, or sometimes completely removes sleep() in the application under test. Due to this heuristic, which we forgot about while writing the sample program, the program prints 00000 with a probability of about one in five hundred. We think of this as a unique privilege that at least some of the bugs that we write turn into features in our sample programs.

[2] The program is so simple that the reader is encouraged to run it and see for himself

```
public class example{

static final int NUM = 5;
static int Global = 0;

public static main(String[] argv){

 Adder[] threads = new Adder[NUM];

 Global = 0;
 threads[0] = new Adder(1);
 threads[1] = new Adder(10);
 threads[2] = new Adder(100);
 threads[3] = new Adder(1000);
 threads[4] = new Adder(10000);

 for(int i=0;i<NUM;i++){
      threads[i].start();
 }
 try{ Thread.sleep(1000); }
   catch(Exception exc){}
 System.out.println(Global);
}
}
```

```
class Adder extends Thread{
 public int Local;

  Adder(int i){
    Local = i;
  }

  public void add(){
    example.Global += Local;
  }

  public void run(){
    add();
  }
}
```

ByteCode for Method add()

```
Method void add()
getstatic #3 <Field int Global>
aload_0
getfield #2 <Field int Local>
iadd
putstatic #3 <Field int Global>
return
```

Fig. 3. Program and ByteCode

5.2 Bugs Due to Java Thread Policy

In this experiment, a program creates n threads recursively. If control shifts to a new thread immediately after it is created, 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 — 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 than the main thread, creates a new thread, control does not immediately shift to the new thread. 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, the program generates concurrent *Thread_Number* threads. If control does not shift to new threads, only one thread at a time is created. We found that we rarely have more than one thread in normal uninstrumented execution. We almost always have one executing thread and one that is waiting for its turn. The exact interleaving depends on the hardware/software combination on which we execute, but tends to be consistent for any such combination. Due to this consistency, without

ConTest, we will see only one of the many possible behaviors, all of which satisfy the Java thread behavior requirements.

With ConTest we see many possible behaviors. For example when the ConTest instrumented program executes a sleep() before passing the parameter to the new thread forcing a thread switch, control will always shift and an exception (the manifestation of a defect in this program) will always be observed.

It is interesting to note that if a yield() is used instead of sleep(), control will shift most of the time. The reason for the difference is that whenever the only running thread performs a yield() it has no effect, while sleep() performed by the only running thread introduces a delay and may result in a context switch.

This experiment shows that behavior of multi-threaded 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 are discussed in the conclusions.

5.3 Dining Philosophers and Deadlock Detection

ConTest is also useful for deadlock detection. To demonstrate this, we implemented the classical symmetric dining philosophers algorithm where the symmetry of the algorithm may cause it to run into a deadlock. While the deadlock did not occur when we run the symmetric dining philosophers algorithm without ConTest for a quarter of an hour, it occurred almost immediately in every run using ConTest.

In the dining philosopher problem there are n philosophers who sit around a round 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 may attempt to pick up the left fork. If successful, the philosopher then picks up the right fork and eats. The philosopher will not release the left fork until it has taken the right fork and has eaten. A deadlock occurs if all of the philosophers pick up their left fork and then all wait for there right fork to be released. For a deadlock to occur, the probability of executing a context switch after the left fork is picked has to be increased. ConTest increases the probability of such event and as a result, the probability of observing the deadlock increases.

This bug will never be detected by a race detection algorithm for the simple reason that no race occurs. In this program races can not occur as all methods (e.g. pick fork) are synchronized. The result of the program depends on the execution order, a legal but dangerous programming practice.

5.4 A Real-life Race Condition Defect

We tested ConTest on a crawler algorithm embedded in an IBM product. The crawler algorithm is implemented using a worker thread design pattern (see [6]

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 wait()s for live connections to be released. A queue of connections is handled by the manager. If the queue is either full or empty, threads must wait() and are eventually interrupted by the manager. In addition, idle workers are retrieved by the manager. Finally, access to shared data structures by different workers are synchronized to merge retrieved information in a consistent manner.

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. 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 is taken. To fix this defect, the above statement should be executed within an appropriate synchronized block.

6 Conclusions

Work on ConTest started after half a person year was spent chasing a single intermittent bug which we knew existed but could not debug, mainly because it was very rare in normal execution and became even less common when debuggers or print statements were added. We decided to create a tool that would enable replay and test generation in concurrent environment. With time and experience we have become more ambitious and added a number of additional components.

In this work we describe an effective method for finding concurrent defects. Concurrent defects are often hard to find in the testing environment and are therefore found by the end user, or in stress test, which makes them very expensive. The major advantages of our method are that: bugs are found earlier in the testing process, no additional user involvement is required, and no false alarms are given.

This technology has great potential as it enables the early detection of concurrent defects. The technology embodied in ConTest is currently applicable to functional and system testing. The main requirement is that the tests can be executed automatically. This usually applies to regression testing and not to unit testing. Coupling the technology with a tool that captures results and then replays the test, will enable the use of ConTest in unit testing as well.

ConTest introduces artificial context switches which degrade performance. This may be a show stopper in a few cases. However, most of the time this degradation is not too bad and is irrelevant in testing small components.

In the future, ConTest can be used to simulate different environments. Different environments interpret the Java thread semantics in different ways (all of which are legal). If the testing is done on one operating system, there is no guarantee that the application will work on another. Indeed the Java slogan "write once run everywhere" can be replaced by "write once test everywhere". Using ConTest, we can reduce testing cost by emulating in a single environment all other environments. This is accomplished by simulating different thread switching policies.

References

1. Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.
2. S. K. Damodaran-Kamal and J. M. Francioni. Nondeterminacy: Testing and debugging in message passing parallel programs. In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging.*, pages 118–128, May 1993.
3. Suresh K. Damodaran-Kamal and Joan M. Francioni. Testing races in parallel programs with an otot strategy. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, August 1994.
4. Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Testing multi-threaded java programs. *submitted to the IBM System Journal Special Issue on Software Testing*, February 2002.
5. Eyal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordehai. Towards integration of data-race detection in dsm systems. *Journal of Parallel and Distributed Computing. Special Issue on Software Support for Distributed Computing*, 59(2):180–203, Nov 1999.
6. Doug Lea. *Concurrent Programming in Java Second Edition.* Addison-Wesley, 2000.
7. Bill Lewis and Daniel J. Berg. *A Guide to Multithreaded Programming. Appendix E.* SunSoft Press. A Prentice Hall Title, 1996.
8. Robert H.B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *Proceeding of the ACM SigPlan Symposium on Principals and Practices of Parallel Programming PPOPP*, 1991.
9. Harini Srinivasan Ravi Konuru and Jong-Deok Choi. Deterministic replay of distributed java applications. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, 2000.
10. B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.
11. Stephen Savage. Eraser: A dynamic race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
12. Scott D. Stoller. Model-checking multi-threaded distributed java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking*, 2000.