Multi-threaded testing with AOP is easy, and it finds bugs!

Shady Copty, shady@il.ibm.com   Shmuel Ur, ur@il.ibm.com

Abstract

Aspect oriented programming (AOP) is a relatively new paradigm that has proven itself in a number of fields. We investigate the suitability of AOP for testing tools by trying to implement the ConTest testing tool using AspectJ, a tool that implements AOP for the Java programming language. We deemed it important to study whether the entire set of features can be implemented this way, in the context of the larger problem where moving to a higher level of abstraction means that some details cannot be implemented.

Our conclusion from this exercise is that AOP is very suitable for the implementation of a number of classes of test tools. These include multi-threaded noise makers such as ConTest, in addition to coverage analyzers, data-race detectors, network traffic simulators, runtime bug pattern detectors, and others. The main advantage is that the instrumentation part of the tool creating method, which usually contains little scientific contribution but consumes most of the work, becomes much easier to perform and requires less expertise. In our specific exercise, a work that took more than half a year, and required specialized knowledge, was reduced to two weeks work by a relative novice.

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. 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 usually created. 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.

There is a lot of research being done on testing multi-threaded programs. Detecting data races was examined in ([15], [16], [10] and many others), replay in several distributed and concurrent contexts was studied in [2], model checking has been applied to the testing of multi-threaded Java programs in [17], static analysis was studied [18] [8] [3], coverage analysis [13] [5], and cloning [7] are also used to improve testing in this domain. In [4] [19] the problem of generating different interleavings for the purpose of revealing concurrent faults was studied. The approach can be simplified by seeding the program with conditional sleep statements at shared memory access and synchronization events, and randomly deciding at run time, whether seeded primitives are to be executed. ConTest [4] is a multi-threaded bug detection architecture that combines a replay algorithm, which is essential for debugging, with the seeding technique, as well as other smaller components such as coverage and race detection.

We realized that it is beneficial to combine many techniques to get a more comprehensive solution to the problem of testing concurrent programs. Toward this goal, in [9] and [6] we suggested a framework that combines the different technologies relevant to the multi-threaded testing domain. We showed how the different technologies described above can all benefit from being in the same framework. All either use output from other technologies or produce information useful to other technologies. As part of this effort, we created a benchmark of multi-threaded Java programs containing documented bugs [1]. Instrumentation is a technology that figures very prominently in our suggested framework, even though it is not specific to the domain. All the dynamic testing technologies need to use sophisticated instrumentation. The fact that instrumentation technology is complicated to develop has been a barrier that prevented the use of many of the tools on commercial sized applications.

A relatively new technology which allows the creation of generic instrumentation is Aspect Oriented Programming (AOP)[12] . AOP does to instrumentation what Lex and Yacc did for compliation. Instead of being a very complicated program that requires lots of expertise, instrumentation became something everyone can use. The central idea of AOP is that while the hierarchical modularity mechanisms of object-oriented languages are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems. In the implementation of any complex system, there will be concerns that inherently crosscut the natural modularity of the rest of the implementation. AOP provides language mechanisms that explicitly capture crosscutting structures. This makes it possible to program crosscutting concerns in a modular way, and achieve the usual benefits of improved modularity [11] From the perspective of our work, AOP is a simple declarative way to write the instrumentation engine.

In this paper we demonstrate how to implement a ConTest like tool using AOP, more specifically AspectJ. We start by listing all the features of ConTest. For each feature we either show how to implement it or explain why it is not related to instrumentation. For the few ConTest features that cannot be implemented due to limitations in AspectJ, we explain the shortcomings of AspectJ and how it can be extended. We think that it is important to go through the entire feature list as a problem common to high level programming models, where not everything can be modeled. This is the reason that while many high level programming languages exist, C/C++/Java remain dominant.

This paper shows in detail how to use AOP for the creation of testing tools, which can be used in many of the testing tools quoted in this paper. We also examine of the appropriateness of AOP for the testing domain in general. This is the first time, to our knowledge, that AOP is used to create a testing tool not specific to logging. This probably explains the missing features of AspectJ. The analysis contained will be used by the AspectJ people to make the tool more useful to the testing community.

2 ConTest

ConTest is a tool for finding bugs caused by concurrency. It alleviates the need to create a complex testing environment with many processors and applications, and works by instrumenting the bytecode of the application with heuristically controlled conditional sleep and yield instructions. ConTest is used by more than one hundred testers and developers in IBM. In this section, we systematically describe the various features of ConTest.

2.1 Instrumentation Scheme

ConTest instruments Java bytecode using cfparse [14]. It adds instrumentation points before and after every potential concurrent event, and other non-concurrent points due to coverage consideration. An event is called ”‘concurrent”’ if the order of its execution may impact the outcome of the program. For example, if two threads are executing but they do not share variables, and each thread output to a different output stream, this program will have no concurrent events. However, if they work with different variables but print to the same output stream, the printing statements are concurrent events. This is because the ordering between the events impacts the output by deciding which threads output is first. Concurrent evnts can potentially occur at the following locations:

1. Shared variable accesses. This is because some access order to a shared variable may be erroneous and cause a faulty value.
2. Synchronization functions, such as join(), start(), wait() and notify() primitives. Such events may change the order of accesses to shared variables.

[1]http://qp.research.ibm.com/QuickPlace/concurrency_testing/Main.nsf.

3. Synchronization block. This is a block or method protected with the keyword *synchronize*. A lock is created whose function is to ensure that at most one thread will be in the code segments protected with a synchronization on that lock. Each synchronization block has a begin and end.
4. Additional coverage related instrumentation. These are added due to specific user requests related to coverage.
   (a) At the entry to every method - used by the method coverage model.
   (b) At the entry to every basic block - used by the branch coverage model.

It is possible to do selective instrumentation based on user directives given as flags to the instrumentation applications.

2.2 Runtime Features

Almost all the ConTest runtime features are based on the instrumentation. As such, they provide additional motivation as to why all the instrumentation features are necessary. We comment on those that are hard or easy to implement, and indicate those beyond the scope of the tool we created with AspectJ. The following are ConTest's runtime features:

1. Coverage information is collected and logged. There are a number of types of coverage models. For each coverage task of any model, we log only the first time it was covered. If it was covered more than once, it is not logged. Adding a line to the log every time a task is covered causes performance problems. Counting and printing at the end is problematic as the code needs to be executed after the application terminates, and this require additional intervention in the user environment.

   Sometimes it is necessary to create more than one coverage trace (for each coverage type) in a run. For example, a server application is tested by repeatedly running a client program. All those runs are handled by one JVM running the (instrumented) server application. Therefore, by default, they will write only one coverage file. If the user considers each run of the client program a different test, and a different coverage file is requrired for each run. It is possible to create many coverage files with one execution, using callback threads (see below). All coverage files thus created will have the same timestamp in their name, but they will have different serial numbers.

   The different coverage models are as follows:

   (a) The basic coverage models are simply logs of the program executing instrumentation coverage statements. These include method coverage, branch coverage, and concurrent point coverage.
   (b) Synchronization coverage collects temporal data, which depends on the order of the specific execution. For example, a synchronization is considered used if the point was reached while another thread was inside the synchronization block on the same object.
   (c) Interfered location pairs coverage. Each line in the trace file of this coverage type contains a pair of program locations that were encountered consecutively in the run, and a third field that is ”t” or ”f”. The field is ”f” if the two locations were run by the same thread, and ”t” otherwise. That is, ”t” means a context switch occurred.
   (d) Shared variables coverage is not exactly coverage, but collects the names of variables detected as shared in a given run.(i.e. accessed by more than one thread).
2. ConTest uses many heuristics to try and increase the likelihood of finding bugs. These heuristics differ in the type of delay added, the frequency, and the activation rules. The general details are as follows:
   (a) ConTest currently supports three different types of noise: yields, sleeps, and synchYields. They differ in the type of synchronization primitive they use to create the noise (java.lang.Thread.yield() or java.lang.Thread.sleep()).
   (b) The amount of noise can be controlled by two properties: noiseFrequency and strength. The first one determines the probability that noise will be done at each concurrent event. The second determines how much noise is done at a given concurrent event, if noise is added at all.
   (c) ConTest can attempt to identify which variables are accessed by more than one thread, and do the heuristic noise only on accesses to those variables. A variable is determined to be shared when two different threads accessed it (read or write). With this option on, the heuristic noise can be any one of those described above (sleep, yield, etc.), with any strength. Note, however, that the noise is only done here on a small subset of the concurrent events.

(d) Halt-one-thread heuristic. This heuristic occasionally causes one thread to stop executing for a long time, until no other thread can advance. It can be powerful in revealing certain types of concurrent bugs.

(e) Tampering with time-out. The method java.lang.Thread.sleep(long millis), which also exists with additional nanoseconds parameter, causes the current thread to stop executing for the specified duration. This time duration alone should not be counted upon, It should not be assumed that other threads have completed some tasks by the time the sleep returns. ConTest helps testing that such wrong assumptions are not done, by randomly reducing the time-out used by these methods. This simulates a condition in which other threads work slower, and thus the bug may be revealed.

(f) Deferring noise generation to a late point in the execution. It may be desirable for ConTest not to begin its perturbations just as the tested program begins. For example, maybe a bug was found using ConTest in the initialization of your program, but the testing the rest of the program should proceed without ConTest causing the bug to appear again and again. ConTest can receive the class name or the method name as a string. Until this class or method is seen, ConTest does no noise. Once it has been seen, noise is done according to other conditions (heuristics, shared variables, etc.)

This mechanism does not affect coverage. If any of the coverage options are on, coverage information will be collected even before the begin-at condition is met.

3. ConTest provides a number of debugging aids that can be used to pinpoint a bug once it has been found. These include the following:

(a) Deadlock support. To aid the debugging of a deadlock, a report can be generated containing the following information: a list of threads waiting on a given lock, a list of which thread is holding each lock, the current line number of each thread.

(b) Orange box. When the program fails, it is often useful to know something about the behavior of the program in the last segments of its execution, similar to the black box on an airplane. The ”orange box” keeps a record of the last two accesses (read or write) to each non-local variable. For example, if the program execution was terminated due to an exception caused by variable foo, having the value null, you could use the orange box feature to check where this value was set.

(c) Partial replay. Finding a synchronization bug once is not always enough for debugging. Once a synchronization bug occurs, the replay feature increases the likelihood that the same synchronization bug will reoccur. This is achieved by saving all ConTest initialization properties and restoring them the next run.

(d) Callback thread. By opening a ”callback thread”, you can make several kinds of requests from ConTest from the outside, while the tested program is running. These requests include: debug reports, closing and restarting coverage sessions, and controlling fault injection. There are two ways you can pass requests to the callback thread: through standard input (normally, the keyboard), or through a network port (i.e., from another process).

3 AspectJ

AspectJ is a simple and practical aspect-oriented extension to Java. With just a few new constructs, AspectJ provides support for modular implementation of a range of crosscutting concerns. AspectJ extends Java with support for two kinds of crosscutting implementation. Dynamic crosscutting makes it possible to define additional implementation to run at certain well-defined points in the execution of the program. Static crosscutting makes it possible to define new operations on existing types; it’s called static because it affects the static type signature of the program. Dynamic crosscutting in AspectJ is based on a small but powerful set of constructs: join points are well-defined points in the execution of the program; pointcuts are a means of referring to collections of join points and certain values at those join points; advice are method-like constructs used to define additional behavior at join points; and aspects are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations. We use dynamic crosscutting to implement the features of ConTest using AspectJ, in a manner similar to that used by ConTest’s instrumentor. [11]

In AspectJ, pointcuts pick out certain join points in the program flow. For example, the pointcut call(void Point.setX(int)) picks out each join point that is a call to a method with the signature void Point.setX(int) (i.e., Point’s void setX method with a single int parameter). A pointcut can be built out of other pointcuts with: *and*, *or*, and *not*. [1] AspectJ also allows us to define pointcuts using wildcards. For example, set(* *) defines all the assignments to all the variables in the program. Pointcuts pick out join points, but they don't do anything apart from picking out join points. To actually implement crosscutting behavior, we use advice. Advice brings together a pointcut (to pick out join points) and a body of code (to run at each of those join points). AspectJ has several different kinds of advice. "'Before advice"' runs as a join point is reached, before the program proceeds with the join point. "'After advice"' on a particular join point runs after the program proceeds with that join point."'Around advice"' on a join point runs as the join point is reached [1]. In simpler words, the pointcut and the advice type define where the instrumentation is being done, and the advice body defines what will actually be instrumented.

3.1 ConTest Instrumentation Scheme Using AspectJ

We listed in Section 2.1 the instrumentation that ConTest performs to the program, we now explain how to implement these in AspectJ

1. Shared variable accesses. The join point *set(* *) or get(* *)* instruments all accesses to variables. Users should do some book keeping to make sure no noise is added to variables that are not shared
2. Synchronization functions, such as join(), start(), wait() and notify() primitives. These can be instrumented using the point cut *call(signature)* for each method.
3. Synchronization block. This kind of point cut is not supported in AspectJ.
4. Additional coverage related instrumentation. These are added due to specific coverage related user requests.
   (a) At the entry to every method - could be easily implemented using the *call* point cut.
   (b) At the entry to every basic block - This is not supported by AspectJ.

It is possible to do selective instrumentation in AspectJ using the point cuts that relate to the static nature of the code.

3.2 Contest Runtime Features Using AspectJ

The section reviews the runtime features mentioned in Section 2.2.

1. Coverage information collection and logging. Logging each coverage event once is not related to instrumentation, but is a back-end issue. The different coverage models are as follows:
   (a) Simple coverage models
      i. Method coverage - simply instrumenting each method call and printing its signature. This is possible using AspectJ and is demonstrated in the following chapter.
      ii. Branch coverage - AspectJ doesn't define a join point of a block of code.
      iii. Concurrent point coverage - AspectJ doesn't define a join point of a synchronization block, which is essential for implementing this coverage model.
   (b) Synchronization coverage - AspectJ lacks a join point of a synchronization block. If we had a join point we could have implemented this feature by recording which thread is waiting and which thread is inside the synchronization blocks.
   (c) Interfered location pairs coverage - this advanced feature requires a lot of bookkeeping on the backend, but little effort on the instrumentor. If it wasn't for the missing synchronization block join point, we could have implemented this feature.
   (d) Shared variables coverage - this feature can be implemented using the get/set join point and some bookkeeping in the aspect's body.
2. Heuristics
   (a) Noise type - independent of the instrumentor.
   (b) Noise amount and frequency - independent of the instrumentor.

(c) Noise on shared variables - this is an enhancement to adding noise on all variable accesses, which is not related to the instrumentor; this could be implemented in AspectJ.

(d) Halt-one-thread heuristic - This is like the other noise types. It's not related to the instrumentor and could be implemented in AspectJ.

(e) Tampering with time-out - AspectJ's around advice allows us to change the parameter for a certain join point. Catching the sleep method and changing its time out parameter, or canceling it altogether, would be sufficient to implement this feature.

(f) Deferring noise generation to a late point in the execution - independent of the instrumentor.

3. Debugging aids

   (a) Deadlock support. We can't provide a list of the threads waiting on a given lock. AspectJ doesn't allow us to instrument all the points where a thread may context switch, specifically a normal block and a synchronization block.

   (b) Orange box. This is possible with AspectJ's get and set join points.

   (c) Partial replay. It is possible to save and restore the initialization properties for the aspects.

   (d) Callback thread. This is not related to instrumentation.

4 Implementing the Tool

We implemented a few aspects to demonstrate the capabilities of an AOP language such as AspectJ. These aspects alter the class files to increase the likelihood of catching concurrent bugs, using ideas already implemented in ConTest. A special emphasis is put on the instrumentation capabilities. The following is the source code for the aspect SleepNoise:

```
public aspect SleepNoise extends Thread{
    private static Random rand = new Random();

    pointcut noiseVictem():
        ((get(* *) || set (* *))&&
          within(!SleepNoise));

    after(): noiseVictem() {
        try{// noise
        if (rand.nextInt(100) == 1){ // activation
           sleep(rand.nextInt(50)); // type
          }
        } catch (Exception e) {};
    }
}
```

SleepNoise is a simple aspect based on a single pointcut and a single advice. The pointcut defines where the instrumentation is being done. In our example, we are weaving the after() advice on all the gets and sets for variables in the instrumented program. The advice is a call to sleep() with a random parameter in the range [0,50] with a probability of 1% for invoking the sleep method. This will add noise to the instrumented application as done by ConTest's instrumentor. The difference, however, is that this aspect inlines the noise, whereas ConTest instruments call back methods, which add some runtime overhead. This example could easily be expanded to instrument special concurrent related methods, such as sleep, yield, notify, notifyAll, and so on. In addition, the type of noise could be altered to other types of noise that affect the interleaving of the program, all creating different kinds of heuristics.

SleepMutator is an aspect that implements the ”‘Tampering with time-out”’ heuristic discussed earlier, the following is its source code:

```
public aspect SleepMutator extends Thread{
    pointcut noiseVictem(long i):
       call(void sleep(long)) &&
       args(i) && within(!SleepMutator);

    private static Random rand = new Random();

    void around(long i): noiseVictem(i) {
    try{
      // [0,3*sleep]
      long newSleepTime = rand.nextInt((int)i*3);

      if (rand.nextInt(5) == 1)
        proceed(newSleepTime);

      } catch (Exception e) {}
    }

};
```

The pointcut we defined for this heuristic is a sleep call, taking into account its parameter and using the *proceed* keyword. The advice calls the sleep method with a new sleep parameter, which is chosen randomly in the range of [0, 3*oldParam].

As described earlier, ConTest provides users with coverage information. Using AspectJ one can only implement ConTest's method coverage because AspectJ doesn't provide a pointcut for synchronization blocks or for simple blocks.

The following aspect prints the methods called to System.out. Of course, we could have created a new class that remembers which strings were already printed and not print them more than once. This aspect could be used for method coverage. As we started working on this paper, there was no way to retrieve all the points that were instrumented in AspectJ. This meant that we had no way of knowing when we reached 100% coverage. In AspectJ 1.2.1c there is a new feature added to AspectJ's compiler (-showWeaveInfo) that accomplishes this.

```
public aspect Coverage extends Thread{
    pointcut methodExecution():
        ((execution(* *(..)))&& within(!Coverage));

    after(): methodExecution(){
        System.out.println("called: " +
            thisJoinPointStaticPart.getSignature());
    }
}
```

5 Experimental Results

We conducted several experiments to show the usability of our approach. We tested the aspect SleepNoise against several programs with documented bugs. These programs get a single parameter, which is the amount of threads running simultaneously, and are catagorized as low, medium, and high. We ran the tests 10 times for each category and for each configuration: once as the uninstrumented program, called original in the figure, then using ConTest with simple noise, and finally with the SleepNoise aspect.

Our first program is a concurrent version of the bubble sort algorithm. The bug in this program is that the programmer assumed his threads will finish their work without interruption and accessed the shared resources without synchronization. Figure 1 shows the results for this program. We can clearly see that using SleepNoise increases the chance of concurrent bugs being manifested.

Our second program is also a bubble sort program, with a different bug. The programmer used a sleep statement to initialize all threads before they start working. The results (Figure 2) of this experiment show the great benefit in using this

Figure 1. Bubble sort

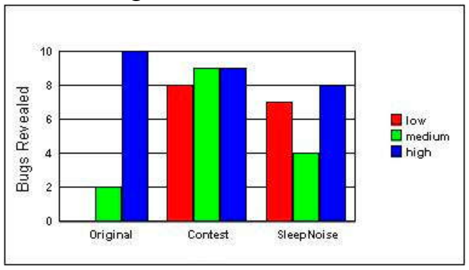
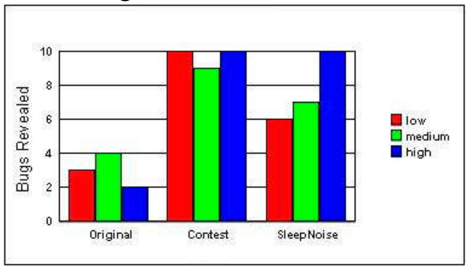
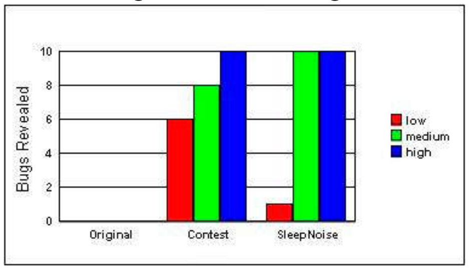


Figure 2. Bubble sort 2



type of testing. We see that SleepNoise increased the chance of finding bugs. Keep in mind that the heuristic is very simple and was not modified to suite the specific program. We believe that by tuning the frequency of adding noise and the type of noise, we can achieve better results.

Figure 3. ID Manager



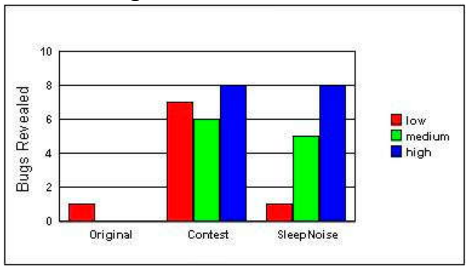
The third program is a program that issues IDs for users. Each user requests an ID and receives a unique ID. The bug is that the programmer assumed that incrementing the ID counter is an atomic operation and didn't protect it. We see in Figure 3 that SleepNoise surpassed ConTest's performance for medium concurrency, This could be the result of a low activation parameter used for ConTest. No further investigation was done since the quality of the heuristics was not the focus of this experiment.

In the last program, the programmer wanted to obtain the locks for two files in different places, but didn't maintain a global partial order on the way she obtained these locks. This introduced a deadlock to the program. SleepNoise increased the chance of the bug being manifested. Had we been able to instrument synchronization blocks, we would have been able to find this bug much easier, consider adding a sleep exactly after the first lock is obtained in one of the two accesses. This would cause the bug to manifest instantly. Unfortunately AspectJ doesn't support point cuts on synchronization blocks.

6 Conclusions and Future Work

We examined the possibility of implementing the instrumentation part of a commercial quality multi-threaded testing tool using AOP. We performed a detailed examination of all the requirements and checked which of them are satisfiable with AspectJ. AspectJ is an open source AOP tool for Java. It was important that not only the concepts be checked, but the fine

Figure 4. Double lock



details. Many of the earlier promises to create a tool that enables time saving due to a higher level of abstraction, such as fourth generation languages, failed to deliver. Each application had some features that could not be implemented, and implementing them in the traditional way reduced much of the benefits of going to higher abstraction level.

We found AOP in general, and AspectJ in particular, to be very well suited for this work. We did find two missing features in AspectJ that are required for complete implementation of the ConTest tool's instrumentation needs. One of them was added while the paper was being written and is now available. The second one, identifying synchronization blocks as places for instrumentation, is still unavailable. Had we decided to move to AspectJ, this point could have been partially mitigated due to the fact that AspectJ is open source and we could have added this capacity ourselves.

We found that with AspectJ we could implement a useful testing tool in two weeks. This is something that would otherwise have taken more than half a year to implement. We consider AOP, and AspectJ, to be very important for implementing high quality open source and academic testing tools. Useful tools in the domains of data race detection, coverage analysis, performance monitoring, trace analysis, concurrent noise making, network load simulation and many others, can now be easily implemented. The developers can now focus on new algorithms and contributions, and the engineering part of instrumentation will become much smaller.

In the experiment described in this paper, we were able to create a commercially useful tool for exposing multi-threaded bugs with very little effort. This exercise demonstrates the viability of our approach. We plan to continue our discussions with the AspectJ community to ensure that the development directions taken will be beneficial to the testing community. We are certain that if more tools follow this approach, AspectJ, together with the artifacts that become available, will create an incentive for additional research in the area, enabling people to concentrate on new contribution and build on existing solutions.

We add the work presented in this paper to the testing benchmark [2] , in the hope that this resource will be utilized by researchers to expand on our work.

References

[1] Aspectj getting started, http://www.elipse.org/aspectj.

[2] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.

[3] J. C. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*. ACM Press, June 2000.

[4] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Testing multi-threaded java programs. *submitted to the IBM System Journal Special Issue on Software Testing*, February 2002.

[5] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002. Also available as `http://www.research.ibm.com/journal/sj/411/edelstein.html`.

[6] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *IPDPS*, 2004.

[7] A. Hartman, A. Kirshin, and K. Nagin. A test execution environment running abstract tests for distributed software. In *Proceedings of Software Engineering and Applications, SEA 2002*, 2002.

[8] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer, STTT*, 2(4), April 2000.

[9] K. Havelund, S. D. Stoller, and S. Ur. Benchmark and framework for encouraging research on multi-threaded testing tools. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, page 286.1. IEEE Computer Society, 2003.

[2]https://qp.research.ibm.com/QuickPlace/concurrency_testing/Main.nsf

[10] E. Itzkovitz, A. Schuster, and O. 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.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[13] Y. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe. Software test coverage and reliability. Technical report, Colorado State University, 1996.

[14] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in java. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, page 10. IBM Press, 2000.

[15] 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.

[16] S. Savage. Eraser: A dynamic race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[17] S. D. Stoller. Model-checking multi-threaded distributed java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking*, 2000.

[18] S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, Oct. 2002.

[19] S. D. Stoller. Testing concurrent java programs using randomized scheduling. In *In Proceedings of the Second Workshop on Runtime Verification (RV)*, volume 70(4) of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.