Automatic Concurrent Debugging Via Minimal Program Mutant Generation with AspectJ

Shady Copty Shmuel Ur

IBM, Haifa Research Labs

{shady,ur}@il.ibm.com

Abstract

Debugging is one of the most time consuming activities in program design. Work on automatic debugging has received a lot of attention and there are a number of symposiums dedicated to this field. The starting point of automatic debugging is that there exists a situation in which a test fails and another where a test succeeds. For example, in one version of the program the test fails and in another it succeeds, or with one scheduler the test fails but succeeds with another. Automatic Debugging searches for the smallest difference which causes the failure. This condition is very useful in identifying and fixing the root cause of the bug.

A new testing method instruments concurrent programs with schedule-modifying instructions to reveal concurrent bugs. The idea is to increase the probability that concurrent bugs (races, deadlocks) will appear. This paper discusses integrating this new testing technology and automatic debugging. Instead of just showing that a bug exists, we can pinpoint its location by finding the minimal set of instrumentations that reveal the bug.

In addition to explaining a methodology for this integration, we show an AspectJ based implementation. We discuss the implementation in detail as it both demonstrates the advantage of open source tools, in their adaptability to change, and how our specific change can be used for other testing tools.

1. Introduction and Motivation

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. Production of mutli-core processors is another trend that focuses on testing and debugging of multi-threaded application in the client space. As a result, commercial enterprises such as Intel, IBM, and Microsoft are giving increased attention to developing methodologies and tools for this domain.

Much research has been done on testing multi-threaded programs. Research has examined data races detection [18], [19], [14], replay in distributed and concurrent contexts[4], static analysis [21] [13] [7], and the problem of generating different interleavings for the purpose of revealing concurrent faults [8] [22]. Model checking [20], coverage analysis [17] [9][3], and cloning [12] are also techniques being to improve testing in this domain.

In a previous paper [6], we demonstrated how to implement a ConTest-like tool using AspectJ. Using 12 lines of AspectJ, we created a testing tool that is useful in finding concurrent bugs. This testing tool works by instrumenting all locations that access global variables with randomly executed sleep statements. When we wanted to implement the full range of ConTest features, we found that AspectJ was missing some features. Because AspectJ is open source, we claimed that user can add the features themselves without waiting for a version that contain them.

In this paper, we describe our work on a new debugging tool that is based on noise creation testing technology. Our tool looks for the minimal set of noise that contain instrumentation that will reveal the bug. The idea is that if you can find one, or several, locations where the instrumentation of noise will reveal the bug, the description of these locations will be very useful to developers. As expected, our experiments found that the knowledge of where a thread switch will cause a bug to manifest is valuable in debugging.

The implementation and the motivation for our tool are similar to that expressed in a thread of papers on delta debugging [5]. [23] [24]. In these papers, a set of program changes was used to induce a bug, with the goal of finding a minimal subset. The set of changes used came from the difference between two program versions: the old one that works and the new one that contains a bug. In this paper, the set of changes that induces bugs is calculated automatically using testing instrumentation technology and is not related to user program changes. Due to the different requirements, we implement a slightly different delta debugging algorithm using AspectH and explain its advantages. The implementation entails the writing of aspects and the tools code along with a modification of AspectJ.

This paper has three contributions. The first shows that the combination of a delta debugging technique and testing via noise generation yields a practical concurrent debugging technique. The second is a new delta debugging algorithm which, in some scenarios, is better than those found in the literature. The third is the actual implementation, which includes modifications to AspectJ that can be applied to other applications.

2. Related work

Debugging is one of the most common activities in the development of computer programs and a lot of thought has been given to its automation. In concurrent programming, one of the domains studied, execution of the same test may sometimes fail and in others succeed. In [5], delta-debugging was used to find places in the interleaving that were indicative of failure. These locations were identified using a replay tool called DEJAVU, used on a special deterministic JVM. In regression testing a new version of the program is examined to see if it contain bugs. Once a test find a bug, the goal of automatic debugging is to find a minimal subset of the changes required to produce the bug. An example of this can be seen in [23] where there are two versions of the program, one that works and another that has a bug. The difference between these programs is 178,000 lines of code. Using delta debugging, they automatically found the single line that caused the bug. Similar ideas are used in another domain where input is reduced to the minimum required to display the bug [24]. This is useful in reducing the number of bug reports (a few may be reduced to the same minimum) and for understanding the core requirement of this bug. The algorithms used in these applications can be found in [23] and can be used to find a group of changes, provided that monotonicity and consistency are guaranteed. This will generally be a problem when testing multi-threaded applications, once the execution of the same test may give different results. This problem can be avoided by using replay on a deterministic JVM.

The testing of multi-threaded applications by inserting schedule modifying statements "noise", such as sleep and yield, has been studied [8] [22]. This is an effective technique for finding out whether a bug exists, but does not look for the root cause of the bug. Studies have been done to find the correct point at which to insert noise [2]. These studies found that noise in many places is not as effective in finding bugs as inserting noise in a few correct places. This means that too much noise may mask the bug, or that the problem is non-monotonic using the definitions of [23].

Studies on bug patterns in multi-threaded programs [11][16] reveal that most bug patterns can be exposed using very few instrumentation points, sometimes even one. However the noise instrumented must be be non-deterministic, i.e., noise that does not impact the interleaving every time it is executed. This requirement means the testing must check whether the noise is in the correct place and is itself non-deterministic, since sometimes, even if the noise is in the correct place, it will fail to produce the bug.

We implemented our work using AspectJ, an aspect-oriented extension to Java. With just a few new constructs, AspectJ can extend Java to provide support for the modular implementation of a range of cross-cutting concerns. Dynamic crosscutting makes it possible to define additional implementations that run at certain well-defined points in the execution of the program. Static crosscutting makes it possible to define new operations on existing types. 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 the ConTest instrumentor. [15]

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 lets you 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 else.

We use advice To implement crosscutting behavior. 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' runs after the program proceeds with that join point. "Around advice" on a join point runs as the join point is reached [1]. The pointcut and the advice type define *where* the instrumentation is done and the advice body defines *what* will actually be instrumented.

3. Algorithms

This section describes the algorithms we use to find the minimal amount of instrumentation needed to uncover the bug. First, we have to deal with the fact that the bugs are not deterministic. If execution succeeds (i.e., found the bug), it does not necessarily means that the instrumentation is in the correct location since it would be found anyway with some probability. When execution fails to find the bug, it does not necessarily mean that the instrumentation is not in the correct place for two reasons. As discussed earlier, the instrumentation must be activated with probability, and in this execution it may have been activated at the wrong time or not activated at all. Additionally, there may have been other thread switches in places that were not instrumented, which masked the bug. We deal with each of these problems separately. The tests we chose for debugging are those in which we find bugs by inserting noise but do not find bugs if we do not add noise. The advantage of looking only at such tests is that there is only a small likelihood the test will find the bug if the instrumentation is in the wrong place. The disadvantage of the approach is that "easy" bugs, that appear even when no noise is used cannot be automatically debugged. We do not have a solution for the numerous cases where the appearance of the bug is common, but finding the root cause of the bug is hard. We address the fact that even correct instrumentation may not produce the bug every time by running each test multiple times and seeing if the bug appears in any of the executions. The number of times the test must be executed depends on how well the bug is hidden and can be fine-tuned when the bug is found. From our experience, between 10 and 30 executions is usually sufficient.

Let $s_1, s_2 ... s_n \in S$ be the set of possible program changes. Program changes are selected such that each change may reveal an existing bug but will not create a new bug in the program. Such changes have to be carefully implemented and the theory and practice of how such changes can be applied to Java programs may be seen in [8] [22]. It is possible that a change, denoted as a bad change, will hide an existing bug. If 'bad changes' exist finding a minimal set of changes becomes more difficult. The work we are currently doing shows that this is very likely. A problem is called monotonic if, for every set that finds bugs, all its supersets also find bugs [23]. The existence of interrelations between instrumentations may cause our problem to be non-monotonic.

A very important issue is the expected size of F - the minimal group of changes needed to reveal a bug. Studies on bug patterns [11][16] have shown that F will generally be very small and will often be a singleton. Finding a singleton is very easy. The simplest algorithm we use creates $|n|$ mutations of the program, each with a single change, and then checks which mutation finds the bug. The advantages of this trivial algorithm are its simplicity and the fact that it is oblivious to the existence of bad changes. One disadvantage is its complexity, as the number of possible changes is linear in S. The number of changes from which we select is dominated by the number of accesses to global variables in the files that contain synchronization elements; this thurs out to be approximately the number of lines of code divided by five. The second disadvantage is that this algorithm will only work if the set of changes is a singleton. If more than one change is necessary, this algorithm will fail.

To alleviate the complexity problem, a second algorithm was implemented to perform a search. In order to search, we need to perform queries on sets of elements. We use query Q, which receives $s \subset S$ and returns Yes if $F \subset s$ and No otherwise (i.e., $\exists x \in F, x \notin s$)

At each stage in this algorithm, we apply half of the remaining changes. If a bug is found, we continue with that half, and if not with the other. The complexity of this algorithm is $log(n)$, which is very good; however, it is still limited to a singleton solution. If the problem is non-monotonic, the search algorithm may not work.

We set out to devise an algorithm that looks for sets, and is optimized to search for small sets, as we expect most of the solutions to be small. Another good property for the algorithm is that every query has a relatively small number of changes. This property is desirable for efficiency, as every instrumentation has costs in runtime and accuracy. We know from experiments [2] that a program with a lot of instrumentation is less likely to exhibit the bug than a program that uses less instrumentation but has it in the correct places. Having less instrumentation is also beneficial from a performance point of view. Due to the existence of bad instrumentation, and the non-monotonicity of the problem, the less instrumentation we have, (as long as we have the right one) the less likely we are to face these problems.

Delta Debugging (DD) is a well known algorithm used to search for sets of changes. The DD algorithm suggested in [23] works as follows: you start with two sets $c \subset c'$, such that the program works with c and does not work with c'. We start with c as the empty set and c' as the full set of changes that finds the bug. We then roughly divide the changes in c' in two. If testing with the first part yields the bug, continue recursively with that part. If not, try the second part; if that yields the bug continue recursively with the second part. If not, we know that a subset of the solution is in the first part and a subset is in the second part. Continue recursively searching the first part, while implementing all the changes that belong to the second. At the same time, search the second part while implementing all the changes to the first. The minimal solution is the union of the two searches. Figure 1(a) [23] describes the search for a minimal subset using this algorithm. Aside from being very simple and proven in practice also lends itself to parallelism. When a search is split, the search done on the first part and on the second one are independent and can be done in parallel. Given enough processors, the complexity of the algorithm is logarithmic in the number of changes investigated. It is sufficient to have the number of processors equal to the number of changes found; this is usually very small.

As stated above, for our application it is desirable to keep the number of changes in each test as low as possible. To alleviate this problem, the algorithm in [23] can be modified to a sequential algorithm. If all the changes are in part a or in part b, there is no change. If the changes are in both parts a and b, we then search for the relevant changes in part a (as before). Next we search for the relevant searches in part b, while holding only the relevant changes in part a (as apposed to holding all the changes). While the algorithm can no longer be parallelized, it is more efficient for our application when run on a single processor.

The algorithm we actually used in the experiments is as follows:

1. Order the modification (give them numbers from 1 to N).
2. Create a set S of instrumentation, which is the output of the process, and initialize it with the empty set.
3. Create an index I equal to the index of the last instrumentation point found and initialize it with N.
4. Repeat until S is a solution (i.e., finds a bug).
   (a) Use binary search to look for the smallest K in 1 ... I, such that Q(1 ... K-1$\cup S$) does not find the bug and Q(1 ... K $\cup S$) does.
   (b) Set I to K-1.

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✓	
2	c_2	.	.	.	.	5	6	7	8	✓	
3	c_1	1	2	.	.	5	6	7	8	✓	
4	c_2	.	.	3	4	5	6	7	8	✗	
5	c_1	.	.	3	.	5	6	7	8	✗	3 is found
6	c_1	1	2	3	4	5	6	.	.	✗	
7	c_1	1	2	3	4	5	.	.	.	✓	6 is found
Result		.	.	3	.	.	6	.	.		

(a) Delta Debugging

Step	c_i	Configuration								*test*	
1	c_1	1	2	3	4	.	.	.	.	✓	
2	c_2	.	.	.	.	5	6	7	8	✓	
3	c_1	1	2	.	.	5	6	7	8	✓	
4	c_2	.	.	3	4	5	6	7	8	✗	
5	c_1	.	.	3	.	5	6	7	8	✗	3 is found
6	c_1	.	.	3	.	5	6	.	.	✗	
7	c_1	.	.	3	.	5	.	.	.	✓	6 is found
Result		.	.	3	.	.	6	.	.		

(b) Modified Delta Debugging

Figure 1. Delta debugging and modified delta debugging

(c) Add K to S.

Example:

```
1 .. 100 are the possible modifications.
                     F = {1, 20, 40, 60}

Look for the one with the largest index
Q(1..50) replies No as there is one outside (60)
Q(1..75)  replies Yes
and so forth.
.... until we found it is 60
add the 60'th modification to S, change I to 59

S is still not a solution, continue

Start looking for the second one
Q(1..30, 60) replies No (because of 40)
Q(1..45,60) replies Yes
and so forth.
... until we found it is 40
add the 40'th modification to S, change I to 39

When S becomes a solution (we find F), we are done
```

This algorithm is slightly better than our modification of the DD algorithm. To find a singleton (if we do not know that the reply is a singleton) the average complexity of the DD algorithm is 1.5log(N). This is because every time we check, we choose with 50 percent probability in the first try and with 50 percent in the second. Roughly the same calculation will hold when the solution is a small number of changes. Another advantage is that our queries, on average, have a smaller amount of instrumentation.

4. Implementation

We used several sub-components to implement our solution:

- Component that extracts the set S of all possible locations where we may want to add noise.
- Component that can instrument noise at any subset $s \subset S$.
- Component that can determine if a program, with noise applied to $s \subset S$, displays a concurrent bug.

The following sections review these sub-components and explain how we implemented them.

4.1 Extracting the initial set of possible changes

Our technique use AspectJ to extract the set of all possible locations to which noise can be added. AspectJ's compiler uses (-showWeaveInfo) option to print out information on all the pointcuts that were advised. The information is presented in the following format:

```
Type 'Test' (Test.java:46) advised by
  before advice from 'Initial' (Initial.java:9)
```

We extract all possible variable sets and gets in a certain program by using the -showWeaveInfo option in AspectJ's compiler with the following aspect:

```
import java.util.*;

public aspect Initial extends Thread{
    pointcut noiseVictem():(
      (get(* *) || set (* *)) &&
      within(!Initial)
    );

    private static Random rand = new Random();
    before(): noiseVictem() {
    if (rand.nextInt(100) == 1){
                //activation probability
      yield();
    }
  }
}
```

This is the same aspect we use to instrument noise in [6]. For our purpose, the advise itself is not important; the important part is getting all the locations.

4.2 Applying a subset of the changes to a program

The information retrieved by -showWeaveInfo prints out locations as pairs of class name and line number. The problem we faced was that AspectJ's pointcuts don't support a specific line number to advise. Hence, there is no way to tell AspectJ to instrument at a specific line number. The good news is that AspectJ is open source and we were able to alter a pointcut type to allow instrumentation of specific line numbers. We changed the "'Within"' pointcut, so it receives two parameters, a type pattern and a line number, with (0) denoting a wildcard line number. Take for example, a program with a class called ClassA, which has an access to a variable at lines 1, 2 and 3. If we want to instrument lines 2 and 3, we create an aspect as follows:

```
import java.util.*;

public aspect NoiseAspect extends Thread{
    pointcut noiseVictem():(
      (get(* *) || set (* *)) &&
      within(ClassA, 2) &&
      within(ClassA, 3) &&
      within(!NoiseAspect,0)
    );

    private static Random rand = new Random();
    before(): noiseVictem() {
    if (rand.nextInt(100) == 1){
         // activation probability
      yield();
    }
  }
}
```

Weaving this aspect with the debugged program would add noise to lines 2 and 3 of ClassA.

A few modifications were required to add this change to AspectJ:

- Changes to the WithinPointcut class
  - Its constructor now receives two parameters: a type, as it did before, and a line number. The line number is kept in a private data member.
  - The methods "'matchInternal"' and "'match"', which check if a certain pattern is matched, now check for line number matching in addition to type pattern.
  - The method "'fastMatch"' can no longer be used for pattern matching since FastMatchInfo doesn't keep the line numbers. We decided not to fix this and now fastMatch returns FuzzyBoolean.MAYBE;.
  - The "'equals"' method now tests for line number in additon to type patterns.
  - The "'write"' and "'read"' methods, which are used for serialization, were changed to keep the line number in addition to the type pattern. In additon to the WithinPointcut class.
- Alterations to the class PatternParser. This class now expects a second argument for the within pointcut from which it creates a new WithinPointcut object using our new constructor.

  To determine whether a program that was instrumented at a subset of all locations reveals the bug, we execute the program a number of times. If the bug appears more than a specific threshold of times we declare it successful.

4.3 Putting it all together

We start with a program that contains a bug that doesn't appear when the program is run normally, but appears when instrumented with noise. We first retrieve the set of all possible locations that can be instrumented with noise. We then use one of the search algorithms described in the previous section. In each iteration, for a given subset of all possible locations, we create an aspect for the specific subset, weave it into the debugged program with our altered version of AspectJ, and then we test to see if the bug appears enough times. We then move on to the next iteration.

5. Experiments

We conducted several experiments to show the feasibility of our approach, mainly on code taken from the concurrent bugs benchmark [10]. We illustrate the approach using synthetic programs created for this work and a program from Sun that demonstrates concurrent issues. For each program, we examine the performance of each search algorithm described in Section 3.

5.1 Increment operator

In Java, the Increment operator is not atomic. A common fault is to consider it as such, as demonstrated by the following program:

```
01. public class Atomic extends Thread {
02.
03.   private static long sharedVariable=0;
04.
05.   public Atomic () {
06.   }
07.
08.   public void run () {
09.         sharedVariable++;
10.   }
11.
12.
13.   public static void main ( String[] args )
            throws InterruptedException {
14.      Atomic a1 = new Atomic();
15.      Atomic a2 = new Atomic();
16.      a1.start();
17.      a2.start();
18.      a1.join();
19.      a2.join();
20.      System.out.println ( sharedVariable );
21.   }
22.}
```

This program has a bug in line 9. For this program to work properly, we should have added a synchronize around the increment operator. When we ran our tool on this program, all three search algorithms reported line 9 as the problematic one. This program has three program locations that are candidates for instrumentation, lines 3, 9 and 20. Table 1 shows the number of iterations it took for each search algorithm to reveal the location of the bug. The binary search worked bestas expected. The binary set search algorithm required an extra iteration, since after each location discovered it checks whether it found a minimal subset or if more searching is needed; this costs an extra iteration.

Algorithm	Number of iterations
Linear	3
Binary	2
Binary Set	3

Table 1. Number of iterations until the bug location was discovered for the Atomic program

5.2 Bank simulator

This program, created by Sun to show concurrent problems, simulates a bank with several customers. Each customer has an account, where he can decide to deposit or withdraw a certain amount of money at random. The bank maintains the balance for all accounts and for the bank itself. The bank's balance is the sum of all accounts. In this program, the programmer keeps a variable for the bank's balance and an array of balances for all the customers. Each time a customer performs an operation, both the bank's balance and the customer's balance are updated. The bug is that the update is not done atomically. The program has 29 possible noise locations. Table 2 shows that the binary search was the most effective for this program. All the search algorithms pointed to line 78 of the bank class.

```
...
76.  public static void Service(int id,int sum){
77.         accounts[id].Balance += sum;
78.         Bank_Total += sum;
79.  }
...
```

Algorithm	Number of iterations
Linear	29
Binary	5
Binary Set	6

Table 2. Number of iterations until the bug location was discovered for the bank simulation program

On average we will expect the number of iterations of the linear search to be half the instrumented locations. In this example it happend to be the last location in the program.

5.3 Interaction between two locations

We synthesized a short program where one location was not enough to reveal the bug and the program's entire code could be shown here. We have seen quite a few examples in the field where one location is not enough.

```
public class TwoChanges extends Thread {

    private int mode;

   private static int x=1;
   private static int z=4;

   public TwoChanges ( int mode ) {
      this.mode = mode;
   }

   public void run () {
      if (mode==0) {
      for (int i=0; i<10000; ++i) {
        if (x != 0){
        try{
          z = 5/x;
          } catch (Exception e)
            {System.out.println("bug");}
          }
        }
      }
      else {
      for (int i=0; i<10000; ++i)
        {
          x=1;
          x=0;
          x=1;
        }
      }
   }

   public static void main ( String[] args )
```

```
33.      throws InterruptedException {
34.       TwoChanges a1 = new TwoChanges(0);
35.       TwoChanges a2 = new TwoChanges(1);
36.
37.       a1.start();
38.       a2.start();
39.       a1.join();
40.       a2.join();
41.       System.out.println ( z );
42.    }
43.}
```

Interleaving that goes through line 26 and then line 17 is required for the bug to appear in this program, therefore, adding noise in one of the two locations is not enough. If we only add it in line 26, line 15 protects the bug. If we add it in line 17, there is little chance the scheduler will choose to perform a context switch in line 26. This program has 11 possible locations at which noise can be added. As expected, both algorithms that attempt to find a single location failed. The set detected by the binary set search included lines 17 and 27, and was found after eight iterations. It is interesting that the linear search had to go over all the possible locations to figure out that it failed, while the binary search needed only two iterations to arrive at the same conclusion.

6. Conclusions and Future Work

This paper contain three contributions: a technique for pin-pointing the location of concurrent faults, a new delta debugging algorithm, and a modification of AspectJ that enables the implemention of more testing technologies.

The technique for pin-pointing the location of concurrent faults is a step in a direction towards automatically fixing concurrent bugs. In previous work we exposed existing bugs and studied bug patterns. After pin-pointing the bug location, the next step is to suggest a fix. This goal is still far away, especially in the unsupervised mode, but we believe the work shown in this paper is an important step in the right direction.

To achieve our goal, we developed a new delta debugging algorithm. This algorithm is superior for our implementation and may be of further use to other applications. Traditional DD algorithm are can easily take advantage of parallel computing. Different usage scenarios lend themselves to different algorithms.

We are now performing experiments on real applications. We are working both on improving the query using statistical technique and on new algorithms, for example reinforcement learning, mainly from the domain of machine learning.

In our previous work [6], we saw that AspectJ can be used for testing but fell short in fulfilling the needs of ConTest [9] because some features were missing. In this paper, we took advantage of the fact that AspectJ is an open source tool and altered it to meet our needs. Performing our changes to AspectJ was relatively simple due to the fact that it is well written and easy to comprehend. Using our altered version of AspectJ we could implement our tool to its full extent. The change we have made is useful for a number of other testing tools. For example when performing coverage measurement and wanting to have minor performance impact. Coverage measurement is usually done by instrumenting the code and measuring which instrumentation points where executed. The main performance impact is due to the commonly executed instrumentations. After each test removing the points that were executed will result in very good performance. Creating such a coverage tools with AspectJ is now feasible due to our enhancement.

It is clear to us that AspectJ is a very powerful solution for academic purpose. When creating an industrial strength tool, some changes will be necessary to AspectJ in order to for all the features to work. A specific study, based on the requirements, will be needed for each industrial tool to check if AspectJ is suitable.

References

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

[2] Yosi Ben-Asher, Yaniv Eytani, and Eitan Farchi. Heuristics for finding concurrent bugs. In *International Parallel and Distributed Processing Symposium, IPDPS 2003, PADTAD Workshop*, 2003.

[3] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 206–212, New York, NY, USA, 2005. ACM Press.

[4] 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.

[5] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 210–220, New York, NY, USA, 2002. ACM Press.

[6] Shady Copty and Shmuel Ur. Multi-threaded testing with aop is easy, and it finds bugs! In *Proceedings of Europar 2005*. Springer-Verlag, 2005.

[7] James C. Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*. ACM Press, June 2000.

[8] 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.

[9] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel 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`.

[10] Yaniv Eytani and Shmuel Ur. Compiling a benchmark of documented multi-threaded bugs. In *IPDPS*, 2004.

[11] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *IPDPS*, page 286, 2003.

[12] 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.

[13] 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.

[14] 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.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[16] Brad Long and Paul A. Strooper. A classification of concurrency failures in java components. In *IPDPS*, page 287, 2003.

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

[18] 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.

[19] Stephen Savage. Eraser: A dynamic race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

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

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

[22] Scott 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.

[23] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

[24] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.