Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 151–165
www.elsevier.com/locate/entcs

Toward Automatic Concurrent Debugging Via Minimal Program Mutant Generation with AspectJ

Shady Copty

IBM, Haifa Research Labs
Haifa University Campus
Haifa, 31905, Israel

Shmuel Ur

IBM, Haifa Research Labs
Haifa University Campus
Haifa, 31905, Israel

Abstract

Debugging is one of the most time-consuming activities in program design. Work on automatic debugging has received a great deal of attention and there are a number of symposiums dedicated to this field. Automatic debugging is usually invoked when a test fails in one situation, but succeeds in another. For example, a test fails in one version of the program (or scheduler), but succeeds in another. Automatic debugging searches for the smallest difference that causes the failure. This is very useful when working to identify and fix the root cause of the bug.
A new testing method instruments concurrent programs with schedule-modifying instructions to reveal concurrent bugs. This method is designed to increase the probability of concurrent bugs (such as races, deadlocks) appearing. This paper discusses integrating this new testing technology with 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 the adaptability of open source tools and how our specific change can be used for other testing tools.

Keywords: Interleaving, Multi-threading, Delta Debugging, Software Engineering, Testing and Debugging

1 Introduction and Motivation

The increasing popularity of concurrent Java programming—for the Internet as well as the server side—has brought the issue of concurrent defect analysis to the

[1] Email: copty@il.ibm.com
[2] Email: ur@il.ibm.com

doi:10.1016/j.entcs.2007.04.001

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 multi-core processors is another trend that highlights the need for testing and debugging of multi-threaded applications 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 race detection [20], [21], [15]; replay in distributed and concurrent contexts [4]; static analysis [23], [14], [7]; and the problem of generating different interleaving to reveal concurrent faults [8], [24]. Model checking [22], coverage analysis [18], [9],[3], and cloning [13] are also techniques used to improve testing in this domain.

In a previous paper [6], we demonstrated how to build a testing tool that randomizes the interleaving on top of AspectJ. AspectJ implements aspect oriented programming for the Java language. Using 12 lines of AspectJ, we created a testing tool similar to ConTest [8], an IBM commercial tool that proved useful in finding concurrent bugs. This kind of testing tool [8],[24] works by instrumenting locations whose timing may impact the program result, such as access to global variables, with randomly executed sleep statements. When we wanted to carry out a full implementation of ConTest, we found that AspectJ was missing some features. Because AspectJ is open source, we claim that test tool makers can add the features themselves without waiting for an AspectJ version that contains them.

In this paper, we describe our work on a new debugging tool that is based on noise creation testing technology. Noise creation, in our context, is insertion of delays, random or otherwise to modify the timing of the program under test. Noise generation is very useful in finding intermittent bugs. Our tool looks for the minimal set of noise that contains instrumentation that reveals the bug. If one or several locations can be found where the instrumentation of noise reveals the bug, the description of these locations can be very useful to developers. As expected, our experiments found that it is valuable, in debugging, to know where a thread switch causes a bug to be manifested. A different approach [10] uses genetic algorithms. In that work, instead of looking for the minimal set of changes, they searched for the set of changes that yields the maximum likelihood of finding the bug.

The implementation and motivation for our tool are similar to those expressed in a thread of papers on delta debugging (DD) [5], [25], [26]. In these papers, a set of program changes is used to induce a bug, with the goal of finding a minimal subset. The set of changes comes 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 DD algorithm using AspectJ and explain its advantages. The implementation entails the writing of aspects and tool code, along with a modification of AspectJ.

This work is part of initial studies in SHADOWS, an EU project whose goal is to

create technology for self-healing. For intermittent bugs that depend on specific interleaving, it may be possible to automatically detect and remove the bug-causing interleaving. While the work is not yet mature, we believe that it is interesting due to the following contributions: we show, at least on small programs, that the combination of a DD technique and testing via noise generation yields a practical concurrent debugging technique. We show a new DD algorithm that, in some scenarios, is better than those found in the literature. In addition, the actual implementation is detailed, 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 much thought has been given to its automation. In concurrent programming, which is one of the domains studied, the same test may sometimes fail and sometimes succeed. In [5], DD found 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 contains bugs. Once a test finds 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 [25], where two versions of the program exist—one that works and another that has a bug. The difference between these programs is 178,000 lines of code. Using DD, the single line that caused the bug was automatically pinpointed.

Similar ideas are applied in another domain where the test is reduced to the essential part required to display the bug [26]. DD is useful in reducing the number of bug reports and for understanding the core requirement of this bug. The algorithms used in these applications can be found in [25] where it is used to find a group of changes, provided that monotonicity and consistency are guaranteed. This is generally a problem when testing multi-threaded applications, when 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], [24]. This is an effective technique for finding out whether a bug exists, but it 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 [25].

Studies on bug patterns in multi-threaded programs [12],[17] reveal that most bug patterns can be exposed using very few instrumentation points, and sometimes only one. However, the instrumented noise must 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. Sometimes, even if the noise is in the correct place, it fails 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 [19]. Dynamic cross-cutting makes it possible to define additional implementations that run at certain well-defined points in the execution of the program. Static cross-cutting makes it possible to define new operations on existing types. Dynamic cross-cutting 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; advices are method-like constructs used to define additional behavior at join points; and aspects are units of modular cross-cutting implementation, composed of pointcuts, advices, and ordinary Java member declarations. We use dynamic cross-cutting to implement the features of ConTest using AspectJ, in a manner similar to that used by the ConTest instrumentor [16].

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 the assignments to all the variables in the program. Pointcuts pick out join points, but don't do anything else.

We use advices to implement crosscutting behavior. An 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* is actually 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., if it finds the bug), it does not necessarily mean that the instrumentation is in the correct location, since the bug would be found anyway with some degree of 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 be other thread switches

in places that were not instrumented that mask 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 no noise is added. When looking only at such tests, there is only a small likelihood of the test finding the bug if the instrumentation is in the wrong place. The disadvantage of this approach is that "easy" bugs that appear even when no noise is used cannot automatically be 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. This number can be fine-tuned once 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 so that each change may reveal an existing bug but does not create a new bug in the program. Such changes have to be carefully implemented. The theory and practice of how such changes can be applied to Java programs is explained in [8], [24]. It is possible that a change, denoted as a bad change, hides an existing bug. If bad changes exist, finding a minimal set of changes becomes more difficult. Our current work shows that this is very likely. A set of changes is monotonic if, for every set that finds bugs, all its supersets also find bugs [25]. The existence of interrelations between instrumentations may cause our problem to be non-monotonic.

A very important issue is the expected size of F, which is the minimal group of changes needed to reveal a bug. Studies on bug patterns [12],[17] have shown that F is generally very small and is often a singleton. Finding a singleton is very easy. The simplest algorithm we use creates $|n|$ mutations of the program, each created with a single addition of a sleep statement, 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. A 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 turns out to be about the number of lines of code divided by five in the program we viewed (mainly industrial middleware programs). The second disadvantage is that this algorithm only works if the set of changes is a singleton. If more than one change is necessary, this algorithm fails.

To alleviate the complexity problem, a second algorithm was implemented to perform a search. 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, we continue 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 solution is not a singleton, then the half we choose may contain a subset of the changes, in which case we continue with the other half

and not find a solution. If the problem is non-monotonic, the search algorithm may not work since not finding the bug does not mean that the solution is not included in the set.

We set out to devise an algorithm that is optimized to search for small sets as we expect most of the solutions to be small. Another advantage of the algorithm is that every query has a relatively small number of changes. This property is desirable for efficiency, as every instrumentation incurs costs in runtime and accuracy. We have seen [2] that a program with more instrumentations 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, the less likely we are to face these problems, assuming, of course, the right instrumentation.

DD is a well-known algorithm for searching for sets of changes. The DD algorithm suggested in [25] works as follows: start with two sets $c \subset c'$, such that the program works with c and does not work with c'. Start with c as the empty set and c' as the full set of changes that finds the bug. 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 part yields the bug, continue recursively. Otherwise a subset of the solution must be in the first part and another subset in the second part. Continue recursively searching the first part, while implementing all the changes from 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) [25] describes the search for a minimal subset using this algorithm. Aside from being very simple and proven in practice, the algorithm also lends itself to parallelism. When a search is split, the two parts of the search 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 number is usually very small.

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

Fig. 1. Delta debugging and modified delta debugging

As stated above, for our application, it is desirable to keep the number of changes in each test as small as possible. The DD algorithm in [25] can be modified to a sequential algorithm that requires less changes. If all the changes are in part a

or part b, there is no change. If the changes are in both a and b, search for the relevant changes in part a (as before). Next, search for the relevant changes in part b, while holding only the relevant changes in part a (as opposed 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.

In the experiments we used the BinarySet algorithm which is better than the modified DD and is implemented using the following procedure:

(i) Arbitrarily assign to the modification numbers from 1 to N.

(ii) Create a set S of instrumentation, which is the output of the process, and initialize it with the empty set.

(iii) Create an index I equal to the index of the last found instrumentation point and initialize it with N.

(iv) Repeat until S is a solution (i.e., finds a bug).
   (a) Use a 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.
   (c) Add K to S.

Example 3.1

```
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 find it is 60
add the 60th 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 find it is 40
add the 40th 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 holds 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. These components perform the following actions:

- Extracts the set S of all possible locations where we may want to add noise. (see [8] as to where this should be done)
- Instruments noise at any subset $s \subset S$.
- Determines 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 uses AspectJ to extract the set of all possible locations to which noise can be added. AspectJ's compiler uses the -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 noiseVictim():(
      (get(* *) || set (* *)) &&
      within(!Initial)
    );

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

This is the same aspect used to instrument noise in [6]. For our purpose, the advice 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 do not 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 access to a variable at lines 1, 2, and 3. 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 into the debugged program adds 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 also check for line number matching.
  - The method "fastMatch" can no longer be used for pattern matching because FastMatchInfo doesn't keep line numbers. We decided not to fix this and now

fastMatch returns FuzzyBoolean.MAYBE;.

- The "equals" method now tests for line number in addition 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.
- The PatternParser class now expects a second argument for WithinPointcut, 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 specified 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 apply 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 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 [11]. 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 synchronization 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 best, as expected. The binary set search algorithm required an extra iteration. This is because after each location was discovered, it checked if it has found a minimal subset or if more searching is needed. This check cost an extra iteration.

Algorithm	Number of iterations
Linear	3
Binary	2
Binary Set	3

Table 1
Number of iterations needed to discover the bug location 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 can decide to deposit or withdraw a certain amount of money at random from their respective bank accounts. The bank maintains the balance of all accounts and of the bank itself. The bank balance is the sum of all accounts. In this program, the programmer keeps a variable for the bank balance and an array of balances for the customers. Each time a customer performs an operation, both the bank balance and the customer's balance are updated. The bug is that this 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 needed to discover the bug location for the bank simulation program

On average, we expect the number of iterations of the linear search to be half the instrumented locations. In this example, it happened to be the last location in the program which is the reason we had so many iterations but even half the number of locations is significantly more than a log.

5.3 Interaction between two locations

We synthesized a short program in which one location is not enough to reveal the bug. We chose this program because its entire code can be shown here. We have seen quite a few examples in the field where one location is not enough.

```
01. public class TwoChanges extends Thread {
02.
03.    private int mode;
04.
05.   private static int x=1;
06.   private static int z=4;
07.
08.   public TwoChanges ( int mode ) {
09.      this.mode = mode;
10.   }
11.
12.   public void run () {
13.      if (mode==0) {
14.      for (int i=0; i<10000; ++i) {
15.        if (x != 0){
16.        try{
17.          z = 5/x;
18.          } catch (Exception e)
               {System.out.println("bug");}
19.          }
```

```
20.         }
21.       }
22.       else {
23.       for (int i=0; i<10000; ++i)
24.         {
25.           x=1;
26.           x=0;
27.           x=1;
28.         }
29.       }
30.   }
31.
32.   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 those two locations is not enough. If we only add it in line 26, line 15 protects the bug. If we only 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. The linear search had to review all possible locations to conclude that it failed, while the binary search needed only two iterations to arrive at the same conclusion.

6 Conclusions and Future Work

This paper contains three contributions: a technique for pinpointing the location of concurrent faults, a new delta debugging algorithm, and a modification of AspectJ that enables the implementation of more testing technologies.

The technique for automatically locating the relevant concurrent faults is a step towards automatically fixing concurrent bugs. In previous work, we exposed existing bugs and studied bug patterns. After pinpointing the bug location, the next step is to suggest a fix. This goal is still far from being attained, 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 DD algorithm. This algorithm is superior for our implementation and may be of further use to other applications. Traditional DD algorithms can easily take advantage of parallel computing. Different usage scenarios lend themselves to different algorithms.

We are now performing experiments on real applications. The key point is the question of monotonicity. If, in practice, the problem proves to be monotonic, then the algorithms suggested in this paper have practical applications. Even if there are 100,000 instrumentation points, due to the logarithmic nature of the algorithm and the use scenario the running time will be reasonable. If the problem turns out to be non-monotonic, then alternative search techniques will be necessary.

Another problem is that even if the bug can be seen, the probability of exposing it depend on the instrumentation points chosen. If it goes down bellow a certain threshold it will be very hard to detect it. To find the bugs automatically many tests need to be run. It will take longer on long running tests. Deadlocks can also be found using this technique as the tests that find them use timeouts or look for circular lock probability.

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, since it is well written and easy to comprehend. Using our altered version of AspectJ, we were able to implement our tool to its fullest extent. The change we made is useful for a number of other testing tools, for example when performing coverage measurement and aiming to reduce the performance impact. Coverage measurement is usually done by instrumenting the code and measuring which instrumentation points were executed. The main performance impact is due to the commonly executed instrumentations. After each test, removing the instrumentation points that were executed results in very good performance. Creating such 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 purposes. When creating an industrial strength tool, some changes must be made to AspectJ 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. *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. Concurrent Java test generation as a search problem. In *Proceedings of the fifth workshop of runtime verification*, 2005.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] Marius Marin, Leon Moonen, and Arie van Deursen. An approach to aspect refactoring based on crosscutting concern types. In *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[20] 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.

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

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

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

[24] 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.

[25] 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.

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