

Lightweight Defect Localization for Java

Valentin Dallmeier, Christian Lindig, and Andreas Zeller

Saarland University, Saarbrücken, Germany
{dallmeier, lindig, zeller}@cs.uni-sb.de

Abstract. A common method to localize defects is to compare the *coverage* of passing and failing program runs: A method executed only in failing runs, for instance, is likely to point to the defect. Some failures, though, come to be only through a specific *sequence* of method calls, such as multiple deallocation of the same resource. Such sequences can be collected from arbitrary Java programs at low cost; comparing object-specific sequences predicts defects better than simply comparing coverage. In a controlled experiment, our technique pinpointed the defective class in 36% of all test runs.

1 Introduction

Of all debugging activities, locating the defect that causes the failure is by far the most time-consuming. To assist the programmer in this task, various automatic methods rank the program statements by the *likelihood* that they contain the defect. One of the most lightweight methods to obtain such a likelihood is to compare the *coverage* of passing and failing program runs: A method executed only in failing runs, but never in passing runs, is correlated with failure and thus likely to point to the defect.

Some failures, though, come to be only through a *sequence* of method calls, tied to a *specific object*. As an example, consider streams in Java: If a stream is not explicitly closed after usage, its destructor will eventually do so. However, if too many files are left open before the garbage collector destroys the unused streams, file handles will run out, and a failure occurs. This problem is indicated by a sequence of method calls: if the last access (say, `read()`) is followed by `finalize()` (but not `close()`), we have a defect.

In this paper, we therefore explore three questions:

1. **Are sequences of method calls better defect indicators than single calls?** In any Java stream, calls to `read()` and `finalize()` are common; but the sequence of these two indicates a missing `close()` and hence a defect.
2. **Do method calls indicate defects more precisely when collected *per object*, rather than globally?** The sequence of `read()` and `finalize()` is only defect-revealing when the calls pertain to the same object.
3. **Do deviating method calls indicate defects in the callee—or in the caller?** For any Java stream, a missing `close()` indicates a defect in the caller.

Generalizing to arbitrary method calls and arbitrary defects, we have set up a tool that instruments a given Java program such that sequences of method calls are collected on

a per-object basis. Using this tool, we have conducted a controlled experiment that answers the above questions. In short, it turns out that (1) sequences predict defects better than simply comparing coverage, (2) per-object sequences are better predictors than global sequences, and (3) the caller is more likely to be defective than the callee. Furthermore, the approach is lightweight in the sense that the performance is comparable to coverage-based approaches. All these constitute the contribution of this paper.

2 How Call Sequences Indicate Defects

Let us start with a phenomenological walkthrough and take a look at the AspectJ compiler—more precisely, at its bug #30168. This bug manifests itself as follows: Compiling the AspectJ program in Fig. 1 produces illegal bytecode that causes the virtual machine to crash (run r_x):

```
$ ajc Test3.aj
$ java test.Test3
test.Test3@b8df17.x

Unexpected Signal : 11 occurred at PC=0xFA415A00
Function name=(N/A)
Library=(N/A)
...
Please report this error at http://java.sun.com/...
$ -
```

In order to fix this problem, we must locate the defect which causes the failure. As the AspectJ compiler has 2,929 classes, this is a nontrivial task. To ease the task, though, we can focus on *differences* in the program execution, in particular the difference between a passing run (producing valid Java bytecode) and the failing run in question. Since the outcome of passing and failing runs is different, chances are that earlier differences in the program runs are related to the defect. For the AspectJ example in Figure 1, we can easily identify a passing run—commenting out Line 32, for instance, makes AspectJ work just fine (run r_v).

Since capturing and comparing entire runs is costly, researchers have turned to *abstractions* that summarize essential properties of a program run. One such abstraction is *coverage*—that is, the pieces of code that were executed in a run. Indeed, comparing the coverage of r_v and r_x reveals a number of differences. The method `getThisJoinPointVar()` of the class `BcelShadow`, for instance, is only called in r_x , but not in r_v , which makes `BcelShadow.getThisJoinPointVar()` a potential candidate for causing the failure.

Unfortunately, this hypothesis is wrong. In our AspectJ problem, the developers have eventually chosen to fix the bug in another class. Although it might be possible to alter the `BcelShadow` class such that the failure no longer occurs, `BcelShadow` is not the location of the defect. In fact, none of the methods that are called *only* within r_x contain the defect.

However, it may well be that the failure is caused not by a single method call, but rather by a *sequence of method calls* that occurs only in the failing run r_x . Such

```

1  package test;
2  import org.aspectj.lang.*;
3  import org.aspectj.lang.reflect.*;
4
5  public class Test3 {
6      public static void main(String[] args) throws Exception {
7          Test3 a = new Test3();
8          a.foo(-3);
9      }
10     public void foo(int i) {
11         this.x=i;
12     }
13     int x;
14 }
15
16 aspect Log {
17     pointcut assign(Object newval, Object targ):
18         set(* test..*) && args(newval) && target(targ);
19
20     before(Object newval, Object targ): assign(newval,targ) {
21         Signature sign = thisJoinPoint.getSignature();
22         System.out.println(targ.toString() + "." + sign.getName() +
23             ":@" + newval);
24     }
25
26     pointcut tracedCall():
27         call(* test..*(..)) && !within(Log);
28
29     after() returning (Object o): tracedCall() {
30         // Works if you comment out either of these two lines
31         thisJoinPoint.getSignature();
32         System.out.println(thisJoinPoint);
33     }
34 }

```

Fig. 1. This AspectJ program causes the Java virtual machine to crash.

sequences can be collected for specific objects, either collecting the incoming or the outgoing calls. This sequence, for instance, summarizes the outgoing method calls of a `ThisJoinPointVisitor` object in r_x :

```

    ThisJoinPointVisitor.isRef(),
    {
    ThisJoinPointVisitor.canTreatAsStatic(),
    MethodDeclaration.traverse(),
    ThisJoinPointVisitor.isRef(),
    ThisJoinPointVisitor.isRef()
    }

```

This sequence of calls does not occur in r_v —in other words, only in r_x did an object of the `ThisJoinPointVisitor` class call these five methods in succession. This difference in the `ThisJoinPointVisitor` behavior is correlated with failure and thus makes `ThisJoinPointVisitor` a class that is more likely to contain the defect. And indeed, it turns out that AspectJ bug #30168 was eventually fixed in `ThisJoinPointVisitor`. Thus, while a difference in coverage may not point to a defect, a difference in call sequences may well.

Comparing two runs usually yields more than one differing sequence. In our case (r_v vs. r_x), we obtain a total of 556 differing sequences of length 5. We can determine the originating class for each of these sequences, assign a weight to each sequence, and

rank the classes such that those with the most important sequences are at the top. In this ranking, the `ThisJoinPointVisitor` class is at position 10 out of 542 executed classes—meaning that the programmer, starting at the top, has to examine only 1.8% of the executed classes or 3.3% of the executed code (0.3% of all classes or 0.8% of the entire code) in order to find the defect. (In comparison, comparing coverage of r_{\checkmark} and r_{\times} yields no difference in the `ThisJoinPointVisitor` class, making coverage difference worse than a random guess.)

Of course, this one experiment cannot be generalized to other defects or other programs. In the remainder of this paper, we first describe in detail how we collect sequences of method calls (Section 3), and how we compare them to detect likely defects (Section 4). In Section 5, we describe our controlled experiment with the NanoXML parser; the results support our initial claims. Section 6 discusses related work and Section 7 closes with conclusion and consequences.

3 Summarizing Call Sequences

Over its lifetime, an object may receive and initiate millions of method calls. How do we capture and summarize these to characterize normal behavior? These are the highlights of our approach:

- Recording a trace of all calls per object quickly becomes unmanageable and is a problem in itself (Reiss and Renieris, 2001). Rather than recording the full trace, we abstract from it by *sliding a window* over the trace and remembering only the observed sequences in a set.
- Collecting a sequence set per object is still problematic, as an application may instantiate huge numbers of objects. We therefore *aggregate the sequence sets* into one set per class, which thus characterizes the behavior of the class.
- An object both receives and initiates method calls; both can be traced, but they tell us different things: the trace of *incoming* (received) calls tells us how an object is *used* by its clients. The trace of *outgoing* (initiated) calls tells us how an object is *implemented*. We consider both for fault localization.
- We keep the *overhead* for collecting and analyzing traces as low as possible. Overall, the overhead is comparable to measuring coverage—and thus affordable even in the field.

The following sections describe these techniques in detail.

3.1 From Traces to Call Sequences

A recording of all calls that an object receives is called a *trace*. To capture an object’s behavior, we like to record the whole trace but the huge number of calls received over the lifetime of an object make the whole trace unmanageable. We are therefore looking for a more abstract representation. One such technique is to remember only *characteristic sequences* of the trace. This abstraction works equally well for a trace of calls initiated by an object, or any other trace, which is why we talk about traces in general.

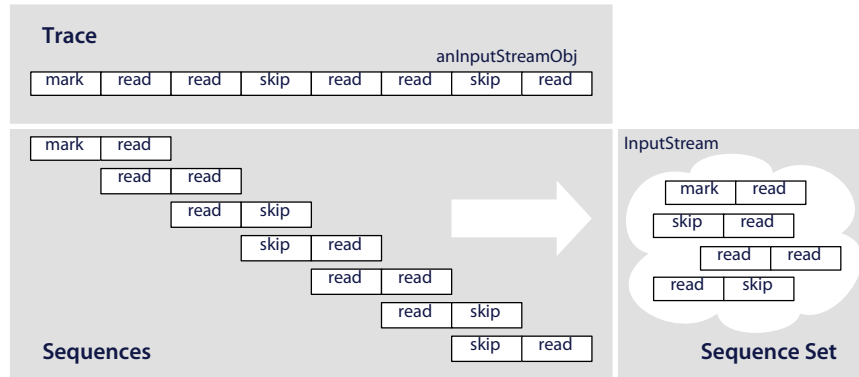


Fig. 2. The trace of an object is abstracted to a sequence set using a sliding window.

When we slide a window over a trace, the contents of the window characterize the trace—as demonstrated in Fig. 2. The observed window contents form a set of short sequences. The wider the window, the more precise the characteristic set will be.

Formally, a trace S is a string of calls: $\langle m_1, \dots, m_n \rangle$. When the window is k calls wide, the set $P(S, k)$ of observed windows are the k -long substrings of S : $P(S, k) = \{w \mid w \text{ is a substring of } S \wedge |w| = k\}$. For example, consider a window of size $k = 2$ slid over S and the resulting set of sequences $P(S, 2)$:

$$S = \langle abcabdc \rangle \quad P(S, 2) = \{ab, bc, ca, cd, dc\}$$

Obviously different traces may lead to the same set: for $T = \langle abcdcdca \rangle$, we have $P(T, 2) = P(S, 2)$. Hence, going from a trace to its sequence set entails a loss of information. The equivalence of traces is controlled by the window size k , which models the context sensitivity of our approach: in the above example a window size $k > 2$ leads to different sets $P(S, k)$ and $P(T, k)$. In the remainder of the paper, we use $P(T)$ to denote the sequence set computed from T , not mentioning the fixed k explicitly.

Note that two calls that are next to each other in a sequence may have been far apart in time: between the two points in time when the object received the calls, other objects may have been active.

The size of a sequence set may grow exponentially in theory: With n distinct methods, n^k different sequences of length k exist. In practice, sequence sets are small, though, because method calls do not happen randomly. They are part of static code with loops that lead to similar sequences of calls. This underlying regularity makes a sequence set a useful and compact abstraction—one could also consider it an *invariant* of program behavior.

3.2 From Objects to Classes

Collecting call traces for each object individually raises an important issue: In a program with millions of objects, we will quickly run out of memory space, just as if we

collected a global trace. As an alternative, one could think about collecting traces at the *class level*. In an implementation of such a trace, an object adds an entry to the trace every time it receives (or initiates) a call. Sliding a window over this trace results in a sequence set that characterizes the class behavior.

As an example of sequence sets aggregated at class level, consider the traces X and Y of two objects. Both objects are live at the same time and their calls interleave into one class-based trace S :

$$\begin{aligned} X &= \langle a \ b \ c \ d \ dc \rangle \\ Y &= \langle a \ a \ b \ c \ ab \ \rangle \\ S &= \langle aaabbcccdabdc \rangle \\ P(S, 2) &= \{aa, ab, bb, bc, cc, cd, da, bd, dc\} \end{aligned}$$

The resulting sequence set $P(S, 2)$ characterizes the behavior of the class—somewhat. The set contains sequences like da or bb that we never observed at the object level. How objects interleave has a strong impact on the class trace S , and consequently on its sequence set. This becomes even more obvious when a class instantiates many objects and when their interleaving becomes non-deterministic, as in the presence of threads.

We therefore use a better alternative: We trace objects *individually*, but rather than aggregating their traces, we aggregate their *sequence sets*. Previously, we collected all calls into one trace and computed its sequence set. Now, we have individual traces, but combine their sequence sets into one set per class. The result $P(X, 2) \cup P(Y, 2)$ is more faithful to the traces we actually observed— bb and da are no longer elements of the sequence set:

$$\begin{aligned} P(X, 2) &= \{ab, bc, cd, dd, dc\} \\ P(Y, 2) &= \{ab, bc, ca, aa\} \\ P(X, 2) \cup P(Y, 2) &= \{aa, ab, bc, cd, dd, dc, ca\} \end{aligned}$$

The sequence set of a class is the union of the sequence sets of its objects. It characterizes the behavior of the class and is our measure when comparing classes in passing and failing runs: we simply compare their sequence sets.

3.3 Incoming vs. Outgoing Calls

Any object receives incoming and initiates outgoing method calls. Their traces tell us how the object is used by its clients and how it is implemented, respectively. Both kinds of traces can be used to detect control flow deviations between a passing and a failing run. However, they differ in their ability to relate deviations to defects.

As an example, consider Fig. 3. A queue object `aQueue` can be implemented as a linked list. The queue object receives calls like `add()` to add an element, and `get()` to obtain the first element and to remove it from the queue. These are *incoming* calls to the queue object.

To implement these methods, the queue object calls `add()` to add an element at the end of `aLinkedList`, `firstElement()` to obtain the first element, and `removeFirst()` to remove it from the list. These calls are *outgoing* calls of the `aQueue` object.

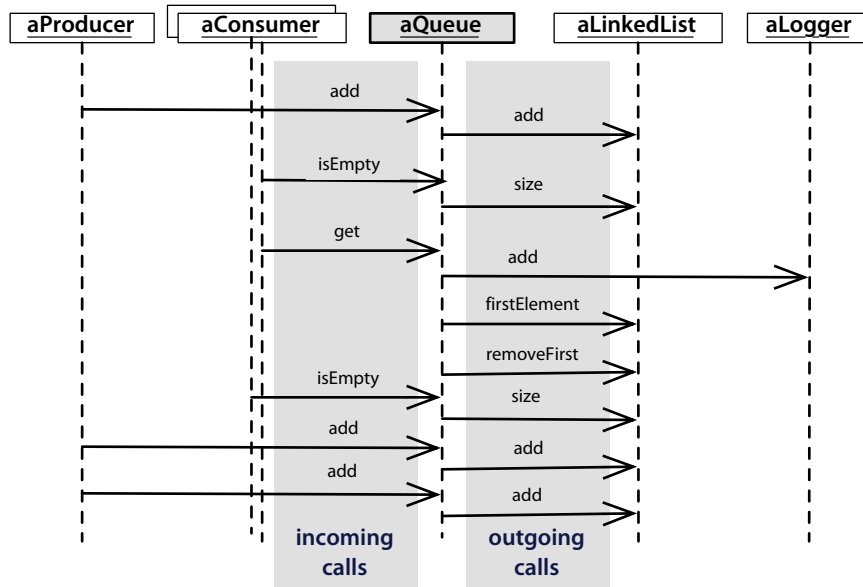


Fig. 3. Traces of incoming calls (left) and outgoing calls (right) for the aQueue object.

Incoming Calls Inspired by the work of Ammons et al. (2002), we first examined *incoming* calls. The technique of Ammons et al. observed clients that called into a part of the X11 API and learned automatically a finite-state automaton that describes how the API is used correctly by a client: for example, a client must call `open()` before it may call `write()`. Such an automaton is an invariant of the API; it can be used to detect non-conforming clients.

By tracing incoming calls, we can also learn this invariant and represent it as a sequence set: each object traces the calls it receives. Since we know the class `Queue` of the receiving object, we have to remember in a sequence only the names of the invoked methods¹. In our example, the trace of incoming calls for the `aQueue` object is

`<add(), isEmpty(), ..., add(), add()>` .

As discussed in Section 3.2, the sequence sets of individual objects are aggregated into one sequence set for the class. After training with one or more passing runs, we can detect when a class receives calls that do not belong to the learned sequence set.

Learning class invariants from incoming calls is appealing for at least two reasons: First, the number of methods an object can receive is restricted by its class. We thus can expect small traces and may even fine-tune the window size in relation to its number of methods. Second, class invariants could be learned across several applications that use the class, not just one. One could even imagine to deploy a class implementation with invariants learned from correct usage.

¹ We also remember their signatures to resolve overloading.

Outgoing Calls In our setting, incoming calls show a major weakness: When we detect a non-conforming usage of a class, it is difficult to identify the responsible client. For example, let us assume we observe a new usage sequence like `(get(), get(), get())`. This sequence could indicate a problem because a consumer should check for an empty queue (`isEmpty()`) before attempting a `get()`. The sequence might also be harmless, for instance, when the `get()` calls stem from three different objects that all previously called `isEmpty()`. In any case, it is not the queue object which is responsible for the new sequence, but its *clients*. Consequently, we turned from incoming to *outgoing* calls: the method calls initiated by an object. For a `Queue`, these are:

```
(LinkedList.add(), LinkedList.size(), Logger.add(), ...)
```

Because an object may call several classes, method names are no longer unique—witness the different calls to `add`. We therefore remember the class *and* method name in a trace. Again, we build one trace per object and aggregate the traces of individual queue objects into one sequence set for the `Queue` class, which represents its behavior.

When we detect a sequence of outgoing calls that is not in a learned sequence set, we know where to look for the reason: the `Queue` class. Unlike a trace of incoming calls, the trace of outgoing calls can guide the programmer to the defect.

3.4 Collecting Traces

We trace a Java program using a combination of off-line and on-line methods. Before the program is executed, we instrument its bytecode for tracing. While it is running, the program collects traces, computes the corresponding sequence sets, and emits them in XML format before it quits; analyzing sequence sets takes place offline.

For program instrumentation, we use the Bytecode Engineering Library (BCEL, Dahm (1999)). This requires just the program's class files and works with any Java virtual machine. We thus can instrument any Java application, regardless of whether its source code is available. While this is not a typical scenario for debugging, it allows us to instrument the SPEC JVM98 benchmark, or indeed any third-party code.

Instrumentation of a class rewrites all call sites and non-static methods. A call is rewritten such that before a call, the caller's class and instance identifications are written to a thread-local variable from where they are read by code added to the prolog of the called method (the callee). Because of dynamic binding, the caller cannot know statically the exact class of the callee, only the callee does. Therefore both must collaborate and only the callee actually adds a new entry to a trace.

Each object builds a trace for its (incoming or outgoing) calls but the trace is not stored within the object. Instead, trace data associated with an object are stored in global hash tables where they are indexed by an object's identity. Since Java's `Object.hashCode()` method is unreliable for object identification, each object creates a unique integer for identification in its constructor. Keeping trace data outside of objects has the advantage that they can be accessed by foreign objects, which is essential for outgoing calls.

For an incoming call, the callee simply adds its name and signature to *its own* trace. But for an outgoing call, the callee must add its name, signature, and class to the trace of the *caller*. To do so, it needs to access the caller's trace using the caller's id.


```

class Caller extends Object {
    ...
    public void m() {
        Callee c;
        ...
        Tracer.storeCaller(this.id);
        c.message(anObject);
        (body of m)
    }
}

class Callee extends Object {
    ...
    public void message(Object x) {
        Tracer.addCall
        (message id for Callee.message);
        (body of message)
    }
}

```

Fig. 4. Instrumentation of caller and callee to capture outgoing calls.

Fig. 4 presents a small example illustrating instrumentation that is done for tracing outgoing calls. (For the sake of readability, we provide Java code instead of byte code.) Statements added during the instrumentation are shown in bold face. Prior to the invocation of `Callee.message` in method `Caller.m`, the id of the caller is stored in the `Tracer`. At the very start of method `Callee.message`, `Tracer.addCall` adds the method id of `Callee.message` to the trace of the calling object—the one which was previously stored in class `Tracer`. Hence, `addCall` only receives the message id—an integer key associated with a method, its class, and signature.

The combined trace of all method calls for all objects quickly reaches Gigabytes in size and cannot be kept in main memory, but writing it to a file would induce a huge runtime overhead. We therefore do not keep the original trace but compute the sequence set for each class online—while tracing. The sequence sets are small (see Section 3.5 for a discussion of the overhead), kept in memory, and emitted when the program quits.

Computing sequence sets and emitting them rather than the original trace has a few disadvantages. To compute sequence sets online, the window size must be fixed (per set) for a program run, where sequence sets for many window sizes could be computed offline from a raw trace. While a trace is ordered, a sequence set is not. We therefore lose some of the trace’s inherent notion of time.

To compute the sequence set of a class online, each object maintains a window for the last k (incoming or outgoing) calls, which is advanced by code in the prolog of the called method. In addition, a sequence set is associated with every traced class. Whenever a method finds a new sequence—a new window of calls—it adds the sequence to the set of the class. Finally, each class emits its sequence set in XML format.

After the program has quit, we use offline tools to read the sequence sets and analyze them. For our experimental setup, we read them into a relational database.

3.5 Overhead

To evaluate the overhead of our tracing method, we instrumented and traced the programs from the SPEC JVM98 benchmark suite (SPEC, 1998). We compared the overhead with JCoverage (Morgan, 2004), a tool for coverage analysis that, like ours, works on Java bytecode, and whose results can point to defects.

The SPEC JVM98 benchmark suite is a collection of Java programs, deployed as 543 class files, with a total size of 1.48 megabytes. Instrumenting them for tracing incoming calls with a window size of 5 on a 3 GHz x86/Linux machine with 1 megabyte of main memory took 14.2 seconds wall-clock time. This amounts to about 100 kilobyte or 38 class files per second. The instrumented class files increased in size by 26%. Instrumentation thus takes an affordable overhead, even in an interactive setting.

Program	Memory			Time			Sequences	
	original MB	JCoverage factor	our approach factor	original seconds	JCoverage factor	our approach factor	count	XML Kb
check	1.4	1.2	1.1	0.14	10.0	1.5	113	3
compress	30.4	1.2	2.2	5.93	1.7	59.8	85	3
jess	12.1	2.1	17.6	2.17	257.1	98.2	1704	37
raytrace	14.2	1.5	22.7	1.93	380.8	541.6	1489	34
db	20.4	1.4	1.2	11.31	1.5	1.2	127	3
javac	29.8	1.5	1.2	5.46	45.7	31.4	15326	334
mpegaudio	12.8	1.6	1.2	5.96	1.2	27.9	587	13
mtrt	18.4	1.4	18.2	2.06	367.9	574.8	1579	36
jack	13.6	1.7	1.7	2.32	40.5	6.3	1261	28
AspectJ	41.8	1.4	1.4	2.37	3.3	3.0	13920	301

Table 1. Overhead measured for heap size and time while tracing incoming calls (with window size 5) for the SPEC JVM 98 benchmark. The overhead of our approach (and JCoverage in comparison) is expressed as a factor relative to the original program. The rightmost columns show the number of sequences and the size of their gzip-compressed XML representation.

Running an instrumented program takes longer and requires more memory than the original program. Table 1 summarizes the overhead in *factors* of the instrumented program relative to the memory consumption and run time of the original program.

The two ray tracers `raytrace` and `mtrt` demonstrate some challenges: tracing them required 380 MB of main memory because they instantiate ten thousands of objects of class `Point`, each of which was traced. This exhausted the main memory, which led to paging and to long run times.

The overheads for memory consumption and runtime varied by two orders of magnitude. At first sight, this may seem prohibitive—even when the overhead was comparable or lower than for JCoverage. We attribute the high overhead in part to the nature of the SPEC JVM 98, which is intended to evaluate Java virtual machines—most programs in the suite are CPU bound and tracing affects them more than, say, I/O-intensive programs.

The database `db` and the `mpegaudio` decoder benchmarks, for instance, show a small overhead. When we traced the AspectJ compiler for the example in Section 1 (with window size 8), we also observed a modest overhead and consider these more typical for our approach.

4 Relating Call Anomalies to Failures

As described in Section 3, we collect call sequences per object and aggregate them per class. At the end of a program run, we thus have a sequence set for each class. These sequence sets now must be compared across multiple runs—or, more precisely, across passing and failing runs. Our basic claim is that differences in the test outcome correlate with differences in the sequence sets, thus pointing to likely defective classes. Therefore, we must *rank* the classes such that classes whose behavior differs the most between passing and failing runs get the highest priority.

4.1 One passing and one failing run

Let us start with a simple situation: We have one failing run r_0 and one passing run r_1 . For each class, we compare the sequence sets as obtained in r_0 and r_1 . Typically, both sets will have several common sequences. In addition, a deviating class may have additional (or “new”) as well as missing sequences. We consider new and missing sequences as symmetrical: a failure may be caused by either.

We are only interested in sequences that are “new” or “missing”. We thus assign each sequence $p \in P(r_0) \cup P(r_1)$ a weight $w(p)$ of

$$w(p) = \begin{cases} 1 & \text{if } p \notin P(r_0) \cap P(r_1)\text{—that is, } p \text{ is “new” or “missing”} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

From the weights of all sequences, we compute the average sequence weight of a class. The calculation uses the sequences from the passing run and the failing run: it is the sum of all weights $w(p)$, divided by the total number t of sequences:

$$\text{average weight} = \frac{1}{t} \sum_{p \in P(r_0) \cup P(r_1)} w(p) \quad \text{where } t = |P(r_0) \cup P(r_1)|. \quad (2)$$

Our claim is that those classes with the highest average weight are most likely to contain a defect. Therefore, the programmer should start searching for the defect in the code of these classes.

4.2 Multiple passing runs

For a programmer, it is often easier to produce passing runs than failing runs. We take advantage of this situation by considering *multiple passing runs* when searching for a defect in a single failing run. We thus have n passing runs and one failing run and therefore must extend the weight calculation to this more general case.

The general idea is as follows: A sequence absent in the failing run is more important when it is present in many passing runs. Conversely, it is less important when the sequence is rare among passing runs. A sequence only present in the failing run is always important.

In Fig. 5, we have illustrated these relationships between the *sequence sets from different runs* (again, all coming from the same class). On the left side, we see two sets for passing runs, and one for a failing run. The set of sequences in the failing run can intersect with the $n = 2$ sets of the passing runs in various ways.

With multiple passing runs, “new” and “missing” sequences are no longer equally important. A sequence that is missing in the failing run does not have to be present in all passing runs. We account for this by assigning each missing sequence a weight according to the number of passing runs where it was found. Again, a new sequence in the failing run is absent in all passing runs and always has the same weight of 1.

The weight of a sequence is assigned according to the scheme on the left side of Fig. 5: a new sequence, occurring in the failing run only, is assigned a weight of 1.

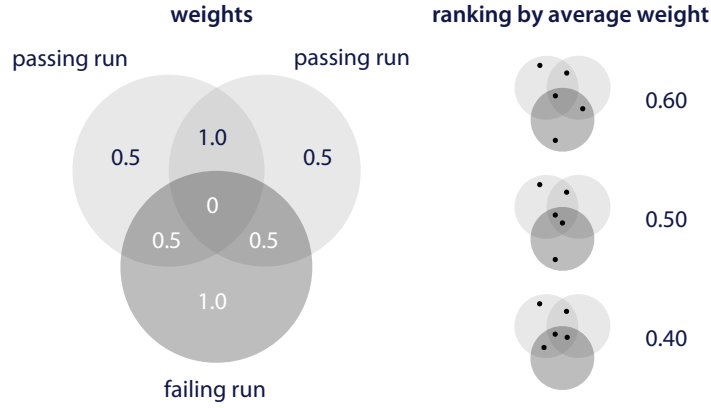


Fig. 5. The weight of a sequence depends on the number of passing and failing runs where it was found; it determines the class ranking.

A sequence missing from the failing run, but present in *all* passing runs, is assigned a weight of 1, as well.

All other missing sequences are assigned a weight between 0 and 1. Intuitively, the weight is the larger the more “new” or “missing” a sequence is. Formally, for one failing run r_0 and n passing runs r_1, \dots, r_n , let $k(p) = |\{r_i \mid p \in P(r_i) \wedge i > 0\}|$ denote the number of passing runs that include a sequence p in their sequence set. Then the weight $w(p)$ is:

$$w(p) = \begin{cases} \frac{k(p)}{n} & \text{if } p \notin P(r_0) \\ 1 - \frac{k(p)}{n} & \text{if } p \in P(r_0) \end{cases} \quad (3)$$

Note that if $n = 1$ holds, we obtain $k(p) = 1$ if $p \in P(r_1)$, and $k(p) = 0$, otherwise; therefore, (3) is equivalent to (1).

Again, from the weights of all sequences, we compute the average sequence weight of a class. The calculation uses the sequences from *all* passing runs r_1, \dots, r_n , plus the failing run r_0 :

$$\text{average weight} = \frac{1}{t} \sum_{p \in \cup_{i=1}^n P(r_i)} w(p) \quad \text{where } t = \left| \bigcup_{i=0}^n P(r_i) \right|. \quad (4)$$

Note that if $n = 1$ holds, (2) is a special case of (4).

Again, we claim that those classes with the highest average weight are most likely to contain a defect (and therefore should be examined by the programmer first). To validate this claim, we have conducted a controlled experiment.

Version	Classes	LOC	Faults	Tests			Drivers
				all	failing		
1	16	4334	7	214	160	79	
2	19	5806	7	214	57	74	
3	21	7185	10	216	63	76	
5	23	7646	9	216	174	76	
total		24971	33		474		

Table 2. Characteristics of NanoXML, the subject of our controlled experiment.

5 A Case Study

As described in Section 4, we rank classes based on their average sequence weight and claim that a large weight indicates a defect. To evaluate our rankings, we studied them in a controlled experiment, with the NanoXML parser as our main subject. As a complementary large subject, we applied our techniques to the AspectJ compiler (Kiczales et al., 2001). Our experiments evaluate class rankings along three main axes: incoming versus outgoing calls, various window sizes, and class-based versus object-based traces.

5.1 Object of Study

NanoXML is a non-validating XML parser implemented in Java, for which Do et al. (2004) provide an extensive test suite. NanoXML comes in five development versions², each comprising between 16 and 23 classes, and a total number of 33 known faults (Table 2). These faults were discovered during the development process, or seeded by Do and others. Each fault can be activated individually, such that there are 33 variants of NanoXML with a single fault.

Faults and test cases are related by a fault matrix: for any given fault and test case, the matrix tells whether the test case uncovers the fault. Each test uses exactly one test driver but several tests may share the same driver. A test driver provides general infrastructure, like reading an XML file and emitting its last element.

5.2 Experimental Setup

Our experiment simulates the following situation: for a fixed program, a programmer has one or more passing test cases, and one failing test case. Based on traces of the passing and failing runs, our techniques ranks the classes of the program. The ranking aims to place the faulty class as high as possible.

In our experiment, we know the class that contains the defect (our techniques, of course, do not); therefore, we can assess the ranking. We express the quality of a ranking as the *search length*—the number of classes above the faulty class in the ranking. The best possible ranking places the faulty class at the top (with a search length of zero).

² We could not use Version 4 because it does not come with a fault description file.

To rank classes, we needed at least one passing run for every failing run. However, we wanted to avoid comparing totally unrelated program runs. For each ranking therefore selected a set of program runs from the suite of programs that met the following conditions:

- All program runs were done with the same version, which had one known bug, and used the same test driver.
- One “failing” run in the set showed the bug, the other “passing” runs did not.

Altogether, we had 386 such sets (Table 2). The test suite contains 88 more failing runs for which we could not find any passing run. This can happen, for example, when a fault always causes a program to crash such that no passing run can be established.

For each of the failing runs with one or more related passing runs, we traced their classes, computed their sequence sets, and ranked the classes according to their average sequence weight. The rankings were repeated in several configurations:

- Rankings based on class and object traces (recall Section 3.2)
- Rankings based on incoming and outgoing calls (recall Section 3.3).
- Rankings based on six window sizes: 1, 2, 3, 4, 5, and 8.

We compared the results of all configurations to find the one that minimizes the search length, and thus provides the best recommendations for defect localization.

5.3 Threats to Validity

Our experiments are not exhaustive—many more variations of the experiment are possible. These variations include other ways to weight sequences, or to trace with class-specific window sizes rather than a universal size. Likewise, we did not evaluate programs with multiple known defects.

NanoXML provides many test cases, but itself is a small program. We therefore can't argue for the scalability of our approach. We do have some evidence, though, that our approach does scale: bytecode instrumentation of the SPEC JVM 98 benchmark suite did not pose a problem, and instrumenting and tracing AspectJ wasn't a problem, either. Our belief is further supported by the fact that our technique ranks the faulty class from AspectJ in 10th place, among 2,929 classes in total.

The search lengths reported in our results are abstract numbers that don't make our potential mistakes obvious. We validated our methods when possible by exploiting known invariants; for example:

- To validate our bytecode instrumentation, we generated Java programs with statically known call graphs and, hence, known sequence sets. We verified that these were indeed produced by our instrumentation.
- When tracing with a window size of one, the resulting sequences for a class are identical for object- and class-based traces: any method called (or initiated) on the object level is recorded in a class-level trace, and vice versa. Hence, the rankings are the same; object- and class-based traces show no difference in search length.

Incoming Calls								
window size	1	2	3	4	5	8	random guess	
							executed	all
object	3.69	3.35	3.38	3.52	3.32	3.19	4.78	9.22
class	3.69	3.35	3.39	3.52	3.31	3.19	4.78	9.22

Outgoing Calls								
object	2.63	2.75	2.55	2.62	2.53	2.22	4.78	9.22
class	2.63	2.81	2.56	2.72	2.55	2.27	4.78	9.22

Table 3. Evaluation of class rankings. A number indicates the average number of classes in atop of the faulty class in a ranking. The two rightmost columns indicate these numbers for a random ranking when (1) considering only executed classes, (2) all classes.

5.4 Discussion of Results

Table 3 summarizes the average search lengths of our rankings for NanoXML, based on different configurations: incoming versus outgoing calls, various window sizes, and rankings based on object- and class-based traces. The search length is the number of classes in atop of the faulty class in a ranking.

For a ranking to be useful, it must be at least better than a random ranking. Each search length in Table 3 is an average over 386 program runs (or rankings). On average, each run utilizes 19.45 classes from which 10.56 are actually executed (excluding the test driver). Random placing of the faulty class would result in an average search length of 9.22 classes, and 4.78, respectively.

All rankings in our experiment are noticeably better than random rankings. They are better even if a programmer had had the additional knowledge of which classes were never executed.

Comparing sequences of passing and failing runs is effective in locating defects.

Sequences vs. Coverage Previous work by Jones et al. (2002) has used coverage analysis to rank source code statements: statements more often executed in failing runs than in passing runs rank higher. Since we are ranking classes, the two approaches are not directly comparable. But ranking classes based on incoming calls with a window size of 1 is analogous to method coverage: the sequence set of a class holds exactly those methods of the class that were called, hence executed. The corresponding search length of 3.69 is the largest in our table, therefore strongly supporting our claim (1): sequences predict defects better than simply comparing coverage.

Comparing sequences of length 2 or greater always performs better than comparing coverage (i.e. sequences of length 1).

Classes vs. Objects Tracing on the object level (rather on the simpler class level) offered no advantage for incoming calls, and only a slight advantage for outgoing calls. We attribute this to the few objects NanoXML instantiates per class and the absence

of threads. Both would lead to increased non-deterministic interleaving of calls on the class level, which in turn would lead to artificial differences between runs.

To validate our hypothesis, we traced AspectJ, which was presented in the introduction, both on the class and the object level. While the class where the developers fixed bug #30168 showed up at position 10 out of 541 executed classes with object-based traces, it showed up at position 16 with class-based tracing. This supports our claim (2): per-object sequences are better defect predictors than global sequences.

Object-based traces are at least slightly better defect locators than class-based traces. For multi-threaded programs, object-based traces should yield a greater advantage.

Window Size With increasing window sizes, search lengths decrease for all traces. The decrease is not always strict, but a strong trend for both incoming and outgoing calls. Moving from a window size of 1 to a window size of 8 reduces the average search length by about 0.5 classes for incoming calls, and by about 0.4 classes for outgoing calls. This corresponds to an improvement of 15 percent. This supports our claim that longer call sequences capture essential control flow of a program.

Given this result, we also experimented with larger window sizes, but moving beyond a window size of 8 did not pay off. A wide window misses any trace shorter than its size and this became noticeable with window sizes greater than 8.

Medium-sized windows, collecting 3 to 8 calls, provide the best predictive power.

Outgoing vs. Incoming Calls Outgoing calls predict faults better than incoming calls. The search length for rankings based on outgoing calls are smaller than those based on incoming calls. Even the worst result for outgoing calls (2.75 for window size of 2) beats the best result for incoming calls (3.19 for window size of 8). This strongly supports our claim (3): the caller is more likely to be defective than the callee.

The inferiority of incoming calls is not entirely surprising: traces for incoming calls show how an object (or a class) is used. A deviation in the failing run from the passing runs indicates that a class is used differently. But the class is not responsible for its usage—its clients are. Therefore, different usage does not correlate with faults.

This is different for outgoing calls, which show how an object (or a class) is implemented. For any deviation here the class at hand is responsible and thus more likely to contain a fault.

Outgoing calls locate defects much better than incoming calls.

Benefits to the Programmer Tracing outgoing calls with a window size of 8, the average search length for a ranking was 2.22. On the average, a programmer must thus inspect 2.22 classes before finding the faulty class—that is, 21.0% of 10.56 executed classes, or 10% of all 23 classes.

Fig. 6 shows a cumulative plot of the search length distribution. Using a window of size 8, the defective class is immediately identified in 36% of all test runs (zero search length). In 47% of all test runs, the programmer needs to examine at most one false positive (search length = 1) before identifying the defect.

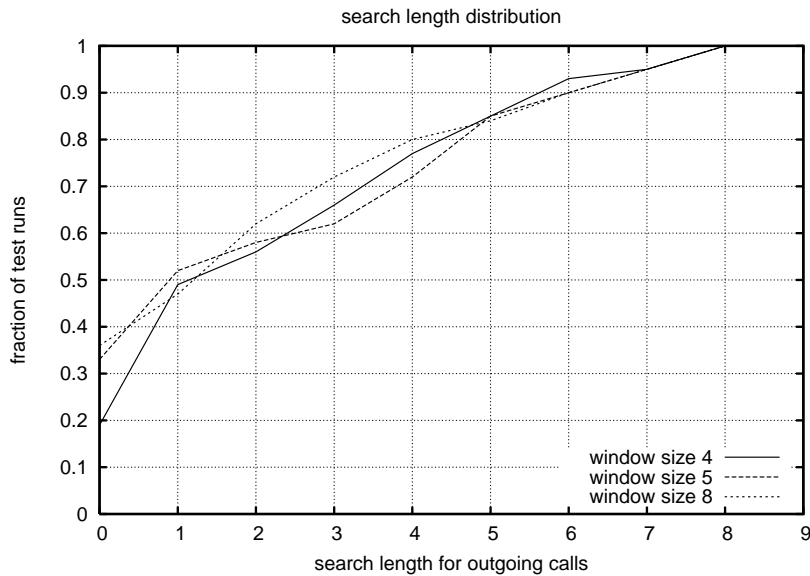


Fig. 6. Distribution of search length for outgoing calls in NanoXML. Using a window size of 8, the defective class is pinpointed (search length 0) in 36% of all test runs.

Because NanoXML is relatively small, each class comprises a sizeable amount of the total application. As could be seen in the example of AspectJ, large applications may exhibit vastly better ratios. We also expect larger applications to show a greater separation of concerns, such that the number of classes which contribute to a failure does not grow with the total number of classes. We therefore believe that the results of our controlled experiment are on the conservative side.

In NanoXML, the defective class is immediately identified in 36% of all test runs. On average, a programmer using our technique must inspect 21% of the executed classes (10% of all classes) before finding the defect.

6 Related Work

We are by no means the first researchers who compare multiple runs, or analyze function call sequences. The related work can be grouped into the following categories:

Comparing multiple runs. The hypothesis that a fault correlates with differences in program traces, relative to the trace of a correct program, was first stated by Reps et al. (1997) and later confirmed by Harrold et al. (1998). The work of Jones et al. (2002) explicitly compares coverage and thus is the work closest to ours. Jones et al. try to locate an error in a program based on the statement coverage produced by several passing and one failing run. A statement is considered more likely to be erroneous the more often it is executed in a failing run rather than in a passing run. In their evaluation, Jones et al. find that in programs with one fault the one faulty statement within a program is almost certainly marked as “likely faulty”, but so is

also 5% to 15% of correct code. For programs with multiple faults, this degrades to 5% to 20% with higher variation. Like ours, this approach is lightweight, fully automatic and broadly applicable—but as demonstrated in the evaluation, sequences have a significantly better predictive power.

Intrusion detection. Our idea of investigating sequences rather than simply coverage was inspired by Forrest et al. (1997) and Hofmeyr et al. (1998)'s work on *intrusion detection*. They traced the system calls of server applications like `sendmail`, `ftpd`, or `lpd` and used the sliding-window approach to abstract them as sequence sets (n -tuples of system calls, where $n = 6, \dots, 10$). In a training phase, they learned the set from normal behavior of the server application; after that, an unrecognized sequence indicated a possible intrusion. As a variation, they also learned sequence that did not match the normal behavior and flagged an intrusion if that sequence was later matched by an application. Intrusion detection is considerably more difficult than defect localization because it has to *predict* anomalous behavior, where we *know* that a program run is anomalous after it failed a test. We found the simplicity of the idea, implementation, and the modest run-time cost appealing. In contrast to their work, though, our approach specifically exploits object orientation and is the first to analyze sequences for defect localization.

Learning automata. Sekar et al. (2001) note a serious issue in Forrest et al. (1997)'s approach: to keep traces tractable, the window size n must be small. But small windows fail to capture relations between calls in a sequence that are n or more calls apart. To overcome this, the authors propose to *learn finite-state automata* from system call sequences instead and provide an algorithm. The interesting part is that Sekar et al. learn automata from traces where they annotate each call with the caller; thus calls by two different callers now become distinguishable. Using these more context-rich traces, their automata produced about 10 times fewer false positives than the n -gram approach. Learning automata from object-specific sequences is an interesting idea for future work.

Learning APIs. While we are trying to locate defects relative to a failing run, Ammons et al. (2002) try to locate defects relative to *API invariants* learned from correct runs: they observe how an API is used by its clients and learn a finite-state automaton that describes the client's behavior. If in the future a client violates this behavior, it is flagged with an error. A client is only required during the learning phase and the learned invariants can later be used to validate clients that did not even exist during the learning phase. However, as Ammons et al. point out, learning API invariants requires a lot of effort—in particular because context-sensitive information such as resource handles have to be identified and matched manually. With object-specific sequences, as in our approach, such a context comes naturally and should yield better automata with less effort.

Data anomalies. Rather than focusing on diverging control flow, one may also focus on differing data. *Dynamic invariants*, pioneered by Ernst et al. (2001), is a predicate for a variable's value that has held for all program runs during a training phase. If the predicate is later violated by a value in another program run this may signal an error. Learning dynamic invariants takes a huge machine-learning apparatus and is far from lightweight both in time and space. While Pytlik et al. (2003) have not been able to detect failure-related anomalies using dynamic invariants, a related lightweight technique by Hangal and Lam (2002) found defects in four Java

applications. In general, techniques that detect anomalies in data can complement techniques that detect anomalies in control flow and vice versa.

Statistical sampling. In order to make defect localization affordable for production code in the field, Liblit et al. (2003) suggest statistical sampling: Rather than collecting all data of all runs, they focus on exceptional behavior—as indicated by exceptions being raised or unusual values being returned—but only for a *sampled set*. If such events frequently occur together with failures (i.e. for a large set of users and runs), one eventually obtains a set of anomalies that statistically correlate with the failure. Our approach requires just two instrumented runs to localize defects, but can be easily extended to collect samples in the field.

Isolating failure causes. To localize defects, one of the most effective approaches is isolating *cause transitions*, as described by Cleve and Zeller (2005). Again, the basic idea is to compare passing and failing runs, but in addition, the delta debugging technique generates and tests *additional runs* to isolate failure-causing variables in the program state (Zeller, 2002). A cause transition occurs at a statement where one variable ceases to be a cause, and another one begins; these are places where cause-effect chains to the failure originate (and thus likely defects). Due to the systematic generation of additional runs, this technique is precise, but also demanding—in particular, one needs an automated test and a means to extract and compare program states. In contrast, collecting call sequences is far easier to apply and deploy.

7 Conclusion and Consequences

Sequences of method calls locate defective classes with a high probability. Our evaluation also revealed that per-object sequences are better predictors of defects than per-class or global sequences, and that the caller is significantly more likely to be defective than the callee. In contrast to previous approaches detecting anomalies in API usage, our technique exploits object orientation, as it collects method call sequences per object; therefore, the approach is fully generic and need not be adapted to a specific API. These are the results of this paper.

On the practical side, the approach is easily applicable to arbitrary Java programs, as it is based on byte code instrumentation, and as the overhead of collecting sequences is comparable to measuring coverage. No additional infrastructure such as automated tests or debugging information is required; the approach can thus be used for software in the field as well as third-party software.

Besides general issues such as performance or ease of use, our future work will concentrate on the following topics:

Further evaluation. The number of Java programs that can be used for controlled experiments (i.e. with known defects, automated tests that reveal these defects, and changes that fix the defects) is still too limited. As more such programs become available (Do et al., 2004), we want to gather further experience.

Fine-grained anomalies. Right now, we are identifying *classes* as being defect-prone. Since our approach is based on comparing *methods*, though, we could relate differing sequences to sets of methods and thus further increase precision. Another interesting option is to identify anomalies in sequences of basic blocks rather than method calls, thus focusing on individual statements.

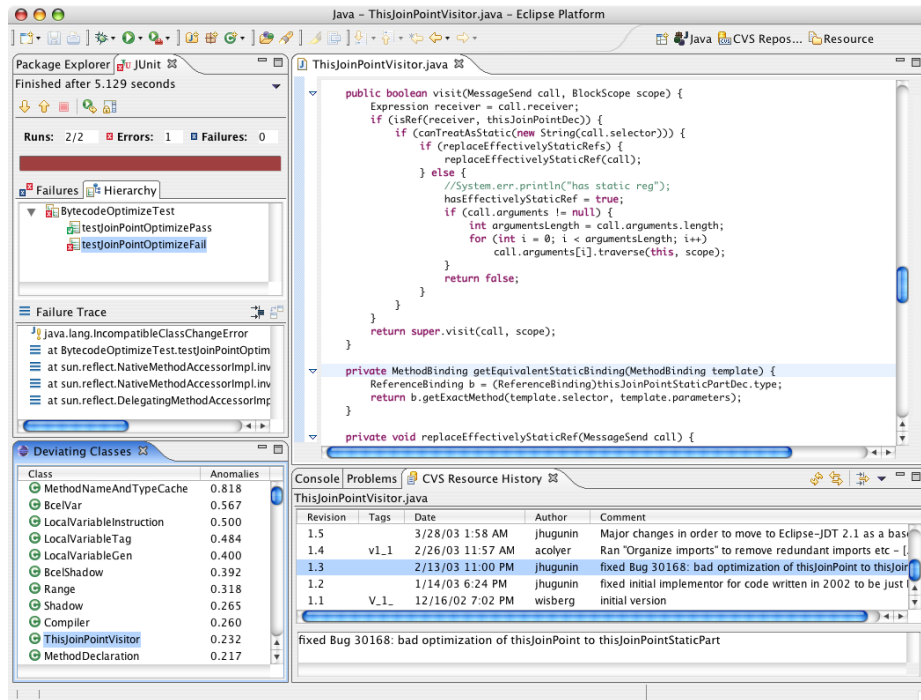


Fig. 7. For the AspectJ bug of Section 2, Eclipse ranks likely defective classes (bottom left)

Sampled calls. Rather than collecting every single method call, our approach could easily be adapted to *sample* only a subset of calls—for instance, only the method calls of a specific class, or only every 100th sequence (Liblit et al., 2003). This would allow to use the technique in production code and thus collect failure-related sequences in the field.

Exploiting object orientation. Our approach is among the first that explicitly exploits object orientation for collecting sequences. Being object-aware might also be beneficial to related fields such as intrusion detection or mining specifications.

Integration with experimental techniques. Anomalies in method calls translate into specific objects and specific moments in time that are more interesting than others. These objects and moments in time could be good initial candidates for identifying failure-inducing program state (Zeller, 2002).

An Eclipse plugin. Last but not least, we are currently turning the prototype into a publicly available Eclipse plugin: As soon as a JUnit test fails, a list shows the most likely defective classes at the top—as in the AspectJ example (Fig. 7).

For future and related work regarding defect localization, see

<http://www.st.cs.uni-sb.de/dd/>

Acknowledgments. Gregg Rothermel made the NanoXML test suite accessible. Tom Zimmermann provided precious insights into the AspectJ history. Holger Cleve and Stephan Neuhaus provided valuable comments on earlier revisions of this paper.

Bibliography

- Glenn Ammons, Rastislav Bodík, and Jim Larus. Mining specifications. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 16–18, 2002.
- Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proc. 27th International Conference of Software Engineering (ICSE 2005)*, St. Louis, USA, 2005. to appear.
- Markus Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, Berlin, Germany, July 07 1999. URL <http://www.inf.fu-berlin.de/~dahm/JavaClass/ftp/report.ps.gz>.
- Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *International Symposium on Empirical Software Engineering*, pages 60–70, Redondo Beach, California, August 2004.
- Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.
- Stephane Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997. ISSN 0001-0782.
- Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 291–301, New York, May 19–25 2002. ACM Press.
- Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, *ACM SIGPLAN Notices*, pages 83–90, Montreal, Canada, July 1998. Published as ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98), *ACM SIGPLAN Notices*, volume 33, number 7.
- Steven A. Hofmeyr, Stephanie Forrest, and Somayaji Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
- ICSE 2002. *Proc. International Conference on Software Engineering (ICSE)*, Orlando, Florida, May 2002.
- James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE 2002*, pages 467–477.
- Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jorgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.
- Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proc. of the SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, June 2003.
- Peter Morgan. JCoverage 1.0.5 GPL, 2004. URL <http://www.jcoverage.com/>.
- Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven Reiss. Automated fault localization using potential invariants. In Michiel Ronsse, editor, *Proc. Fifth Int. Workshop on Automated and Algorithmic Debugging (AADEBUB)*, Ghent, Belgium, September 2003. URL <http://xxx.lanl.gov/html/cs.SE/0309027>.
- Stevn P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE-01)*, pages 221–232, Los Alamitos, California, May 12–19 2001. IEEE Computer Society.

- Thomas Reps, Thomas Ball, Manuvir Das, and Jim Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 432–449. Lecture Notes in Computer Science Nr. 1013, Springer-Verlag, September 1997.
- R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In Francis M. Titsworth, editor, *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P-01)*, pages 144–155, Los Alamitos, CA, May 14–16 2001. IEEE Computer Society.
- SPEC. SPEC JVM 98 benchmark suite. Standard Performance Evaluation Corporation, 1998.
- Andreas Zeller. Isolating cause-effect chains from computer programs. In William G. Griswold, editor, *Proceedings of the Tenth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-02)*, volume 27, 6 of *Software Engineering Notes*, pages 1–10, New York, November 18–22 2002. ACM Press.