Evaluating a Lightweight Defect Localization Tool

Valentin Dallmeier Christian Lindig Andreas Zeller
Saarland University
Department of Computer Science
Saarbrücken, Germany
{dallmeier,lindig,zeller}@cs.uni-sb.de

ABSTRACT

AMPLE locates likely failure-causing classes by *comparing method call sequences* of passing and failing runs. A difference in method call sequences, such as multiple deallocation of the same resource, is likely to point to the erroneous class. In this paper, we describe the implementation of AMPLE as well as its evaluation.

1. INTRODUCTION

One of the most lightweight methods to locate a failure-causing defect 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*. For instance, a failure may occur because some API is used in a specific way, which is not found in passing runs.

To detect such failure-correlated call sequences, we have developed AMPLE[1], a plugin for the development environment Eclipse that helps the programmer to locate failure causes in Java programs. AMPLE works by comparing the method call sequences of passing JUnit test cases with the sequences found in the failing test (Dallmeier et al., 2005). As a result, AMPLE presents a class ranking with those classes at the top that are likely to be responsible for the failure. A programmer looking for the bug thus is advised to inspect classes in the presented order.

Figure 1 presents a programmer's view of AMPLE as a plugin for Eclispe. The programmer is working on the source code for the AspectJ compiler for which a JUnit test case has failed, as was reported in AspectJ bug report #30168. AMPLE instruments the classes of AspectJ on the byte-code level and runs the failing test for observation again, as well a passing test case. As a result, it presents a class ranking

[1]Analyzing Method Patterns to Locate Errors

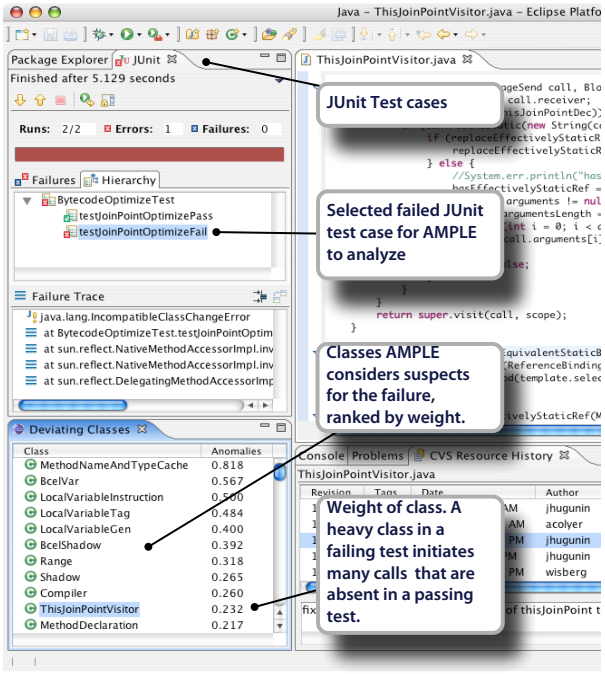

Figure 1: The AMPLE plugin in Eclipse: based on one passing and one failing JUnit test case, AMPLE presents a class ranking in the view *Deviating Classes*. High-ranking classes are suspect because their behavior deviated substantially between passing and failing runs. The AspectJ bug #30168 shown was fixed in the class at position #10, out of 2,929 classes. Our improved ranking algorithm now places the class at position #6.

in view *Deviating Classes*; each class is associated with a weight—its likeliness to be responsible for the failure. The actual bug was fixed in class `ThisJoinPointVisitor`, ranked at position 6, out of 997 classes.

While such anecdotal evidence for the predictive power of AMPLE is nice, we had to evaluate AMPLE in a more systematic way. In this paper, we briefly describe the implementation of AMPLE (Section 2), before discussing its evaluation (Section 3) as well as related work (Section 4). In Section 5, we describe our experiences from the evaluation, and make

suggestions for future similar evaluations.

2. AMPLE IN A NUTSHELL

AMPLE works on a hypothesis first stated by Reps et al. (1997) and later confirmed by Harrold et al. (1998): faults correlate with differences in traces between a correct and a faulty run. A trace is a sequence of actions observed over the lifetime of a program. AMPLE traces the control flow of a class by observing the calls invoked from within its methods. To rank classes, AMPLE compares the method call sequences from multiple passing runs and one failing run. Those classes that call substantially different methods in the failing run than in a passing run are suspect. These are ranked higher than classes that behave similarly in both runs.

AMPLE captures for each object the sequence of methods it calls. To gather such a *trace*, it instruments the program on the byte-code level using BCEL (Dahm, 1999). However, capturing the trace of calls for every object in a program is unfeasable for a number of reasons: the amount of trace data would lead to a high runtime overhead (Reiss and Renieris, 2001). While objects initiate calls they have no source-code representation, only classes do. We therefore rather need a characterization for classes. And finally, the differences between traces need to be qualified, just comparing traces for equality would be too coarse.

AMPLE's solution to these issues are *call-sequence sets*. A call-sequence set contains short sequences of consecutive calls initiated by an object. A call-sequence set is computed from a trace by sliding a window over it: given a string of calls $S = \langle m_1, \ldots, m_n \rangle$ and a window width k, the call-sequence set $P(S,k)$ holds 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 $P(S,2)$:

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

Call-sequence sets have many advantages over traces: (1) they are compact because a trace typically contains the same substring many times; (2) call-sequence sets can be aggregated: we obtain a characterization of a class by aggregating the call-sequence sets of its objects; and (3), call-sequence sets are meaningful to compare, in particular the call-sequence sets from different runs of the same class.

To find classes whose behavior differs between passing and failing runs, AMPLE computes a call-sequence set for each class in the failing and passing runs. Looking at all call sequences observed for a class, it finds some call sequences common to all runs, some that occurred only in passing runs, and others that occurred only in failing runs. Each call sequence is weighted such that sequences that occur in the failing run, but never or seldom in passing runs, are assigned a larger weight than common call sequences.

Given these weights for sequences, a class is characterized by its *average call-sequence weight*. Classes with a high average sequence weight exhibit many call sequences only present in the failing run, and thus are prime suspects. As a result, classes are ranked by decreasing average call-sequence weight, as shown in Figure 1.

Version	*Classes*	*LOC*	*Faults*	*Tests*	*Drivers*
1	16	4334	7	214	79
2	19	5806	7	214	74
3	21	7185	10	216	76
5	23	7646	9	216	76
total		24971	33		

Table 1: Four versions of NanoXML, the subject of our controlled experiment.

Call-sequences sets can be computed directly, without capturing a trace first. The resulting runtime and memory overhead varies widely, depending on the number of objects that a program instantiates. We measured for the SPEC benchmark (SPEC, 1998) a typical runtime-overhead factor of 10 to 20, as well as a memory-overhead factor of two. While this may sound prohibitive, it is comparable to the overhead of a simpler coverage analysis with JCoverage (Morgan, 2004). We also believe that statistical sampling, as proposed by Liblit et al. (2003), can reduce the overhead for programs that instantiate many objects.

3. EVALUATION

For the systematic evaluation of AMPLE, we picked NanoXML as our test subject. NanoXML is a small, non-validating XML parser implemented in Java that Do et al. (2004) have pre-packed as a test subject. It is intended for the evaluation of testing methods and can be obtained[2] by other researchers directly from Do and others, which ensures reproducibility and comparability of our results.

The package provided by Do et al. includes the source code of five development versions, each comprising between 16 and 22 classes (Table 1). Part of the package are 33 defects that may be individually activated in the NanoXML source code. These defects were either found during the development of NanoXML, or seeded by Do and others.

The NanoXML package also contains over 200 test cases. Related test cases are grouped by test drivers, of which there are about 75. Test cases and defects are related by a fault matrix, which is also provided. The fault matrix indicates for any test, which of the defects it uncovers. Because development version 4 lacked a fault matrix, we could only use the other four versions listed in Table 1 for the evaluation of AMPLE.

3.1 Experimental Setup

The main question for our evaluation was: How well does AMPLE locate the defect that caused a given failure? To model this situation, we selected test cases from NanoXML that met all of the following conditions:

- We let AMPLE analyze a version of NanoXML with *one known defect*, which manifests itself in a faulty class.
- As *failing run*, we used a test case that uncovered the known defect.

[2]from `http://csce.unl.edu/~galileo/sir/`

- As *passing runs*, we selected *all* test cases that did not uncover the known defect.
- All test cases for passing and failing runs must belong to the *same driver*. This limits the number of passing runs to those that are semantically related to the failing run.

Altogether, we found 386 such situations, which therefore lead to 386 class rankings. Each ranking included one class with a known defect.

Note that AMPLE also works for test cases whose failure is caused by a combination of bugs or bugs whose fix involves more than one class. However, we did not evaluate these settings.

3.2 Evaluation Results

To evaluate a class ranking, we consider a ranking as advice for the programmer to inspect classes in the presented order until she finds the faulty class. In the experiment, the position of the known faulty class reflects the quality of the ranking: the higher the faulty class is ranked, the better the ranking. More precisely, we took the *search length* as the measure of quality: the number of classes atop of the faulty class in the ranking. This is the number of classes a programmer must inspect *before* she finds the faulty class.

Table 2 shows the average search length values over 386 rankings for various window sizes k. The numbers in row *Object* present the search length computed from sequence sets as discussed in Section 2. For example, with a window size $k = 7$, a programmer would have to inspect 1.98 classes in vain before finding the faulty class.

The numbers in row *Class* stem from an alternative way to compute sequence sets: rather than computing sequence sets per object and joining them into one sequence set per class, one sequence set per class is computed directly. This should be problematic for threaded programs (which NanoXML isn't)—details can be found in Dallmeier et al. (2005).

3.3 Discussion

A comparison with random guessing provides a partial answer for how good the numbers in Table 2 are. On average, a test run of NanoXML utilizes 19.45 classes, from which 10.56 are actually executed. Without any tool support or prior knowledge, a programmer on average would have to inspect half of these before finding the faulty class. This translates into a search length of $(10.56-1)/2 = 4.78$ when considering only executed classes, or 9.22 when considering all classes. All observed search lengths are better than random guessing, even under the assumption that the programmer could exclude non-executed classes (which is unlikely without tool support). Hence, AMPLE's recommendations are useful.

The search length in Table 2 is minimal for window sizes around 4 and 5. We have not analyzed this in detail but find the following plausible: larger window sizes in general provide more useful context than smaller window sizes, which leads to smaller search lengths. But large window sizes require objects to be long-lived in order to fill the window.

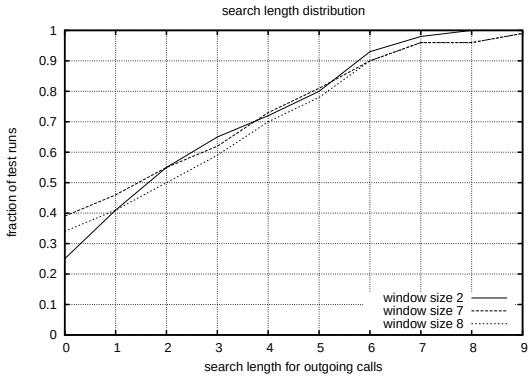

Figure 2: Distribution of search length for NanoXML. Using a window size of 7, the defective class is pinpointed (search length 0) in 38% of all test runs.

With larger window sizes fewer such objects exist, leading to a higher search length. These two opposing forces seem to balance out for window sizes around 4 and 5.

For the average search length to be meaningful, the search length must be distributed normally. Since we could not be sure of this, we examined the actual distribution of the search length. Figure 2 shows the distribution for selected window sizes and confirms the usefulness of AMPLE: with a search length of 2, the faulty class can be found in 50% to 60% of all cases, and in 38% of all cases for $k = 7$, the faulty class is right at the top of the ranking.

Does AMPLE perform better than existing techniques? This question is much harder to answer because the technique closest to ours works on the level of statements, rather than classes: defect localization based on coverage in TARANTULA compares the statement coverage of passing and failing runs (Jones et al., 2002). Statements executed more often in failing runs than in passing runs are more likely to have caused a failure. While this technique also assigns a weight to source code entities, it does it at a much finer granularity, which makes a direct comparison infeasible.

We have argued in Dallmeier et al. (2005) by analogy. A window size of one is equivalent to coverage analysis: the sequence-call set for a window size of $k = 1$ holds just the executed methods. Since the search length for all call-sequence sets with a window size $k \geq 2$ is smaller, the extra context provided by a larger window is obviously useful. This suggests that call-sequence sets outperform pure coverage analysis.

We admit that suggesting entire classes for inspection is quite coarse-grained. Individual methods could be suggested for inspection by taking into account not the class, but the method that invokes a call. This is planned for future work.

3.4 Does it Scale?

While we were satisfied with the results from our systematic evaluation of NanoXML, we expected critics to find NanoXML too small a subject. After all, AMPLE saved us only the inspection of less than three classes on average, compared to random guessing. We therefore recently complemented our evaluation with the AspectJ compiler as another test subject (Dallmeier, 2005).

		Window Size										*Random Guess*	
Subject	*Trace*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*Executed*	*All*
NanoXML	Object	2.53	2.31	2.19	2.17	2.04	2.00	1.98	2.12	2.15	2.14	4.78	9.22
	Class	2.53	2.35	2.22	2.14	2.03	2.04	2.03	2.02	2.22	2.25	4.78	9.22
AspectJ	Object	32.4	31.8	30.8	10.2	8.6	23.4	22.6	23.8	24.4	24.0	209	272
	Class	32.4	32.2	34.8	12.8	12.4	25.2	24.8	25.2	25.2	25.6	209	272

Table 2: Evaluation of class rankings. A number indicates the average *search length*: the number of classes 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.

			Size (LOC)	
Bug ID	*Version*	*Defective Class*	*Class*	*Fix*
29691	1.1b4	org.aspectj.weaver.patterns.ReferencePointcut	294	4
29693	1.1b4	org.aspectj.weaver.bcel.BcelShadow	1901	8
30168	1.1b4	org.aspectj.ajdt.internal.compiler.ast.ThisJoinPointVisitor	225	20
43194	1.1.1	org.aspectj.weaver.patterns.ReferencePointcut	299	4
53981	1.1.1	org.aspectj.ajdt.internal.compiler.ast.Proceed	133	19

Table 3: Bugs in AspectJ used for the evaluation of AMPLE.

AspectJ is a compiler implemented in Java and consists in version 1.1.1 of 979 classes, representing 112,376 lines of code. Unlike NanoXML, it does not come pre-packed with a set of defects, and therefore it was not possible to use it in a systematic evaluation. But its developers have collected bug reports and provide a source code repository which documents how bugs were fixed. From these we could reconstruct passing and failing test cases for the further evaluation of AMPLE.

In order to obtain results comparable with our evaluation using NanoXML, we restricted ourself to bugs whose fixes involved only one Java class; Table 3 shows the 5 bugs that we found, and Table 2 the observed average search lengths for window sizes up to 10.

The average search lengths in Table 2 confirm that AMPLE scales to large programs and works for real-world bugs. Again, rankings for a window size of $k = 1$ perform worse than wider windows. Compared with random guessing, AMPLE saves the programmer the inspection of 177 classes.

4. RELATED WORK

Locating defects that cause a failure is a topic of active research that has proposed methods ranging from simple and approximative to complex and precise.

Comparing multiple runs. The work closest to ours is TARANTULA by Jones et al. (2002). Like us, they compare a passing and a failing run for fault localization, albeit at a finer granularity: TARANTULA computes the statement coverage of C programs over several passing and one failing run. While a direct comparison is difficult, we have argued by analogy in Section 3.3 that sequence sets as a generalization of coverage perform better.

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; a more lightweight technique for Java was proposed by Hangal and Lam (2002).

Isolating failure causes. To localize defects, one of the most effective approaches is isolating *cause transitions* between variables, 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). Due to the systematic generation of additional runs, this technique is precise, but also demanding—in particular, one needs a huge apparatus to extract and compare program states. In contrast, collecting call sequences is far easier to apply and deploy.

5. CONCLUSIONS AND CONSEQUENCES

Like other defect detection tools, AMPLE can only make an *educated guess* about where the defect in question might be located. At a very fundamental level, this is true for any kind of defect detection: If we define the defect as the part of the code that eventually was fixed, no tool can exactly locate a defect, as this would mean predicting future actions of the programmer. With AMPLE, we make that guess explicit—by *ranking* source code according to the assigned probability.

Such a ranking has the advantage of providing a straightforward measure of the tool's precision. The model behind the ranking is that there is an ideal programmer who can spot defects by looking at the code, and thus simply work her way through the list until the "official" defect is found. Thus, the smaller the search length, the better the tool.

To demonstrate the advance beyond the state of the art,

the ranking must obviously be better than a random ranking (which we showed for AMPLE), but also be better than a ranking as established by previously suggested techniques (which we also showed for AMPLE, but using an analogon rather than the real tool). This also requires that the test suites are publicly available, and that the rankings, as obtained by the tools, are fully published. This way, we can establish a number of benchmarks by which we can compare existing tools and techniques.

All this comes with a grain of salt: AMPLE starts with a given failure. Hence, we know that a defect must exist somewhere in the code; the question is where to locate it. Static analysis tools, in contrast, are geared towards the future, and help preventing failures rather than curing them. For a static analysis tool, the central issue is not so much where a defect is located, but whether a code feature should be flagged as defect or not.

Nonetheless, ranking is still a way to go here: Rather than setting a threshold for defects and non-defects, a static analysis tool could simply assign each piece of code a computed probability of being defective. If a caller and a callee do not match, for instance, both could be flagged as potential defects (although only one needs to be fixed). Again, such a probability could end in a ranking of locations, which can be matched against the "official" defects seeded into the test subject, or against a history of "official" fixes: "If the programmer examines the 10% of the ranked locations, she will catch 80% of the defects." Again, these are figures which can be compared for multiple defect detection tools.

While developing AMPLE, we found that such benchmarks help a lot in directing our research—just like test-driven development, we established a culture of "benchmark early, benchmark often". Comparing against other work is fun and gives great confidence when defending the results. We therefore look forward to the emergence of standard benchmarks which will make this young field of defect detection rock-solid and fully respected.

References

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`.

Valentin Dallmeier. Detecting failure-related anomalies in method call sequences. Diploma thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, March 2005.

Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In Andrew Black, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer, July 2005. To appear. Also available from `http://www.st.cs.uni-sb.de/papers/dlz2004/`.

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.

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.

James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proc. International Conference on Software Engineering (ICSE)*, pages 467–477, Orlando, Florida, May 2002.

Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In Jr. James B. Fenwick and Cindy Norris, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI-03)*, volume 38, 5 of *ACM SIGPLAN Notices*, pages 141–154, New York, June 9–11 2003. ACM Press.

Peter Morgan. JCoverage 1.0.5 GPL, 2004. URL `http://www.jcoverage.com/`.

Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 221–232, Los Alamitos, California, May12–19 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.

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.