Lightweight Bug Localization with AMPLE

— Demo paper —

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. Such sequences can be collected from arbitrary Java programs at low cost; comparing object-specific sequences predicts defects better than simply comparing coverage. AMPLE comes as a plug-in for the Java IDE Eclipse that is automatically invoked as soon as a JUnit test fails.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Tracing*

General Terms

Algorithms, Languages

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 regression 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 on the following page presents AMPLE in action. The programmer is working with Eclipse on the source code for the AspectJ compiler; AspectJ is an open-source compiler that implements the aspect paradigm for Java. Regression testing is done within the JUnit framework. Here, the JUnit view shows two test cases, one of which has passed and one that has failed, as reported in AspectJ bug report #30168. By default, AMPLE kicks in automatically as soon as a test fails, comparing the failing run with all passing test runs; however, the programmer can also select related passing tests manually.

To compute its diagnosis, AMPLE instruments these passing and failing test cases on the bytecode level and runs them again. Each run computes for each class the sequences of method calls. Next, AMPLE compares for each class the call sequences in the failing and in the passing run, and computes the class ranking shown in the *Deviating Classes* view. Classes ranked at the top differed substantially between their passing and failing runs. In Figure 1, the defect was fixed in the class at position #10, out of 2,929 classes in AspectJ from which 542 were actually executed—that is, AMPLE has automatically pointed the programmer to the most suspect classes.

The main value of AMPLE is that it easily integrates with existing workflows: After some JUnit test fails, AMPLE can be directly invoked to guide the search for the defect. At the same time, AMPLE is lightweight, meaning that running it requires no more resources that computing coverage; and finally, it brings a higher precision than simply comparing coverage. In this paper, we briefly describe how AMPLE works, and then give an overview of our demonstration.

2. HOW AMPLE WORKS

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 classes by observing what methods their objects call. To rank classes, it compares the method call sequences from multiple passing runs and one failing run. Those classes that call substantially different methods in the failing run

[1]Analyzing Method Patterns to Locate Errors

AADEBUG'05, September 19–21, 2005, Monterey, California, USA.

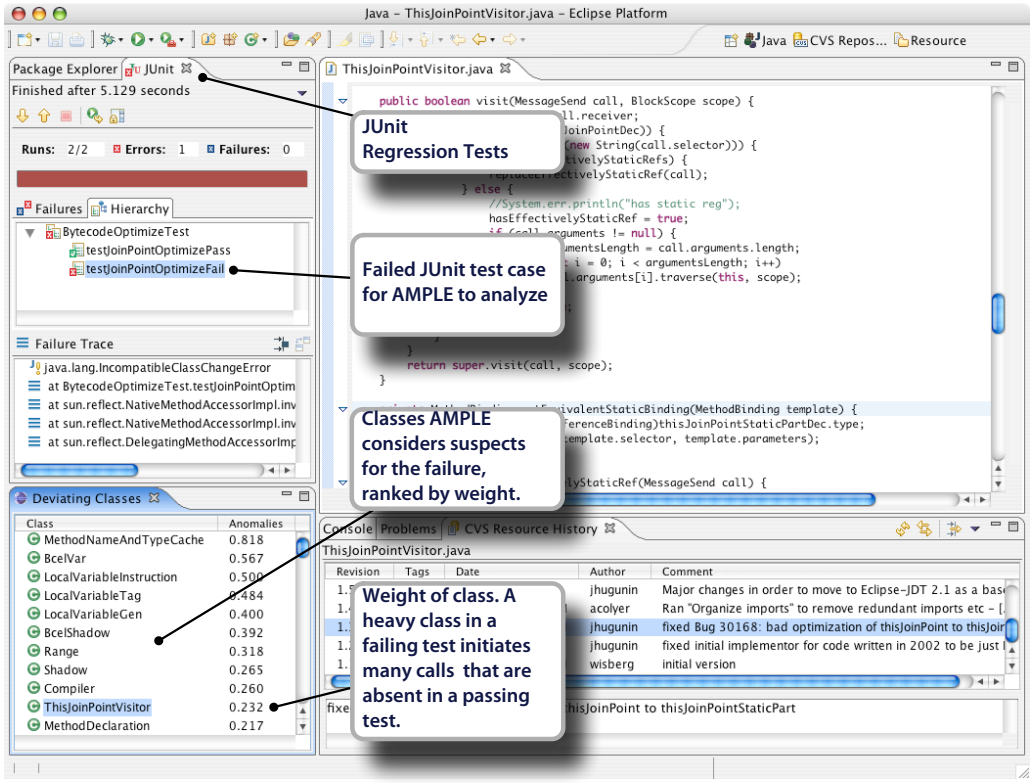

Figure 1: The AMPLE plugin in Eclipse while working on the source code for AspectJ. Based on one passing and one failing JUnit regression test, AMPLE presents a class ranking in the view *Deviating Classes*. High-ranking classes are suspect because their behavior deviated substantially between passing and failing runs. The AspectJ bug #30168 shown was fixed in the class at position #10, out of 2,929 classes.

than in a passing run are suspect. These are ranked higher than classes that behave similarly in both runs.

Comparing the behavior of classes using method calls is not as straight-forward as may seem. In our work, we have met the following challenges:

1. Method calls are issued by objects, while we would like to rank classes. While a class represents the implementation of objects, it is not active at run time. We thus have to abstract from the calls initiated by objects to the behavior of their class.

2. An application may initiate many calls. An uncompressed trace of these is unmanageable and leads to an unacceptable overhead (Reiss and Renieris, 2001). We thus need a more abstract representation of traces.

3. A trace is just a sequence of calls. Two traces become already different when a trace contains one additional call:

$$X = \langle aabcacdeda \rangle$$
$$Y = \langle aabca\ deda \rangle$$

   Trace X and Y are the same, except for one missing call in trace Y. Just calling them different would be too coarse, hence we need a method to *quantify* the difference.

Our solution to these issues are *call-sequence sets*. A call-sequence set contains short sequences of consecutive calls initiated by an object; it is a much more compact representation than a trace. Because of their set nature, sequence sets for objects may be aggregated into one sequence set per class, which then characterizes the class' behavior. Also, because of their set nature, sequence sets are meaningful to compare across several runs of the same class.

2.1 Computing Call-Sequence Sets

Let's suppose for a moment that we already have a method to obtain for each object the trace of methods it invokes. Such a *raw trace* is a long sequence of calls, where each call is characterized by its class, name, and signature (to resolve overloading).

A *call sequence set* is obtained by sliding a window over a raw trace. The contents of the window characterize the trace, as demonstrated in Figure 2: a window of width two is slid over a raw trace of an object that calls `InputStream` (IS) and `OutputStream` (OS) objects.

The observed window contents form a set of characteristic call sequences. Note that the call sequence ⟨`IS.read`, `OS.skip`⟩ appears twice in the raw trace, but only once in the call-sequence set.

Formally, a trace S is a string of calls: $\langle m_1, \ldots, m_n \rangle$. When the window is k calls wide, the set $P(S,k)$ of observed windows are the k-long substrings of S: $P(S,k) = \{w \mid w$ is

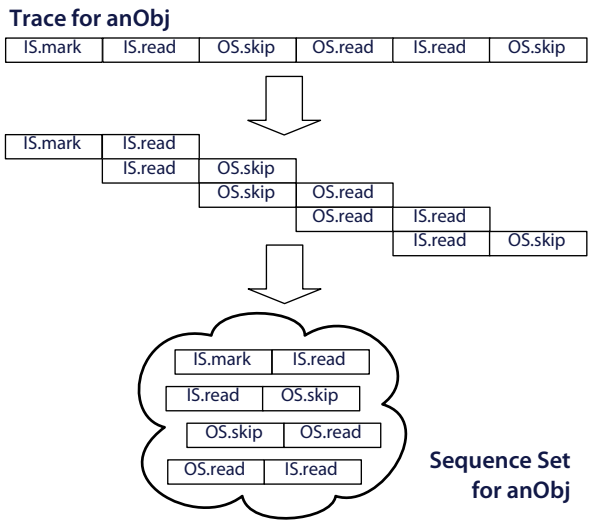

Figure 2: A raw trace of calls is abstracted to a set of characteristic call-sequences using a sliding window of width 2.

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 $P(S, 2)$:

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

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 call trace to its call-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 $P(T, k)$.

AMPLE lets the programmer control the window size k, where typical values range between 5 and 10. The window size is set as a preference and may be changed any time.

The size of a call-sequence set may grow exponentially in theory: With n distinct methods, up to 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 call-sequence set a useful and compact abstraction—one could also consider it an *invariant* of program behavior.

2.2 Aggregating Call-Sequence Sets

Since objects have no source-code representation, only classes do, we like to rank classes rather than objects. We therefore need to abstract from one call-sequence set *per object* to one call-sequence set *per class*. We obtain such a set by aggregating the sequence sets of all *its* objects into one.

Just like going from a raw trace to a sequence set entails some loss of information, so does going from many sequence sets to one. This is demonstrated by the following call traces of two objects:

$$P(\langle abcddc \rangle, 2) = \{ab, bc, cd, dd, dc\}$$
$$P(\langle abcaa \rangle, 2) = \{ab, bc, ca, aa\}$$

The union of the two call-sequence sets is $\{aa, ab, bc, cd, dd, dc, ca\}$, and therefore representation of their class. The calls-sequence set of a hypothetical sequence $U = \langle cabcdd \rangle$ is $P(U, 2) = \{ca, ab, bc, cd, dd\}$. As such it is a subset of the class representation, although it contains call sequences that we never observed over the lifetime of a *single* object. Again, the amount to which this can happen is controlled by the window size k.

2.3 Ranking Classes

After running a passing and a failing JUnit test case, AMPLE has two sequence sets for each class: one from the passing and one from the failing run. We consider sequences observed only in the passing or failing run as suspect, and assign them a higher weight than those that occur in both runs. Missing or extra sequences (relative to the other run) are treated symmetrical because either of them could cause a failure.

Given a class C, let us assume we have a failing run c_0 and a passing run c_1 and their associated sequence sets $P(c_0)$ and $P(c_1)$, all for some fixed window width k. We assign to a sequence p that we observed in both runs a weight $w(p) = 0$, and to a "new" or "missing" sequence a weight of $w(p) = 1$.

$$w(p) = \begin{cases} 1 & \text{if } p \notin P(c_0) \cap P(c_1) \\ 0 & \text{otherwise} \end{cases}$$

The average weight $W(C)$ of the sequences belonging to class C now reflects how much the behavior of C differs between the runs c_0 and c_1:

$$W(C) = \frac{1}{|P_C|} \sum_{p \in P_C} w(p) \quad \text{where } P_C = P(c_0) \cup P(c_1)$$

The AMPLE plugin in Figure 1 shows the average sequence weight next to each class in view *Deviating Classes*. Telling from the average weight of 0.818 in the top-ranking class, almost all sequences in its failing run were either new or missing.

AMPLE can take several passing runs into account, not just one. It lets the user select several passing runs and uses a slightly more general formula to compute the weight of sequences. The details can be found in Dallmeier et al. (2005).

3. EXPERIENCES

We have described our experiences with AMPLE in a separate paper (Dallmeier et al., 2005). Our experiences can be summarized as follows:

1. For bytecode instrumentation, we use the Bytecode Engineering Library (BCEL, Dahm (1999)). This requires just the program's class file and works with any Java virtual machine. The instrumentation rate on a 3 GHz $x86$/Linux machine is about 100 kilobyte of class file per second. On-demand instrumentation, together with small class files, ensures that even large applications are instrumented within seconds.

2. The runtime overhead of tracing varies widely. A computationally intense application—like a ray tracer—that instantiates an extreme number of objects, can be slowed down by a factor of 100 or more. A factor between 10 and 20 is more typical for applications

that also perform some I/O operations. The memory overhead is typically below a factor of two, except for applications that instantiate many (small) objects. While the runtime overhead may sound prohibitive, it is comparable to a simpler coverage analysis with JCoverage (Morgan, 2004).

3. To evaluate our rankings, we studied them in a controlled experiment, with the NanoXML parser as our main subject. In this experiment, we found that

   (a) 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.

   (b) Comparing sequences of passing and failing runs is noticeably better than random rankings. That is, AMPLE is effective in locating defects.

   (c) Comparing sequences of length 2 or greater always performs better than sequences of length 1. That is, comparing call sequences is better than comparing coverage.

4. The window size k controls the sensitivity of AMPLE. Large window sizes lead to many distinctive sequence sets. On the other hand, a large window needs to be filled before it becomes useful. With large values of k, the number of objects increases that perform fewer than k calls over their lifetime. We found window sizes in the range 5 to 10 the most useful and use 5 as a default.

5. Sequences themselves, as well as the number of sequences in sequence sets also increase with k. This however, was of no practical concern. The applications in the SPEC JVM 98 (SPEC, 1998), for example, exhibited several hundred sequences (across all classes, for $k = 5$), and the large AspectJ compiler showed several thousand sequences.

6. AMPLE captures control flow differences in passing and failing runs and may fail to pickup algorithmic errors in a single method. These may become visible only as a differing data flow. But since control flow is induced by data flow, many difference in data become finally visible in method call sequences. AMPLE therefore works best in truly object-oriented programs where each method represents a small and well-defined task, and otherwise relies on method calls.

4. RELATED WORK

Automating debugging is a topic of active research that has proposed methods ranging from simple and approximative to complex and precise. Typically they exclusively analyze either the control, or the data flow of an application. Few approaches have matured into tools.

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. A statements executed more often in a failing run than in passing runs is suspect and colored red in a visualization. TARANTULA is available for the Sparc/Solaris platform.

As the main difference, call sequences take the dynamic execution context into account. A window size of one leads to a sequence set that is equivalent to the class' method coverage. A larger window captures that a call only happened in the context of the other calls captured in the window. The different granularities of TARANTULA and AMPLE make it difficult to compare their accuracy and predictive power. But our experiments (Dallmeier et al., 2005) suggest that the context provided by sequences lead to better results: window sizes of $k > 1$ perform better than those with $k = 1$.

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. Dynamic invariants can be detected with DAIKON (Ernst et al., 2001) for Java, Perl, and C. A related lightweight technique for Java was proposed by Hangal and Lam (2002).

Trace-based Debugging. When a trace for an entire program run is recorded literally, it can be used to re-examine a program run interactively after the fact. This idea is implemented in the Omniscient Debugger by Lewis (2003) for Java. It gives the user the ability to "travel back in time" to find an event that lead to an error. Unlike with AMPLE, this requires manual inspection of events.

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 means to extract and compare program states. In contrast, collecting call sequences is far easier to apply and deploy. Delta debugging and isolation of cause transition are available as a web service[2] for C programs on Linux.

5. OUR DEMO

Our demo of AMPLE will be organized as follows:

1. **Introduction** We give a brief motivation to the topic and discuss the state of the art (Section 4).

2. **How AMPLE works** Using an ongoing example, we show how AMPLE collects and aggregates method call sequences, and how AMPLE ranks the suspect classes. This will give just the general idea; people interested in precise definitions can look at the paper.

3. **Actual demo** We run AMPLE on a real-world program (such as the AspectJ unit test in Figure 1), and show how AMPLE presents the ranking of suspect classes.

4. **Experiences** Generalizing from the demo, we present our experiences, as summarized in Section 3.

[2] `http://www.askigor.org/`

5. Discussion. We close with a general discussion of the approach and a number of open issues to spark discussion with and within the audience.

At the time of the workshop, we plan to make AMPLE freely available for download, such that interested researchers and programmers can try out AMPLE for their own projects. All material related to AMPLE can be found at the project page:

`http://www.st.cs.uni-sb.de/ample/`

Acknowledgements Tom Zimmermann and anonymous reviewers provided valuable comments on earlier revisions of this paper.

References

Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proc. 27th International Conference of Software Engineering (ICSE 2005)*, pages 342–351, St. Louis, Missoury, USA, May 2005. ACM Press.

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, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In Andrew Black, editor, *European Conference on Object-Oriented Programming (ECOOP)*, 2005. To appear. A preliminary version can be found at `http://www.st.cs.uni-sb.de/papers/dlz2004/`.

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.

Bil Lewis. Debugging backwards in time. In M. Ronsse and K. De Bosschere, editors, *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, September 2003.

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.