Generating Fixes from Object Behavior Anomalies

Valentin Dallmeier · Andreas Zeller
Dept. of Computer Science
Saarland University, Germany
{dallmeier, zeller}@cs.uni-saarland.de

Bertrand Meyer
Chair of Software Engineering
ETH Zürich Switzerland
bertrand.meyer@inf.ethz.ch

***Abstract*—Advances in recent years have made it possible in some cases to *locate* bugs automatically. But debugging is also about *correcting* bugs. Can tools do this automatically? The results reported in this paper, from the new PACHIKA tool, suggest that such a goal may be reachable.**

PACHIKA leverages *differences in program behavior* to generate program fixes directly. It automatically infers object behavior models from executions, determines differences between passing and failing runs, generates possible fixes, and assesses them via the regression test suite. Evaluated on the ASPECTJ bug history, PACHIKA generates a valid fix for 3 out of 18 crashing bugs; every fix pinpoints the bug location and passes the ASPECTJ test suite.

***Keywords*-debugging; fixing; object usage patterns**

I. Introduction

When a program fails, debugging starts—the process of locating and fixing the bug that causes the failure. Recent years have seen considerable advances in automated debugging. Even with automated bug localization, the programmer must still assess these locations to choose where and how to fix the program. The goal of this work is to automate this final step, effectively automating the entire debugging process for a significiant subset of programming errors.

The following example illustrates the approach. The APACHE MINA project provides a framework for building network applications. The project's bug database contains an entry for bug 293, complaining that test `VmPipeBindTest` crashes with an assertion error. To debug the failure, we first want to know how the failing run differs from passing runs; we are searching for *anomalies* that correlate with the failure. In earlier work [3], we have shown how to extract *object behavior models* from executions—models that characterize behavior as finite state machines over object states and method calls. Figure 1 shows such a model for the MINA `BaseIoAcceptor` class; the solid transitions occur in the passing runs. We can see individual states of the object, characterized by the properties of its attributes. In the passing run, clients call `setLocalAddress()`, then `setHandler()` to set up the attributes; a sequence of alternating `bind()` and `unbind()` calls then alters the state.

The failing run follows different transitions, shown by dashed lines in the figure. Besides a different method call order when setting up the object, the client now calls

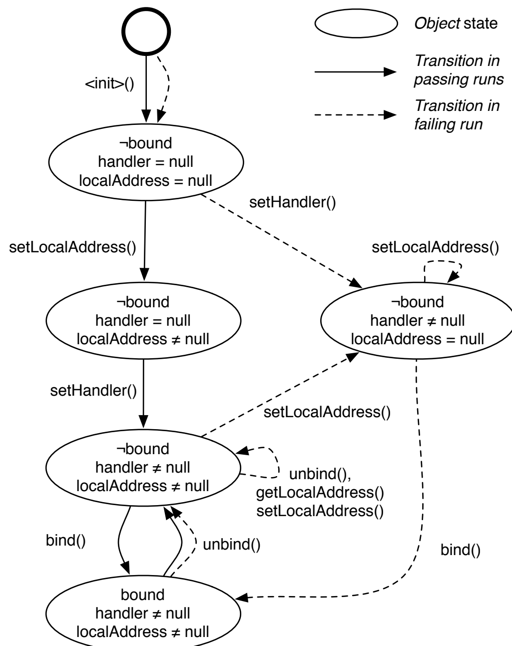

Figure 1. A combined model of passing and failing runs for the MINA `BaseIoAcceptor` class.

`unbind()` multiple times in a row—even when the `bound` attribute is already true. This behavior occurs only in the failing run. But is it also the cause of the failure? To investigate this, we *systematically generate patches* that alter the failing run to match the behavior from the passing runs. If a patch fixes the failure and does not break the regression test suite, we consider it valid.

In the example, there are several ways to change the behavior from failing to passing: we can (a) make the call to `unbind()` conditional such that it only occurs when `bound` is true (as in the passing run), or (b) insert a `bind()` call to reach the correct state in which `unbind()` can be called. All of these *fix candidates* would be valid at the abstraction level—but would they also work for the concrete program?

We have built a tool called PACHIKA that extracts the above models from passing and failing runs of programs (currently in Java), compares the models to determine anomalies, and automatically generates possible fixes. PACHIKA validates the fixes against the original failing run,

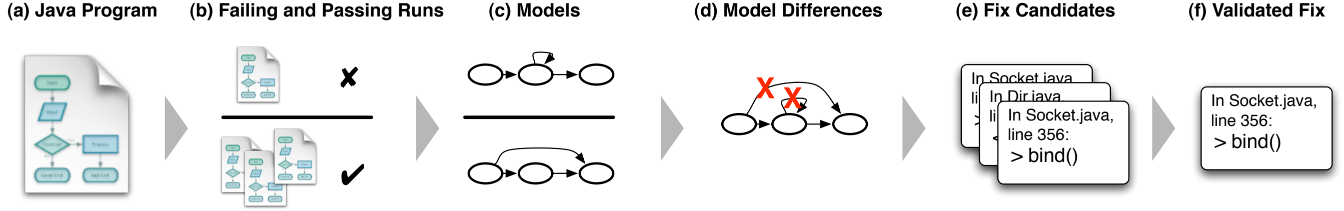

Figure 2. PACHIKA takes a Java program (a) and out of its passing and failing runs (b), it mines object behavior models (c). From differences (d) between the models, it derives fix candidates (e), which it then validates against the regression test suite. Only validated fixes remain (f).

ensuring that the fix indeed solves the problem at hand; it also runs the program's regression test suite to minimize the risk of introducing new problems. Only fixes that pass this validation will eventually be presented to the programmer.

In the MINA example, PACHIKA finds that the fix candidate (a) introduces an alternate failure in the failing run, while candidate (b)—inserting an additional `bind()` call—passes all the tests; this candidate is the fix PACHIKA suggests to the programmer. This is also how the real MINA bug was eventually fixed as indicated by the project's history.

The rest of this paper presents the details of the above approach, and we evaluate its performance on real-life programs with real-life bugs. We make the following contributions:

- A technique to automatically *derive fix candidates* from anomalies in program executions. To our knowledge, this is the first time that fixes are directly generated from mined specifications.
- A method for *validating these fix candidates* using the failing run and automated quality assurance, eventually suggesting the best fix.
- An evaluation of the *effectiveness and the efficiency of the approach* on the iBUGS collection of real-life bugs.

II. Mining Models

To obtain information about passing and failing runs, PACHIKA must *trace* the executions—that is, collect all information required to mine models. For this purpose, PACHIKA uses the ASM framework to inject additional statements into the program. The injected instructions trace events like access to fields and method calls.

The next step reads the trace file and generates an object behavior model for each object. In essence, the model miner builds and maintains a representation of the heap and updates models whenever a method changes the state of an object.

Unfortunately, using concrete values for primitive fields generates models that are too fine-grained. To deal with this issue, PACHIKA uses *abstract values* rather than concrete values. The problem is to choose the right level of abstraction. If the abstraction is too strong, information vital for detecting violations may be lost. If the abstraction is too weak, though, we might still end up with models that are too fine-grained and thus discover too many anomalies.

Table I
Abstraction methods.

Type	Categories
Complex	$x = \texttt{null}$, $x \neq \texttt{null}$
Numerical	$x < 0$, $x = 0$, $x > 0$
Boolean	x, $\neg x$

Our method for abstraction is inspired by Liblit et al. [8], who have successfully applied abstraction in the context of statistical bug localization. Table I summarizes how concrete values are mapped to abstract values. The successful application of this approach in the context of bug localization by Liblit et al. gives us reason to believe that it provides a suitable level of abstraction when mining anomalies. However, we did not investigate other abstraction methods and therefore do not claim that our approach is the best.

III. Finding Violations

Our approach for detecting anomalies compares models of *passing* and *failing* runs. From the passing models, PACHIKA learns preconditions for a method invocation and checks the failing model for violations of these preconditions.

Even a very short run of an object oriented program creates a large number of objects. In MINA, for example, the failing run lasts only 0.3 seconds but generates over 18,000 objects. Analyzing all these models, while possible in principle, would take too much time in practice. We need to find a heuristic that reduces the search space by only considering a subset of all objects. A good heuristic selects all objects whose behavior is relevant for the failure, and only few objects that are irrelevant.

In its current state, PACHIKA requires the failing run to abort with an exception and extracts models for all objects that are reachable through the parameters of the methods on the stack. This approach was inspired by work of Artzi et al. [1], who use a similar technique to reproduce crashes.

The final step for detecting violations is to check all method invocations from the failing model to see whether they violate any of the preconditions mined from the passing model. If an invocation violates at least one precondition, PACHIKA remembers the violated preconditions, and the state before the invocation.

Table II
SUBJECTS USED IN THE PAPER.

Program	Crashing Bugs	Size (LOC)	Number of Tests
ASPECTJ	18	75,123	1,178
RHINO	8	37,902	1,499

Table III
TRACING OVERHEAD AND EXECUTION TIMES FOR ALL SUBJECTS.

	Tracing Overhead (factor)	Trace File Size (MB)	Model Mining (s)
ASPECTJ	9	223	110
RHINO	26	11	8

IV. GENERATING FIXES

For each invocation of a method *m* that violates at least one precondition, PACHIKA generates fix candidates based on the passing and failing models. In general, there are two possibilities to fix a violation based on models. The first is to satisfy the preconditions of *m* by *inserting calls* that make the necessary changes to the state. The second strategy is to avoid the violation by *deleting the violating call to m*.

We refer to the non-validated fixes generated by PACHIKA as the set of *fix candidates*. Each fix candidate is applied in isolation and evaluated in two steps. First, we execute the failing test. If the fix changes the outcome to passing, we call it a *potential fix*. For each potential fix, we subject it to the program's automated quality assurance—in our case, all tests of the program's regression test suite. If the fix does not alter the outcome of any one test, we refer to it as a *validated fix*. Only validated fixes will be presented to the programmer as proposed fixes for the failure.

V. EXPERIMENTAL EVALUATION

To evaluate the effectiveness of our approach, we ran PACHIKA on the two subjects provided by the iBUGS repository [4]. iBUGS contains programs together with test runs and bugs as they actually occurred in the history of the project. For a subset of the bugs, iBUGS also provides test cases that reproduce the problem, which we refer to as failing tests. In our experiments, we use the projects' regression test suites as passing runs.

A. Subjects

Table II summarizes information about the subjects used in the iBUGS study. The column "Crashing Bugs" gives the number of bugs that caused the program to crash. We included all these bugs in our study. For each bug in the repository, iBUGS contains a snapshot of the project right before and right after the bug was fixed. Thus, the size of the project and the number of tests varies from bug to bug. Columns "Size" and "Number of Tests" therefore list only the values for the latest bug included in the study.

B. Running the Experiments

We performed experiments with all crashing bugs of ASPECTJ (18) and RHINO (8). For each bug we mine models from the failing run and compare them against the models of a randomly chosen subset of all passing runs. All candidate fixes generated by PACHIKA are verified against the whole regression test suite.

As is to be expected, tracing incurs a huge amount of runtime overhead. Since both ASPECTJ and RHINO contain over 1,000 tests, tracing and mining the test suite was the most time-consuming part in our experiments. For example, tracing and mining all 1,038 runs in the test suite of bug 87376 takes a little less than two days. Unfortunately this needs to be done for each investigated bug, since each bug is fixed in a different version of the code base.

In practice, however, tracing and mining the test suite only takes place once for each released version of a program. As soon as a new version is released to the public, we can trace the test suite, mine models for all objects in the traces, and store them for reuse. For every bug report filed for the new version, we can reuse the cached models.

C. Results

For RHINO, PACHIKA generates fix candidates for three out of eight bugs. None of these fixes turns the failing test into a passing one. We examined the results in detail and found two causes:

- RHINO is considerably smaller than ASPECTJ and contains only a very small number of classes that have complex models. Thus, PACHIKA finds only a small number of violations per bug.
- In many cases where a violation is found, technical restrictions such as the limitation to methods without parameters prevent PACHIKA from generating a fix. We hope to remove some of these restrictions in the near future and thus be able to generate more fixes for RHINO.

The results for ASPECTJ are summarized in Table IV. PACHIKA generates fix candidates for 14 out of 18 bugs. For 6 bugs, PACHIKA finds at least one fix that causes the failing run to pass. For 3 out of those 6 bugs, there is at least one validated fix. The fixes for bugs 173602 and 121616 turned out to be semantically equivalent to the fix applied by the developers. Due to space restrictions, we will only discuss bug 51322 in detail.

The failing run for bug 51322 crashes ASPECTJ by causing a `NullPointerException` in method `build()` of class `InterType MethodDeclaration`. Figure 3 shows the relevant parts of this method, together with the fix as applied by the developers, and the fix generated by PACHIKA. The failing run contains two invocations of method `build()`, of which only the last one fails. For

```
  public EclipseTypeMunger build(ClassScope classScope) {
    ...
    if(ignoreFurtherInvestigation) { return null;
    } else {
      binding = classScope.referenceContext.
                    binding.resolveTypesFor(binding);
>     // Fix generated by PACHIKA
>     binding.constantPoolDeclaringClass().
>          addDefaultAbstractMethods();
>     binding.constantPoolDeclaringClass().methods();
>     // Fix from source repository
>     if (binding == null) {
>       throw new AbortCompilation();
>     }
      ResolvedMember sig = new ResolvedMember(...);
         ...
    }
  }
```

Figure 3. The proposed fix for bug 51322 invokes methods that initialize values, avoiding the illegal access in a subsequent invocation of `build()`.

Table IV
RESULTS OF THE EXPERIMENTAL EVALUATION FOR ASPECTJ.

	Fix Candidates		Potential	Validated
Bug	Insert	Delete	Fixes	Fixes
34858	420	50	0	0
43033	219	65	0	0
51322	**112**	**190**	**56**	**1**
67774	0	72	0	0
70619	6	1	0	0
75129	0	0	0	0
87376	20	218	0	0
107858	405	235	1	0
109614	0	0	0	0
120474	0	0	0	0
121616	**123**	**0**	**38**	**1**
125475	72	122	7	0
128237	283	4	123	0
131933	0	50	0	0
152631	0	783	0	0
158412	2895	310	0	0
158624	0	0	0	0
173602	**17**	**13**	**7**	**1**

the first invocation, PACHIKA detects a precondition violation for the `declaringClass` attribute in the `binding` variable. The model from the passing run contains a path that repairs this violation, which consists of invoking `addDefaultAbstractMethods()` and `methods()`. When this fix is applied to ASPECTJ, the state of `binding` is altered such that the second invocation of `build()` no longer occurs and the failing run passes. The fixed version also passes all the other tests.

The developer's fix for this problem is simply to abort the execution of `build()`, which is very different from PACHIKA's fix. However, both fixes comply with the specification as given by the program's test suite.

D. Discussion

Our results show that PACHIKA works much better on ASPECTJ (validated fixes for 3 out of 18 bugs) than on RHINO (no validated fixes). A closer examination of the log files revealed that there are much fewer violations of preconditions than in ASPECTJ, the reason being that there is only a small number of classes that have models with preconditions.

The validated fix for bug 51322 highlights a problem for approaches that validate fixes using the test suite: The quality of validated fixes is highly dependent on the quality of the test suite. A bad test suite will cause many fixes to be validated successfully and thus a lot of false positives to be presented to the user. However, in the absence of a formal specification, a test suite is still the best way to automatically assess the impact of a change on the program.

E. Threats to Validity

The scope of our study is limited, as it only investigates 26 bugs in two programs. Therefore, the results of our experiments are hardly generalizable. However, it is difficult to conduct a controlled experiment with realistic data since there is only little such data available.

PACHIKA is a complex system that consists of almost 30,000 lines of code. We verified the correctness of model mining and fix generation for several small artificial test cases. However, the huge amount of data and the complexity of the system make it impossible to check every step for realistic examples.

There also is a risk that PACHIKA generates fixes that only apply to the symptom at hand, rather than the problem root cause ("The method crashes when `p` is `null`, so let's insert a check for it"). This risk is best countered by quality assurance; in particular, any increased level of automated validation (such as contracts or widespread program proofs) will automatically filter out more bad fix candidates as generated by PACHIKA.

VI. RELATED WORK

The most frequent work in automated debugging deals with the problem of *bug localization*—that is, relating a failure to possible bug locations. Milestones in that direction include the TARANTULA approach by Jones et al. [6] as well as *statistical debugging* [8] by Liblit et al., who allow the programmer to focus on a small percentage of the code.

Most related to PACHIKA is the recent work by Weimer et al. [11] on automatic patch generation. Weimer et al. use adaptive random search to systematically mutate a failing C program by inserting, swapping, and deleting statements. Rather than using adaptive random search, PACHIKA relies on behavioral differences between passing and failing runs, which keeps the search space focused. Such a focus is very much needed: It is unknown whether the approach of Weimer et al. scales up to a program like ASPECTJ, with more than 75,000 lines of code and a test suite where *one single run* already takes a minute.

PACHIKA is an instance of *specification mining* tools. The behavior models as mined by PACHIKA were first implemented in the ADABU tool [3]. The concept was later adapted by Ghezzi et al. [5]. Their ADIHEU tool uses

models generated by ADABU to support recovering algebraic specifications from program runs. This approach could also be used in PACHIKA to capture object behavior and find anomalies.

Our work on generating fixes was heavily inspired by recent work on generating tests. Ciupa et al. [2] generate random sequences of method calls, leveraging existing contracts to retain only valid sequences. When a test case fails, the approach of Leitner et al. [7] automatically extracts a test case that reproduces the failure. Both generation and extraction of call sequences to characterize passing and failing runs are key concepts of PACHIKA.

VII. Conclusion and Consequences

The future of automated debugging lies in the automatic generation of fixes. Applied to real-life Java programs, our PACHIKA tool can generate fixes for 3 out of the 18 post-release bugs that crash ASPECTJ. By leveraging the difference between normal and abnormal behavior, we successfully constrain the search space to quickly generate potential fixes that not only remove the problem at hand, but also have a high diagnostic quality. Starting with behavioral differences, coupled with strict filtering via the test suite ensures a zero rate of false positives, ensuring that PACHIKA increases productivity. The approach can easily be extended to quality assurance beyond testing: As soon as a specification can be automatically validated, PACHIKA can leverage it to filter fix candidates—such that only true corrections remain.

Our future work will focus on the following topics:

Adaptive fix generation. With a larger set of possible fixes, one could consider adaptive techniques to systematically explore the search space, as in the approach of Weimer et al. [11]. One interesting possibility could be to use fixes from behavioral differences as a basis for further mutations.

Assessing the impact of fixes. What happens if there are multiple fix candidates that all pass the test suite? In this case, we also would like to minimize the *impact* on passing executions—impact as measured using dynamic invariants [10] or object behavior models.

Leveraging contracts. A key aspect of the approach is the need to ascertain, before calling a method, whether its precondition is satisfied. In the present work, as noted, preconditions have to be inferred from a model. Although assertion inference has made considerable advances, it still falls short of inferring all assertions that programmers would write, as indicated in particular in a recent study by some of the authors [9]. One of the next steps in our work is to apply the ideas to the Eiffel language, where programmer-written contracts not only filter out invalid fixes, but can also serve as boilerplates for generating alternative fixes.

The PACHIKA tool is available for download as an open source Java system. The package also includes all the necessary data to replicate and extend the above experiments. For more information on PACHIKA, visit its Web site:

http://www.st.cs.uni-saarland.de/models/

Acknowledgments. This work is funded by Deutsche Forschungsgemeinschaft, Ze509/4-1 and Hasler-Stiftung, Grant no. 2327. The concept of generating fixes from differences in passing and failing runs was conceived with Andreas Leitner. We thank Andrzej Wasylkowski, David Schuler, and the anonymous reviewers for their helpful comments on earlier revisions of this paper.

References

[1] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP 2008*, pages 542–565, Paphos, Cyprus, July 9–11, 2008.

[2] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *ISSTA '07*, pages 84–94, New York, NY, USA, 2007. ACM.

[3] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA '06*, May 2006.

[4] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *ASE '07*, pages 433–436, November 2007.

[5] C. Ghezzi, A. Mocci, and M. Monga. Efficient recovery of algebraic specifications for stateful components. In *IWPSE '07*, pages 98–105, New York, NY, USA, 2007. ACM.

[6] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE '05*, pages 273–282, 2005.

[7] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva. Contract driven development = test driven development − writing test cases. In *ESEC-FSE '07*, pages 425–434, New York, NY, USA, 2007. ACM.

[8] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05*, pages 15–26, June 2005.

[9] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA '09*, pages 93–104, June 2009.

[10] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA '09*, pages 69–80, June 2009.

[11] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09*, pages 364–374, Vancouver, Canada, May 2009.