# Generating Fixes from Object Behavior Anomalies

Valentin Dallmeier · Andreas Zeller
Dept. of Computer Science
Saarland University, Saarbrücken, 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* a bug (the source of a failure) 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.

## 1. 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: sophisticated program analysis guides the programmer along dependencies [15], statistical debugging highlights execution features that correlate with failures [14, 17], and experimental techniques automatically isolate failure causes in the input [23] or program changes [4]. All these techniques narrow down the set of possible bug locations, presenting the programmer with a list of likely locations.

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 as well, effectively automating the entire debugging process for a significiant subset of programming errors.

The following example, simple but addressing a real-life application 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 [5], 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 object state.

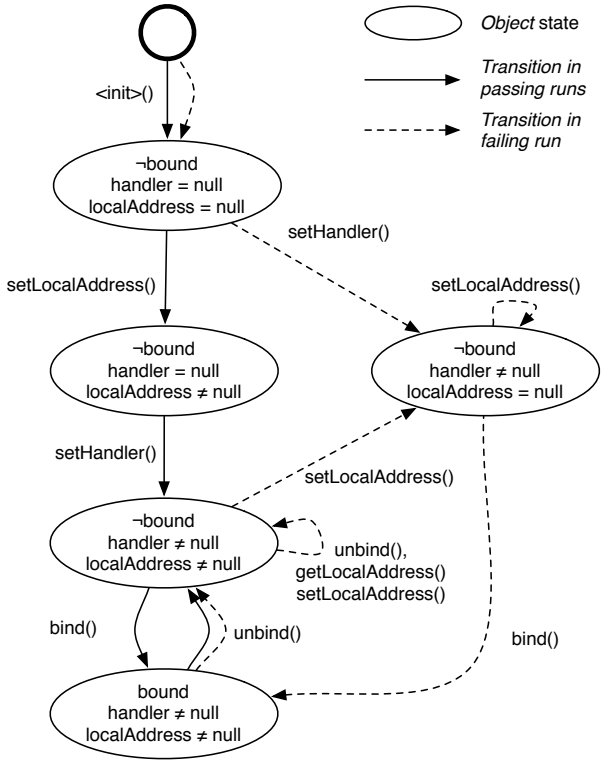


**Figure 1: A combined model of passing and failing runs for the MINA `BaseIoAcceptor` class.**

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 `unbind()` multiple times in a row—even when the `bound` attribute is already false. 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


*ASE* 09 Auckland, New Zealand

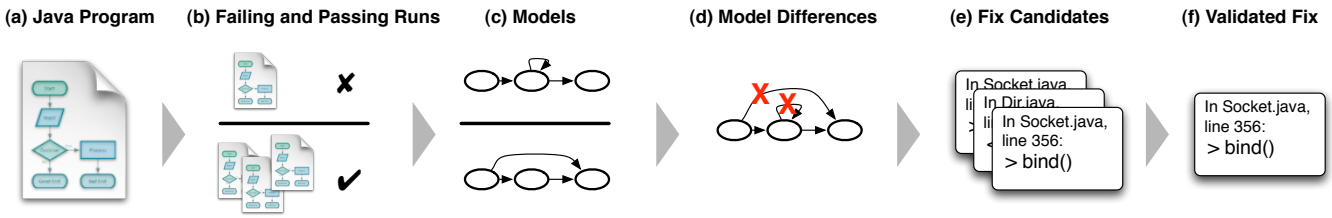


**Figure 2: How PACHIKA works. PACHIKA takes a Java program (a) and out of its passing and failing runs (b), it mines object behavior models (c). From differences (d) between the models, it derives fix candidates (e), which it then validates against automated quality assurance (e.g., a regression test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

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[1] 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, 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 bug. After recapitulating existing techniques to extract object behavior models (Sections 2 and 3) and detecting anomalies (Section 4), we make the following contributions:

- We present a technique to automatically *derive fix candidates* from anomalies in program executions (Section 5). To our knowledge, this is the first time that fixes are directly generated from mined specifications.
- We present a method for *validating these fix candidates* using the failing run as well as automated quality assurance (Section 6), eventually suggesting the best fix.
- We evaluate the *effectiveness and the efficiency of the approach* on the iBUGS collection of real-life bugs (Section 7).

Our evaluation results are promising. PACHIKA, as expected, is applicable only to a fraction of real-life bugs: In ASPECTJ, it generates a fix for 3 out of 18 post-release crashing bugs. For each of these bugs, however, it produces a validated and reasonable fix; in all other cases, it stays silent. The close-to-zero[2] rate of false positives is due to the extensive fix validation: Even if PACHIKA is able to produce a fix only for 15% of all failures, it will significantly reduce debugging effort for these 15% while not negatively affecting the other 85%. Even a low chance of synthesizing a fix thus brings an overall gain in productivity. Furthermore, such synthesized fixes may eventually be deployed automatically as a "first aid" while being refined by the programmer.

The remainder of the paper is organized along the individual stages of PACHIKA (Figure 2). In the first step, PACHIKA traces the execution of a failing and one or more passing runs (Section 2). In the next step, PACHIKA analyzes the traces to identify relevant objects and mines models for these objects (Section 3). The tool then searches the models from the failing run and identifies method invocations that violate preconditions as specified by the models of the passing run (Section 4). PACHIKA then derives fix candidates from model differences (Section 5). These fix candidates are then validated against the test suite to find the best fix (Section 6). Throughout these sections, we will use the MINA example as well as another real-life example taken from the APACHE JDO project to illustrate our approach.

After evaluating effectiveness and efficiency of PACHIKA (Section 7), we discuss the general applicability to real-life bugs (Section 8). We close with related work (Section 9), and conclusion and consequences (Section 10).

## 2. TRACING

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 [13] framework to inject additional statements into the program. The injected instructions cause a flow of events to be written to a trace file. Table 1 summarizes the event types and instrumentation sites for all events handled by the tracer.

PACHIKA traces those events for all classes loaded during the program run. The downside of this approach is that it incurs significant runtime overhead. On the other hand, this makes the implementation of the tracer much less complex and thus less error-prone. Another advantage of tracing everything is that the set of objects that can be investigated in later stages is not limited a priori.

For MINA, the tracer collects 15MB of trace data from the failing run and 7MB of trace data from the passing run. When trace data is collected, execution time increases from 0.3 seconds to 11 seconds, which is a factor of roughly 33. Section 7.4 provides more information on runtime overhead and discusses implications on the applicability of PACHIKA.

## 3. MINING MODELS

The next step reads the trace file and generates object behavior models for a subset of all traced objects (for details on how this subset is chosen, see Section 4). In essence, the model miner builds and maintains a representation of the heap and updates models whenever a method changes the state of an observed object.

[1]"Pachika" is the Swahili word for "fix", "insert".

[2]The actual rate of false positives depends on the quality of the test suite. This is discussed in Section 7.7.

**Table 1: Types of events traced by PACHIKA. Access to arrays is handled the same way as access to fields.**

| Event | Traced Data | Instrumentation Sites |
|---|---|---|
| *METHODSTART* | Thread, Method, Parameters | Start of each method / constructor |
| *METHODEND* | Thread, Method | `ARETURN`, `DRETURN`, `FRETURN`, `IRETURN`, `LRETURN`, `RETURN` |
| *OBJECT_NEW* | Thread, Object | `NEW`, `NEWARRAY`, `ANEWARRAY`, `MULTIANEWARRAY` |
| *FIELD_WRITE* | Thread, Object, Value | `PUTFIELD`, `PUTSTATIC`, `AASTORE`, `BASTORE`, `CASTORE`, `DASTORE`, `FASTORE`, `IASTORE`, `SASTORE` |
| *FIELD_READ* | Thread, Object, Value | `GETFIELD`, `GETSTATIC`, `AALOAD`, `BALOAD`, `CALOAD`, `DALOAD`, `FALOAD`, `IALOAD`, `SALOAD` |

- For an *OBJECT_NEW* event, a new `State` object that represents the current state of the object is created. This object stores the identifiers and values of all fields in the object. For primitive values, a string representation of the value is stored. For complex values, PACHIKA stores the identifier of the object referred to by the field.
- *FIELD_WRITE* events cause updating of the object's representation with the new value.
- For every *METHODSTART* event invoked on a relevant object, PACHIKA extracts the object's state right before invocation and pushes it on the stack of method start object states. If the method is a constructor call, a new empty model is created for this object.
- For every *METHODEND* event of a method invoked on a relevant object, PACHIKA extracts the state at the end of the method. The state for the corresponding method start is taken from the stack of object states as described above. Finally, PACHIKA adds a transition from the start state to the end state labeled with the method name and signature.

Unfortunately, using concrete values for primitive fields generates models that are too fine-grained. As an example, consider the model of a `Vector` whose state consists of the number of elements currently in the `Vector`. If 100 objects are added to the `Vector`, the miner generates a model with over 100 different states. However, `Vector` behaves the same as soon as it contains one or more elements. Although the model is a precise representation of what happened, it is difficult to compare the behavior of different objects. If there were a second `Vector` to which 200 elements were added, PACHIKA would find many model differences although, on an abstract level, the two objects behave the same.

One way to deal with this is to use *abstract values* rather than concrete values in the model. The problem is to choose the right level of abstraction. If the abstraction is too strong, we might lose information that is vital for detecting violations. On the other hand, if the abstraction is too weak, we might still end up with models that are too fine-grained and thus discover too many anomalies.

**Table 2: Abstractions used by PACHIKA.**

| 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. [17], who have successfully applied abstraction in the context of statistical bug localization. Table 2 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.

In the case of the MINA failing run (Figure 1), the model has 5 different states, one of which is the empty starting state. Each state contains the values for three attributes, which are mapped according to Table 2.

## 3.1 Model Depth

The models that PACHIKA mines for the MINA bug contain all the information required to fix it. PACHIKA detects an incorrect value of `bound` and synthesizes a fix from the passing model. In more advanced examples, many methods also have preconditions on fields that are not part of the object, but rather part of other objects referenced by this object. In that case, PACHIKA would be unable to fix the bug, as it does not include the state of transitively reachable objects.

We refer to models that contain state of transitively reachable objects as *deep models*. To mine such models:

- We introduce a *depth* parameter which defines the number of indirections on the heap PACHIKA will consider when including state. For example, in MINA a model of depth 0 for `BaseIOAcceptor` includes the values of attributes `handler`, `localAddress` and `bound`. Depth 1 also includes the values for attributes of `handler` and `localAddress`.
- State extraction is changed such that it includes the state of transitively reachable objects up to the configured depth.
- For each method that changes at least one object, PACHIKA calculates which models are affected by the changes and adds transitions in the models for all objects.

Let us show another real-life example to illustrate the concept of deep models, taken from the bug database of the APACHE JDO project. In this example, a `PersistenceManager` class manages objects stored in a database. Internally, `PersistenceManager` uses a `Transaction` object to synchronize access to the database. The `Transaction` object is available to clients via a getter method. For consistency reasons, access to persistent objects requires an active `Transaction`. In the failing run, a client requests an object by calling `getObjectById()` when the `Transaction` is inactive. In all passing runs, this is handled correctly.

Figure 3 shows a simplified version[3] of the passing and failing model for the `PersistenceManager`. The state contains the transaction `tx`, as well as the `Transaction` object's active flag if the `Transaction` is not null. Transitions in the state of the `PersistenceManager` now also occur if a method changes the

[3]The actual models mined for this example are too large to present in a paper but can be viewed at the project's web page (Section 10).

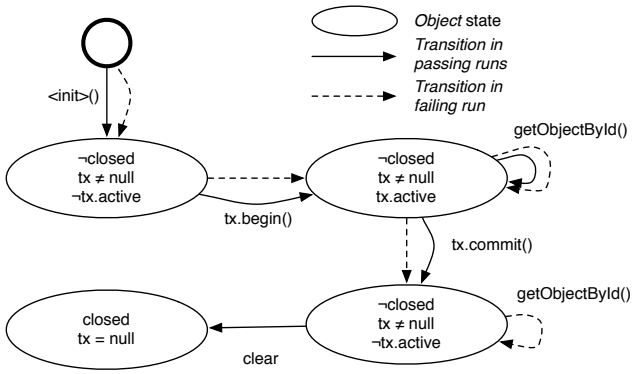


**Figure 3: A deep model of `PersistenceManager` for the passing and failing runs of bug JDO 28. In the failing run, the second invocation of `getObjectById()` violates the precondition that `tx.active` is `true`.**

state of the `Transaction`. This model captures the interplay between calls to `getObjectById()`, `tx.begin()` and `tx.commit()`.

## 3.2 Mining Preconditions

Every method has a (potentially empty) set of preconditions that need to be satisfied in order to invoke the method successfully. For example, the `unbind()` method in the MINA example has the precondition that `bound` needs to be `false`.

In some languages, such as Eiffel, Spec# and JML, programmers would be able to provide preconditions explicitly. In this paper we are working with plain Java programs where preconditions have to be inferred. Section 10 discusses this idea further.

A first approach to mining preconditions from models would be to search for common properties of attributes in states in which a method is invoked. This approach has two disadvantages. First, it limits preconditions to the state of the object the method is invoked on. Second, a method typically does not read all attributes of the state; PACHIKA would thus generate spurious preconditions.

To solve these problems, PACHIKA traces *the set of fields that are read by a method invocation* and generates preconditions only for those fields. For example, in the case of `unbind()`, PACHIKA detects that the method only reads fields `bound` and `handler`, and therefore only looks for preconditions that affect those two fields. If a method reads a field that is part of a parameter, the field will also be included in the set and thus PACHIKA also detects preconditions for fields of parameters.

In practice, identifying the set of relevant fields is more complex than only tracing field reads for every method invocation:

- Many methods create and use *temporary objects.* Field reads on such objects cannot yield preconditions since those objects did not exist when the method was invoked. We therefore only include field reads on objects that existed prior to the invocation.

- Many programs make extensive use of *getter and setter methods.* To retrieve the value of a field, a method invokes a getter rather than accessing the field directly. To deal with this, PACHIKA propagates a field access to the calling method if the accessed object is also visible in the caller.

When generating models, PACHIKA annotates each method invocation in the model with the set of fields read. This information is then used in the next step to detect violations.

# 4. DETECTING VIOLATIONS

The basic technique for detecting anomalies is to compare 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.

One way to approach this is to identify suspicious points in the execution of the program and include all objects that are accessible at those points. For non-crashing bugs, this is difficult since all points in the execution are equally suspicious. To alleviate this problem, we could use an anomaly-based bug localization tool such as TARANTULA [14] or DIDUCE [12] to identify suspicious points. However, in that case, the quality of the generated fixes strongly depends on the bug localization tool's ability to identify the right locations. To avoid this dependency, we limit the evaluation in this paper to crashing bugs where it is easy to identify suspicious program points.

For crashing bugs, we can use the crashing method as a suspicious point and include all objects that are accessible to methods on the stack. This heuristic assumes that a failure occurs close to where the infection (a faulty program state) originates. This assumption does not hold for all bugs. In our experience, however, many crashing bugs are fixed in a method that is active when the program crashes. We therefore believe that this heuristic will include the relevant objects in sufficiently many cases.

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. [2], who use a similar technique to reproduce crashes. Unlike that approach, however, PACHIKA does not include all transitively reachable objects, but only follows references up to a certain depth (see Section 3.1).

Once PACHIKA has mined models for all relevant objects from the failing run, the next problem is to choose passing models against which to compare the failing models. PACHIKA currently takes the following approach:

- First, PACHIKA searches the passing run for invocations of the same methods as for the failing run. For every such invocation, PACHIKA extracts objects accessible from the method and compares models for objects that were accessible through the same path in the passing and failing runs. For example, if a method *m* has a first parameter that is of complex type, PACHIKA compares passing and failing models for the first parameter.

- If no method invocation is found in the passing run, PACHIKA identifies the set of classes for which models were mined from the failing run. It then extracts models for all instances of those classes from the passing run and then compares models for the corresponding classes.

- If there are no suitable models in the failing run, PACHIKA is unable to detect any violations and therefore exits without generating a fix.

If PACHIKA is able to find comparable models, it will search the models of the passing run for preconditions of method invocations.

For every method *m* that is part of the model, PACHIKA examines all invocations of *m* and extracts the values for all fields accessed by *m* (see Section 3.2). PACHIKA then mines the values for each field and tries to derive simple preconditions such as a field having the same value before all invocations of a method. The tool currently has its own engine to detect preconditions. If necessary, however, it could use DAIKON's [9] invariant detection engine to mine more complex preconditions.

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 a method invocation violates at least one precondition, PACHIKA remembers the violated preconditions, as well as the state in which the violating method was invoked.

For the MINA example, PACHIKA finds three relevant objects. The passing run does not include an invocation of the crashing method, and therefore PACHIKA compares models based on classes. PACHIKA only finds one model with violations, shown in Figure 1. The violation is that `unbind()` in the passing run is only being called when `bound` is `true`. Note that PACHIKA does not extract preconditions for `setLocalAddress()` and `setHandler()`, as those methods do not read fields.

For the JDO example, PACHIKA mines three models from the failing run and compares them based on their classes. Altogether, there are 4 violations, one of which is that `getObjectById()` requires `tx.active` to be `true`.

# 5. 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*.

## 5.1 Inserting calls

In order to satisfy the preconditions of a method *m*, PACHIKA searches the failing and passing models for states that satisfy the preconditions and searches for a path to any of them. For example, the violating method call in MINA happens in a state where `bound` is `false`. The precondition from the passing run requires `bound` to be true. PACHIKA finds one state that satisfies this condition and two possible paths from the violating state to the correct state:

1. The first path is to invoke `setLocalAddress()` first and then `bind()`. This path is not considered because `setLocalAddress()` requires an argument, and PACHIKA cannot synthesize arguments.[4]

2. The second path is to invoke only `bind()`. This is a fix candidate as produced by PACHIKA.

Every feasible path is translated into code which injects calls to all methods on the path right before the violating method call.

## 5.2 Deleting calls

The second strategy is to avoid the violation by deleting the method call if at least one precondition is violated. Depending on where the fix is to be applied, we can remove the call at either the caller or the callee site. To remove callee invocations, PACHIKA generates an if-block that checks the precondition at the beginning of the method, and adds a return instruction as the content of the if-block. At the caller site, PACHIKA also creates an if-block that suppresses the call if the precondition is violated. If the removed method has a return type other than void, we try default values such as `true`, `false` or `null`.

For MINA, PACHIKA generates another fix candidate consiting of an if-block around the call to `unbind()` such that the method is only invoked if `bound` is `true`.

[4]Generally, PACHIKA is limited to methods that do not take arguments. We are aware that this is a severe restriction. However, synthesizing arguments for method invocations is a problem in itself and is therefore left for future work.

# 6. CHOOSING THE BEST FIX

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.

In the case of MINA, PACHIKA generates two candidate fixes, out of which one is successfully validated against the test suite. The fix is to add a call to `bind()` which ensures that the precondition for `unbind()` is satisfied. For JDO, PACHIKA generates 8 fix candidates, of which only one is a potential fix that is validated successfully against the test suite. The fix is to insert a call to `tx.begin()` right before the second call to `getObjectById()`.

Both fixes are semantically equivalent to the fixes that were applied by the developers, and thus can be considered to be valid fixes for the failures.

The notion of "best fix" raises the question whether PACHIKA can produce "bad" fixes, too. If a suggested fix passes all tests but is considered incorrect, the test suite should be improved—very much like, in mutation testing [7], an undetected mutation implies a weakness in the test suite. As soon as the test suite (or generally, automated quality assurance) is set up to catch the invalid fix, PACHIKA will filter it out.

# 7. EXPERIMENTAL EVALUATION

In the previous sections, we have seen how PACHIKA was able to generate successful fixes for two bugs as they occurred in real-life. The two examples were found by analyzing the bug databases of MINA and JDO, manually inspecting the bug reports, extracting the faulty version from the source repository, building and running the test suites. This is a lot of manual effort and is not feasible for a larger study.

To evaluate the effectiveness of our approach, we ran PACHIKA on the two subjects provided by the iBUGS repository [6]. 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.

## 7.1 Subjects

Table 3 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.

**Table 3: Subjects used in the paper. The first two rows show characteristics of the examples used. The last two rows give details on the subjects used in the evaluation. Size was measured using David A. Wheeler's *sloccount*.**

| Program | Crashing Bugs | Size (LOC) | Number of Tests |
|---|---|---|---|
| MINA | 1 | 14,773 | 89 |
| JDO | 1 | 64,017 | 437 |
| ASPECTJ | 18 | 75,123 | 1,178 |
| RHINO | 8 | 37,902 | 1,499 |

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.

## 7.2 Experimental Setup

Currently, PACHIKA requires only one configuration parameter: the depth for searches on the heap (cf. Section 3.1). For our experiments, we used a depth of 1, which we believe to be a reasonable compromise between speed and the range of violations that PACHIKA can detect and possibly fix. This is further discussed in Section 7.6.

## 7.3 Running the Experiments

To conduct the experiments, we perform the following steps:

1. First, we examine all bugs in the repository for which there is at least one test case that reproduces the failure. Since PACHIKA is currently limited to crashing bugs, only those bugs that crash the program are included in the study. This yields 18 usable bugs for ASPECTJ, and 8 usable bugs for RHINO. For each such bug, PACHIKA executes the failing test and parses the stack trace of the failure. The remaining steps are performed for each of those bugs.

2. PACHIKA traces the failing run and identifies the set of objects that are accessible to methods on the stack when the crashing method executes (cf. Section 4). For each such object, a model is mined from the failing run. The remaining steps are performed for each passing test in the test suite.

3. PACHIKA traces the passing run and searches the trace for executions of the crashing method. If at least one invocation is found, models for all visible objects are mined just like for the failing run. If no invocation is found, PACHIKA mines models for all classes for which at least one model was extracted from the failing run (cf. Section 4).

4. If the previous step yields at least one model, PACHIKA compares models to generate candidate fixes for all active methods as described in Section 5. Each candidate fix is first checked against the failing test and then against the test suite (cf. Section 6).

## 7.4 Performance

Our experiments were performed on a 2 GHz AMD machine with a maximum of 2 Gigabytes of memory. Table 4 lists information about overhead and execution times. For MINA and JDO, results are averages over all runs in the test suite. For ASPECTJ and RHINO, we give averages for the latest version used in the experiments. Tracing overhead is expressed as the factor by which execution time increases when tracing is turned on. The third column gives the execution time the model miner takes to extract models for depth 1 (cf. Section 3.1).

**Table 4: Tracing overhead and execution times for all subjects.**

| | Tracing Overhead (factor) | Trace File Size (MB) | Model Mining (s) |
|---|---|---|---|
| MINA | 29 | 42 | 34 |
| JDO | 16 | 356 | 212 |
| ASPECTJ | 9 | 223 | 110 |
| RHINO | 26 | 11 | 8 |

Table 4 does not list times PACHIKA takes to generate fixes, since these are negligible compared to the other steps. The time needed to validate a candidate fix is equivalent to the execution time of the test suite for almost all candidates. In some cases, a fix candidate causes the program to loop endlessly. In that case, we terminate the run after a timeout of two minutes and consider the test as failed.

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 needs to happen 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.

## 7.5 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 (see Section 8). 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 5. For each investigated bug, we give the number of candidate, potential, and validated fixes (cf. Section 6). 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 following sections discuss each of those bugs in detail.

### *7.5.1 Checking for a null reference*

Bug 173602 causes a `NullPointerException` to be thrown in method `resolve()` in class `InterTypeMethodDeclaration`. PACHIKA detects one violation for the invocation of `resolve()`, namely that `binding` must not be null. The *delete method call* strategy generates the fix as shown in Figure 4. The original fix

**Table 5: Results of the experimental evaluation for ASPECTJ. For 3 bugs out of 18, PACHIKA is able to generate a fix that fixes the failure and passes the regression test suite.**

| | 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** |

```
   public void resolve(ClassScope upperScope) {
>       // Fix from source repository
>       if (binding == null) ignoreFurtherInvestigation = true;
>       // Fix generated by PACHIKA
>       if (binding == null) {
>          return;
>       }
       if (munger == null) ignoreFurtherInvestigation = true;
       if (ignoreFurtherInvestigation) return;
          ...
       }
   }
```

**Figure 4: The proposed fix for bug 173602 is to not execute method `resolve()` if the precondition for `binding` is violated.**

also amounts to a conditional return which additionally sets the `ignoreFurtherInvestigation` flag. This flag is later used by ASPECTJ to stop processing the declaration object. However, not setting the flag in this situation does not cause any problems, since none of the tests in the test suite later fails.

### 7.5.2 *Checking for error conditions*

In the failing run of bug 121616, method `resolve()` in class `ValidateAtAspectJAnnotationsVisitor`. PACHIKA detects a precondition violation for parameter `methodDeclaration`, namely that the `ignoreFurtherInvestigation` flag which is returned by `hasErrors()` is `true`. The delete method strategy generates a conditional return in case that the precondition is violated (Figure 5). In this case, the generated fix is equal to the fix applied by the developers.

### 7.5.3 *Invoking methods to set default state*

The failing run for bug 51322 crashes ASPECTJ by causing a `NullPointerException` in method `build()` of class `InterType MethodDeclaration`. Figure 6 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 the first invocation, PACHIKA detects a precondition violation for the

```
public boolean visit(MethodDeclaration methodDeclaration,
                     ClassScope scope) {
>  // Fix generated by PACHIKA
>  // is the same as in the source repository
>  if (methodDeclaration.hasErrors()) {
>    return false;
>  }
   ContextToken tok = CompilationAndWeavingContext.
         enteringPhase(...);
     ...
}
```

**Figure 5: The fix for bug 121616 surpresses the violation by aborting the execution in case `methodDeclaration` has errors.**

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

`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.[5]

## 7.6 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 study in Section 8 compares the number of classes with preconditions for all subjects in this paper.

In most cases where PACHIKA detects a violation, both fix strategies generate fixes. In terms of the numbers of generated fixes, both strategies are also roughly equivalent. In some cases, the *insert method call* strategy generates a large number of fixes due to many different paths through the model. However, with two out of three validated fixes, the *delete method call* strategy in our experiments is more successful when it comes to generating a correct fix.

[5]If PACHIKA's fix would be considered incorrect, a simple remedy is to extend the test suite appropriately, as discussed in Section 6. We asked the developer who committed the original fix for his opinion, but did not get a reply before the submission deadline.

In our experiments we used a depth of 1 for mining models. Of the three validated fixes found by PACHIKA, only the fix for bug 51322 actually requires a depth of 1. The two remaining fixes would also have been found with depth 0. It may be that larger values for depth would have caused PACHIKA to generate more validated fixes. A thorough investigation of how larger depths affect fix generation is left for future work.

The validated fix for bug 51322 (Section 7.5.3) 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.

## 7.7 Threats to Validity

As with any empirical study, the interpretation of the results is subject to several limitations.

**External 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. A manual investigation, as we did it on MINA and JDO, requires a lot of effort and is also difficult to reproduce for other researchers. Although we are aware of these limitations, we believe that our evaluation is realistic since it uses real post-release bugs[6] and relies only on test runs from a bug database or the test suite.

**Internal Validity** 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. It may well be that PACHIKA contains errors which cause fixes to be missed or invalid fixes to be generated. However, verifying potential fixes against the test suite ensures that there are no false positives. We encourage other researchers to validate our results. All bugs used in the evaluation are available in the iBUGS dataset. PACHIKA is also available for download; see Section 10 for details.

**Construct Validity** PACHIKA uses the test suite as a source of program runs. As such, it depends on the tests to correctly classify a run as passing or failing. In some cases, this check is not precise enough. For example, some tests in ASPECTJ simply check the output for a certain keyword, which may lead to a test outcome incorrectly being classified as passing. However, we observed this problem only for a small number of tests and are confident that the huge number of tests ensures a high quality of fixes that are presented to the user.

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. Indeed, our evaluation indicates that this is already the case.

[6]We expect an evaluation of PACHIKA on artificially seeded bugs to yield much better results—in particular if seeding includes addition or deletion of method calls, as most mutation testing approaches do.

**Table 6: How prevalent are classes with preconditions? With the exception of MINA, roughly one third of all classes are complex enough to be misused.**

| | Number of Classes | Classes with preconditions |
|---|---|---|
| MINA | 166 | 15 |
| JDO | 377 | 116 |
| ASPECTJ | 443 | 154 |
| RHINO | 52 | 17 |

# 8. APPLICABILITY

After coming to the conclusion that automatic fixing of failing programs was indeed feasible for some cases, we wanted to investigate the general applicability of tools like PACHIKA. In the experimental evaluation in Section 7, our tool was only able to generate fix candidates for a small number of bugs in RHINO, since only few bugs actually revealed violations of preconditions. Obviously, PACHIKA's applicability is limited to bugs that cause a precondition violation. In order to get a feeling of PACHIKA's potential, we wanted to know *how many bugs actually show precondition violations.*

For this purpose, we investigated a sample of bugs from the bug databases of the projects used in the evaluation. For each bug, we tried to determine whether or not the bug would have caused a violation of a precondition. However, we quickly came to the conclusion that it is not possible to realiably answer this question by only looking at the bug report and source code. On the other hand, manually building and executing each snapshot for a large enough set of bugs is too time-consuming.

PACHIKA's ability to detect bugs correlates with the number of classes that may potentially be used in a wrong way. A high percentage of such classes would mean that there is a big potential for wrong usage that causes violations. To measure the percentage, we generated models for all classes used in our subjects and classified models as having preconditions or not. A model has preconditions if there is at least one method invocation other than that of a getter method which requires another method to be invoked before. For example, in Figure 1, the model for `VmPipeAcceptor` has preconditions, because in order to satisfy the precondition of `unbind()`, method `bind()` has to be invoked before.

Table 6 lists the number of classes for which we mined at least one model (column 2), and the number of classes with preconditions (column 3). Except for MINA, approximately *one out of three classes has a model with preconditions.* Thus, roughly one third of the classes in our projects are complex enough to be misused. Since there are typically several objects with different types in the scope at any point in the program, there is a big potential for detecting anomalies based on violated preconditions.

# 9. RELATED WORK

## 9.1 Locating Bugs

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. [14] as well as *statistical debug-*

*ging* [17] by Liblit et al., who allow the programmer to focus on a small percentage of the code.

Like these approaches, PACHIKA leverages the *difference* between passing and failing executions; rather than suggesting locations, however, it produces *fixes*. By leveraging the test suite (and all other forms of automated validation), PACHIKA can thus successfully weed out invalid candidates, resulting in either a valid fix—or nothing. This "no-false-positives" approach is where our approach greatly differs from existing bug localization techniques. Nonetheless, it can be easily combined with bug localization: When PACHIKA cannot generate a fix, then bug localization may at least suggest a location; or one could use locations as suggested by a bug localization technique as suspicious locations for PACHIKA (cf. Section 4).

## 9.2 Repairing Programs

Most related to PACHIKA is the recent work by Weimer et al. [22] on automatic patch generation. Weimer et al. systematically mutate a failing C program by inserting, swapping, and deleting statements. Their approach then uses an extended form of genetic programming to evolve those mutants that pass (1) the (previously failing) test and (2) as many tests as possible from a regression test suite. The approach produces repairs in less than three minutes on average on a set of ten selected bugs.

Our approach is similar to their technique in that it also generates potential fixes and assesses them via a regression test suite. The contribution and potential of their approach over PACHIKA is clearly the wide range of possible mutations, as well as the adaptive approach in generating fixes.

Rather than using adaptive random search, however, PACHIKA starts right away with 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; it is also unknown how much fine-tuning of parameters is required to quickly find fixes. It is also unclear how the approach of Weimer et al. could integrate bug localization or mined specifications, as PACHIKA does. Last but not least, we evaluate PACHIKA on *all* previously documented crashing bugs of ASPECTJ and RHINO—and thus get an idea of scalability and applicability on real programs and real bugs.

## 9.3 Leveraging Specifications

Prior to [22], Weimer developed a method for automatically and soundly patching programs with a given specification [21]. However, as Weimer states in [22], a formal specification is seldom available—which is why PACHIKA mines and leverages behavior models from passing and failing executions.

In the long run, we expect automatic fix generation to rely on both search-based techniques (as in the approach of Weimer et al.) as well as specification mining (as in PACHIKA)—in addition to the wide range of information that is available via static analysis, theorem provers, bug history, and other techniques.

## 9.4 Repairing State

Demsky et al. [8] show how to automatically fix data structures at run-time, again according to a given specification. Rinard et al. [19] suggest similar repair techniques for invalid memory accesses. In both these works, only the program state is fixed. Weimer's and our work, though, look for repairs not only to the program state of the current run, but to its actual *code* (which as a side-effect yields repairs to the state as well). This requires many more checks, such as contracts or a regression test suite, but also increases confidence in the correctness of the repairs—besides, hopefully, providing a permanent fix to the problem.

## 9.5 Mining Specifications

PACHIKA is an instance of *specification mining* tools. The behavior models as mined by PACHIKA were first implemented in the ADABU tool [5]. The concept was later adapted by Ghezzi et al. [10]. 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.

Dynamic invariants, as conceived by Ernst et al. [9], express properties of data that hold at specific moments during the observed executions. By checking object attribute states, one could use the DAIKON tool to extract pre- and postconditions for method calls and thus object behavior models.

The concept of learning models from actual program runs was first explored by Amons et al. [1], applying a probabilistic NFA learner on C traces. Their approach relies on manual annotations to relate functions to objects (such as C sockets or X11 selections) and to distinguish object definers from object users.

## 9.6 Generating Tests

Our work on generating fixes was heavily inspired by recent work on generating tests. Ciupa et al. [3] 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. [16] 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.

# 10. 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.

There will always be bugs that cannot be fixed automatically. Still, automatic fix generation has much room for improvement. Our future work will focus on the following topics:

**Alternate differences.** Right now, the set of differences we observe and the set of fixes we can generate is limited to conditional method calls. However, there are many more potential fixes that could be generated. For instance, assigning a value to an attribute could instantly fix the object state.

**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. [22]. One interesting possibility could be to start with behavioral differences as fix candidates (as PACHIKA does), and to use these 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 [20], coverage [11], 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 [18] . 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 the project's 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 under the title "AutoFix—Programs that fix themselves". The concept of generating fixes from differences in passing and failing runs was conceived with Andreas Leitner. Andrzej Wasylkowski and David Schuler provided helpful comments on earlier revisions of this paper. We also thank Wesley Weimer for providing us with a preprint of [22].

# 11. REFERENCES

[1] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, New York, NY, USA, 2002. ACM.

[2] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conf.*, pages 542–565, Paphos, Cyprus, July 9–11, 2008.

[3] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *ISSTA '07: Proceedings of the 2007 International symposium on Software testing and analysis*, pages 84–94, New York, NY, USA, 2007. ACM.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *Proceedings of 27th International Conference on Software Engineering (ICSE)*, pages 342–351. ACM, 2005.

[5] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with adabu. In *WODA 2006: ICSE Workshop on Dynamic Analysis*, May 2006.

[6] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, November 2007.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[8] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *ICSE '05: Proceedings of the 27th International Conference on Software engineering*, pages 176–185, New York, NY, USA, 2005. ACM.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[10] C. Ghezzi, A. Mocci, and M. Monga. Efficient recovery of algebraic specifications for stateful components. In *IWPSE '07: Ninth International workshop on Principles of software evolution*, pages 98–105, New York, NY, USA, 2007. ACM.

[11] B. J. Gruen, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *4th International Workshop on Mutation Analysis*, January 2009.

[12] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM.

[13] http://asm.objectweb.org/. ASM 3.1, 2009.

[14] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering*, pages 273–282, 2005.

[15] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *ICSE '08: Proceedings of the 30th International Conference on Software engineering*, pages 301–310, New York, NY, USA, 2008. ACM.

[16] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva. Contract driven development = test driven development - writing test cases. In *ESEC-FSE '07: Proceedings of the ACM Symposium on The foundations of software engineering*, pages 425–434, New York, NY, USA, 2007. ACM.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 15–26, June 2005.

[18] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *ISSTA '09: Proceedings of the 2009 ACM SIGSOFT International symposium on Software testing and analysis*, June 2009. To appear.

[19] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, pages 82–90, Washington, DC, USA, 2004. IEEE Computer Society.

[20] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA '09: Proceedings of the 2009 ACM SIGSOFT Int. symposium on Software testing and analysis*, June 2009. To appear.

[21] W. Weimer. Patches as better bug reports. In *GPCE '06: Proceedings of the 5th International Conference on Generative programming and component engineering*, pages 181–190, New York, NY, USA, 2006. ACM.

[22] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proc. of the Int. Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009. To appear.

[23] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.