On Test Repair Using Symbolic Execution

Brett Daniel[1] Tihomir Gvero[2] Darko Marinov[1]

[1]University of Illinois at Urbana-Champaign
Urbana, IL 61081, USA
{bdaniel3,marinov}@illinois.edu

[2]Ecole Polytechnique Fédérale de Lausanne
Lausanne, VD, Switzerland
tihomir.gvero@epfl.ch

ABSTRACT

When developers change a program, regression tests can fail not only due to faults in the program but also due to out-of-date test code that does not reflect the desired behavior of the program. When this occurs, it is necessary to repair test code such that the tests pass. Repairing tests manually is difficult and time consuming. We recently developed ReAssert, a tool that can automatically repair broken unit tests, but only if they lack complex control flow or operations on expected values.

This paper introduces *symbolic test repair*, a technique based on symbolic execution, which can overcome some of ReAssert's limitations. We reproduce experiments from earlier work and find that symbolic test repair improves upon previously reported results both quantitatively and qualitatively. We also perform new experiments which confirm the benefits of symbolic test repair and also show surprising similarities in test failures for open-source Java and .NET programs. Our experiments use Pex, a powerful symbolic execution engine for .NET, and we find that Pex provides over half of the repairs possible from the theoretically ideal symbolic test repair.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object-oriented programming*

General Terms

Verification

1. INTRODUCTION

Automated regression testing is an integral aspect of software development because it offers several potential benefits: fewer bugs, greater reliability, and lower long-term development cost. However, such testing can have high costs, not only to develop tests but also to maintain them so that they retain their benefits as the system under test (SUT) evolves [35]. Software evolution often causes tests to fail. Ideally, failures reveal regression errors in the SUT. Unfortunately, failures can also occur when changing requirements invalidate existing tests. When this happens, we say the tests are *broken*.

Broken tests cause many problems. Updating broken tests takes time. The fear of broken tests can even create a disincentive to write thorough test suites because they are more likely to have broken tests. Developers may not take the time to inspect all failing tests to distinguish regression failures from broken tests. They may instead choose to ignore or delete some failing tests from the test suite, thereby reducing its effectiveness. While these approaches are obviously undesirable, anecdotal evidence suggests that they are very common [3, 6, 32, 36, 38, 41, 44]. Similarly, if regression (also called characterization [4]) tests are generated automatically using a test generation tool [46], developers often find it easier to re-generate tests than address all the failures, reducing the benefit of using the tool.

To avoid the problems that arise from removing or ignoring tests, it is much more desirable to repair broken tests. To do so, developers must update the test code (and perhaps the SUT) such that the tests pass. Repairing tests is tedious and time-consuming, especially when a large number of tests fail. Fortunately, our prior work has shown that it is possible to automate repair of some broken tests [14], reducing the effort required to maintain tests.

Our ReAssert tool [14] was, to the best of our knowledge, the first general-purpose test repair tool. ReAssert automatically suggests repairs for broken tests, allowing developers to repair broken tests with the push of a button. In particular, ReAssert suggests simple yet reasonable changes to test code that are sufficient to cause failing tests to pass. If the user agrees with the suggested changes, ReAssert modifies the test code automatically. Our implementation repaired unit tests written in JUnit, but the technique generalizes to other languages and test frameworks.

We evaluated ReAssert's ability to repair broken tests in several ways [14]. We described two case studies, performed a controlled user study, and repaired some failures from real open-source applications. Our results showed that ReAssert could repair a large portion of test failures (nearly 100% in the case studies and user study) and that its suggested repairs were often useful to developers (75% in the case studies and 86% in the user study). However, ReAssert also has two significant problems. First, ReAssert could repair only about 45% of failures in open-source applications. Second,

ISSTA'10, July 12–16, 2010, Trento, Italy.

in many cases, ReAssert suggested a suboptimal repair, and a more useful repair was possible.

The main reason why ReAssert could not repair some failures was related to expected values. Unit tests commonly prepare the expected results of a computation, then *assert* that they match the actual values returned from the SUT. ReAssert could repair many failing tests by changing the expected value appropriately. However, if a failing test modified expected values, created complex expected objects, or had multiple control-flow paths, then ReAssert could not determine what expected values needed to be changed and in what way. Similarly, ReAssert would suggest a suboptimal repair that would effectively remove some computation of the expected value. A better repair would preserve as much of the test's computation as possible. Section 3 describes three examples that illustrate the ReAssert's deficiencies.

This paper introduces *symbolic test repair*, a new technique that uses symbolic execution [1, 5, 7, 8, 12, 18, 20, 31, 33, 37, 39, 42, 46, 49] to address some of the ReAssert's deficiencies. Symbolic execution treats values not as concrete data but as symbolic expressions that record the state of the computation. The state additionally includes the *path constraints* required to direct the program down a particular execution path. Constraint solving [17] can produce concrete values that direct the program down a path. Most frequently, symbolic execution is used to search for values that make the SUT *fail*. We cast test repair as a dual problem where symbolic execution is used to search for values that make the tests *pass*.

If successful, symbolic execution would provide several benefits for test repair. Most importantly, it would overcome the problems that prevented ReAssert from repairing many failures. It would also naturally complement ReAssert's existing capabilities. Finally, it would provide more useful repair for many broken tests.

This paper makes two contributions.

Propose Symbolic Test Repair: We define a procedure in which symbolic execution can repair tests by changing literals in test code. In addition, we show how such a procedure would fit in ReAssert's general test repair process. Unlike with ReAssert, we do not offer a complete, ready-to-use symbolic test repair tool (primarily because ReAssert is for Java [14] and Pex is for .NET [46], as described in Section 2.2.1), but we do evaluate how well such a tool would work in practice with the current state-of-the-art symbolic execution engines.

Evaluate Symbolic Test Repair: We thoroughly evaluate the applicability of symbolic execution to test repair. Our evaluation addresses three research questions:

Q1: How many failures can be repaired by replacing literals in test code? That is, if we had an *ideal* way to discover literals, how many broken tests could we repair?

Q2: How do literal replacement and ReAssert compare? How would an ideal literal replacement strategy affect ReAssert's ability to repair broken tests?

Q3: How well can existing symbolic execution discover appropriate literals? Can symbolic execution produce literals that would cause a test to pass?

To answer the first two questions, we recreate prior experiments from ReAssert's evaluation [14] on Java applications and also add new experiments on .NET applications. We determine which failures ReAssert and ideal literal replacement would be able to repair, and find that they could together repair 66% (155 of 235) of failures, with literal replacement being able to repair 19% (44 of 235) that ReAssert cannot. In several cases, the repairs that literal replacement would suggest are more useful than the repairs that ReAssert would suggest. Surprisingly, we find that the total fraction of repairable tests is nearly identical between Java and .NET test suites.

To answer the third question, we use Pex [46], a publicly available, state-of-the-art symbolic execution engine from Microsoft Research. We manually mark which values should be symbolic in .NET applications, and use Pex to search for concrete values that make the tests pass. We then perform a similar process for passing tests to see if Pex can recreate literals that would cause the test to pass. Overall, we find that Pex can solve between 53% and 92% of the cases that ideal literal replacement could solve, which is quite impressive for the challenging, complex code used in our experiments.

Our experimental setup and results are available from `http://mir.cs.illinois.edu/reassert/`.

2. BACKGROUND

This work builds on prior research in automated test repair and symbolic execution. It uses ReAssert [14], our unit test repair tool, and Pex [46], a symbolic execution engine developed at Microsoft Research.

2.1 Automated Test Repair

Tests can fail not only due to problems in the SUT but also due to incorrect test code. When test code has problems, developers should change it to repair the failures such that the tests pass. Automated test repair attempts to make this process easier.

Naturally, there is more to test repair than making the test pass. One could trivially "repair" a failing test by deleting the test's code; the test would then pass, but it would not reveal anything about the correct or incorrect behavior of the SUT. Thus, we attempt to produce "good" repairs by adhering to the following requirements:

Make All Tests Pass: Not only should test repair make a given failing test pass, but it should also not cause other passing tests to fail.

Minimal Changes to Test Code: It is important that the repaired test exercises the same behavior of the SUT (to a reasonable approximation) as the original test did before the SUT evolved. Therefore, it is necessary to make simple changes to test code and attempt to retain as much of the test's computation as possible.

Leave SUT Unchanged: Test repair assumes that tests must change, not the SUT. A dual problem is to change SUT to make the tests pass, which is an active area of research known as *automatic debugging* [2, 24, 26, 51, 56] because it automatically repairs bugs in the SUT.

Do not Introduce Bugs: A test should verify correct behavior, so the repair process should not introduce a repaired test that is inconsistent with the intended code specification, even if it is consistent with the SUT implementation. However, it is difficult for a tool to know what "correct" means when the SUT evolves and tests fail. Therefore, it is usually necessary to delegate to the developer to approve suggested repairs. Developer approval is not foolproof, but it reinforces the need for simple, understandable changes to test code.

Our proposed technique for symbolic test repair adheres to the listed requirements: it makes tests pass with low likelihood of causing additional failures, produces minimal changes to test code by only changing literal values, requires no modification to the SUT, and repairs tests in a way that is in many cases easier to understand and inspect than with ReAssert alone. In particular, symbolic test repair focuses on *literal replacement*: it only replaces some literal value(s) in the test code to attempt to repair it to pass.

The primary assumption underlying all these requirements and the key insight leading to automated test repair can be stated in the following manner:

> *For test repair, we use the SUT as the oracle for test correctness, not the other way around as usually done.*

Both ReAssert and symbolic test repair use this insight as the basis for their repair processes. ReAssert records the values returned from the SUT and serializes those values into test code. Symbolic test repair finds values that satisfy constraints introduced by the SUT.

2.1.1 Test Repair with ReAssert

ReAssert was, to our knowledge, the first automated unit test repair tool. Given a failing test, it suggested simple changes to the test code that would be sufficient to cause the test to pass. The developer could inspect and modify the suggested changes. If the changes appeared reasonable, and the developer approved them, then ReAssert modified the test code automatically.

We implemented ReAssert as both an Eclipse plugin and standalone command-line utility. Both acted on unit tests written using JUnit, a popular unit testing framework for Java. While our ideas and the repair process easily generalize to other languages and test frameworks, there is a substantial amount of engineering necessary to reimplement ReAssert for another language.

To illustrate ReAssert's repair process, we describe a very simple failure derived from the user study performed to evaluate ReAssert [14]. In the study, participants tested a shopping cart application. Each test initialized a shopping cart object, added or removed products and coupons, then verified that the bill was calculated correctly. For example, the following test code verifies that the printed bill matches the expected string:

```
Cart cart = ...
cart.addProduct(...);
String expected = "Total: $9.00";
assertEquals(expected, cart.getPrintedBill());
```

The system evolved such that the total was calculated differently—it became $6.00 instead of $9.00—causing this and several other tests to fail. Instead of fixing every failure manually, participants could invoke ReAssert from Eclipse, which would then display suggested repairs. In this example, ReAssert replaced the value of the `expected` variable with the correct string:

```
String expected = "Total: $6.00";
```

This repair is deceptively simple, but producing it is nontrivial. To produce it, ReAssert recorded the actual value returned from the SUT, traced the `expected` variable from its use in the failing assertion back to its declaration, and then suggested to replace the value in code.

More generally, ReAssert provided several *repair strategies* tailored to common kinds of test failures [15]. Each strategy encapsulated a particular code transformation depending on the static structure of the code, the exception thrown by the failing test, and the runtime values that caused the failure. Most strategies—including the *trace declaration-use path* strategy used in the example above—transformed tests by serializing into the code the values returned from the SUT. Strategies used the exception's stack trace to find location in code to repair. ReAssert was also extensible, allowing a user to create custom repair strategies.

ReAssert used a five-step repair process around its library of repair strategies:

1. Instrument assertion methods such that they record their arguments if the assertion fails.
2. Execute the failing test, capturing the exception it throws and the values recorded by the instrumentation.
3. Apply the appropriate strategy to transform the source code of the failing test.
4. Recompile the modified test code.
5. Repeat the process until the test passes or ReAssert reaches a limit on the total number of repairs.

Symbolic test repair would complement ReAssert's existing repair strategies and fit well into its repair process. We envision that it would be implemented as an additional strategy for use when ReAssert's default strategies do not apply. In this way it could take advantage of ReAssert's existing user interface and source code transformations. Section 4 describes our proposed extension in detail.

2.2 Symbolic Execution

Symbolic execution [10, 31, 46] is a program analysis technique that performs computation on symbolic rather than concrete values. Each symbolic value records a symbolic expression representing some operations performed during computation. When the program encounters a conditional dependent on symbolic variables, the symbolic execution engine needs to decide which branch to take, and accumulates a so called *path constraint* that represents the branches that were taken. Symbolic execution can then explore different feasible paths in the program.

A recent approach to symbolic execution is to combine it with concrete execution [7, 8, 18, 20, 33, 42, 46]. The approach differs from the traditional symbolic execution in that it executes a program on concrete inputs while simultaneously recording symbolic path constraints. Which branch to take is then determined by concrete values. By adjusting the path constraints—usually by negating one of its branch conditions—standard constraint-solving techniques [17] can produce new concrete values that force the program execution down a different program path.

2.2.1 Symbolic Execution with Pex

Pex is a state-of-the-art symbolic execution engine developed at Microsoft Research for analyzing .NET code. It uses the Z3 constraint solver [17], which provides decision procedures for most constraints encountered in the .NET intermediate language as well constraints involving low-level pointers and memory references.

To instruct Pex that some values should be symbolic, one can either use parametrized unit tests [47], where test parameters are symbolic, or provide explicit calls to *nondeterministic choice generators*. Conceptually, Pex re-runs a program many times on different values, attempting to maximize the number of paths executed. For symbolic variables, Pex initially chooses very simple concrete values (e.g., `0`, an empty string, `null`, etc.) and builds more complex values based on the constraints encountered in the program. To illustrate Pex, we consider a simple example that uses Pex's method called `ChooseValue`, which produces a symbolic value of a given type. The call to `ChooseValue` in the following code makes the `input` variable symbolic:

```
int input = ChooseValue<int>();
if (input > 5) {
  throw new ArgumentOutOfRangeException();
}
```

Pex first executes the code with `input` equal to `0`. The program completes successfully, but the branch condition introduces the constraint that the symbolic value was less than or equal to `5`. Pex negates this constraint on the next execution, and (using Z3) finds that, for instance, `6` is a value that satisfies the new constraint. It re-executes the code and follows the opposite branch of the conditional, which throws an `ArgumentOutOfRangeException`. After this point, Pex has executed both feasible paths in the program, so the exploration can stop.

In brief, we cast test repair into a symbolic execution problem as follows. First, we make certain literal values in the test code symbolic by replacing the literals with the Pex's choice generators. (In our experiments, we currently perform this step manually with a number of ad-hoc Visual Studio macros, but it is possible to automate this step as discussed in Section 4.2.) Then, we run Pex on this code and accept the values for which the test pass. Section 4 describes this process in greater detail.

We chose Pex for this work because it is a robust and powerful symbolic execution engine. Unfortunately, Pex and ReAssert act on different languages: .NET and Java, respectively. This language mismatch makes it challenging to implement a complete symbolic test repair tool: we either need to reimplement ReAssert ideas in .NET and Visual Studio, or we need to use a symbolic execution engine for Java. We have experimented with several engines for Java but found them to be much less powerful than Pex at this time.

3. MOTIVATING EXAMPLES

In our evaluation of ReAssert, we encountered several failures that ReAssert could not repair or that it could have repaired in a better way. Here we describe three such examples that illustrate how replacing the appropriate literal(s) in a failing test would cause it to pass. In Section 4, we describe how these these and similar tests can be repaired using symbolic execution.

3.1 Modifications to Expected Values

Many unit tests prepare expected values and then compare them against the actual values returned from the SUT. If a test computed expected values rather than declaring them directly as literals, then ReAssert could not correctly determine what values needed to change.

Several examples from our user study illustrate this problem. As shown in Section 2.1.1, participants' tests verified the printed bill returned from a simple shopping cart application. Two of the 18 participants prepared the expected bill by concatenating strings, as in the following example:

```
// original test code
double total = 9.00;
String expected = "Total: $" + total;
assertEquals(expected, cart.getPrintedBill());
```

The system evolved, causing this test to fail as the total value should be `6.00`. A naive repair could change the assertion, making the test to pass but cutting off some computation and likely not producing a useful repair:

```
double total = 9.00;
String expected = "Total: $" + total;
// naive repair
assertEquals("Total: $6.00", cart.getPrintedBill());
```

ReAssert repaired the failure by removing the concatenation and replacing the initial value of the `expected` variable:

```
double total = 9.00;
// ReAssert repair
String expected = "Total: $6.00";
assertEquals(expected, cart.getPrintedBill());
```

A more useful repair would instead replace the value of the `total` variable:

```
// literal replacement
double total = 6.00;
String expected = "Total: $" + total;
assertEquals(expected, cart.getPrintedBill());
```

A similar pattern in which a developer performed operations on expected data also appeared in several of the open-source applications we examined. ReAssert would simply replace the *result* of the operation, rather than the appropriate *operands*.

3.2 Expected Object Comparison

Many assertions compare against primitive values (e.g., integers) or strings. The rest compare against more complex objects. The latter assertions are more difficult to repair because the value cannot be written directly as a literal into the code as in the previous example. ReAssert overcame this limitation by changing *one* failing assertion into *many* that compared against the primitive values returned from the actual object's accessors.

For example, the JFreeChart application from our evaluation asserted against the default font used in an object's label:

```
ValueMarker m = new ValueMarker(...);
Font expected = new Font("SansSerif", Font.PLAIN, 9);
assertEquals(expected, m.getLabelFont());
```

The application evolved such that the default font changed from `"SansSerif"` to `"Tahoma"`. This caused the test to fail. ReAssert repaired the failure by expanding the failing assertion into several new assertions that passed:

```
ValueMarker m = new ValueMarker(...);
Font expected = new Font("SansSerif", Font.PLAIN, 9);
// ReAssert repair
{
```

```
  Font actual = m.getLabelFont();
  assertEquals("Tahoma", actual.getName());
  assertEquals(expected.getSize(), actual.getSize());
  assertEquals(expected.isPlain(), actual.isPlain());
  ... // a dozen more assertions
}
```

This repaired test passed because the first new assertion verified that the font had the correct name. The subsequent two assertions showed that the expected and actual size and style were the same. There were a dozen more assertions. While these assertions made the program evolution explicit, they could become very verbose with deeply-nested recursive objects. A more useful repair would simply replace the literal value in the `Font` constructor:

```
ValueMarker m = new ValueMarker(...);
// literal replacement
Font expected = new Font("Tahoma", Font.PLAIN, 9);
assertEquals(expected, m.getLabelFont());
```

This is one example of a common pattern in which ReAssert should have changed an expected object's initialization rather than a failing assertion.

3.3 Conditional Expected Value

ReAssert could trace an expected value from its use in a failing assertion back to its *declaration* (not definition). However, this process required that there be only one possible declaration-use path with a single expected value. If an expected value for a particular failure depended on a condition or had multiple definitions, then ReAssert could not determine which value(s) to repair.

Several failing tests in our subject applications declared different expected values based on an external system property. The following example shows a simplified test for an XML serialization library that serialized collections differently based on the version of an external library:

```
String expected;
if (LIB.is15) {
  expected = "<coll><string>value</string></coll>";
} else {
  expected = "<list><string>value</string></list>";
}
List list = ...;
list.add("value");
assertEquals(expected, toXml(list));
```

The library evolved to use different XML tags, causing this test and several others to fail. Since the `expected` variable could take one of two values, ReAssert only suggested to replace the expected side of the failing assertion with a literal value that made the test pass:

```
// ReAssert repair
assertEquals(
  "<c><string>value</string></c>",
  toXml(list));
```

In the actual test suite, doing so would have actually caused other tests to fail, since the assertion was located in a helper method called from multiple places. A better repair would replace the value in the appropriate branch of the conditional:

```
if (LIB.is15) {
  // literal replacement
  expected = "<c><string>value</string></c>";
  ...
```

4. SYMBOLIC TEST REPAIR

The examples in the previous section illustrate failures that ReAssert should have repaired in a better way. In each case, ReAssert could not determine how literal values in test code should change to make the test pass. Symbolic execution can overcome this deficiency. We propose a technique that brings the benefits of symbolic execution to test repair and a tool like ReAssert. We refer to the hybrid technique as *symbolic test repair*.

Symbolic test repair could be implemented as a repair strategy that would integrate into ReAssert's normal repair process (Section 2.1.1). Like other strategies, it would have access to the parse tree of the failing test, the location of the failure (derived from the stack trace), and the concrete values that caused the failure. It would in the end output a modified parse tree.

The strategy would have its own five-step repair process distinct from ReAssert's:

1. Use the stack trace to find the location of the failing assertion.
2. Analyze the source code to determine the "expected side" of the assertion. That is, determine the expression(s) that represent the expected result(s) of the behavior being tested.
3. Create symbolic values for all literals that contribute only to the calculation of the expected side. Fail if none can be found. Do not change any values that contribute to the actual side, since doing so would alter the behavior of the SUT, violating the requirements listed in Section 2.1.
4. Execute the test symbolically and solve accumulated constraints. Fail if no solution can be found within a given time limit.
5. Replace appropriate literals in source code.

To illustrate this process, we revisit the example given in Section 3.3. We emulated parts of this process manually in the evaluation described in Section 5. We discuss here several challenges that one must address when implementing a symbolic test repair tool.

4.1 Example Revisited

Consider the example given in Section 3.3. The assertion on the last line fails because the expected string does not match the value returned from the SUT. The two XML strings potentially contribute to the calculation of the expected value. Only these values could change when the test is repaired; changing the list contents would change the behavior of the SUT. The repair strategy makes the two expected-side literals symbolic using Pex's choice generators (Section 2.2.1):

```
String expected;
if (LIB.is15) {
  // replace literal with symbolic value
  expected = ChooseValue<string>();
} else {
  // replace literal with symbolic value
  expected = ChooseValue<string>();
}
List list = ...;
list.add("value");
assertEquals(expected, toXml(list));
```

Then, the strategy executes the test under Pex. Pex executes the condition concretely and encounters the first symbolic literal. Pex sets its value to an empty string and continues execution. The `assertEquals` fails because the empty string does not equal the value returned from `toXml`. The failure introduces the constraint that the symbolic value is not equal to the returned string. On the next execution, Pex inverts the condition and sets the expected value to the correct string, causing the test to pass. ReAssert would then write the string into the original test code. The alternate expected value would not change because Pex's execution never encountered it:

```
String expected;
if (LIB.is15) {
  // symbolic test repair
  expected = "<c><string>value</string></c>";
} else {
  expected = "<list><string>value</string></list>";
}
List list = ...;
list.add("value");
assertEquals(expected, toXml(list));
```

4.2 Challenges

The high-level symbolic test repair process described above ignores several important points that would impact tool implementation. We encountered each of the following issues in our evaluation.

Identify Expected Computation: In the example, the left argument of the `assertEquals` represents the expected result, and the right represents the actual value. Thus, it was easy to determine the "expected side" of the computation. Other tests and assertions lack this well-known convention, making it difficult to determine automatically what computation should be repaired.

In our evaluation, we manually determined the expected side using the context of the test and simple heuristics. Generally, we found that arguments to methods used in an assertion represented expected values. The methods themselves represented the behavior being tested.

For example, the AdblockIE application from our evaluation in Section 5 contains many assertions that verify that certain strings match a given pattern:

```
Glob target = new Glob("*eggs");
Assert.IsTrue(target.IsMatch("hamandeggs"));
Assert.IsTrue(target.IsMatch("eggs"));
Assert.IsFalse(target.IsMatch("hamandeggsandbacon"));
```

In this example, the initialization of the `target` variable, the `Glob` constructor, and the `IsMatch` methods represent the actual computation. Their values should *not* change when the test is repaired. Only the three strings used as arguments to `IsMatch` represent the expected computation that can change.

Find Expected Literals: After identifying the expected computation, a tool must accurately find all literals that contribute to *only* that side of the computation. Doing so can be a challenge since some literals may contribute to both the expected and actual sides. In our evaluation, we manually differentiated the values. A semi automated tool could ask the user to point to the values to be changed. A fully automatic tool, however, could solve this problem as an instance of program slicing [48]. It would compute dynamic backward slices from both the expected and actual side of a failing assertion. Then, only those literals that appear in the slice for the expected side but do not appear in the slice for the actual side would be made symbolic.

Choosing the "Correct" Solution: There is rarely just one solution that would cause a test to pass. Many literals may satisfy the constraints encountered when executing a test symbolically. Our evaluation is only concerned with whether any one such solution exists. However, an actual tool would need to choose which solution(s) to present to the user. Pex produces simpler solutions earlier in its exploration, and we found that the first set of satisfying values was often the best.

Multiple Failures: Many broken tests fail for multiple reasons. Therefore, it is important that a symbolic test repair tool be able to handle multiple failures in a single test. Multiple failures make it difficult for the symbolic execution engine to know when the conditions for a particular failure are satisfied. In our manual evaluation, we simply added symbolic literals for each failure that the symbolic exploration encountered. An actual tool would need to iteratively explore failures, as ReAssert does [15].

5. EVALUATION

To evaluate our proposed symbolic test repair technique, we measure how often literal replacement can repair tests and how well symbolic execution can discover those literals. Specifically, we ask the following research questions:

Q1: How many failures can be repaired by replacing literals in test code? That is, if we had an *ideal* way to discover literals, how many broken tests could we repair?

Q2: How do literal replacement and ReAssert compare? How would an ideal literal replacement strategy affect ReAssert's ability to repair broken tests?

Q3: How well can existing symbolic execution discover appropriate literals? Can symbolic execution produce literals that would cause a test to pass?

To answer these questions, we recreate experiments from the evaluation of ReAssert [14] and perform several new experiments. First, we consider repair of test failures in open-source applications caused by real and potential program evolution (Section 5.2). Then, we revisit results from the user study done with ReAssert (Section 5.3). Along the way, we evaluate Pex's ability to discover literals that would make failing tests to pass. Finally, we also evaluate Pex's ability to discover the correct literals in passing tests (Section 5.4).

5.1 Experimental Setup

Figure 1 lists the subject applications used in our experiments. The Java applications are drawn from our original evaluation of ReAssert. We chose them because several had been used in other studies [13,40,57] and all evolved in ways that would cause tests to fail. We found the .NET applications by searching public code repositories for applications with unit tests. Of those listed, six evolved in ways that would cause unit tests to fail. We use the remainder in our potential evolution and literal recreation experiments.

All experiments dealt with standard unit tests, each of which verified the behavior of a small component of the subject application. We did not consider larger system or integration tests. The unit tests for the Java applications all

Java

Application		Version(s)	Description	Tests
Checkstyle	checkstyle.sourceforge.net	3.0, 3.5	Code style checker	143
JDepend	clarkware.com/software/JDepend.html	2.8, 2.9	Design quality metrics	53
JFreeChart	jfree.org/jfreechart/	1.0.7, 1.0.13	Chart creator	1696
Lucene	lucene.apache.org	2.2.0, 2.4.1	Text search engine	663
PMD	pmd.sourceforge.net	2.0, 2.3	Java program analysis	448
XStream	xstream.codehaus.org	1.2, 1.3.1	XML serialization	726

.NET

Application		Version(s)	Description	Tests
AdblockIE	adblockie.codeplex.com	18785	Ad blocker for Internet Explorer	6
CSHgCmd	bitbucket.org/kuy/cshgcmd/	99	C# interface to Mercurial	16
Fudge-CSharp	github.com/FudgeMsg/Fudge-CSharp/	8e3654, 85952	Binary message encoding	73
GCalExchangeSync	code.google.com/p/google-calendar-connectors/	6, 7	Google Calendars and Exchange Server interoperability	33
Json.NET	json.codeplex.com	35127, 44845	JSON serialization	673
MarkdownSharp	code.google.com/p/markdownsharp/	116	Convert structured text to HTML	48
NerdDinner	nerddinner.codeplex.com	1.0	Lunch planning website	68
NGChart	code.google.com/p/ngchart/	0.4.0.0, 0.6.0.0	Wrapper for Google Charts API	25
NHaml	code.google.com/p/nhaml/	300, 446	XHTML template system	53
ProjectPilot	code.google.com/p/projectpilot/	446, 517	Source code statistics and metrics	103
SharpMap	sharpmap.codeplex.com	0.9	Geospatial mapping	49

Figure 1: Subject applications

used JUnit, while the .NET tests used one of several similar frameworks. These we converted to instead run under Pex.

We ran Pex version 0.91.50418.0 in Microsoft Visual Studio 2009 on a dual-processor 1.8Ghz laptop. We used the default Pex configuration and limited Pex's running time to 60 seconds. We believe that this limit is appropriate for an interactive test repair tool. If Pex reached other limits, such as execution depth or number of branch conditions, we increased the appropriate configuration setting until Pex met the time limit.

Links to subject applications, test code, software tools, and all experimental data are publicly available from `http://mir.cs.illinois.edu/reassert/`.

5.2 Program Evolution

We address all three of our research questions by mimicking the evolution of real open-source applications. In the ReAssert paper [14], we obtained *version difference failures* (VDFs) by executing the test suite from an older version of an open-source Java application on a newer version of its SUT. Here we do the same with .NET applications and additionally introduce *potential program evolutions* that also cause failures.

5.2.1 Version Difference Failures

Our goal with version difference failures is to reconstruct a test failure as the developer would encounter it while evolving the SUT from the older version to the newer version. For both Java and .NET subject applications, obtaining VDFs requires several steps.

First, we choose two application versions, either nearby minor releases or revisions that the version control log indicates would contain changes that would cause tests to fail. Second, we download and compile both versions, each of which has a SUT and a test suite. Third, we apply the test suite from the older version on the newer version of the SUT. This step creates several challenges. The code could have evolved such that the old tests fail to compile against the new SUT. Often such changes are due to refactorings, which change the SUT and tests without changing their semantics (e.g., renaming a method or a class). Fourth, we reverse-engineer the refactorings that occurred in the SUT and apply them manually on the tests. Fifth, we remove any remaining tests that fail to compile. Finally, we ensure that we have actual test failures due to semantic changes in the SUT, much like what the developer would observe. The goal of test repair is to repair these failures.

We produce VDFs for all of the Java applications and the top six .NET applications listed in Figure 2.

5.2.2 Potential Program Evolution

The .NET applications from Figure 1 have fewer unit tests than the Java applications. Thus, the .NET applications have understandably fewer VDFs. To make up for this difference, we produce more failures by considering reasonable evolutions in two .NET applications that lacked VDFs: MarkdownSharp and AdblockIE.

We did not define "reasonable" ourselves but instead asked 15 colleagues to provide "suggestions of reasonable software changes" in these simple .NET applications. Respondents did not need to download or run the applications, but they could browse the source code online. We asked them to avoid "refactorings and compilation-breaking changes" and instead suggest "modifications that change the current behavior of the system". We implemented nine solutions that appeared feasible, and four of them caused tests to fail. Figure 2 shows that a total of 17 tests failed in these two applications.

5.2.3 Counting Repairs

Using our two sets of failing tests—one in Java from ReAssert's evaluation and the other in .NET from this work—we determine the number of tests that could be repaired with literal replacement, ReAssert, and Pex. For literal replacement, we manually count how many failures could be repaired by changing only the literal(s) in the test. We use our judgment of the program evolution and our reasoning about how a particular test behaves to deduce which literal(s) to replace and with what value(s) to make the test

Java

Application	Failures	ReAssert	Lit. Repl.
Checkstyle	34	9 (26%)	12 (35%)
JDepend	6	6 (100%)	4 (66%)
JFreeChart	15*	15 (100%)	11 (61%)
Lucene	47	12 (25%)	7 (15%)
PMD	5	5 (100%)	2 (40%)
XStream	60	28 (47%)	51 (85%)
Total	167	75 (45%)	87 (52%)

*Originally reported as 18 in [14], but recent inspection revealed 3 spurious failures

.NET

Application	Failures	ReAssert	Lit. Repl.	Pex
Fudge-C#	2	1 (50%)	1 (50%)	1 (50%)
GCalExSync	9	8 (89%)	4 (44%)	1 (11%)
Json.NET	14	7 (50%)	6 (43%)	4 (29%)
NGChart	1	1 (100%)	1 (100%)	1 (100%)
NHaml	9	6 (67%)	4 (44%)	4 (44%)
ProjectPilot	16	2 (13%)	10 (63%)	0
AdblockIE	3	3 (100%)	3 (100%)	1 (33%)
Markdown#	14	8 (57%)	7 (50%)	7 (50%)
Total	68	36 (53%)	36 (53%)	19 (28%)

Figure 2: Failures in open-source applications that ReAssert, literal replacement, and Pex can repair

pass. For ReAssert, we use the actual results from our ReAssert tool on the Java programs, and we determine the number of .NET failures that a ReAssert-like tool would be able to repair. Since ReAssert does not act on .NET code, we use our knowledge of the tool to determine whether it could repair a particular .NET failure. For Pex, we use the actual tool to produce literal(s) that make each test pass.

5.2.4 Analysis of Results

Figure 2 summarizes the results of these experiments. The "Lit. Repl." column shows how many failures literal replacement can repair. This provides an answer to our first research question:

A1: Replacing literals in test code can repair about half of all failures (52% in Java and 53% in .NET).

For the second research question, we compare the number of failures repairable by literal replacement and ReAssert. Comparing the total numbers from Figure 2, we see that the number is somewhat smaller for ReAssert in Java (45% vs. 52%), and the numbers are same (53%) in .NET. We also see that the totals are affected by a few applications whose failures require literal replacement: Checkstyle and XStream in Java and ProjectPilot in .NET. Indeed, XStream is the application from which we drew the example used in sections 3.3 and 4.1. Recall from those sections that ReAssert can repair some failures that require literal replacement, but ReAssert cannot repair all such failures and sometimes provides less than desirable repairs.

Figure 3 further shows how literal replacement and ReAssert complement/overlap. We can see that the overall number of failures that ReAssert, literal replacement, or both could repair is 66% (51+24+36 of 167) in Java (an improvement of 22% over ReAssert) and 65% (28+8+8 of 68) in .NET (an improvement of 12% over ReAssert). When both ReAssert and literal replacement could repair a failure, our examples in Section 3 show that literal replacement can often produce more realistic repairs. Together, these results provide an answer to our second question:

Java

		Literal Replacement	
		Repairable	¬Repairable
ReAssert	Repaired	51 (31%)	24 (14%)
	¬Repaired	36 (22%)	56 (34%)

.NET

		Literal Replacement	
		Repairable	¬Repairable
ReAssert	Repairable	28 (41%)	8 (12%)
	¬Repairable	8 (12%)	24 (35%)

Figure 3: How failures in Java and .NET applications would be repaired

A2: Literal replacement would provide benefit over ReAssert, both quantitatively (reducing the number of unrepairable failures from 92 to 56 in Java and from 32 to 24 in .NET) and qualitatively (providing better repairs when both can repair).

Note the surprising similarity between the Java and .NET results. Even though we found fewer failures in .NET applications, the fractions of failures repairable by ReAssert, literal replacement, both, or neither were very similar. We did not expect or notice this similarity prior to performing all the experiments. While the similarity may be simply a consequence of the particular artifacts we used in our study, we speculate that the similarity may be actually more general and due to common testing practices (e.g., JUnit vs. NUnit) and object-oriented language design for code and tests in Java and .NET.

For the third research question, we consider the number of failures that Pex can repair. The last column of Figure 2 for .NET shows that Pex repairs about half as many failures as ideal literal replacement (19 of 36). In all 19 cases, ReAssert could have repaired the failure without symbolic execution but in a less desirable way. We inspected why Pex could not repair the remaining 17. The reasons were many and varied, but generally Pex had difficulty (1) creating long strings for parsing, (2) solving constraints involving floating-point values, and (3) finding valid keys for indexing into data structures. Even so, these results are quite impressive for the complex, real-world code used in our experiments. We have passed our findings along to the Pex developers.

To estimate how well symbolic execution would work for Java, we also convert 12 of the "hardest" Java VDFs to .NET. We translated both the test code and parts of the SUT from Java to C#. This is a semi automated translation; we used a source-to-source conversion tool called Sharpen [16] and manually fixed the constructs that it did not handle properly. ReAssert did not repair any of these 12 failures on its own, but Pex successfully repairs 75% of them (9 of 12). In summary, these results provide an answer to our third question:

A3: Powerful symbolic execution tools such as Pex can repair over half of all the failures that can be repaired with literal replacement. Symbolic test repair would thus improve on ReAssert.

5.3 User Study

In the original ReAssert work, we performed a user study to evaluate how ReAssert assists developers in writing and maintaining unit tests [15]. We asked participants to per-

form several simple development tasks in a small but realistic shopping cart application. Tasks required users to augment an existing test suite, implement changes designed to break the tests, and then repair the failures.

Our study had 18 participants (13 graduate students, 3 undergraduate students, and 2 industry professionals) with varying amounts of expertise and experience with Java, JUnit, and Eclipse. We randomly divided the participants into two groups: a control group repaired the broken tests manually, and the experimental group used ReAssert. After the study, we measured how many tests either group could repair and how many of the repairs matched (modulo source code formatting) ReAssert's suggestions. In the control group, the number of matches showed the number of times ReAssert's suggestions were identical to developers' actual repairs. In the experimental group, the number of matches showed the number of times participants accepted ReAssert's suggestions without modification. Both indicated how useful were ReAssert's suggestions to the developers. We found that ReAssert could repair 97% (131 of 135) of failures and 87% (113 of 131) matched. Three of the four unrepaired failures were repairable with a newer version of ReAssert, and the remaining failure cannot be repaired regardless of the tool used because it tests non-deterministic behavior. The 18 non-matching repairs were split across four participants.

Here we revisit these results in the context of symbolic test repair, which also provides additional data to address our second and third research questions. Since ReAssert can already repair nearly all of the failures, we assess whether symbolic test repair can increase the number of repairs that match developers' intentions.

We perform the following experiment:

1. Convert the non-matching tests from Java to C# using Sharpen with manual patching of incorrect translation.
2. Apply the tests to our reference implementation of the SUT.
3. Run the tests and repair any failures using a procedure similar to that described in Section 4.
4. Compare Pex's result to the participant's actual repair using the matching criteria from our earlier evaluation.

We find that Pex repairs all of the 18 failures that ReAssert also repaired but did not match. Moreover, three of Pex's repairs exactly match developers' repairs, and we feel three more match the developers' intentions although the source does not match exactly. Five of these six "better" repairs involve string concatenation as in the example from Section 3.1. The remaining one replaces literals in assertions of the following form: `assertTrue(cart.getTotal() == 6.0)`. ReAssert could repair this form of assertion but only after converting it to use `assertEquals`. In total, two of the four participants would have benefited from symbolic test repair. In terms of our research questions, we find that literal replacement would produce better repair than ReAssert for these cases, and we also find that in these cases Pex can repair all failures that require literal replacement.

5.4 Recreate Literals

Our third question asks whether symbolic execution—in particular Pex—can discover literals that cause a test to pass. The previous experiments evaluate this question using failing tests (from real/potential evolutions of open-source code and the ReAssert's user study). Here we assess Pex's ability to rediscover appropriate values in *passing* tests. We turn certain literals into symbolic values in passing tests and ask Pex to provide concrete values. Had these literals contributed to a failure for this code version, then Pex's success would determine whether the test could have been repaired.

Application	Literals	Matching	Passing	Unsolved
AdblockIE	52	19 (37%)	31 (60%)	2 (3%)
CSHgCmd	69	48 (70%)	5 (7%)	16 (23%)
Json.NET	262	242 (92%)	0	20 (8%)
NerdDinner	84	68 (81%)	15 (18%)	1 (1%)
SharpMap	267	187 (70%)	59 (22%)	21 (8%)
Total	734	564 (77%)	110 (15%)	60 (8%)

Figure 4: Recreated Literals

We again use several .NET applications that lack VDFs. In every passing test, we make each expected-side literal symbolic. We then run Pex to solve one literal at a time. There are three possible outcomes: Pex produces the same value as in the original test; Pex produces a different value that nevertheless causes the test to pass; or Pex fails to produce a value that makes the test pass.

Figure 4 shows the results of these experiments. We found that Pex successfully discovered a matching or passing value for 92% (674 of 734) of literals. These results show that Pex can frequently discover a correct value when one is guaranteed to exist. The results in Section 5.2 show that the percentage is lower in actual failing tests, since they lack such a guarantee; Pex finds 53% (19 of 36) of literal replacement repairs, but those are just 53% (36 of 68) of all failures. In summary, these results support the previous conclusion that symbolic execution, as embodied in Pex, can find appropriate literal values for over half literal replacement cases.

6. RELATED WORK

It is well known that software evolution can cause problems: tests can break, software artifacts can become inconsistent, and bugs can appear. ReAssert and symbolic test repair address breaking tests. Researchers have proposed many other techniques to address similar problems and reduce the burden on developers. Most techniques act on the system itself to repair faults, which is a problem dual to test repair. Those techniques that act on tests either maintain the test suite or repair tests within a particular domain such as testing user interfaces or web applications.

6.1 Program Repair

Software evolution can introduce faults. Automatically repairing these faults is an important and active area of research. There are two stages to program repair: finding faults and repairing the system to remove the faults.

The first step is referred to as *fault localization* [11]. Most closely related to our work is that by Jeffrey et al. [27], which alters literal values in the SUT code to narrow down the possible location of a fault. Other techniques generally use test coverage in various ways [55], though Wang et al. [50] and Jiang and Su [28] include program context as a way to improve localization accuracy.

The second step of program repair is referred to as *automated debugging* [56]. Proposed techniques include those based on genetic algorithms [2, 26, 51], path conditions [24], and backward data flow analysis [43, 48].

Sometimes failures appear not due to faults in the program but by incorrect component integration or environment. In these cases, more extensive repairs are necessary. Chang et al. [9] address the problem of integration faults in the context of component-based software. They provide several "healing connectors" conceptually similar to ReAssert's repair strategies. When failures appear due to incorrect execution environment, Tallam et al. [45] propose a method for patching future executions based on the event log of the failing execution.

6.2 Domain-Specific Test Repair

As far as we are aware, ReAssert was the first general purpose test repair tool for unit tests. However, evolving software often causes test failures in specific application domains, prompting the need for domain-specific test maintenance tools. In particular, several researchers have proposed repair techniques tailored to graphical user interfaces (GUIs) and web applications.

When testing GUIs, it is common for developers to create test scripts using record-and-replay testing tools [25]. Such scripts are exceptionally fragile. To address this problem, Memon et al. [34] and Grechanik et al. [21] have both proposed techniques to bring GUI test scripts back in sync with the tested application.

In the context of web applications, Harman and Alshahwan [22] test an underlying application with user session data. When the application evolves, it can invalidate the data. They propose an automatic technique that detects the changes to the underlying system and suggests repairs to the session data.

Other researchers address test evolution not by repairing individual tests but by maintaining the larger test suite. Wloka et al. [52], for example, create new unit tests that specifically exercise changes to the SUT. When software evolution invalidates existing tests such that they no longer exercise any useful part of the SUT, Zheng and Chen [58] and Harrold et al. [23] use *change impact analysis* to remove the obsolete tests.

6.3 Symbolic Execution

Our work applies symbolic execution to the domain of test repair and attempts to find tests that pass. Most other applications of symbolic execution take the opposite approach: they attempt to find test failures [5, 37, 39, 49]. Other researchers have applied symbolic execution to invariant detection [12, 29], security testing [19, 30, 53], string verification [54], and a host of other domains.

7. CONCLUSION

Test maintenance is an important but time-consuming task in software evolution. As code under test changes, tests may fail such that they also need to be changed. Test repair aims to automate these test changes. Our previous work proposed ReAssert, a tool for test repair. This paper showed that *symbolic test repair*, based on symbolic execution, can improve on ReAssert to repair more test failures and provide better repairs. Our experiments showed that the current state-of-the-art symbolic tools such as Pex for .NET can work very well in this domain. We hope that such a powerful tool for Java will be also available soon such that we can integrate it with ReAssert. In the future, we hope to study and improve how developers manually write, automatically generate, and evolve their automated unit tests.

Acknowledgments

We would like to thank Nikolai Tillmann for help with Pex, colleagues who suggested software changes for potential evolution, and Milos Gligoric, Vilas Jagannath, and Viktor Kuncak for valuable discussions about this work. This work was supported in part by the US National Science Foundation under Grant No. CCF-0746856.

8. REFERENCES

[1] S. Anand, C. Păsăreanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *TACAS*, 2007.

[2] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *CEC*, 2008.

[3] K. Beck. Where, oh where to test? `http://www.threeriversinstitute.org/WhereToTest.html`.

[4] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. In *ISSTA*, 2006.

[5] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX*, 2007.

[6] C2 Wiki. Deleting broken unit tests. `http://c2.com/cgi-bin/wiki?DeletingBrokenUnitTests`.

[7] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.

[9] H. Chang, L. Mariani, and M. Pezzè. In-field healing of integration problems with COTS components. In *ICSE*, 2009.

[10] L. Clarke and D. Richardson. Symbolic evaluation methods for program analysis. In *Program Flow Analysis: Theory and Applications*, chapter 9. 1981.

[11] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, 2005.

[12] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *ICSE*, 2008.

[13] B. Daniel and M. Boshernitsan. Predicting effectiveness of automatic testing tools. In *ASE*, 2008.

[14] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In *ASE*, 2009. `http://mir.cs.illinois.edu/reassert/`.

[15] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. Technical Report `http://hdl.handle.net/2142/13628`, University of Illinois at Urbana-Champaign, 2009.

[16] db4objects. Sharpen. https://developer.db4o.com/Documentation/Reference/db4o-7.12/java/reference/html/reference/sharpen.html.

[17] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008. http://research.microsoft.com/projects/z3/.

[18] P. Godefroid. Compositional dynamic test generation. In *POPL*, 2007.

[19] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, 2008.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, 2005.

[21] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *ICSE*, 2009.

[22] M. Harman and N. Alshahwan. Automated session data repair for web application regression testing. In *ICST*, 2008.

[23] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *TOSEM*, 2(3), 1993.

[24] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. In *FASE*, 2004.

[25] J. H. Hicinbothom and W. W. Zachary. A tool for automatically generating transcripts of human-computer interaction. In *HFES*, 1993.

[26] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. BugFix: A learning-based tool to assist developers in fixing bugs. In *ICPC*, 2009.

[27] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *ISSTA*, 2008.

[28] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control flow paths. In *ASE*, 2007.

[29] Y. Kannan and K. Sen. Universal symbolic execution and its application to likely data structure invariant generation. In *ISSTA*, 2008.

[30] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE*, 2009.

[31] J. C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

[32] R. Kitts. Is bad software really my fault? http://artima.com/weblogs/viewpost.jsp?thread=231225.

[33] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[34] A. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *TSE*, 18(2), 2008.

[35] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In *Software Evolution*. 2008.

[36] NoMoreHacks. Shocker: Changing my code broke my tests, a developer confesses. http://nomorehacks.wordpress.com/2009/08/18/shocker-changing-my-code-broke-my-tests-a-developer-confesses/.

[37] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.

[38] K. Rutherford. Why I broke 89 tests. http://silkandspinach.net/2009/10/18/why-i-broke-89-tests/.

[39] M. Sama, F. Raimondi, D. S. Rosenblum, and W. Emmerich. Algorithms for efficient symbolic detection of faults in context-aware applications. In *ASE Workshops*, 2008.

[40] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. Technical report, Universitaet des Saarlandes, 2009.

[41] M. Schwern. On fixing a broken test suite, step one: Break the cycle of failure. http://use.perl.org/∼schwern/journal/32782.

[42] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.

[43] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold. Fault localization and repair for Java runtime exceptions. In *ISSTA*, 2009.

[44] Stack Overflow. Program evolution and broken tests. http://stackoverflow.com/questions/2054171/.

[45] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Avoiding program failures through safe execution perturbations. In *COMPSAC*, 2008.

[46] N. Tillmann and J. de Halleux. Pex–white box test generation for .NET. In *Tests and Proofs*. 2008. http://research.microsoft.com/projects/Pex/.

[47] N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/FSE*, 2005.

[48] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[49] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA*, 2007.

[50] X. Wang, S.-C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *ICSE*, 2009.

[51] W. Weimer, T. V. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.

[52] J. Wloka, B. G. Ryder, and F. Tip. JUnitMX – A change-aware unit testing tool. In *ICSE*, 2009.

[53] R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *ISSTA*, 2008.

[54] F. Yu, T. Bultan, and O. H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *TACAS*, 2009.

[55] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *ICSE*, 2008.

[56] A. Zeller. Automated debugging: Are we close? *Computer*, 34(11), 2001.

[57] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei. Time-aware test-case prioritization using integer linear programming. In *ISSTA*, 2009.

[58] X. Zheng and M.-H. Chen. Maintaining multi-tier web applications. In *ICSM*, 2007.