TestEvol: A Tool for Analyzing Test-Suite Evolution

Leandro Sales Pinto
Politecnico di Milano, Italy
pinto@elet.polimi.it

Saurabh Sinha
IBM Research – India
saurabhsinha@in.ibm.com

Alessandro Orso
Georgia Institute of Technology, USA
orso@cc.gatech.edu

Abstract—**Test suites, just like the applications they are testing, evolve throughout their lifetime. One of the main reasons for test-suite evolution is test obsolescence: test cases cease to work because of changes in the code and must be suitably repaired. There are several reasons why it is important to achieve a thorough understanding of how test cases evolve in practice. In particular, researchers who investigate automated test repair—an increasingly active research area—can use such understanding to develop more effective repair techniques that can be successfully applied in real-world scenarios. More generally, analyzing test-suite evolution can help testers better understand how test cases are modified during maintenance and improve the test evolution process, an extremely time consuming activity for any non-trivial test suite. Unfortunately, there are no existing tools that facilitate investigation of test evolution. To tackle this problem, we developed TESTEVOL, a tool that enables the systematic study of test-suite evolution for Java programs and JUnit test cases. This demonstration presents TESTEVOL and illustrates its usefulness and practical applicability by showing how TESTEVOL can be successfully used on real-world software and test suites.**
Demo video at http://www.cc.gatech.edu/~orso/software/testevol/

I. MOTIVATION AND OVERVIEW

Test suites are not static entities: they constantly evolve along with the applications they test. For instance, new tests can be added to test new functionality, and existing tests can be refactored, repaired, or deleted. Often, test cases have to be modified because changes in the application break them. In these cases, if the broken test covers a valid functionality, it should ideally be repaired. Alternatively, if the repair is unduly complex to perform, or if the test was designed to cover a functionality that no longer exists in the application, the test should be removed from the test suite. To illustrate with an example, Figure I shows two versions of a unit test case from the test suite of PMD, one of the programs we analyzed in previous work [8]. A change in PMD's API broke the original version of the test case, which had to be fixed by adding a call to method `SourceCode.readSource` and removing one parameter from the call to method `Tokenizer.tokenize` (lines 6 and 7 in Figure I(b), respectively).

Because test repair can be an expensive activity, automating it—even if only partially—could save a considerable amount of resources during maintenance. This is the motivation behind the development of automated test-repair techniques, such as the ones targeted at unit test cases [3, 2, 7] and those focused on GUI (or system) test cases [1, 4, 5, 6].

We believe that, to develop effective techniques for assisting manual test repair, we must first understand how test suites evolve in practice. That is, we must understand when and how tests are created, removed, and modified. Such understanding can (1) provide evidence that test cases do get repaired, (2) support the hypothesis that test repairs can be (at least partially) automated, and (3) suitably guide research efforts. Otherwise, we risk to develop techniques that may not be generally applicable and may not perform the kind of repairs that are actually needed in real-world software systems.

More generally, the ability to analyze how test suites evolve can be beneficial for developers, as it can help them better understand the cost and tradeoffs of test evolution (*e.g.,* how often tests must be adapted because of changes in the system's API). Similarly, project managers can use information about test-suite evolution to get insight into the test maintenance process (*e.g.,* how often tests are deleted and cause loss of coverage) and ultimately improve it.

Until recently, there were no empirical studies in the literature that investigated how unit test suites evolve, and no tools that could support such studies existed. To address this issue, we defined an approach that combines various static- and dynamic-analysis techniques to (1) compute the differences between the test suites associated with two versions of a program and (2) categorize such changes along two dimensions: the static differences between the tests in the two test suites and the behavioral differences between such tests [8]. By applying our approach to a set of real-world programs, we were able to discover several important aspects of test evolution. For example, we found evidence that, although test repairs are a relatively small fraction of the activities performed during test evolution, they are indeed relevant. We also found that repair techniques that just focus on oracles (*i.e.,* assertions) are likely to be inapplicable in many cases, that test cases are rarely removed because they are difficult to fix, but rather because they have become obsolete, and that test cases are not only added to check bug fixes and test new functionality, as expected, but also to validate modified code.

This demonstration presents TESTEVOL, a tool that implements our technique and enables other researchers and practitioners to perform additional studies on test evolution. Given two versions of a program and the corresponding test suites, TESTEVOL automatically computes and classifies both static and behavioral differences between the test suites for the two program versions. Specifically, TESTEVOL identifies deleted, added, and repaired test cases, along with the effects that such deletions, additions, and repairs have on code coverage.

After presenting TESTEVOL's features and technical details, the demonstration shows how the tool can be used to study

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
Formal Demonstrations

```
1  public void testDiscardSemicolons() throws Throwable {
2      Tokenizer t = new JavaTokenizer();
3      SourceCode sourceCode = new SourceCode("1");
4      String data = "public class Foo {private int x;}";
5      Tokens tokens = new Tokens();
6      t.tokenize(sourceCode, tokens, new StringReader(data));
   // Broken statement
7      assertEquals(9, tokens.size());
8  }
```

(a)

```
1  public void testDiscardSemicolons() throws Throwable {
2      Tokenizer t = new JavaTokenizer();
3      SourceCode sourceCode = new SourceCode("1");
4      String data = "public class Foo {private int x;}";
5      Tokens tokens = new Tokens();
6      sourceCode.readSource(new StringReader(data));   // Added statement
7      t.tokenize(sourceCode, tokens);                   // Modified statement
8      assertEquals(9, tokens.size());
9  }
```

(b)

Fig. 1. Two versions of a test case from PMD's unit test suite: (a) version 1.4, broken, and (b) version 1.6, repaired.

the evolution, over the years, of a software project's test suite. To do so, we show examples of application of TestEvol to several real-world open-source software systems. In particular, we show how to use TestEvol to investigate relevant questions on test evolution, such as what types of test-suite changes occur in practice and with what frequency, how often test repairs require complex modifications of the tests, and why tests are deleted and added. We demonstrate how TestEvol is a useful and practically applicable tool for researchers and practitioners interested in test repair (and test evolution in general), and for developers and testers who want to better understand their test maintenance process.

II. The TestEvol Technique and Tool

In this section, we first summarize our technique, implemented in the TestEvol tool, and then discuss the main characteristics of the tool implementation. A detailed description of the approach can be found in Reference [8].

Before describing the characteristics of our technique, we introduce some necessary terminology. A *system* $S = (P, T)$ consists of a program P and a test suite T. A *test suite* $T = \{t_1, t_2, \ldots, t_n\}$ consists of a set of unit test cases. $Test(P, t)$ is a function that executes test case t on program P and returns the outcome of the test execution. A *test outcome* can be of one of four types:
1) $Pass$: The execution of P against t succeeds.
2) $Fail_{CE}$: The execution of P against t fails because a class or method accessed in t does not exist in P.[1]
3) $Fail_{RE}$: The execution of P against t fails due to an uncaught runtime exception (*e.g.,* a "null pointer" exception).
4) $Fail_{AE}$: The execution of P against t fails due to an assertion violation.

We use the generic term *Fail* to refer to failures for which the distinction among different types of failures is unnecessary.

Given a system $S = (P, T)$, a modified version of S, $S' = (P', T')$, and a test case t in $T \cup T'$, there are three possible

[1]These failures can obviously be detected at compile-time. For consistency in the discussion, however, we consider such cases to be detected at runtime via "class not found" or "no such method" exceptions. In fact, TestEvol detects such failures at runtime by executing the tests compiled using the previous version of P on P.

(a) Test t exists in S and S' and is modified

$Test(P', t) = Fail \wedge Test(P', t') = Pass$	t is repaired [TestRep]
$Test(P', t) = Pass \wedge Test(P', t') = Pass$	t is refactored, updated to test a different scenario, or is made more/less discriminating [TestModNotRep]

(b) Test t is removed in S'

$Test(P', t) = Fail_{RE} \mid Fail_{AE}$	t is too difficult to fix [TestDel$_{(AE\|RE)}$]
$Test(P', t) = Fail_{CE}$	t is obsolete or is too difficult to fix [TestDel$_{(CE)}$]
$Test(P', t) = Pass$	t is redundant [TestDel$_{(P)}$]

(c) Test t' is added in S'

$Test(P, t') = Fail_{RE} \mid Fail_{AE}$	t' is added to validate a bug fix [TestAdd$_{(AE\|RE)}$]
$Test(P, t') = Fail_{CE}$	t' is added to test a new functionality or a code refactoring [TestAdd$_{(CE)}$]
$Test(P, t') = Pass$	t' is added to test an existing feature or for coverage-based augmentation [TestAdd$_{(P)}$]

Fig. 2. Scenarios considered in our approach, given two system versions $S = (P, T)$ and $S' = (P', T')$: (a) t exists in T and T' and is modified, (b) t exists in T but not in T', (c) t' exists in T' but not in T.

scenarios to consider: (1) t exists in T and in T', (2) t exists in T but not in T' (*i.e.,* t was removed from the test suite), and (3) t exists in T' but not in T (*i.e.,* t was added to the test suite). These scenarios can be further classified based on the behavior of t in S and S', as summarized in Figure 2 and discussed in the rest of this section. (Note that we assume that all tests in T pass on P and all tests in T' pass on P'.)

Test Modifications: Figure 2(a) illustrates the scenario in which t is present in the test suites for both the old and the new versions of the system. To study different cases, we consider whether t is modified (to t') and, if so, whether the behaviors of t and t' differ. For *behavioral differences* there are two cases, shown in the two rows of the table: either t fails on P' and t' passes on P' or both t and t' pass on P'.

a) Category TestRep *(Repaired Tests)* corresponds to cases where t is repaired so that, after the modifications, it passes on P'. (See Figure I and discussion in Section I.)

For this category, we wish to study the types of modifications that are made to t. A test repair may involve changing the sequence of method calls, assertions, data values, or control flow. Based on our experience, for *method-call sequence changes*, we consider five types of modifications:

1) *Method call added*: a new method call is added.
2) *Method call deleted*: an existing method call is removed.
3) *Method parameter added*: a method call is modified such that one or more new parameters are added.
4) *Method parameter deleted*: a method call is modified such that one or more existing parameters are deleted.
5) *Method parameter modified*: a method call is modified via changes in the values of its actual parameters.

A test repair may involve multiple such changes. For example, the repair shown in Figure I involves the addition of a method call (line 6) and the deletion of a method parameter (line 7). For *assertion changes*, we consider cases in which an assertion is added, an assertion is deleted, the expected value of an assertion is modified, or the assertion is modified but the expected value is unchanged. Finally, we currently group together *data-value changes* and *control-flow changes*.

The rationale underlying this classification into subcategories is that different classes of changes may require different types of repair techniques. If nothing else, at least the search strategy for candidate repairs would differ for the different classes of changes. Consider the case of method-parameter deletion, for instance, for which one could attempt a repair by simply deleting some of the actual parameters. Whether this repair would work depends on the situation. For the code in Figure I, for example, it would not work because deleting one of the parameters in the call to `tokenize()` is insufficient by itself to fix the test—a new method call (to `readSource()`) has to be added as well, for the test to work correctly.

b) Category TESTMODNOTREP *(Refactored Tests)* captures scenarios in which a test t is modified in S' even though t passes on P'.

Test Deletions: Figure 2(b) illustrates the scenario in which a test t is deleted. To study the reasons for this, we examine the behavior of t on the new program version P' and consider three types of behaviors.

a) Category TESTDEL$_{(AE|RE)}$ *(Hard-To-Fix Tests)* includes tests that fail on P' with a runtime exception or an assertion violation. These may be instances where the tests should have been fixed, as the functionality that they test in P still exists in P', but the tests were discarded instead. One plausible hypothesis is that tests in this category involve repairs of undue complexity, for which the investigation of new repair techniques to aid the developer might be particularly useful.

b) Category TESTDEL$_{(CE)}$*(Obsolete Tests)* includes tests that are obsolete because of API changes, and thus fail with a compilation error on the new program version. Although for this category of deletion too, one could postulate that the tests were removed because they were too difficult to fix, we believe this not to be the case in most practical occurrences. Instead, the more likely explanation is that the tests were removed simply because the tested methods were no longer present.

c) Category TESTDEL$_{(P)}$*(Redundant Tests)* includes tests that are removed even though they pass on P'. These tests would typically be redundant according to some criterion, such as code coverage.

Test Additions: Figure 2(c) illustrates the cases of test-suite augmentation, where a new test t' is added to the test suite. The behavior of t' on the old program can indicate the reason why it might have been added.

a) Category TESTADD$_{(AE|RE)}$ *(Bug-Fix Tests)* includes added tests that fail on P with a runtime exception or an assertion violation. In this case, the functionality that t' was designed to test exists in P but is not working as expected (most likely because of a fault). The program modifications between P and P' would ostensibly have been made to fix the fault, which causes t' to pass on P'. Thus, t' is added to the test suite to validate the bug fix.

b) Category TESTADD$_{(CE)}$ *(New-Features Tests)* includes tests that fail on P with a compilation error, which indicates that the API accessed by the tests does not exist in P. Thus, the added test t' is created to test new code in P'.

c) Category TESTADD$_{(P)}$ *(Coverage-Augmentation Tests)* considers cases where the added test t' passes on P. Clearly, t' would have been a valid test in the old system as well. One would expect that the addition of t' increases program coverage. Moreover, if t' covers different statements in P and P', the plausible explanation is that t' was added to test the changes made between P and P'. However, if t' covers the same statements in both program versions, it would have been added purely to increase code coverage, and not to test added or modified code.

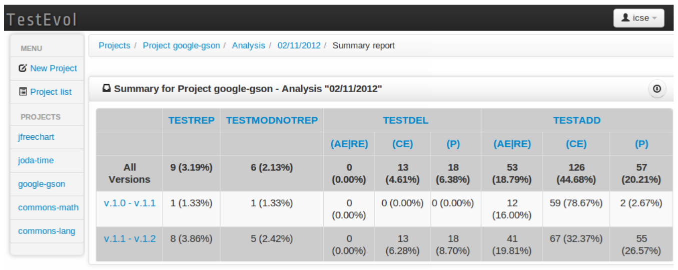
III. DETAILS OF THE IMPLEMENTATION

TESTEVOL works on Java programs and JUnit test suites (http://www.junit.org/). We chose Java and JUnit because the former is a widely used language, and the latter is the defacto standard unit-testing framework for Java. TESTEVOL analyzes a sequence of versions of a software system, where each version can be an actual release or an internal build and consists of application code and test code.

TESTEVOL is implemented as a Java-based web application that runs on Apache Tomcat (http://tomcat.apache.org/). Although the tool can be installed locally, users can also run it by accessing it remotely at http://cheetah.cc.gt.atl.ga.us:8081/testevol/ and specifying user "icse" and password "icse2013" (or requesting an account).

Due to space limitations, we can only provide a few examples of TESTEVOL's graphical interface. Figure 3 shows the summary report generated by TESTEVOL for two pairs of versions of project `google-gson`. The report shows the number of tests, both total and per version, that fall in each of the eight test-evolution categories. By clicking on a pair of versions, the user can obtain a detailed report for those two versions. To illustrate, Figure 4 shows the detailed report for versions $v.1.1$–$v.1.2$ and for a specific category of differences (TESTREP). By further clicking on a test case in the list, users can also inspect, in case the test is modified, the differences between the two versions of that test, as shown in Figure 5 for test case `TypeInfoTest`.

In terms of design, TESTEVOL consists of five components, as illustrated in Figure 6. The *compiler* component builds each system version and creates two jar files, one containing the application classes and the other containing the test classes. The *test-execution engine* analyzes each pair of system versions (S, S'), where $S = (P, T)$ and $S' = (P', T')$. First, it executes T on program P and T' on program P' (*i.e.,* it runs the tests on the corresponding program versions). Then, it executes T' on P and T on P'. For each execution, it records the test outcome: $Pass, Fail_{CE}, Fail_{AE}, or Fail_{RE}$. The *differencing component* compares T and T' to identify modified, deleted, and added tests. It matches tests between T and T' by comparing their fully qualified names and

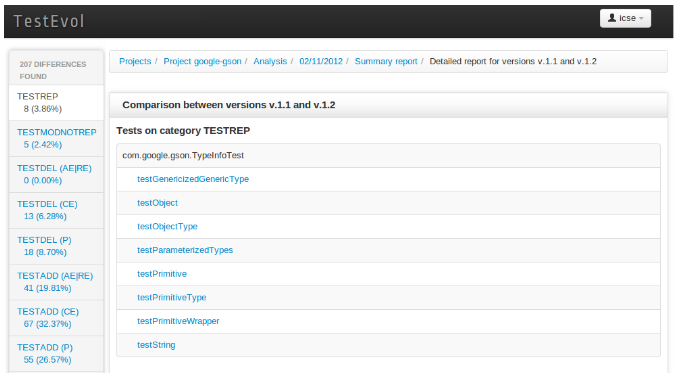


Projects / Project google-gson / Analysis / 02/11/2012 / Summary report

Summary for Project google-gson - Analysis "02/11/2012"

	TESTREP	TESTMODNOTREP	TESTDEL			TESTADD		
			(AE|RE)	(CE)	(P)	(AE|RE)	(CE)	(P)
All Versions	**9 (3.19%)**	**6 (2.13%)**	**0 (0.00%)**	**13 (4.61%)**	**18 (6.38%)**	**53 (18.79%)**	**126 (44.68%)**	**57 (20.21%)**
v.1.0 - v.1.1	1 (1.33%)	1 (1.33%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	12 (16.00%)	59 (78.67%)	2 (2.67%)
v.1.1 - v.1.2	8 (3.86%)	5 (2.42%)	0 (0.00%)	13 (6.28%)	18 (8.70%)	41 (19.81%)	67 (32.37%)	55 (26.57%)

Fig. 3. Summary report for two pairs of versions of a project.



207 DIFFERENCES FOUND
TESTREP 8 (3.86%)
TESTMODNOTREP 5 (2.42%)
TESTDEL (AE|RE) 0 (0.00%)
TESTDEL (CE) 13 (6.28%)
TESTDEL (P) 18 (8.70%)
TESTADD (AE|RE) 41 (19.81%)
TESTADD (CE) 67 (32.37%)
TESTADD (P) 55 (26.57%)

Projects / Project google-gson / Analysis / 02/11/2012 / Summary report / Detailed report for versions v.1.1 and v.1.2

Comparison between versions v.1.1 and v.1.2

Tests on category TESTREP

com.google.gson.TypeInfoTest
testGenericizedGenericType
testObject
testObjectType
testParameterizedTypes
testPrimitive
testPrimitiveType
testPrimitiveWrapper
testString

Fig. 4. Detailed report for a pair of versions.

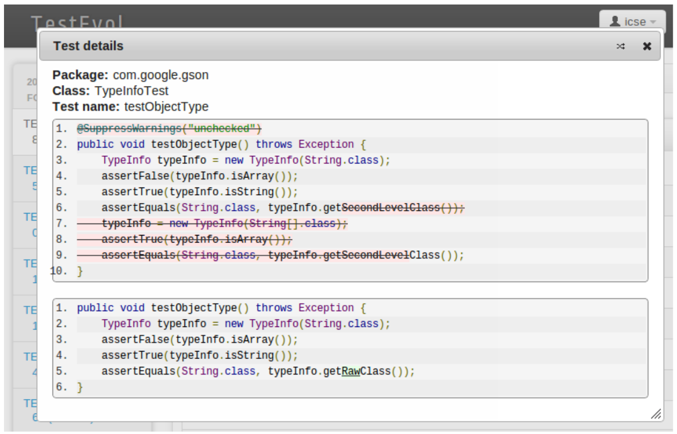
signatures. This component is implemented using the WALA analysis infrastructure for Java (http://wala.sourceforge.net).

The test outcomes, collected by the test-execution engine, and the test-suite changes, computed by the differencing component, are then passed to the *test classifier*, which analyzes the information about test outcomes and test updates to classify each update into one of the eight categories presented in Section II. For each pair, consisting of a broken test case and its repaired version, the test classifier also compares the test cases to identify the types of repair changes, as we discussed in Section II. This analysis is also implemented using WALA.

TESTEVOL performs a further step for the test cases in categories TESTDEL$_{(P)}$ and TESTADD$_{(P)}$. For these tests, the test classifier leverages the *coverage analyzer* to compute the branch coverage achieved by each test; this facilitates the investigation of whether the deleted or added tests cause any variations in the coverage achieved by the new test suite.

IV. Conclusion

Test suites evolve throughout their lifetime and change together with the applications under test. Adapting existing test cases manually can be extremely tedious, especially for large test suites, which has motivated the recent development of automated test-repair techniques. To support the development of more effective repair techniques, and to help developers and testers better understand test maintenance in general, we developed TESTEVOL. TESTEVOL is a tool for systematic investigation of test-suite evolution that can provide its users with a comprehensive understanding of how test cases evolve throughout the lifetime of a software system. This demonstration presents TESTEVOL, its main features, and its technical characteristics. It also shows how TESTEVOL can be run on real-world software systems and produce a wealth of useful information on how test-suite changes.



Package: com.google.gson
Class: TypeInfoTest
Test name: testObjectType

```
~~@SuppressWarnings("unchecked")~~
public void testObjectType() throws Exception {
    TypeInfo typeInfo = new TypeInfo(String.class);
    assertFalse(typeInfo.isArray());
    assertTrue(typeInfo.isString());
    assertEquals(String.class, typeInfo.get~~SecondLevelClass());~~
~~    typeInfo = new TypeInfo(String[].class);~~
~~    assertTrue(typeInfo.isArray());~~
~~    assertEquals(String.class, typeInfo.getSecondLevel~~Class());
}
```

```
public void testObjectType() throws Exception {
    TypeInfo typeInfo = new TypeInfo(String.class);
    assertFalse(typeInfo.isArray());
    assertTrue(typeInfo.isString());
    assertEquals(String.class, typeInfo.getRawClass());
}
```

Fig. 5. Differences between two versions of a test case.

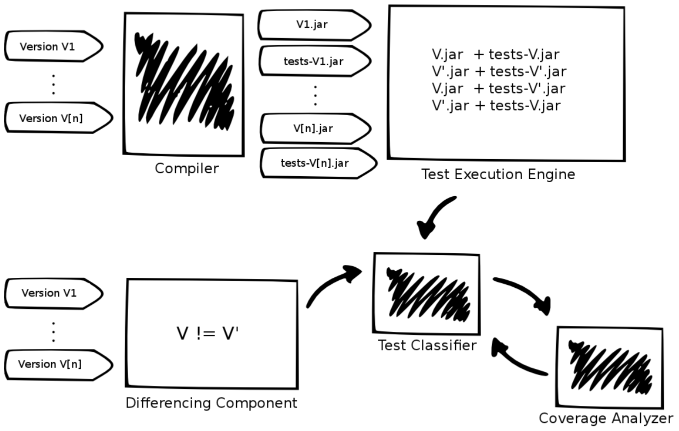


Fig. 6. High-level design of TESTEVOL.

Complete and updated information on TESTEVOL can be found at http://www.cc.gatech.edu/~orso/software/testevol/.

Acknowledgments

This work was supported in part by NSF awards CCF-1161821, CNS-1117167, and CCF-0964647 to Georgia Tech, and by funding from IBM Research and Microsoft Research.

References

[1] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. WATER: Web Application TEst Repair. In *Proc. of the First International Workshop on End-to-End Test Script Engineering*, pages 24–29, 2011.

[2] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 207–218, 2010.

[3] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In *Proc. of the International Conference on Automated Software Engineering*, pages 433–444, 2009.

[4] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *Proc. of the International Conference on Software Engineering*, pages 408–418, 2009.

[5] S. Huang, M. B. Cohen, and A. M. Memon. Repairing GUI test suites using a genetic algorithm. In *Proc. of the International Conference on Software Testing, Verification and Validation*, pages 245–254, 2010.

[6] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM TOSEM*, 18:1–36, November 2008.

[7] M. Mirzaaghaei, F. Pastore, and M. Pezzé. Supporting test suite evolution through test case adaptation. In *Proc. of the International Conference on Software Testing, Verification and Validation*, pages 231–240, 2012.

[8] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Proc. of the International Symposium on the Foundations of Software Engineering*, pages 33:1–33:11, 2012.