ReAssert: A Tool for Repairing Broken Unit Tests

Brett Daniel[1] Danny Dig[1] Tihomir Gvero[2] Vilas Jagannath[1] Johnston Jiaa[1]
Damion Mitchell[1] Jurand Nogiec[1] Shin Hwei Tan[1] Darko Marinov[1]

[1]University of Illinois at Urbana-Champaign, USA
[2]Ecole Polytechnique Fédérale de Lausanne, Switzerland
{bdaniel3,dig,vbangal2,jiaa1,mitche34,jnogiec2,stan6,marinov}@illinois.edu
tihomir.gvero@epfl.ch

ABSTRACT

Successful software systems continuously change their requirements and thus code. When this happens, some existing tests get broken because they no longer reflect the intended behavior, and thus they need to be updated. Repairing broken tests can be time-consuming and difficult.

We present ReAssert, a tool that can automatically suggest repairs for broken unit tests. Examples include replacing literal values in tests, changing assertion methods, or replacing one assertion with several. Our experiments show that ReAssert can repair many common test failures and that its suggested repairs match developers' expectations.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Unit Testing, Testing Tools, Test Repair, ReAssert

1. INTRODUCTION

Successful software systems continue to evolve long after deployment, for example in response to requirements changes, continuous hardware improvements, and better understanding of user needs. Changes incur high maintenance costs not only for the system under test (SUT) but also for the regression test suites which can often be larger than the SUT itself [9]. When requirements change and the SUT evolves, some existing tests break because they reflect the old behavior, and not the new intended behavior.

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 break. 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 practices are undesirable, anecdotal evidence suggests that they are common (as discussed elsewhere [4]). Similarly, if regression tests are generated automatically using a test generation tool [2, 8], developers often find it easier to re-generate tests than address all the failures, reducing the benefit of using the tool.

Rather than 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. However, repairing tests is tedious and time-consuming, especially when a large number of tests fail.

This paper presents our tool, ReAssert [3, 4], that suggests repairs for failing unit tests while retaining their power to detect regressions. When tests (manually written or automatically generated) fail, ReAssert can suggest changes to test code that cause the tests to pass. If the suggested repairs match the developer's intentions, then ReAssert can repair the tests with the push of a button rather than a tedious editing process.

The key challenge in repairing tests is to retain as much of the original test logic as possible. One could trivially "repair" a failing test by removing all of its code so that it passes but reveals nothing about the correct (or incorrect) behavior of the SUT. Our design requirements for ReAssert include the following criteria:

- **Make minimal changes:** Retain as much of the test code as possible and leave application code unchanged.
- **Only change if needed:** If no changes cause a test to pass, then leave the test code unchanged. ReAssert may not repair all failures, but those it does will pass.
- **Require developer approval:** Allow a developer to inspect, modify, and approve the suggested repairs.
- **Produce understandable test code:** Produce code that a developer can understand and could write manually; use normal method calls and assertions similar to any other unit test.

ReAssert is available as an Eclipse plugin or a standalone Java library and command-line tool. Both can repair unit tests written using JUnit version 3.* or 4.*. Our initial release of ReAssert was in 2009. We have recently added two extensions. First, we integrated ReAssert with Infinitest [5], an Eclipse plugin for continuous testing [7]. Second, we added the capability to repair `assertThat` kind of assertions, provided since JUnit version 4.4. ReAssert code, related experimental data and demonstration video are publicly available from `http://mir.cs.illinois.edu/reassert`.

2. ORIGINAL REASSERT TOOL

We now describe how a developer can use ReAssert to repair tests using our Eclipse plugin. ReAssert adds a "Repair with ReAssert" option to Eclipse's JUnit runner. When one or more tests fail during a test run, the developer can choose this option to get repair suggestions for the failing tests. This brings up a compare dialog which shows the original test (on the left), and the repaired test (on the right). The developer can inspect the repairs and further modify them if necessary. Pressing the "Confirm" button applies the repairs to the test code. The developer can fix one test at a time, or can fix all the failing tests at once.

If there is more than one assertion that needs repair in one test case, ReAssert suggests repairs for all assertions at once, although the first failure would normally "hide" the subsequent failures. This increases productivity, because the developer would otherwise have to revisit the subsequent failures in the same test.

We next describe how ReAssert repairs a single failure in a test. ReAssert follows a five-step process. It first instruments the test classes to record values of method arguments for failing assertions. It then re-runs the test and catches the failure exception (which contains the stack trace) and records the actual values that might make the test pass. It next traverses the stack trace to find the code to repair. It then examines the structure of the code and the recorded values to choose a repair strategy to apply to change the code. It finally recompiles the changed code.

If the test has more than one failure, ReAssert iteratively repeats these steps until the test passes, no strategies apply, or the iteration limit is reached.

2.1 Repair Strategies

Internally, ReAssert uses one of its repair strategies. It provides several general strategies and is extensible so that developers can add new strategies tailored to a particular project or test suite. Each strategy requires analysis of the dynamic execution of the test and the static code structure. We summarize the seven original ReAssert strategies [4].

Replace Assertion Method: This strategy replaces a failing assertion with a similar assertion that passes or that can be repaired by another strategy. For example, if the argument to `assertTrue` is a call to `equals`, then the strategy creates an `assertEquals` invocation, which other strategies can repair when the test is re-executed. This strategy often serves as a "preprocessing step" for other strategies.

Invert Relational Operator: This strategy inverts a relational operator in the argument to an `assertTrue` or `assertFalse` assertion. This approach works particularly well for assertions against objects that implement the `java.lang.Comparable` API.

Replace Literal in Assertion: This strategy replaces the expected (left) side of an `assertEquals` assertion with the literal value computed by the actual (right) side. It is applicable to primitive types and certain reference types such as `String` and `Class` that can be written directly into code.

Replace with Related Method: This strategy applies when the argument to `assertTrue` or `assertFalse` is a call to a common library method that is closely related to another. There are many examples of such related methods in common APIs, including `java.util.Date`'s `after` and `before`, `String`'s `contains` and `indexOf`, and `java.util.Collection`'s `isEmpty` and `size`. In each case, the assertion fails on a boolean accessor (e.g., `isEmpty`) and can be repaired by asserting against the value returned from the related method.

In the following example, a call to `isEmpty` on a non-empty collection object causes the `assertTrue` to fail.

```
assertTrue(collection.isEmpty());
```

Rather than trivially changing `assertTrue` to `assertFalse`, this strategy asserts against the actual size of the nonempty collection:

```
assertEquals(5, collection.size());
```

Trace Declaration-Use Path: It is common for developers to write helper methods that bundle several assertions for reuse across many tests. To repair such helper methods, ReAssert traces an argument used in a failing assertion back to its definition (following dynamic call chain and static declaration-use paths) and replaces the value there.

Accessor Expansion: The previous strategies mostly operate on primitive values; this strategy differs in that it repairs failures related to object inequality. It replaces a failing `assertEquals` whose arguments are reference types with a list of assertions that test values returned from both arguments' accessor methods.

Surround with Try-Catch: Occasionally developers change code to throw an exception rather than return an error value or silently fail. In these cases, the exception is expected, and a test should verify that it was thrown. This repair strategy surrounds a failing method call with a try-catch block that asserts that a particular exception is caught.

Custom Repair Strategies: ReAssert provides an extension API which can be used to instrument arbitrary methods and to define custom repair strategies. This capability allows one to repair application-specific failures or tests written in a custom test framework. An extension needs to provide a class that implements the repair strategy interface and give ReAssert the names of methods to instrument.

While the strategies implemented in ReAssert handle many common test repair scenarios, there are other scenarios [10] which are not handled by these strategies. However, there is ongoing research on this topic developing techniques for repairing other scenarios [6].

2.2 Empirical Evaluation

To evaluate ReAssert's effectiveness, we addressed the following research questions:

Q1 How many failures can ReAssert repair?

Q2 Are ReAssert's suggested repairs useful?

Q3 Does ReAssert reveal or hide regressions?

We evaluated effectiveness in three ways. First, we conducted two case studies in which other researchers used ReAssert to repair failures in their evolving software. The results show that ReAssert was able to suggest repairs for 100% of 37 failing tests, of which 78% were accepted by developers and 22% exposed regressions.

Second, we performed a controlled experiment with 18 participants to quantitatively answer all three questions. ReAssert could repair 98% (131 of 135) of failures caused by the participants' code changes, the participants accepted 86% of the suggestions, and ReAssert hid 9% of regressions. The participants also answered a survey. The majority of the participants found ReAssert useful for the user study tasks and thought that ReAssert would also be useful for their own development and testing tasks. Most would recommend ReAssert to other people and thought that it should be included as part of the Eclipse IDE.

Third, we assessed ReAssert's ability to suggest repairs for failures in six open-source projects, considering both manually written and automatically generated test suites. ReAssert was able to repair 45% (76 of 170) of failures.

Our follow-up work [3] showed that using symbolic execution [8] could repair even more failures. Recreating a set of experiments on Java applications and adding new experiments on .NET applications, we found that ReAssert and ideal literal replacement could together repair 66% (155 of 235) of failures, with literal replacement repairing 19% (44 of 235) that ReAssert could not. We also found that the Pex symbolic-execution tool [8] could solve between 53% and 92% of the cases that ideal literal replacement could solve.

3. NEW EXTENSIONS

We recently extended ReAssert with new repair strategies to handle tests that contain the `assertThat` assertions. We also integrated ReAssert with a continuous testing tool.

3.1 Handling `assertThat`

JUnit version 4.4 introduced `assertThat` assertions with the following syntax:

```
assertThat([value], [matcher statement]);
```

The second parameter of an `assertThat` statement is a `Matcher`, which allows one to define "match" rules declaratively, e.g., using the Hamcrest library [1]. This allows one to create a large combinatorial set of assertions within calls to `assertThat`, for example:

```
assertThat(aString, either(containsString("red")).
                    or(containsString("blue")));
```

Analyzing 27 randomly selected open-source projects that use `assertThat`, we found 614 uses of matchers. The most popular matchers were: `is` (376 uses), `equalTo` (141 uses), `nullValue` (49 uses), and `not` (24 uses). We added a new set of repair strategies to ReAssert to support `assertThat` and commonly used matchers.

AssertThat Repair: These strategies trigger when the failing assertion is written using `assertThat`. The strategies invoke some of ReAssert's other strategies (e.g., *Replace Literal in Assertion* or *Trace Declaration-Use Path*) and can also negate the predicate written in the matcher.

3.2 Continuous Testing Integration

Continuous testing [7] is a technique that improves developer productivity and testing efficiency by providing immediate testing feedback to developers as they make code changes. This is achieved by (1) continuously monitoring the changes made by a developer, (2) executing in the background unit tests relevant to the changes, and (3) notifying the developers about any failures due to the changes. The technique is similar in spirit to the continuous notification of compilation errors in most modern IDEs and spelling errors in text editors. In recent years, popular IDEs, including Eclipse, have suggested fixes for compilation errors. In Eclipse, these fixes are called quick fixes and allow developers to easily fix many common errors like invalid class name, missing method declaration, invalid package name, etc. Similarly, ReAssert could naturally be used to improve continuous testing by suggesting repairs for failing tests.

We integrated ReAssert with *Infinitest* [5], a continuous-testing tool for Eclipse. While using the ReAssert integration, developers are provided suggestions for how the failing tests can be fixed. Note that the suggestions are not displayed automatically, which would be very disruptive to the developers workflow. Rather, as for quick fixes, the developer has to select a failing test and request the suggestions for fixing that test (or all failing tests).

Acknowledgments

The first author passed away less than a week before the submission deadline. Brett will be missed dearly by his family, friends, and colleagues.

We would like to thank Rob Bocchino, Milos Gligoric, Bobak Hadidi, Munawar Hafiz, Viktor Kuncak, Steven Lauterburg, Yun Young Lee, Samira Tasharofi, Nikolai Tillmann, and Mohsen Vakilian for help and valuable discussions about this work. We would also like to thank anonymous colleagues who provided code evolutions and participated in our user study. This material is based upon work partially supported by the US National Science Foundation under Grant No. CCF-0746856.

4. REFERENCES

[1] Hamcrest – Library of matchers for test expressions. `http:/http://code.google.com/p/hamcrest/`.

[2] 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.

[3] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *ISSTA*, 2010.

[4] 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/`.

[5] Improving Works. Infinitest. `http://improvingworks.com/products/infinitest/`.

[6] M. Mirzaaghaei, F. Pastore, and M. Pezze. Automatically repairing test cases for evolving method declarations. In *ICSM*, 2010.

[7] D. Saff and M. D. Ernst. An experimental evaluation of continuous testing during development. In *ISSTA*, 2004.

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

[9] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. Tech report, Microsoft Research, 2005.

[10] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In *ICST*, 2008.