ReAssert: Suggesting Repairs for Broken Unit Tests*

Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov

University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {bdaniel3,vbangal2,dig,marinov}@illinois.edu

***Abstract*—Developers often change software in ways that cause tests to fail. When this occurs, developers must determine whether failures are caused by errors in the code under test or in the test code itself. In the latter case, developers must repair failing tests or remove them from the test suite. Reparing tests is time consuming but beneficial, since removing tests reduces a test suite's ability to detect regressions. Fortunately, simple program transformations can repair many failing tests automatically.**

We present ReAssert, a novel technique and tool that suggests repairs to failing tests' code which cause the tests to pass. Examples include replacing literal values in tests, changing assertion methods, or replacing one assertion with several. If the developer chooses to apply the repairs, ReAssert modifies the code automatically. Our experiments show that ReAssert can repair many common test failures and that its suggested repairs correspond to developers' expectations.

I. Introduction

Unit testing is becoming an important and widely practiced activity in software development. For example, reports from Microsoft show that 79% of developers use unit tests [2], and the code for unit tests is often larger than the application code under test [3]. Developers manually write (or automatically generate) unit tests for their application code and frequently run them while changing the code.

When changes cause existing unit tests to fail, developers should inspect the failures. There are two possible outcomes. First, if failures are caused by regressions, then developers must revise the application code to make the tests pass. Second, if failures are caused by tests that no longer reflect the intended behavior of the software, then the tests are *broken*—developers must repair the broken tests or remove them from the test suite. (In some cases, developers need to change both the application code and the test code.) Repairing broken tests is time consuming but often preferable to removing tests since removing tests reduces a test suite's ability to detect regressions that appear in later versions of the software.

The scenario described above assumes that developers first change application code, then repair any broken tests. In contrast, test-driven development advocates changing the tests (to reflect the new requirements) before changing the application code (to implement these requirements) [4], [5]. However, even with advances in change-impact analysis [6], [7], [8], developers do not always know *a priori* all tests which will be affected by a particular change, much less how a failure will manifest. Therefore, *a posteriori* repair remains the most common practice.

Unfortunately, developers may not take the time to repair broken unit tests. The problem becomes particularly acute when developers make a "deep" change that breaks many tests or when developers have a large test suite produced by an automatic test-generation tool [9], [10], [11], [12], [13], [14]. Automatically generated tests tend to be more fragile than manually written tests, e.g., dozens of automatically generated tests can fail on a seemingly simple code change. Developers are often reluctant to manually repair a large number of broken tests—particularly if they were automatically generated—opting instead to remove them.

We present *ReAssert*, a novel technique and tool 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 one mouse click rather than a tedious editing process.

One can compare ReAssert to automated refactoring tools commonly included in modern integrated development environments. Both are code transformation tools that involve well-defined sequences of structural changes [15]. Developers could perform these transformations manually, but doing so is tedious and error-prone. Just as one can perform a refactoring by selecting an item from a menu, ReAssert allows one to repair a failing test at the push of a button.

Both automated refactorings and ReAssert perform nontrivial analyses prior to transformation. ReAssert differs from refactoring tools in that its transformations require analysis of a test's runtime execution (i.e., the values that caused a test to fail) in addition to the static structure of the code. Also, ReAssert suggests repairs that change the behavior of test code (from failing to passing), while refactorings perform structural code changes but preserve code behavior.

* This technical report is an extended version of [1]

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 application code. Our design of ReAssert follows these 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 test 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.

This paper makes the following contributions.

Idea: We propose automatic repair of failed unit tests as a means toward reducing the manual effort required to update test code.

Technique: We develop a technique that can automatically suggest repairs to the test code that make failing tests pass. While we present the technique for Java, the concepts generalize to other languages. The technique combines analysis of a test's dynamic execution (to recover failing values and control flow) with analysis and transformation of the static structure of test code.

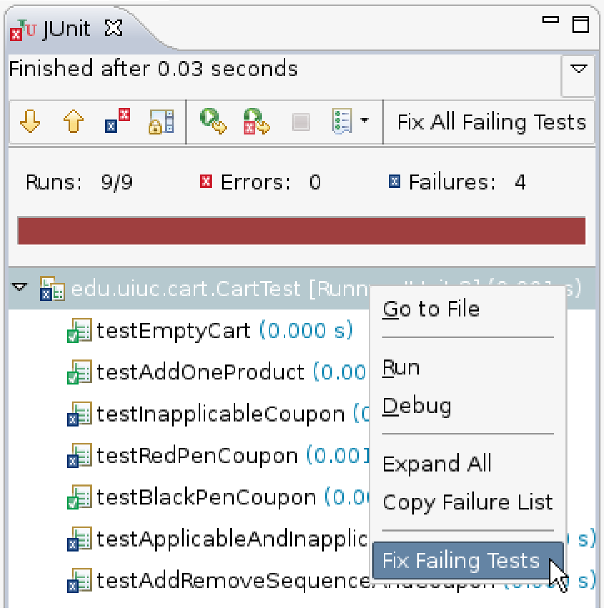
Tool: We implemented our technique in a tool called *ReAssert* that can repair Java unit tests written using the JUnit framework (http://junit.org). We also built a plugin for the Eclipse IDE (http://eclipse.org) that integrates seamlessly with the built-in test runner. It allows the developer to easily compare the failing and repaired test code and to update the code with one mouse click. ReAssert can repair many common types of test failures and is also extensible, allowing developers to write project-specific repair strategies.

Evaluation: We evaluate ReAssert's effectiveness in three ways. First, we describe two case studies in which researchers used ReAssert to repair failures in their evolving software. Second, we perform a controlled user study to evaluate whether ReAssert's suggested repairs match developers' expectations. Third, we assess ReAssert's ability to suggest repairs for failures in open-source projects, considering both manually written and automatically generated test suites, as well as failures introduced by actual software evolution and a mutation testing tool.

Our tool and experimental results are publicly available at http://mir.cs.illinois.edu/reassert.

```
public void testRedPenCoupon() {
    Cart cart = new Cart();
    cart.addProduct(new RedPen());
    cart.addProduct(new RedPen());
    cart.addCoupon(new AnniversaryCoupon());
    assertEquals(3.0, cart.getTotalPrice());
    assertEquals(
        "Discount: -$3.00, Total: $3.00",
        cart.getPrintedBill());
}
```

Figure 1. Example test



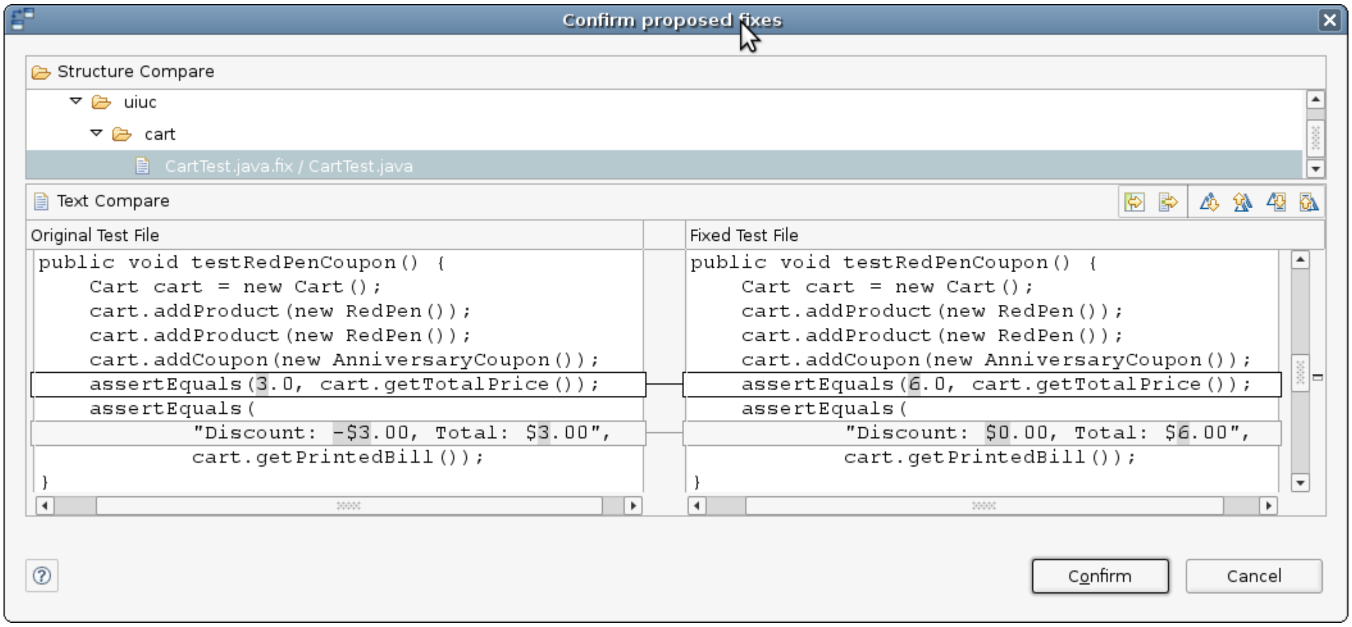
```
Finished after 0.03 seconds
Runs: 9/9    Errors: 0    Failures: 4
edu.uiuc.cart.CartTest [Runn
    testEmptyCart (0.000 s)
    testAddOneProduct (0.00
    testInapplicableCoupon (0
    testRedPenCoupon (0.001
    testBlackPenCoupon (0.0
    testApplicableAndInapplic
    testAddRemoveSequence
```

Figure 2. ReAssert's extension to Eclipse's JUnit runner

II. Example

We illustrate the use of ReAssert through a simplified example based on the shopping cart application from the user study described in detail in Section V-B. The example considers a common code evolution scenario: a developer starts from application code and passing tests, receives a change request due to a change in requirements, and has to update the code and/or tests to match the new requirements. We discuss how ReAssert helps in repairing broken (manually written or automatically generated) tests after code changes.

In our example, the application logic is in the classes `Cart` (which represents a shopping cart that can contain products and coupons), `Product` (whose subclasses represent products such as red or black pens), and `Coupon` (whose subclasses represent certain discounts that may apply). The classes provide usual methods for adding/removing products and coupons to/from the cart, computing prices, printing bills, etc. One example coupon celebrates the store's anniversary with a buy-one-get-one-free discount for any pen.



Original Test File:

```
public void testRedPenCoupon() {
    Cart cart = new Cart();
    cart.addProduct(new RedPen());
    cart.addProduct(new RedPen());
    cart.addCoupon(new AnniversaryCoupon());
    assertEquals(3.0, cart.getTotalPrice());
    assertEquals(
            "Discount: -$3.00, Total: $3.00",
            cart.getPrintedBill());
}
```

Fixed Test File:

```
public void testRedPenCoupon() {
    Cart cart = new Cart();
    cart.addProduct(new RedPen());
    cart.addProduct(new RedPen());
    cart.addCoupon(new AnniversaryCoupon());
    assertEquals(6.0, cart.getTotalPrice());
    assertEquals(
            "Discount: $0.00, Total: $6.00",
            cart.getPrintedBill());
}
```

Figure 3. The suggested repair in Eclipse

Figure 1 shows an example, manually written JUnit test for the anniversary coupon. In object-oriented code, each test creates one or more objects, invokes a sequence of methods on these objects, and *asserts* that certain properties of these objects hold. This test checks that the discount correctly applies when the cart contains two red pens, which in the example cost $3 each without the coupon. JUnit's `assertEquals` method compares the expected (first argument) value and the actual (second argument) value. This test is one of the many tests for coupons and products.

All the tests pass for the original code, but the requirements change: the anniversary coupon should now apply to black pens only. When the developer correctly changes the `AnniversaryCoupon` class, this test and several others fail because they no longer reflect the correct application logic.

Figure 2 shows the failing tests in the Eclipse's JUnit runner. ReAssert adds the "Fix Failing Tests" option to the context menu. Choosing the option brings up a compare dialog such as that shown in Figure 3. The top part lists test files with suggested repairs, and the bottom right part shows the repairs. The user can inspect the repairs and modify the code if necessary. Pressing the "Confirm" button applies the repairs to the test code.

In this example, ReAssert suggests to repair the assertions at lines 6 and 7 by changing the expected values in the assertions. Note that ReAssert suggests repairs for both assertions at once, although the failure at line 6 would normally "hide" the failure at line 7. To determine these repairs, ReAssert re-runs the test to record the actual values and then writes the recorded values back into the code.

```
public void testRedPenCoupon() {
  Cart cart = new Cart();
  cart.addProduct(new RedPen());
  cart.addProduct(new RedPen());
  cart.addCoupon(new AnniversaryCoupon());
  checkCart(cart, 3.0,
      "Discount: -$3.00, Total: $3.00");
}
protected void checkCart(
    Cart cart, double total, String bill) {
  assertEquals(total, cart.getTotalPrice());
  assertEquals(bill, cart.getPrintedBill());
}
```

Figure 4. Example test with a helper method

Internally, ReAssert applied one of its repair strategies called *Replace Literal in Assertion*. This is just one of several strategies described in Section IV. It is a simple but effective strategy that illustrates two important characteristics of all repair strategies. First, the strategy requires analysis of the dynamic execution of the test. In this case, the actual values are a double and a string, which can be written directly as literals into the code. Second, each strategy is tailored to a particular code structure. The *Replace Literal in Assertion* strategy only applies to an `assertEquals` invocation located in the test method (not a helper method). If these conditions are not met, ReAssert chooses a different strategy.

For example, Figure 4 shows another test for the same functionality, but which uses a helper method to check the price and bill. Developers commonly write such helper methods to reuse several assertions across many tests. (One of the participants in our user study wrote such a helper method for several tests.) The *Replace Literal in Assertion* strategy does not apply because changing the failing assertions directly in the helper method would break other tests that call the same helper method.

ReAssert can still suggest a repair for this test but applies another strategy, *Trace Declaration-Use Path*. This strategy traces a variable's use from an assertion back to its definition (across the dynamic call chain from the test execution) and then writes the values recorded from the test run into the definition. This strategy exploits the fact that test helpers rarely modify the expected value, so it suffices to track declarations rather than control flow. In this example, the strategy changes the arguments to `checkCart` on line 6.

```
6   checkCart(cart, 6.0,
7         "Discount: $0.00, Total: $6.00");
```

While the previous two examples have shown the use of ReAssert to repair manually written tests, ReAssert is also useful for repairing and maintaining automatically generated test suites. For example, we applied the Randoop tool [14] on the shopping cart application, and Randoop generated a regression test suite with 1,598 tests (for the default time limit for generation). The assertions in those tests are similar to the assertions in Figure 1, but tests generated by Randoop are usually longer than manually written tests. After changing the shopping cart code (Section V-B), 834 of these tests failed. We applied ReAssert, and it repaired all tests such that they passed.

III. Repair Process

ReAssert's repair process starts when the user chooses a set of failing tests to repair. For each test (which can have more than one failing assertion), ReAssert iteratively attempts repairs until the test passes, no strategies apply, or the iteration limit is reached.

To repair a single failure in a test, ReAssert follows the five-step process shown graphically in Figure 5. ReAssert first instruments the test classes to record values of method arguments for failing assertions (Section III-A). It then re-executes the test and catches the failure exception that contains both the stack trace and recorded values (Section III-B). It next traverses the stack trace to find the code to repair (Section III-C) and examines the structure of the code and the recorded values to choose a repair strategy to apply to change the code (Section III-D). It finally recompiles the code changes and repeats these steps if the test has another failure (Section III-E).

A. Instrumenting Assertions

ReAssert instruments some methods to dynamically record the runtime values that the repair strategy will use when repairing the test code. ReAssert does not instrument all methods; only those that are assertion methods and are declared in classes referenced (perhaps indirectly) by the current test. By default, JUnit's standard assertions (such as `assertEquals`) are instrumented as assertion methods, and ReAssert provides a general mechanism for instrumenting additional methods.

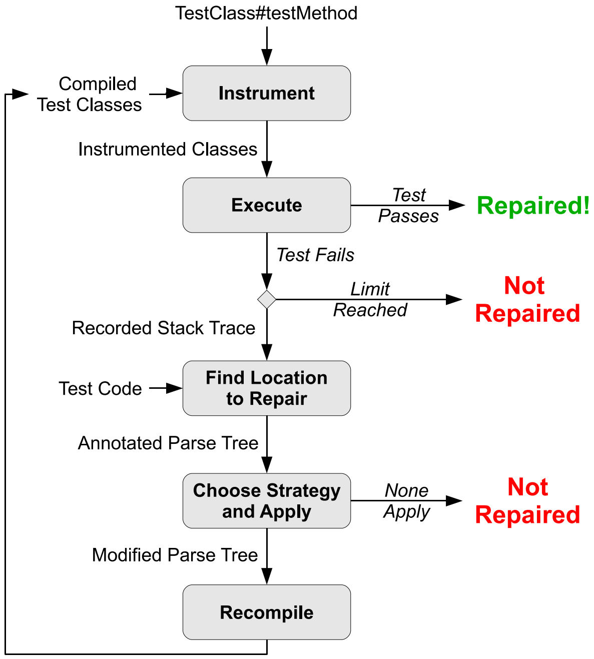


Figure 5. ReAssert's repair process

```
public static void assertEquals(
      Object expected,
      Object actual) {
    try {
      // ... original code of assertEquals
    } catch (Error e) {
      throw new RecordedAssertFailure(
          e, expected, actual);
    }
}
```

Figure 6. Instrumentation of an assertion method

The instrumentation records values by wrapping the exception thrown by a failing assertion with an exception holding the values of the assertion's arguments. Recalling the example from Figure 1, ReAssert instruments the `assertEquals` method to record the expected and actual values. Normally, `assertEquals` throws a `java.lang.AssertionError`. ReAssert instruments the method such that it catches any exception and re-throws ReAssert's own `RecordedAssertFailure`. This exception holds the original exception and the expected and actual values. Figure 6 shows conceptually how ReAssert instruments the method.

This form of instrumentation has several good characteristics. First, it re-throws an exception only on the failing assertions that ReAssert cares about. There is no global state and no runtime overhead for non-failing calls. Second, ReAssert instruments bytecode dynamically using Java's class loaders, allowing one to instrument arbitrary methods without requiring recompilation or source code (though ReAssert necessarily requires the source of *calls* to the method so it can repair them). Finally, one can record values from multiple assertions at different points on the call stack. These last two benefits enable custom repair strategies described in Section IV-H.

B. Re-Executing Tests

ReAssert uses JUnit to re-execute failing (instrumented) tests. JUnit catches the exception thrown by a failing test (likely a `RecordedAssertFailure`) and returns it to ReAssert. Even though ReAssert delegates to JUnit in our current implementation, ReAssert is effectively decoupled from a particular test framework. It could just as easily repair tests written using TestNG (http://testng.org/) or another framework. ReAssert simply requires the exception thrown by the test. This exception contains recorded values of method arguments and information about the cause of the test failure, including the stack trace.

C. Finding the Code to Repair

Repair strategies explore the stack trace of a failing test to find the static code location to repair. Java's stack traces include source code line numbers as shown in Figure 7. ReAssert parses the test code into an *annotated parse tree* [16] that contains all line numbers and type information. Thus, given a location from the stack trace, a particular strategy can easily find the appropriate code fragment.

Most repair strategies simply change the failing assertion's call site one stack frame below the instrumented method. Some strategies require a more complex analysis. For instance, the *Trace Declaration-Use Path* strategy follows the stack trace to find the location in the source code of the literal value that flowed into the expected side of a failing `assertEquals` call (if such a value exists). Figure 4 shows one test that uses this strategy, and Figure 7 shows the stack trace produced by the failure of the assertion at line 11.

The *Trace Declaration-Use Path* strategy examines the code at the failing assertion's call site (line 11) and finds that the expected side is a variable. The variable is declared as the second formal argument of the helper method `checkCart`, so the strategy retrieves the code one stack frame lower at line 6. The tracing stops because the second actual argument of the call is the literal value `3.0`, and the strategy can replace it with the recorded value. Had the argument been another variable, tracing would have continued, following the dynamic call chain from the stack trace and looking for declaration of variables. (Since test code rarely changes method arguments, it suffices to follow declarations rather than control flow.) The repair may occur many frames lower in the stack trace than the failed assertion.

```
edu.illinois.reassert.RecordedAssertFailure:
org.junit.AssertionFailedError:
expected:<3.0> but was:<6.0>
  at org.junit.Assert.assertEquals(Assert.java:116)
  at CartTest.checkCart(CartTest.java:11)
  at CartTest.testRedPenCoupon(CartTest.java:6)
  ...
  at org.junit...JUnitCore.run(JUnitCore.java:109)
Caused by: java.lang.AssertionError:
expected:<3.0> but was:<6.0>
  at org.junit.Assert.fail(Assert.java:71)
  at org.junit.Assert.failNotEquals(Assert.java:451)
  at org.junit.Assert.assertEquals(Assert.java:99)
  at org.junit.Assert.assertEquals(Assert.java:116)
  ...
```

Figure 7. Stack trace for the failure shown in Figure 4

```
public class ReplaceLiteralStrategy
      implements RepairStrategy {
  public void fix(
      Error failure,
      MethodInvocation assertion,
      Object[] recordedArgValues) {
    if (failure instanceof AssertionError
        && "assertEquals".equals(assertion.getName())
        && isLiteral(recordedArgValues[1])) {
      assertion.setArgument(0, recordedArgValues[1]);
    }
  }
}
```

Figure 8. Pseudocode for the *Replace Literal in Assertion* strategy

D. Repairing Test Code

ReAssert chooses a repair strategy by looping over all strategies to determine which one can apply to the given failure. ReAssert passes the exception, the parse tree, and the recorded values to each strategy. Each strategy checks a set of *preconditions* to determine whether it can repair the failure. The first strategy whose preconditions are satisfied updates the parse tree. If no strategies apply, then ReAssert cannot repair the failure. The next section describes individual strategies in more detail. We give here only one example.

In the test shown in Figure 1, the *Replace Literal in Assertion* strategy applies. Figure 8 shows this strategy's simplified pseudocode. Its preconditions require that the failure be an instance of `java.lang.AssertionError`, the failing method be `assertEquals`, and the recorded actual value can be written directly into the code. (There is also the implicit precondition that the expected side is not a variable, since if it was, the *Trace Declaration-Use Path* strategy would apply.) The test code in Figure 1 meets these conditions, so the strategy writes the recorded actual value into the expected side of the `assertEquals` as shown in Figure 3.

E. Iteratively Repairing Failures

Once a repair strategy completes, ReAssert recompiles the changed test code and repeats the entire repairing process until the test passes, no more strategies apply, or a limit on the maximum number of repair attempts is reached. In this

```
public void testCloning() {
    LegendItem item = ...
    assertFalse(item instanceof Cloneable);
}
```

Figure 9. The original failing test code

way, ReAssert is able to repair multiple assertions in a single test: it repairs the first, recompiles, repairs the second, and so on.

IV. Repair Strategies

ReAssert's library of repair strategies determines its ability to repair broken tests in meaningful and useful ways. ReAssert currently implements seven general repair strategies for common types of assertions and failures. Additionally, ReAssert is extensible so that developers can add new strategies tailored to a particular project or test suite. Each strategy analyzes the type of failure, the code at the point of failure, and any recorded values to determine whether its preconditions apply. Only if the preconditions are met does it transform the code.

This section lists each repair strategy, describes the analysis and transformation it performs, and provides an example from the evaluation described in Section V.

A. Replace Assertion Method (RAM)

This strategy replaces a failing assertion with a similar assertion that passes or that can be repaired by another strategy. This strategy maps one assertion method to another, taking the arguments to the assertion into account. Since the mapping is static, the strategy only analyzes the structure of the failing assertion to determine which mapping to apply. This strategy often serves as a "preprocessing step" for other strategies.

ReAssert implements the following mappings:

```
          assertTrue(x)   →   assertFalse(x)
         assertFalse(x)   →   assertTrue(x)
    assertFalse(x != y)   →   assertEquals(x, y)
      assertTrue(x == y)  ⇝   assertEquals(x, y)
assertTrue(x.equals(y))   ⇝   assertEquals(x, y)
          assertNull(x)   ⇝   assertEquals(null, x)
```

The pseudovariables `x` and `y` represent arbitrary expressions. Mappings denoted ⇝ are not fixed immediately; they require additional repairs provided by other repair strategies. For example, the *Replace Assertion Method* strategy changes `assertTrue(x == y)` to `assertEquals(x, y)`, and then delegates to the *Replace Literal in Assertion* or *Accessor Expansion* strategy.

Example: The example in Figure 9 is from the JFreeChart application. In version 1.0.7, the developers had a class that was not required to implement `Cloneable` (unlike many of the other classes in the library). By version 1.0.13, they had revised this decision, breaking a test. ReAssert repairs the failure by changing the `assertFalse` to `assertTrue`.

B. Invert Relational Operator (IR)

This strategy inverts a relational operator in the argument to an `assertTrue` or `assertFalse` assertion. It analyzes the structure of the code to determine whether the argument to the assertion is one of the four binary relational operators: <, >, <=, and >=. The *Replace Assertion Method* strategy handles failures involving the equality and disequality operators.

This strategy transforms the following assertions:

```
 assertTrue(x < y)   →   assertTrue(x >= y)
 assertTrue(x > y)   →   assertTrue(x <= y)
assertTrue(x <= y)   →   assertTrue(x > y)
assertTrue(x >= y)   →   assertTrue(x < y)
```

Example: This strategy is particularly useful for assertions against objects that implement the `java.lang.Comparable` API. For example, our mutation evaluation of TimeAndMoney version 0.5.1 mutated the return value of the `TimePoint` class' `compareTo` method. This caused the following assertion in `TimePointTest` to fail:

```
assertTrue(dec19_2003.compareTo(dec20_2003) < 0);
```

This strategy inverts the less-than operator:

```
assertTrue(dec19_2003.compareTo(dec20_2003) >= 0);
```

C. Replace Literal in Assertion (RL)

This strategy replaces the expected (left) side of an `assertEquals` assertion with the literal value computed by the actual (right) side. This strategy is applicable to primitive types and certain reference types such as `String` and `Class` that can be written directly into code. The running example in sections II and III describes this strategy in more detail.

This strategy also replaces the assertion method if necessary. For example, if the actual recorded value is found to be `null`, then this strategy repairs it with `assertNull(actual)` rather than the more verbose `assertEquals(null, actual)`.

D. Replace with Related Method (RRM)

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.

ReAssert currently implements a strategy for `isEmpty` and `size`. This strategy analyzes the `assertTrue` or `assertFalse` to determine whether its argument is an invocation of the `isEmpty` method. If so, it transforms the `assertTrue` or `assertFalse` into `assertEquals`, sets the actual argument to a `size` invocation, and sets the expected argument to an arbitrary value. The *Replace Literal in Assertion* strategy corrects the arbitrary value when the test is re-executed.

Example: This strategy was used most heavily to repair mutation failures found in Apache Commons Collections version 3.2. In many places, a call to `isEmpty` on a nonempty collection caused `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());
```

E. Accessor Expansion (AE)

This strategy applies to a failing `assertEquals` whose arguments are reference types. It replaces the failing assertion with a list of assertions that test the values returned from both arguments' accessor methods. The list of assertions is similar to what some automatic test generation [11] or test augmentation tools [13] use to produce assertions.

The motivation for this strategy lies in the fact that the `assertEquals` likely fails because one or two fields in the arguments no longer match. This strategy makes this behavior explicit by matching the actual side's accessors with the expected side's. If the two do not match, the strategy asserts against a literal value or, if the values are reference types, expands one level deeper.

The downside of this strategy is that it can create a large number of assertions. However, a developer can easily modify the suggested repair to, say, include only those assertions that test unmatched accessors. There is little danger to accepting all assertions; if they break, ReAssert can repair them.

Example: Figure 10 illustrates this behavior on a failure found when running JFreeChart. Between versions 1.0.7 and 1.0.13, the developers changed the default offsets of a category plot from 0 to 4. This change broke the test shown in Figure 10(a). ReAssert's suggested repair, shown in Figure 10(b), asserts against the literal values of the changed accessor methods in lines 6 through 9. On line 10, the expected and new actual value's `getUnitType` accessors match. Thus, ReAssert asserts that they are equal.

The actual repair performed by the JFreeChart developers, shown in Figure 10(c), is less verbose, but encodes the same changed behavior. We argue that ReAssert's suggested repair can be more powerful than the actual repair since it tests more accessors and is more likely to detect regressions. Furthermore, it encodes explicitly how a change affected a test, which is not apparent from the developers' real repair.

F. Trace Declaration-Use Path (TD)

It is common for developers to write helper methods that bundle several assertions for reuse across many tests. Repairing such helper methods requires tracing an argument used in a failing assertion back to its definition (following dynamic call chain and static declaration-use paths) and replacing the value there. Section III-C explains this process in detail.

```
1  public void testConstructor() {
2    CategoryPlot plot = new CategoryPlot();
3    assertEquals(
4      RectangleInsets.ZERO_INSETS,
5      plot.getAxisOffset());
6  }
```

(a) The original failing test code

```
1  public void testConstructor() {
2    CategoryPlot plot = new CategoryPlot();
3    {
4      RectangleInsets actual0 =
5        plot.getAxisOffset();
6      assertEquals(4.0, actual0.getBottom());
7      assertEquals(4.0, actual0.getTop());
8      assertEquals(4.0, actual0.getLeft());
9      assertEquals(4.0, actual0.getRight());
10     assertEquals(
11       RectangleInsets.ZERO_INSETS.getUnitType(),
12       actual0.getUnitType());
13     assertEquals(
14       "RectangleInsets[t=4.0,l=4.0,b=4.0,r=4.0]",
15       actual0.toString());
16   };
17 }
```

(b) ReAssert's suggested repair

```
1  public void testConstructor() {
2    CategoryPlot plot = new CategoryPlot();
3    assertEquals(
4      new RectangleInsets(4.0, 4.0, 4.0, 4.0),
5      plot.getAxisOffset());
6  }
```

(c) The actual repair

Figure 10. Example of the *Accessor Expansion* strategy applied to a failure in JFreeChart's test suite.

Example: Figure 11 shows a test and helper method found in version 1.2 of XStream. XStream is an XML serialization tool, and this test verifies that a particular object is serialized to the given string literal.

Between versions 1.2 and 1.3.1, the XStream developers changed the default ordering of the XML elements, breaking many tests like this example. The assertion at line 15 in the helper method fails, but the literal string at line 4 needed to be changed. Both ReAssert and XStream's developers repaired the test by changing the literal string to the correct value.

G. Surround with Try-Catch (STC)

Sometimes 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.

```
public void testHandlesInheritanceHierarchies() {
  OpenSourceSoftware openSourceSoftware = ...
  String xml =
    ”<oss>\n” +
    ”  <license>license</license>\n” +
    ”  <vendor>apache</vendor>\n” +
    ”  <name>geronimo</name>\n” +
    ”</oss>”;
  ...
  assertBothWays(openSourceSoftware, xml);
}
```

(a) The failing test

```
protected Object assertBothWays(
    Object root, String xml) {
  String resultXml = toXML(root);
  assertEquals(xml, resultXml);
  ...
}
```

(b) The helper method

Figure 11. Failing test from the XStream project that is repairable with the *Trace Declaration-Use Path* strategy.

Example: Figure 12 shows an example failure that this strategy can repair. The test comes from an automatically generated test suite produced for version 1.0.7 of JFreeChart. When run on version 1.0.13, the test fails because unlike in version 1.0.7, the `CategoryPlot` class in version 1.0.13 throws a `java.lang.IllegalArgumentException` when passed a negative number.

ReAssert captures the runtime type of the exception, and wraps the failing call with a try-catch block. The body of the catch performs accessor expansion on the exception as described in Section IV-E.

H. 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. Section V-A2 describes a case study that required custom repair strategies to update the content of external files.

V. Evaluation

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

Q1 How often can ReAssert suggest a repair for a failing unit tests?

Q2 Are the repairs suggested by ReAssert usable for developers? Do the developers accept the changes suggested by ReAssert?

Q3 Does ReAssert give false confidence to the developers such that they mask regression errors by repairing failing tests?

```
public void test16() {
  ...
  CategoryPlot var6 = new CategoryPlot();
  ...
  Integer var8 = -1;
  ValueAxis var9 =
    var6.getRangeAxisForDataset(var8);
  ...
}
```

(a) The failing test

```
public void test16() {
  ...
  CategoryPlot var6 = new CategoryPlot();
  ...
  Integer var8 = -1;
  ValueAxis var9 = null;
  try {
    var6.getRangeAxisForDataset(var8);
    fail();
  } catch (IllegalArgumentException e) {
    assertNull(e.getCause());
    assertEquals(”Negative \’index\’.”,
      e.toString());
    assertEquals(”Negative \’index\’.”,
      e.getMessage());
    assertEquals(”Negative \’index\’.”,
      e.getLocalizedMessage());
  };
  ...
}
```

(b) The suggested repair

Figure 12. Failing automatically generated test for JFreeChart that is repairable with the *Surround with Try-Catch* strategy.

Project	Tests	Failures	Accepted Repairs
Basset	-	17	15 (88%)
DPJ compiler	86	6	5 (83%)
DPJizer	78	14	9 (64%)

Figure 13. Using ReAssert to repair tests in Basset, DPJ, and DPJizer projects.

We describe two case studies in which ReAssert was used to repair failures in evolving research software, which provided some quantitative data for Q1 and Q2, and some qualitative data for Q3. We also performed a controlled user study that addressed all three questions directly. For Q1, we assess ReAssert's ability to repair failures in open-source applications. Finally, we discuss reasons why ReAssert is unable to repair certain failures.

A. Case Studies

We asked two teams of researchers to try ReAssert when changes to their evolving research software caused unit tests to fail. ReAssert successfully (Q1) and usefully (Q2) repaired the failures and revealed regressions in the software (Q3). The following sections describe these case studies.

1) Case Study: Basset: Java PathFinder (JPF) [17] is a system for verifying Java programs using an explicit-state model checker that acts on Java bytecode. JPF offers a powerful extension mechanism that allows one to verify

```
public class TestReductionModes
extends TestActor {
  private static String language = ...
  ...
  @Test public void clientserver_jpfcomp() {
    options = ...
    runAndCheck(language, Driver.class, options,
      null, 5, 1, 24, 2, 5, 0, 0);
  }
  ...
}
```

Figure 14. Example Basset unit test

programs in many domains. Basset [18] is one such extension currently under development that verifies programs that utilize the actor model of concurrency. This extension has a large suite of JUnit tests that often break as both JPF and Basset improve.

Figure 14 shows a simplified example of a Basset unit test. Each test does nothing except pass several arguments to a helper method called `runAndCheck` declared in the superclass `TestActor`. This method executes Basset using the first four arguments. After Basset completes, `runAndCheck` verifies that the latter seven arguments (which we will call the "postcondition numbers") match numbers derived from Basset's internal state.

These tests are exceptionally fragile and difficult to repair manually. First, the postcondition numbers often change as Basset improves. Second, The postcondition numbers are difficult to calculate prior to running the test (yet reasonably easy to check afterward). Third, each test takes tens of seconds to run due to JPF's high overhead. Therefore, when tests failed due to changes in Basset, the developers were forced to spend many minutes manually re-running each failing test to calculate each of the seven postcondition numbers.

ReAssert removes much of this manual labor since it automates re-execution and can fix multiple failures. Internally, ReAssert applies the *Trace Declaration-Use Path* strategy to replace postcondition numbers.

On one failing test run, shown in Figure 13, the Basset developers accepted 15 of ReAssert's 17 suggested repairs (Q1 and Q2). This corresponds to replacing 105 postcondition numbers. The remaining two failures were due to null pointer dereferences, which, while repairable using the *Surround with Try-Catch* strategy, were caused by regressions (Q3).

2) Case Study: Deterministic Parallel Java: Deterministic Parallel Java (DPJ) [19] is an extension to the Java language. DPJ gives static guarantees that a program that type-checks with the DPJ compiler will be deterministic when executed by any number of parallel tasks. DPJ incorporates a sophisticated effect system to verify noninterference of parallel tasks. DPJizer [20] is a tool that infers effect annotations in DPJ programs.

Because the DPJ language is an evolving research language, the DPJ compiler and the DPJizer tool must also evolve. This evolution frequently causes tests to break and requires a substantial manual effort to repair them.

Both DPJ compiler's and DPJizer's unit tests verify that the output from the tool matches the expected content in a prepared file. We implemented a custom repair strategy that is conceptually similar to the *Replace Literal in Assertion* strategy: it must record the expected content and write it to the expected file.

This strategy instruments two "levels" of assertions. The highest level is a helper method called `assertFilesAreTheSame` that takes the expected and actual file names. The expected file contains the prepared output, and the actual file contains the output from DPJ/DPJizer. The helper method reads the content of both files to strings that it passes to JUnit's `assertEquals`. Thus, the instrumented `assertFilesAreTheSame` records the expected file name, and `assertEquals` records the actual content.

Figure 13 shows how many tests were repaired with ReAssert and this custom strategy. The repairs were correct—in that they would have caused the test to pass (Q1)—in all cases, and the developers accepted 14 out of 20 of the suggested repairs (Q2). The remainder revealed regressions in the system under test (Q3).

The developers expect to continue using ReAssert as DPJ and DPJizer evolve. They commented that ReAssert was useful not only for updating existing tests but also for writing new tests: they left the expected content blank, and ReAssert filled it in with the actual content.

B. Controlled User Study

We performed a controlled user study to quantitatively and qualitatively evaluate ReAssert. Quantitatively, we collected and analyzed data required to answer our three research questions. Qualitatively, we used a questionnaire to find out whether participants found ReAssert useful for performing the tasks in the user study and for software development and testing in general. The questionnaire was also used to find out whether the participants, thought that ReAssert should be included as part of the Eclipse IDE and whether they would recommend ReAssert to other people.

1) Participants: We had 18 participants in the user study: 13 graduate students, 3 undergraduate students, and 2 industry professionals. Figure 15 shows their experience demographics. No rewards were offered for participating in the study. Invitations to the study were sent out through class and departmental mailing lists as well as individual emails.

We randomly split the participants into two groups: a control group who were asked to perform the tasks described below using the Eclipse IDE but without ReAssert, and a ReAssert group who were asked to perform the tasks using the Eclipse IDE with ReAssert. In the end, nine members of

Experience	Mean	Stdev	Min	Max
Programming	11.2	6.1	2.0	28.0
Java	5.1	3.4	1.0	11.0
JUnit	2.4	2.3	0.0	8.0
Eclipse	2.7	2.0	0.0	6.0

Figure 15. Participants experience demographics (numbers in years)

Control Group

Test Suite	Failures	Repairs	Matches
User written	26	26 (100%)	25 (96%)
Provided	47	47 (100%)	42 (89%)

ReAssert Group

Test Suite	Failures	Repairs	Matches
User written	19	15 (79%)	12 (80%)
Provided	43	43 (100%)	34 (79%)

Figure 16. User study results.

Question	Responses
Useful for the study	very 50%, yes 38%, no 12%
Use for own projects	yes 56%, maybe 33%, no 11%
Include in Eclipse	yes 67%, maybe 33%
Recommend to others	yes 67%, maybe 33%

Figure 17. Responses to Qualitative Questions

the control group and nine members of the ReAssert group finished the study. One participant from the ReAssert group did not finish all tasks because the participant used an older version of ReAssert that did not have the *Trace Declaration-Use Path* strategy.

2) Tasks: The participants were given the application code and passing unit tests for the shopping cart application introduced in Section II. They were asked to first familiarize themselves with the code/tests and then to perform the following five tasks:

Task 1: Write some unit tests of their own to test previously untested functionality.

Task 2: Implement a requirement change which could potentially cause some of *their tests* to fail.

Task 3: Repair all failing tests.

Task 4: Implement another requirement change which would cause some of the initially *provided tests* to fail.

Task 5: Repair all failing tests.

The participants were asked to perform the tasks in order and click on a button at the end of each task to indicate completion. This helped us record the state of the code and tests at the end of each task.

3) Quantitative Results: The quantitative results of the user study provide some data for all three of our research questions.

Q1: The first two columns of Figure 16 summarize the number of failures caused by the participants' code changes in tasks 2 and 4 and the number of those failures that ReAssert could repair such that they pass. Note that the participants from the control group did not actually run ReAssert, so we evaluated post-mortem whether ReAssert would have suggested a repair that made their failing tests pass.

In total, ReAssert could repair 98% (131 of 135) of failures caused by the participants' code changes. Three of the failures were not repairable because a participant was using an older version of ReAssert which did not implement the *Trace Declaration-Use Path* strategy; those failures can now be repaired with the latest version of ReAssert. The remaining unrepaired failure was caused by the nondeterministic iteration order of `java.util.HashMap`; common practice dictates that tests should be deterministic.

Q2: The final column of Figure 16 summarizes the number of ReAssert's suggested repairs that *exactly matched* the changes made by the participants to repair the failing tests in tasks 3 and 5. We determined this match by structurally comparing tests recorded at the end of task 3 or 5 with the tests obtained by applying ReAssert's suggested repairs on the code written by the participants in the preceding task 2 or 4, respectively. Our structural comparison ignored white spaces, comments, variable names, and other code differences that don't affect code behavior.

For task 3, in which participants repaired tests they had written themselves, 90% (37 of 41) of ReAssert's suggested repairs matched the participants' repairs. For task 5, in which participants repaired tests provided to them, 84% (76 of 90) of ReAssert's repairs matched.

These results show that the repairs suggested by ReAssert are very useful because they very frequently match the repairs made by the participants. In cases where the repairs suggested by ReAssert did not match, the participants' repairs most often involved changes to the test setup or the addition of new assertions. We suspect that ReAssert could have been useful even in these cases, and thus the exact match is the lower limit for the number of usable repairs.

The participants found the *Replace Literal in Assertion* strategy particularly useful when they changed how the shopping cart printed the final bill. Those without the tool most often copied and pasted the output from JUnit, essentially replicating this strategy manually.

Q3: The participants introduced a total of 20 bugs in their code. We expected some bugs since our requirements were fairly vague, simulating real-life situations. Eleven of the bugs were introduced in task 2, and nine were introduced in task 4. Twelve of the bugs were introduced by participants who used ReAssert, and eight were introduced by participants who did not use ReAssert. One of the reasons why ReAssert users introduced more bugs could be that they become overly reliant on the tool. This can be mitigated by training users to carefully inspect the repairs suggested by ReAssert rather than accepting them blindly.

Project	Version(s)	Description
Apache Coll.	3.2	Collection library
Apache Lucene	2.2.0, 2.4.1	Text search engine
Barbecue	1.5 beta	Bar code creator
Checkstyle	3.0, 3.5	Code style checker
java.util Coll.	1.6	JDK collection classes
JDepend	2.8, 2.9	Design quality metrics
JFreeChart	1.0.7, 1.0.13	Chart creator
Joda Time	1.6	Date & time library
JSAP	2.1	Console args parser
PMD	2.0, 2.3	Java program analysis
Time&Money	0.5.1	Time & money library
XStream	1.2, 1.3.1	XML obj serialization

Figure 18. Subject applications

```
public void testclasses25() {
  ...
  int var25 = 0;
  ArrayDeque var26 = new ArrayDeque(var25);
  ...
  short var28 = 1;
  var26.addLast(var28);
  Object var30 = var26.peekLast();
  Object var31 = var26.pop();
  ...
  assertEquals(1, var30);
  assertEquals(1, var31);
}
```

Figure 19. A simplified test generated by Randoop

4) Qualitative Results: We obtained qualitative feedback though a simple questionnaire and post-study interviews. We modeled our questionnaire and analysis after a study by Saff and Ernst [21].

Questionnaire Responses: We asked the nine participants in the ReAssert group four questions after they completed their tasks. Figure 17 summarizes their responses. The majority of the participants found ReAssert useful for the user study tasks and thought that it 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.

Positive Feedback: The participants who found ReAssert useful liked the fact that ReAssert repairs all the assertions in a test method at once rather than individually. Some participants mentioned that they would write more tests if they had a tool like ReAssert to help them maintain tests. They also noted that some assertions were very tedious to write (especially those involving large strings) and mentioned that they used ReAssert to build such assertions by leaving assertions “empty” and letting the tool compute the real expected value. This shows that ReAssert could also be helpful when creating new tests.

Cautionary Feedback: Most participants expressed concerns that ReAssert could induce carelessness in developers. These concerns do hold merit as shown by the results for Q3 in Section V-B3. However, we believe that this problem can be mitigated by explicitly informing developers that they should carefully inspect each repair suggested by ReAssert before applying it.

C. Failures in Open-Source Projects

We address Q1 by measuring how many failed tests ReAssert can successfully change such that they pass. Doing so requires many examples of failing tests in real applications. We consider the open-source projects listed in Figure 18. All are widely used and actively developed Java applications whose source code is available online. Several have been used in previous studies [22], [23], [24]. More importantly, they each had a large, manually written test suite and evolved in ways that would cause some tests to fail.

We obtained failing tests in two ways. First, we executed tests from an early version of the software on a newer version of the system under test. This procedure mimics the evolution of the system under test (but is not exactly the same since the tests would have also evolved with the application). We refer to these failing tests as *version difference failures*. Second, we executed tests from the later version of the software on mutants of that version obtained using the Jumble mutation testing tool [25]. We refer to these failing tests as *mutation failures*.

We executed both manually written and automatically generated test suites. Manually written tests came from the projects’ source code repositories. For a subset of the projects, we generated many regression tests using Randoop [14], a feedback-directed random test generator. These tests are created for the early version or non-mutated version of the software.

Thus, we have four evaluation scenarios: manually written or automatically generated tests executed on a newer or mutated system under test.

1) Generating Tests with Randoop: Randoop is one example of an automatic test generator. Given a system to test, it produces a regression test suite with minimal user input. It uses feedback-directed test generation in which it starts from shorter method sequences, then extends them by adding more method calls (randomly selected), runs the resulting sequences and feeds back the results into the generation process. It can output regression tests that assert some observed values from the generated sequences. Figure 19 shows a simplified test generated by Randoop. Like many automatic test-generation tools, the tests it produces are particularly well-suited to automatic repair. First, many are likely to break on seemingly simple changes. Second, the generated tests tend to have very simple structure and assertions (although they can be quite long).

For our evaluation, we directed Randoop to generate tests for a subset of classes in the subject applications. The process we used for generating the tests consisted of the following four steps:

1) When projects had existing manual tests, we identified the classes in the older version that caused the most failures when tests from the older version were run against the newer version of the project. We then used Randoop to generate tests for the identified classes and classes that were imported by those classes. When projects had no existing manual tests, we generated tests for the main classes in the project and the classes that were required to call methods on them.
2) The tests generated by Randoop were not always deterministic, i.e., some tests passed during some runs and failed during other runs against the same older version of the subject application. The generated tests were non-deterministic for various reasons; the most common reason was that some assertions checked the hashcode of objects using the default implementation of hashcode which relies on the internal address of the object. We removed all such non-deterministic tests by running the generated tests many times against the same older version of the subject application and removing any tests that failed during any of the runs.
3) After performing the first two steps, we obtained automatically generated deterministic test suites for the older verions of the subject applications. However, in many cases, these tests would not compile when compiled against the newer versions of the subject applications. This was because the newer versions included many refactorings or changes to interfaces. Where possible, we performed the same refactorings (mostly renamings) on the generated tests to make them compile. When this was not possible, we removed the non-compiling tests so that the test suites could be run against the newer versions of the subject applications.
4) In some cases the automatically generated test suites we obtained at the end were not interesting enough because they exercised only a small portion of the subject applications. In such situations, we provided Randoop with helper classes that had methods that could be called by Randoop to obtain instances of various classes that would bootstrap Randoop in the directions required to expand the coverage of the tests it generated.

Once we obtained suitable automatically generated test suites using Randoop, we evaluated ReAssert with respect to the automatically generated tests that passed when run against the older version of a subject application but failed when run against the newer version of the subject application. While these failures were mostly caused by changes between versions due to requirements changes, we did detect one instance in JFreeChart where a failure was caused due to a regression.

Project	Tests	Failures	Repairs
Checkstyle	143	34	9 (26%)
JDepend	53	6	6 (100%)
JFreeChart	1696	18	16 (89%)
PMD	448	5	5 (100%)
Apache Lucene	663	47	12 (25%)
XStream	726	60	28 (47%)
JFreeChart	2050	14	9 (64%)
Lucene	1077	278	75+203* (100%)
XStream	692	33	27 (82%)

Figure 20. Tests that fail due to version differences and how many ReAssert repairs such that they pass. Above the line are manually written tests, below are Randoop-generated tests.

We also used Randoop to generate tests that were run against mutated versions of code. We had to perform similar steps as above but did not need to change Randoop-generated tests to compile on the mutated versions since Jumble (like other mutation tools) does not mutate the API of the code.

2) Version Difference Failures: Running an old test suite against a new system under test creates tests that no longer reflect the correct behavior of the code. Repairing such tests brings them back into sync with the system under test. It is important—especially with manually written tests—not only that the repaired tests pass (Q1) but that the repair is something that the developer would expect (Q2).

For each of our subject applications, we retrieved two versions from the application's source control repository. For those applications without manually written tests, we generated a test suite on the old version using Randoop. We applied the old test suite (manually written or automatically generated) to the new system under test. We removed any tests that did not compile, ran the tests, and recorded the number of failures. ReAssert attempted to repair the failures, and we recorded the number of successful repairs as well as the repair strategies used. If the application had a manually written test suite, we manually inspected ReAssert's suggested repairs to see how they compared with the actual repairs written by developers.

Figure 20 shows version difference failures for both manually written and Randoop-generated test suites. ReAssert's success rate, measured as a ratio of repaired to failing tests, is 45% on manually written tests and 97% (with caveats explained in Section V-D) on Randoop-generated tests.

ReAssert's success rate is more dependent on the structure of the tests than the type of failure. For example, almost all of the failures in JDepend, JFreeChart, and PMD can be repaired since their tests have simpler control flow than CheckStyle's and Lucene's. Similarly, Randoop-generated tests are exceptionally simple, allowing ReAssert to repair nearly all of them.

The usefulness of the fixes suggested by ReAssert was measured by comparing them against the actual fixes made

Project	Tests	Mutants	Failures	Repairs
Apache Coll.	2025	695	8451	7885
Barbecue	269	587	2902	1903
JodaTime	2093	1318	8115	6670
JSAP	27	461	385	333
java.util	6370	1971	97217	95486
TimeAndMoney	6840	685	38300	37235

Figure 21. Tests that fail due to mutations and how many ReAssert has a strategy to repair. Above the line are manually written tests, below are Randoop-generated tests.

Strategy	Version Failures		Mutation Failures	
	Manual	Randoop	Manual	Randoop
RAM	14	0	1960	28663
IR	2	0	143	0
RL	95	130+203*	5258	14908
RRM	0	0	444	0
AE	5	0	1680	22
TD	11	0	1553	0
STC	12	28	8353	89128

Figure 22. Strategies used.

by developers (for manually written tests) and the most probable fixes (for automatically generated tests). It was easy to discern that most of the fixes suggested for manually written tests were useful because they matched the fixes made by the developer in the next version. However in some cases, the fixes made by the developers were not straight forward; for example they added additional code checking other properties, deleted assertions, created helper methods etc. In such cases it was harder to discern the usefulness of the fixes because the tests had been changed so drastically in the next version. The only way to confidently measure usefulness in such scenarios would be to monitor developer changes at a finer granularity (rather than just version control diffs). In the case of tests generated by Randoop, it was easy to discern that all the fixes were usable because Randoop only used simple assertions (asserting literals against the value of variables) and the fixes suggested changing the literal value to fix the assertion.

3) Mutation Failures: Mutation testing [26] is a technique used to assess the effectiveness of a test suite. A mutation tool changes, i.e., "mutates", the system under test to produce a large number of "mutants". If one or more tests fail when executed on a mutant (i.e., the result on the mutant differs from the result on the original code), the test suite "kills" the mutant; otherwise, the mutant remains "alive". Developers can use the ratio of killed and alive mutants to decide whether to extend the test suite.

We use mutation to simulate how software could change and to reveal a sample of how tests *could* fail. Unlike traditional mutation testing, our evaluation is not concerned with how many mutants a test suite kills. Instead, we provide some data to answer Q1: how many failures introduced by mutants have an appropriate repair strategy. We do not address Q2 through this experiment since we have no real evolution of code or tests.

We created mutants using the Jumble mutation testing tool [25]. Jumble mutates conditionals, binary arithmetic operations, unary increment and decrement, constants, and return values. We chose this tool because it mutates bytecode dynamically, works seamlessly with JUnit, and was easily extended to work with ReAssert.

We chose a subset of classes in our subject applications to mutate. Jumble mutated each class individually, one mutation at a time. Then, it executed a short list of tests for the mutated class. We inserted ReAssert into Jumble's pipeline at this point. ReAssert attempted to repair any failures introduced by the mutation. It recorded the repair strategy used and the location of the repair. For technical reasons having to do with class loading, ReAssert did not attempt to recompile and re-execute the test; it merely counted the repair strategies attempted. Thus, it revealed a distribution of how tests can fail and what strategies would apply.

Figure 21 shows the number of mutants generated, how many failures occur due to the mutations, and how many ReAssert has a strategy to repair. For manually written tests, ReAssert has a strategy for 85% of the failures; for Randoop-generated tests, 98%. Even though the previous section shows that not all strategies produce passing tests, these numbers show that ReAssert implements a strategy for the majority of failures.

4) Repair Strategies Used: Figure 22 shows which repair strategies each evaluation scenario used. It is clear that *Replace Literal in Assertion* is a particularly useful strategy despite its simplicity; all scenarios used it heavily. The distribution of other strategies depends on the system under test, the structure of the test code, and the types of failures.

The dependence on test code is particularly apparent in the different strategies used to repair manually written versus Randoop-generated tests. Since Randoop only produces simple `assertTrue`, `assertFalse`, `assertNull`, and `assertEquals` calls, it has little need for complex strategies like *Accessor Expansion* or *Trace Declaration-Use Path*.

Repairs for mutation failures were largely dependent on the type of failure. The mutator tended to produce objects that threw many exceptions, so the *Surround with Try-Catch* strategy applied most often in those cases.

D. Unrepairable Failures

ReAssert cannot repair all tests, nor would we expect any tool to be able to do so in a usable manner. Removing all failing assertions would trivially cause a test to pass but would not reveal anything about the correct (or incorrect) behavior of the system under test. Alternately, some failures require large changes to both test and application code. Since

ReAssert makes minimal changes to test code and leaves application code unchanged, it is inevitable that it cannot repair all broken tests. The following are common reasons why this is the case.

Test Framework Restrictions: The starred number in figures 20 and 22 represents repairable assertions that cannot be expressed in the test framework. In particular, Randoop produced many `assertNull` invocations that failed when passed a non-null array of primitive types. The *Replace Literal in Assertion* strategy can repair these failures by changing the `assertNull` to `assertArrayEquals`. However, Randoop's tests require JUnit version 3 while `assertArrayEquals` is implemented in JUnit version 4.

Shared Logic: If a failure occurs in code that is executed multiple times in different contexts—for example if a common setup or teardown method fails, an assertion fails in a loop, or a helper method cannot be repaired by the *Trace Declaration-Use Path* strategy—then the failure is likely unrepairable since a repair may break subsequent executions.

Too Many Assertions: Exceptionally large tests with many failing assertions may exceed the maximum number of failures that ReAssert can repair. By default the limit is 10, but the number can be increased if needed.

Nondeterminism: Tests should generally execute deterministically, but we encountered some that do not, causing unrepairable failures. Similarly, if the *Accessor Expansion* strategy asserts against a nondeterministic accessor, then the test will probably fail. This is a common problem encountered by many test generation tools, including Randoop.

External Resources: Certain tests depend on external resources such as input files. As discussed in Section V-A2, such failures can be repaired but require custom strategies tailored to the applications.

No Applicable Strategies: Certain failures may not meet the preconditions of any repair strategy. For example, a combination of helper methods, complex control flow, and data operations may prevent both *Trace Declaration-Use Path* and *Replace Literal in Assertion* from applying.

Repair Would Remove Test Logic: In certain cases a repair is possible but would remove rather than update test logic. In these cases, we do not implement a repair strategy. For example, a hypothetical Remove Try-Catch strategy would repair several failures in Randoop-generated tests, but would be equivalent to removing an "assertion" that an exception was thrown.

VI. Related Work

While there has been much prior work in program repair and maintenance, we are unaware of any other tools or techniques that automatically repair general-purpose unit tests. Most existing work focuses on repairing the system under test rather than the test code. Those techniques that act on test code often rely on test (re)generation or are applicable only to a particular domain such as GUI test scripts.

A. Fault Localization and Repair

Many researchers have proposed techniques for locating and fixing faults in the system under test. The first step, finding those statements likely relevant to a particular failure, is referred to as *fault localization* [27], [28], [29]. Our technique—particularly the *Trace Declaration-Use Path* strategy—locates statements to repair by traversing the failure trace; this is conceptually similar to fault localization using backward dynamic program slices [30], [31], [32], [33].

The second step, repairing the system under test to remove the fault, is sometimes referred to as *automated debugging* [34]. He and Gupta [35] introduce the concept of *path-based weakest preconditions* and uses this formalism to locate and repair erroneous statements. Arcuri [36], Jeffrey et al. [37], and Weimer et al. [38] all model automated debugging as a search problem solvable using genetic algorithms and machine learning. Finally, Abraham and Erwig [39] describe a semi-automatic debugger for spreadsheet formulas.

Other fault repair techniques focus on repairing program state rather than code. Elkarablieh et al. [40] and Demsky et al. [41] both describe techniques that repair faulty data structures while a program executes.

B. Automatic Test Generation

Automatic test generation tools produce a suite of regression tests (sometimes called *characterization tests*) with minimal user input. We use one such tool, Randoop [14], in our evaluation, but there are many other examples utilizing automatic invariant detection [10], [42], symbolic execution [11], software version differences [43], and static program analysis [9].

When automatically generated tests fail due to changes in the system under test, users rarely take the time to examine the failures in detail or repair the tests. Instead it is much more common to throw away the broken tests and re-run the tool to generate new tests. ReAssert is complementary to automatic test generation in that it provides a way to maintain unit tests without re-generation.

ReAssert most frequently repairs unit tests by modifying or creating assertions. Thus, it is closely related to test augmentation techniques [13], [44] that aim to improve regression test oracles.

C. Maintaining GUI Test Scripts

Testing graphical user interfaces (GUIs) requires a great deal of manual effort. Test engineers manually write test scripts that interact with GUI widgets or use a record-and-replay testing tool [45]. These scripts or recordings are very fragile, and several researchers have proposed techniques for maintaining GUI tests [46], [47]. In particular, Memon et al. have done extensive work in generating [48] and repairing [49], [50] GUI tests.

Techniques for GUI test repair are conceptually similar to ReAssert's *Replace Literal in Assertion* strategy: both must replace the incorrect portions of test code with the correct value(s) derived from a test execution. Unlike GUI test maintenance tools, ReAssert repairs general-purpose unit tests and provides an extension mechanism for domain-specific repair strategies.

VII. Conclusion

We presented ReAssert, a novel technique and tool that automates repair of broken unit tests. ReAssert performs a combined dynamic and static analysis to find repairs that developers are likely to accept with a mouse click, relieving them from a tedious editing process. Our experiments show that ReAssert can propose repairs for a large percentage of failing tests and that developers often find the suggested repairs useful. We believe that by using ReAssert, developers can write more unit tests or more effectively use automatic test generation tools, improving their ability to detect regressions.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. CCF-0746856. We would like to thank Rob Bocchino and Mohsen Vakilian for their help with DPJ and DPJizer; Steve Lauterburg and Bobak Hadidi for their help with Basset; the participants of our user study for providing an hour of their time; and Milos Gligoric, Munawar Hafiz, Yun Young Lee, and Samira Tasharofi for their valuable comments.

References

[1] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting repairs for broken unit tests," in *ASE*, 2009.

[2] G. Venolia, R. DeLine, and T. LaToza, "Software development at Microsoft observed," Microsoft Research, TR, 2005.

[3] N. Tillmann and W. Schulte, "Unit tests reloaded: Parameterized unit testing with symbolic execution," Microsoft Research, TR, 2005.

[4] K. Beck, *Test-Driven Development By Example*, 2003.

[5] S. Fraser, D. Astels, K. Beck, B. W. Boehm, J. D. McGregor, J. Newkirk, and C. Poole, "Discipline and practices of TDD," in *OOPSLA Companion*, 2003.

[6] B. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *PASTE*, 2001.

[7] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *OOPSLA*, 2004.

[8] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *ICSE*, 2005.

[9] Parasoft, "Jtest." [Online]. Available: http://www.parasoft.com/jsp/products/home.jsp?product=Jtest

[10] M. Boshernitsan, R.-K. Doong, and A. Savoia, "From daikon to agitator: lessons and challenges in building a commercial tool for developer testing," in *ISSTA*, 2006.

[11] N. Tillmann and J. de Halleux, "Pex-white box test generation for .NET," in *TAP*, 2008.

[12] C. Csallner and Y. Smaragdakis, "DSD-Crasher: A hybrid analysis tool for bug finding," in *ISSTA*, 2006.

[13] T. Xie, "Augmenting automatically generated unit-test suites with regression oracle checking," in *ECOOP*, 2006.

[14] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007.

[15] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[16] J. Purtilo and J. Callahan, "Parse tree annotations," *CACM*, 1989.

[17] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering Journal*, vol. 10, no. 2, 2003.

[18] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha, "A framework for state-space exploration of Java-based actor programs," in *ASE*, 2009.

[19] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A Type and Effect System for Deterministic Parallel Java," in *OOPSLA*, 2009, (To Appear).

[20] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson, "Inferring method effect summaries for nested heap regions," in *ASE*, 2009.

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

[22] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," Universitaet des Saarlandes, TR, 2009.

[23] B. Daniel and M. Boshernitsan, "Predicting effectiveness of automatic testing tools," in *ASE*, 2008.

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

[25] S. Irvine, T. Pavlinic, L. Trigg, J. Cleary, S. Inglis, and M. Utting, "Jumble java byte code to measure the effectiveness of unit tests," *TAICPART*, 2007.

[26] W. Howden, "Weak mutation testing and completeness of test sets," *TSE*, 1982.

[27] S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in *ICSE*, 2002.

[28] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *ICSE*, 2002.

[29] H. Cleve and A. Zeller, “Locating causes of program failures,” in *ICSE*, 2005.

[30] F. Tip, “A survey of program slicing techniques,” *Journal of Programming Languages*, vol. 3, no. 3, 1995.

[31] B. Korel and J. W. Laski, “Dynamic slicing of computer programs,” *Journal of Systems and Software*, vol. 13, no. 3, 1990.

[32] H. Agrawal and J. R. Horgan, “Dynamic program slicing,” in *PLDI*, 1990.

[33] M. Weiser, “Program slicing,” in *ICSE*, 1981.

[34] A. Zeller, “Automated debugging: Are we close,” *Computer*, vol. 34, no. 11, 2001.

[35] H. He and N. Gupta, “Automated debugging using path-based weakest preconditions,” in *FASE*, 2004.

[36] A. Arcuri and X. Yao, “A novel co-evolutionary approach to automatic software bug fixing,” in *CEC*, 2008.

[37] D. Jeffrey, M. Feng, N. Gupta, , and R. Gupta, “Bugfix: A learning-based tool to assist developers in fixing bugs,” in *ICPC*, 2009.

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

[39] R. Abraham and M. Erwig, “Goal-directed debugging of spreadsheets,” in *VL/HCC*, 2005.

[40] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid, “Assertion-based repair of complex data structures,” in *ASE*, 2007.

[41] B. Demsky and M. Rinard, “Automatic detection and repair of errors in data structures,” in *OOPSLA*, 2003.

[42] C. Csallner, Y. Smaragdakis, and T. Xie, “DSD-Crasher: A hybrid analysis tool for bug finding,” *TOSEM*, 2008.

[43] S. Person, M. Dwyer, S. Elbaum, and C. Pasareanu, “Differential symbolic execution,” in *ESEC/FSE*, 2008.

[44] Y. Song, S. Thummalapenta, and T. Xie, “UnitPlus: assisting developer testing in Eclipse,” in *OOPSLA workshop on Eclipse Technology eXchange*, 2007.

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

[46] M. Grechanik, Q. Xie, and C. Fu, “Maintaining and evolving GUI-directed test scripts,” in *ICSE*, 2009.

[47] C. Kaner, “Improving the maintainability of automated test suites,” *Software QA*, vol. 4, no. 4, 1997.

[48] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan, “Dart: A framework for regression testing ”nightly/daily builds” of GUI applications,” in *ICSM*, 2003.

[49] A. Memon, “Automatically repairing event sequence-based GUI test suites for regression testing,” *TSE*, vol. 18, no. 2, 2008.

[50] A. Memon and M. L. Soffa, “Regression testing of GUIs,” in *ESEC/FSE*, 2003.