AUTOMATED FIXING OF WRONG ASSUMPTIONS ON UNDERDETERMINED SPECIFICATIONS

BY

PEILUN ZHANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Advisers:

Professor Darko Marinov
Associate Professor Victoria Stodden

ABSTRACT

Software specifications assist in the implementation of software by stating expected software behavior. Many software specifications are deterministic, i.e., the software produces the same output for the same input. However, some specifications are underdetermined, meaning that the software may produce different outputs given the same input. Underdetermined specifications are not uncommon because they can offer some advantages over deterministic ones. For instance, underdetermined specifications can allow for optimization as developers can be more flexible when conducting speed optimization. We encounter potential problems when deterministic implementations produce different outputs. For example, even though the Java standard library does not specify the order of elements returned by method `getDeclaredFields`, prevailing implementations, like Oracle JDK 1.8, return fields in the order as they are listed in the class file (which is itself the order in which they are in the source file when using the standard `javac` compiler). There is no reason to expect the implementation will not change because the specifications allow flexibility, and popular vendors have historically changed the implementations of several widely used library methods.

Unfortunately, software library users may write code that relies on a specific *implementation* rather than on the *specification*, e.g., assuming mistakenly that the order of elements cannot change in the future. If users write software tests that involve methods with underdetermined specifications, those tests can therefore produce unexpectedly non-deterministic outputs, meaning that tests can intermittently fail or pass without changing the production and test code (but changing the library code). Prior work proposed the NonDex approach to proactively detect such wrong assumptions in the production and test code.

The goal of this thesis is to propose automated code changes that help resolve these issues by either making the output deterministic or making the test assertion order-agnostic. We present a novel approach, called *DexFix*, to fix wrong assumptions on underdetermined software specifications in an automated way. To demonstrate these efforts, we run the NonDex tool on 200 open-source Java projects and detect 275 tests that fail due to wrong assumptions. We find that the majority of failures are based on `HashMap`/`HashSet` class iterations and the `getDeclaredFields` method. We provide several new automated fix strategies that can fix these violations in both the production and test code, which are implemented in the DexFix tool. Our experiments show that DexFix proposes fixes for 101 tests from our 275 tests. We have reported fixes for 84 tests to the developers as GitHub pull requests: 57 have been merged, with only 2 rejected, and the remaining are pending.

To my mother, for her endless love and support.

ACKNOWLEDGMENTS

I have received a lot of help and support from many people towards completing this thesis.

First, I would like to thank my adviser, Professor Darko Marinov. I am extremely grateful that Darko keeps encouraging me to get involved in research activities. It is also Darko who provides me with the opportunity to work with him and other awesome students on this research topic. I appreciate the advice Darko has given me on both research and life.

I would like to thank my co-adviser, Associate Professor Victoria Stodden, for her advice and hands-on guidance on this thesis. I am also grateful for Victoria's encouragement to me during the current special period that the whole society is going through.

I would like to thank August Shi, who has spent hours and hours discussing and working together with me. Working with August is one of the most enjoyable experiences I have in graduate school. I wish him all the best in his upcoming career in academia. I would like to thank Yanjie Jiang and Anjiang Wei for their collaborations, and my research experience would not have been as enjoyable as it was without them. I would like to thank Yizhi Huang and Hao Wu for their friendship. It would be significantly more difficult for me to survive multiple hard times in my graduate school life without them. I would like also to thank Minho Song, who helped me get through the period when I hit my lowest point. He is always a fount of inspiration to me.

Finally, and most significantly, I would like to thank my mother, Ruimei Zhang. She supported me to study in a country that is thousands of miles away from my home. She gave me the freedom to pursue my dream and constantly encouraged me when I hesitated about my decision. Words cannot express my gratitude and love to her.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 SOFTWARE TESTING

Modern society runs on software and the continuing rise in computing power will continue to enable software to automate more tasks in our daily lives [1]. The increasing dependency on software also contributes to the prevailing demand for increasing quality of software [2].

Software testing aims to increase the quality of software by evaluating whether the functionality of software applications matches the specified requirements. This evaluation is usually done by executing a piece of program, namely a test, which encodes developer expectations of software functionality [2]. Software testing plays an important role in modern software development, especially for the typical case of large-scale software applications that are collaboratively developed in multiple iterations. Software testing catches faults in earlier stages, which makes fixing those faults easier with fewer costs. In addition, developers write software tests as documentation to demonstrate to library users the correct usage of the code's underlying functionality [2].

Regression testing is one of the most widely used types of software testing [3]. When a developer makes and commits changes, regression test systems run regression test suites and report test results to the developer, i.e., the tests pass or fail. A test suite passes only if all the tests from the test suite pass. If any test fails after the new changes, regression test systems will block the changes from being merged so as to prevent changes from breaking the existing functionalities in production [4]. The new changes become ready for review only when regression test suites pass. The main assumption of regression testing is that a test that fails after the new changes but passes before the changes indicates that the changes introduce the fault and, thus, flags to developers to investigate the changes to debug [3].

1.2 FLAKY TESTS

Regression testing heavily relies on the assumption that a test failure indicates some fault in the new changes, and uses the test failures as flags to draw developers' attention for further investigation. In practice, however, this assumption may not always hold because test outcomes can be unreliable [5]. There are many reasons tests can become unreliable [6, 7, 8, 9, 10], and one cause is having wrong assumptions on underdetermined specification (Section 1.3) [11]. Unreliable tests intermittently pass or fail, which hinders developers from using test results as an indicator of the quality of changes. Developer cannot be certain if a test failure indicates fault in their changes or in the test itself, which reduces the test suite's ability to detect regressions [12].

Tests that can pass or fail without the developer changing the code are often called *flaky tests* [13]. Flaky tests can waste developers' time to debug test failures that are not directly related to their change, and debugging flaky tests that are written by other developers tends to be more costly. Even worse, as tests can unpredictably pass or fail, it can become more dangerous for developers to ignore the test failures wrongly considering that tests fail due to flakiness instead of faults in the changes. Real faults can be buried in the production code and cause greater loss [14].

1.3 UNDERDETERMINED SPECIFICATIONS

Underdetermined specifications [15] admit multiple implementations. These different implementations can return different output for the same input, even if each implementation itself is deterministic. For example, consider the method `getDeclaredFields` from the Java standard library class `java.lang.Class`. The Javadoc specification [16] for this method states that it "Returns an array of `Field` objects reflecting all fields declared by the class or interface represented by this `Class` object" and also "The elements in the returned array are not sorted and are not in any particular order." For example, consider the code snippet of a test in Figure 1.1 that calls `getDeclaredFields` on a class `C` that has a set of fields with different visibilities (private and public in this example) and different names (each declared field has to have a unique name). When running `CTest#test` with different JDKs, the test outcome may vary. One implementation of `getDeclaredFields` may return an array where the fields are sorted by name, which means the output of `Arrays.toString(fields)` in `CTest#test` is `"[private int a, public int b]"`. Another implementation may return an array where the fields are sorted by visibility, e.g., public methods first, then package methods, then protected methods, and finally private methods. In this case the output becomes `"[public int b, private int a]"`. Implementations (as of this writing) from both Oracle JDK and OpenJDK provide the fields in the order in which they are declared in the class. If library users write code that relies on a specific (deterministic) *implementation* rather than on the (underdetermined) *specification* of a method from the library (like in the code snippet of `CTest#test`), the code can break when the library developers provide a new implementation of the same specification. In some cases the code dependence on a specific implementation may be intentional, but most cases are unintentional and stem from a *wrong assumption* that the implementation will not change in the future. There is no reason to expect that the future implementation will not change. For example, the Java standard library (from Sun

```java
// Class C source
public class C {
   private int a;
   public int b;
}
...
// Test source
public class CTest {
   @Test
   public void test() throws Exception {
      Field[] fields = C.class.getDeclaredFields();
      assertEquals("[private int a, public int b]", Arrays.toString(fields));
   }
}
```

Figure 1.1: Example test that produces unreliable outcome due to wrong assumptions on underdetermined specifications

and then Oracle) has changed the implementation over time of several widely used methods such as `Object#hashCode`, `HashMap` and `HashSet` iterators, and `Class#getMethods` [17], which on several occasions broke substantial amounts of code [18, 19].

Shi et al. [13] developed an approach, called NonDex, to proactively find code that makes such wrong assumptions. Specifically, they manually identified a set of methods with underdetermined specifications in the Java standard library, implemented a tool that can automatically randomize the choices made by the implementation (e.g., permuting the order of fields in the array returned from `getDeclaredFields`), and ran the tool on tests with various random seeds to check if any test fails due to wrong assumptions. While Shi et al. [4] detected 60 failing tests in (21 out of 195) open-source projects, they did not provide support for debugging: "In the future, we plan to investigate how to automate debugging of failures that NonDex reports" [13, p. 10].

1.4 AUTOMATED SOFTWARE TEST FIXING

When changes cause a regression test to fail, a developer should inspect the failure to understand whether the failure is caused by regression or indicates a change of expected behavior. If the failure is caused by regression, developers need to revise the new code to meet the expectations of the test suite and requirements. If the test failure indicates a change of expected behaviors, tests are considered broken and the developer needs to fix or remove those tests from test suites [12]. As fixing broken tests can be time-consuming, especially when new changes can break many tests related to the same functionality, automated software test fixing therefore aims to propose fixes for broken tests in an automated manner while retaining the original logic as much as possible.

Generally, automated program fixing techniques generate patches by searching and mutating existing code, learning from prior patches, or through symbolic execution [20, 21, 22, 23, 24, 25, 26]. These techniques rely on tests to guide them where test failures indicate that the fault in the production code still exists, and the aim is then to make the test pass by fixing the fault in the production code. Prior work focused on fixing test code includes ReAssert [12], which fixes tests that become outdated when production code evolves. ReAssert aims to fix test code instead of production code to make the test pass assuming that evolutions in production code reflect the correct behavior (Section 2.2 presents an example of a ReAssert run).

1.5 OVERVIEW OF THIS THESIS

We present a novel technique, called *DexFix*, to automatically fix wrong assumptions on underdetermined specifications. Inspired by the growing body of work on program repair [20, 21, 22, 23, 25, 26] (with a recent survey [24]) and test repair [12, 27, 28, 29, 30, 31], we provide a set of simple and effective fix strategies that can fix wrong assumptions on underdetermined specifications in *both* the test code and the production code. We derive our fix strategies from the predominant causes we found for test failures that NonDex reports. Existing program or test repair tools cannot handle these cases because these tools do not have the necessary fix strategies to correct wrong assumptions on underdetermined specifications.

In this thesis, we seek to answer the following three research questions:

RQ1: What is the breakdown of the root causes for tests that fail due to wrong assumptions on underdetermined specifications?

RQ2: How many tests can DexFix fix, and what fix strategies are the most effective?

RQ3: How effective is DexFix at proposing fixes that developers actually accept?

We first carry out a formative study. We run the NonDex tool on 200 open-source Java projects, detecting 275 tests that fail due to wrong assumptions and where NonDex provides a specific root cause. Our inspection of these root causes finds the majority are from the `HashMap`/`HashSet` class iterations (152) and the `getDeclaredFields` method (93). We also identify a number of tests that fail due to test assertions comparing JSON strings: the serialization of Java objects into JSON strings can produce JSON strings with a different order of fields; the JSON specification [32] does not specify the order of fields, so any order is valid. To automate analysis and repair of failing assertions we use the ReAssert tool, developed by Daniel et al. [12]. Our initial manual exploration of these cases shows that the outputs of NonDex and ReAssert do *not* provide debugging support that can greatly help in localizing and repairing the real causes of failures, so we extend both of these tools to produce additional info for debugging.

We finally derive new, automated fix strategies that can fix the failing tests by changing the code to properly work with underdetermined specifications. **Intuitively, our strategies aim to make each output deterministic or each test assertion order-agnostic.** For example, consider some code that calls `getDeclaredFields` and then a test exercising this code fails because an assertion expects a particular order of elements in the array returned by `getDeclaredFields`. One fix strategy is to sort the fields in the array, e.g., by field name. The order then does not depend on the particular implementation of `getDeclaredFields`. However, the new order may differ from the old order (for a specific implementation), so some assertion may fail after sorting. We use ReAssert [12] to fix such failing assertions. We implement our strategies in a prototype tool, named DexFix.

This thesis makes several contributions:

- **Dataset:** We provide a publicly available dataset [33] of 275 tests (from 37 projects) that fail due to wrong assumptions made on underdetermined specifications. This dataset is the largest for NonDex, and shows the continued prevalence of this problem among open-source projects.

- **Debugging Support:** We extend the existing NonDex and ReAssert tools to provide more info for debugging tests that fail with NonDex. Specifically, for `HashMap`/`HashSet` iteration, we record where the object is allocated and, for failed string comparisons, we identify the likely kind of strings.

- **Strategies:** We derive novel strategies that can help to automatically repair the code exercised by the tests that fail with NonDex. Our new strategies are complemented by the existing ReAssert strategies for repairing tests [12]. We implement a prototype of our DexFix technique as an extension of NonDex and ReAssert.

- **Evaluation:** We apply our prototype to the 275 tests, and find that DexFix can propose fixes for 101 tests. After DexFix makes code changes, we check that the fix passes with NonDex. We have reported 84 of these fixes as GitHub pull requests from anonymous accounts, and 57 have been already merged, while only 2 have been rejected, and the remaining are still pending.

The rest of this thesis is structured as follows: Chapter 2 describes the NonDex and ReAssert techniques on which we build our work. Chapter 3 describes several examples how DexFix fixes faults with different fixing strategies. Chapter 4 describes the design and implementation of DexFix technique for automated fixing of tests that fail due to underdetermined specifications. Chapter 5 describes our experimental setup including how projects are selected, how we use NonDex to detect tests that have wrong assumptions for underdetermined specifications, how we use DexFix to propose fixes, and how we send those fixes as pull requests to developers. Chapter 6 describes the results of our experiment and how our evaluation answers the research questions. Chapter 7 describes limitations of DexFix and why DexFix cannot fix all the tests; it also discusses performance overhead of fixes proposed by DexFix. Chapter 8 discusses threats to the validity of our evaluation. Chapter 9 presents some related work. Chapter 10 discusses our visions of the future work. Chapter 11 concludes this thesis.

CHAPTER 2: BACKGROUND

Our tool, DexFix, automates fixing of tests with wrong assumptions on underdetermined specifications by extending the existing NonDex [13] and ReAssert [12] tools. In this chapter, we describe the relevant background of NonDex and ReAssert techniques, as needed to understand the rest of this thesis. Section 2.1 describes NonDex, and Section 2.2 describes ReAssert.

2.1 NONDEX

NonDex is a technique and tool for detecting tests that fail due to wrong assumptions on underdetermined specifications in the Java standard library [13, 34, 35]. NonDex detects such tests by modifying the Java standard library during class loading to randomize the output of several methods with underdetermined specifications [13]. NonDex utilizes an instrumentation engine that selects corresponding APIs and modifies them to add helper methods that explore different orders. NonDex is implemented as a Maven plugin [34, 35] that can be integrated into any Maven-based project, and runs using Java 8.

For example, when running tests with NonDex (invoked via command `mvn nondex:nondex` instead of the usual `mvn test`), the execution can invoke the `getDeclaredFields` method, which has an underdetermined specification that the returned array of fields is not in any particular order. At this point, NonDex performs a "random choice" and randomizes the order of the returned array of fields. A test may later fail due to this random choice, indicating that the the production code or test code made a wrong assumption on some underdetermined specification. NonDex runs tests for multiple rounds, where each round uses a different random seed, leading to different random choices and potentially detecting different tests. NonDex reports any tests that pass when run normally (without any random choice) but fail during one of these rounds.

NonDex also provides a debugging feature (invoked via the command `mvn nondex:debug`). Given a detected test that fails for some random seed, NonDex attempts to find the one random choice location that makes the test to fail [34]. NonDex uses binary search across all the random choices executed during the round where the test fails, localizing to the one point where a single random choice can make the test fail. NonDex then reports the stack trace of the single problematic random choice location.

2.2 REASSERT

ReAssert is a technique and tool for fixing test assertions [12, 28, 36]. When changes cause existing regression tests to fail, developers need to manually inspect and try to fix the test when the existing test fails to reflect the intended behavior. The goal of ReAssert is to propose fixes for tests that need to be updated after developers change the underlying production code (with the assumption that the production code is correct) and also retain as much of the original test logic as possible [36].

To illustrate how ReAssert works, consider the code adapted in Figure 2.1 from one of the examples from Daniel et al. [12]:

```java
public void testPenCoupon() {
    ShoppingCart shoppingCart = Utils.initEmptyShoppingCart();
    shoppingCart.addProduct(new Pen());
    shoppingCart.addProduct(new Pen());
    shoppingCart.addCoupon(new PenCoupon());
    assertEquals(5.0, shoppingCart.getTotalPrice());
    assertEquals(
        "Discount: -$5.00, total: $5.00",
        shoppingCart.getBill()
    );
}
```

Figure 2.1: Example test code that fails after developer makes changes

This code snippet in Figure 2.1 is a simplified test for a shopping application, and the developers aim to use this test to check if the coupon for pen is implemented correctly. The test `testPenCoupon` initializes an empty `ShoppingCart` object and then adds two pens and the coupon for pen. Initially, the store decides to provide a buy-one-get-one-free coupon for pens with a unit price of 5 dollars. So after adding two pens and the coupon, the total price of this cart is 5 dollars, as expected in this test. Later, the shop decides to replace this buy-one-get-one-free coupon to be a discount of \$3 if two pens are purchased. After the developer makes changes according to the requirements, the total price should be 7 dollars, and the discount 3, which means this test will fail because it no longer reflects the specification of the software application. At this point, developers need to fix the broken test by changing the total price and the string for bill (including the discount value) accordingly. This task may become particularly time-consuming when this new change breaks multiple tests. In practice, changes in requirements can happen relatively often. ReAssert aims to

```
public void testPenCoupon() {
    ShoppingCart shoppingCart = Utils.initEmptyShoppingCart();
    shoppingCart.addProduct(new Pen());
    shoppingCart.addProduct(new Pen());
    shoppingCart.addCoupon(new PenCoupon());
-   assertEquals(5.0, shoppingCart.getTotalPrice());
+   assertEquals(7.0, shoppingCart.getTotalPrice());
    assertEquals(
-     "Discount: -$5.00, total: $5.00";
+     "Discount: -$3.00, total: $7.00";
        shoppingCart.getBill()
    );
}
```

Figure 2.2: Changes proposed by ReAssert to fix the test

help developers in this scenario by assuming the production code is correct and tries to fix the test code. In this example, ReAssert proposes the code changes as in Figure 2.2 to fix the test.

Given a (failing) test, ReAssert first instruments assertion methods to dynamically record the expected and actual values of tests. For example, ReAssert instruments JUnit's assertion methods like `assertEquals` in class `Assert` to record values by replacing the exception with ReAssert's own exception with runtime values wrapped [12]. After instrumentation, ReAssert executes the test via standard JUnit by default, and uses the instrumented exception to get the expected and actual values for comparisons. ReAssert then applies several *fix strategies* to attempt to fix the assertion so the test no longer fails. For example, one strategy, `ReplaceLiteralInAssertion` (which is also used to fix the example test) works on assertions where the expected value is a literal (e.g., a string or an integer), and replaces that literal with the actual value it observes during the test execution [12].

CHAPTER 3: EXAMPLES

We next discuss several example tests for which (1) NonDex finds wrong assumptions on underdetermined specifications and (2) DexFix proposes fixes for those wrong assumptions. The examples show a variety of different root causes of underdetermined specifications and a variety of fix strategies used to change the code. Our GitHub pull requests, based on fixes proposed by DexFix, for all these examples have been accepted by the developers of the respective projects.

3.1 SIMPLE FIX IN PRODUCTION CODE

In this example test from Apache Hadoop [37], DexFix proposes a simple fix in the production code. Apache Hadoop is a widely used and popular open-source project with over 10K stars and 6K forks on GitHub. We first ran the NonDex tool on the commit 14cd969 of this project. Using `mvn nondex:nondex`, we detected several tests that contain wrong assumptions on some underdetermined specifications, which shows that even well-tested projects can have problems with underdetermined specifications.

One of the tests from Hadoop was `TestMetricsSystemImpl#testInitFirstVerifyCallBacks`. This test passes without the randomizations from NonDex (as discussed in Section 2.1), but after we turn on the randomizations, the test fails (for a number of random seeds) with an error message (as shown in Figure 3.1) that would be rather difficult to debug by itself.

```
java.lang.AssertionError:
Element 0 for metrics expected:<MetricCounterLong{info=MetricsInfoImpl
  {name=C1, description=C1 desc}, value=1}>
but was:<MetricGaugeLong{info=MetricsInfoImpl
  {name=G1, description=G1 desc}, value=2}>
```

Figure 3.1: Error message after running test `testInitFirstVerifyCallBacks` with NonDex

Fortunately, NonDex (via `mvn nondex:debug`) can provide debugging info [34] for each failing test, specifically the "root cause" random choice that affects the failure. From the debugging info provided by NonDex (as shown in Figure 3.2), we can observe that `getDeclaredFields` is used in line 353 in the file `ReflectionUtils.java`. Figure 3.3 shows the relevant code snippet from the `ReflectionUtils.java` file.

Since the specifications for `getDeclaredFields` states "the elements in the returned array are not sorted and are not in any particular order" (as discussed in Section 1.3), the list of `Field` objects returned by the method `getDeclaredFieldsIncludingInherited` can contain

```
java.lang.Class.getDeclaredFields(Class.java:1916)
org.apache.hadoop.util.ReflectionUtils.getDeclaredFieldsIncludingInherited(ReflectionUtils.
 java:353)
org.apache.hadoop.metrics2.lib.MetricsSourceBuilder.<init>(MetricsSourceBuilder.java:68)
org.apache.hadoop.metrics2.lib.MetricsAnnotations.newSourceBuilder(MetricsAnnotations.java
 :43)
[...]
```

Figure 3.2: Root cause location reported by NonDex debugging module

```java
import java.util.ArrayList;
import java.util.List;
...
public static List<Field> getDeclaredFieldsIncludingInherited (Class<?> clazz) {
   List<Field> fields = new ArrayList<Field>();
   while (clazz != null) {
       for (Field field : clazz.getDeclaredFields()) {
           fields.add(field);
       }
        clazz = clazz.getSuperclass();
   }
   return fields;
}
```

Figure 3.3: Problematic code snippet used by `testInitFirstVerifyCallBacks`

elements in arbitrary order, and tests that depend on this method can produce unreliable results. DexFix has a general, automated fix strategy for sorting arrays of fields such as those returned by `getDeclaredFields` (as discussed in Section 4.2.2). Based on the NonDex debugging output pertaining to the `ReflectionUtils` class, DexFix proposes the fix as shown in Figure 3.4. Rather than introducing its own sorting, DexFix uses the Java standard library classes `java.util.Arrays` and `java.util.Comparator` for sorting the fields. As a minor point, DexFix imports these classes when needed and adds them to the import block sorted by the fully qualified class names. Because the `Arrays#sort` method from the standard library sorts the input array in place and does not return the sorted array, DexFix cannot simply replace `clazz.getDeclaredFields()` with some inline code like `sort(clazz.getDeclaredFields())` in the `for` loop. Instead, DexFix introduces a fresh variable `sortedFields`, sorts the array (comparing fields by name) in place, and uses `sortedFields` in the `for` loop. After DexFix makes a change, we automatically compile and rerun the test with NonDex to check if it still fails; in this case, the test passed after the above fix.

We submitted this fix as a GitHub pull request [38]. In gory detail, the Hadoop change process requires first opening an issue ticket on Jira, which we did [39], before submitting a pull request on GitHub that references the Jira ticket. Having a particular process for

```
import java.util.ArrayList;
+ import java.util.Arrays;
+ import java.util.Comparator;
import java.util.List;
...
  while (clazz != null) {
-     for (Field field : clazz.getDeclaredFields()) {
+     Field[] sortedFields = clazz.getDeclaredFields();
+     Arrays.sort(sortedFields, new Comparator<Field>() {
+         public int compare(Field a, Field b) {
+           return a.getName().compareTo(b.getName());
+         }
+     });
+     for (Field field : sortedFields) {
         fields.add(field);
...
```

Figure 3.4: Fix for test `testInitFirstVerifyCallBacks` proposed by DexFix

submitting contributions was not specific to Hadoop, but we found many other projects having their specific requirements; in fact, it often took us *more* time to understand how to submit a fix to a project than it took for DexFix to propose the fix and for us to inspect that fix. The developer promptly accepted our fix with the message: "+1, committing. we all hate flaky tests. thanks for this" [38].

3.2 MULTIPLE CHANGES WITH THE SAME STRATEGY

In this example, DexFix proposes a fix with multiple changes in both production and test code, but all these changes follow the same strategy. In the Quarkus project [40], commit 84128ce, NonDex detected several tests that depend on some underdetermined specifications. One of the tests was `CompilerFlagsTest#defaulting`. When the test failed with randomizations from NonDex, it produced an error message that was not easy to debug, as shown in Figure 3.5. While NonDex's prior debugging output [34] provides some info for this failing

```
org.opentest4j.MultipleFailuresError:
org.opentest4j.AssertionFailedError: expected: <CompilerFlags
  @{-b, -a}> but was: <CompilerFlags@{-a, -b}>
org.opentest4j.AssertionFailedError: expected: <CompilerFlags
  @{-b, -a, -c, -d}> but was: <CompilerFlags@{-a, -b, -c, -d}>
[...]
at io.quarkus.dev.CompilerFlagsTest.defaulting(CompilerFlagsTest.java:25)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
```

Figure 3.5: Error message after running test `defaulting` with NonDex

test, unfortunately it does not provide enough info. Specifically, the prior NonDex debugging output provides that the "root cause" random choice (as shown in Figure 3.6) affects the failure. We can see that the failure stems from an iteration over a `HashMap` object, whose order is underdetermined, but the NonDex output did not provide (1) the code location that allocated this object and (2) whether it indeed allocated an object of the class `HashMap` or potentially of the class `HashSet` (whose internal implementation uses `HashMap`).

```
java.util.HashMap$HashIterator$HashIteratorShuffler.<init>(Unknown Source)
java.util.HashMap$HashIterator.<init>(HashMap.java:1435)
java.util.HashMap$KeyIterator.<init>(HashMap.java:1467)
java.util.HashMap$KeySet.iterator(HashMap.java:917)
java.util.HashSet.iterator(HashSet.java:173)
java.util.AbstractCollection.toArray(AbstractCollection.java:137)
java.util.ArrayList.addAll(ArrayList.java:581)
[...]
io.quarkus.dev.CompilerFlags.toList(CompilerFlags.java:44)
io.quarkus.dev.CompilerFlags.equals(CompilerFlags.java:68)
org.junit.jupiter.api.AssertionUtils.objectsAreEqual(AssertionUtils.java:189)
```

Figure 3.6: Root cause reported by NonDex for test `defaulting`

We have extended NonDex debug output to include the code location (as explained in Section 4.3). Our extension reported that the object was allocated on line 30 of the class `CompilerFlags`. This class comes from the file `CompilerFlags.java`. Figure 3.7 shows the relevant line of the allocation place for the HashSet. DexFix has a general automated strategy for replacing allocations of `HashMap`/`HashSet` with allocations of `LinkedHashMap`/`LinkedHashSet`. The `LinkedHash*` [1] classes have a precisely defined iteration order, e.g., the Javadoc specification [41] for `LinkedHashMap` states: "Hash table and linked list implementation of the `Map` interface, with predictable iteration order. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order)" and also "This implementation spares its clients from the unspecified, generally chaotic ordering provided by `HashMap`." Based on the debugging output from our NonDex extension, DexFix proposed the change as shown in Figure 3.8 (together with appropriate changes in `import`). Each `LinkedHash*` class is a subclass of its respective `Hash*` class, so after this change, the changed code can compile without further changes to the declared types (in this case of the field `this.defaultFlags`).

After DexFix makes a change and reruns the test with randomization from NonDex, unlike in the first example where the test passed after the first change, this test again failed (for a number of seeds). The new failure was again due to iteration over a `HashMap`, and our

[1] We will use `LinkedHash*` to refer to either `LinkedHashMap` or `LinkedHashSet`.

```
this.defaultFlags = defaultFlags == null ? new HashSet<>() :
   new HashSet<>()(defaultFlags);
```

Figure 3.7: The allocation place of the HashSet that causes the test failure reported by DexFix

```
- this.defaultFlags = defaultFlags == null ? new HashSet<>() :
       new HashSet<>(defaultFlags);
+ this.defaultFlags = defaultFlags == null ? new LinkedHashSet<> :
       new LinkedHashSet<>()(defaultFlags);
```

Figure 3.8: The first fix proposed by DexFix for test defaulting

extension reported that the object was allocated on line 41 of the same class CompilerFlags. DexFix then proposed the change as shown in Figure 3.9.

After this change, the test failed with NonDex yet again. The new failure was yet again due to iteration over a HashMap, and our extension reported that the object was allocated on line 88 of the class CompilerFlagsTest. DexFix yet again proposed the change as shown in Figure 3.10. After DexFix changed all these three lines (along with adding import statements), the test finally passed with NonDex. We submitted this fix as a GitHub pull request, "Make tests more stable by using LinkedHashSet for deterministic iterations" [42], and the developer accepted it with the message, "Merged, thanks!"

3.3 MULTIPLE CHANGES WITH DIFFERENT STRATEGIES

This example illustrates a case where DexFix proposes a fix that changes both production and test code, specifically updating a test assertion. In the Alibaba Fastjson project [43], commit d4a6271, NonDex detected several tests that depend on some underdetermined specifications, including WriteDuplicateType#test_dupType2. Similar to the previous example, the problem was with a HashMap iteration, specifically line 38 of the class WriteDuplicateType , and DexFix proposes the change as shown in Figure 3.11.

```
- Set<String> effectiveDefaultFlags = new HashSet<>(this.defaultFlags);
+ Set<String> effectiveDefaultFlags = new LinkedHashSe<>(this.defaultFlags);
```

Figure 3.9: Fix proposed by DexFix for test defaulting after the first fix

```
- return new HashSet<>()(Arrays.asList(strings));
+ return new LinkedHashSet<>()(Arrays.asList(strings));
```

Figure 3.10: Fix proposed by DexFix for test defaulting after the second fix

```
- HashMap<String, Object> obj = new HashMap<>();
+ HashMap<String, Object> obj = new LinkedHashMap<>();
```

Figure 3.11: Fix proposed by DexFix for test `test_dupType2`

After this change, when the test is rerun with NonDex, it fails with an assertion violation, but unlike previous cases where a test failed for *some* random seeds, this test fails for *all* seeds. This deterministic failure means that the test fails even without NonDex, indicating that the actual cause is likely not an underdetermined specification but some other reason. In particular, this test fails on line 44 of `WriteDuplicateType.java` (as shown in Figure 3.12), which compares two strings (we simplified some irrelevant parts with [...]).

```
Assert.assertEquals("[pre]\"@type\":[...],\"id\":1001[post]", text1);
```

Figure 3.12: The test assertion of `test_dupType2` that fails after the fix

At this point we run ReAssert [12] to fix the failing assertion. For these cases of `assertEquals` with a string literal, ReAssert replaces the expected string with the actual string that the test produces. Specifically, ReAssert generates the following change as shown in Figure 3.13.

```
- Assert.assertEquals("[pre]\"@type\":[...],\"id\":1001[post]", text1);
+ Assert.assertEquals("[pre]\"id\":1001,\"@type\":[...][post]", text1);
```

Figure 3.13: The new assertion proposed by ReAssert for `test_dupType2`

These two changes now make the test pass both with and without NonDex. We submitted this fix [44], and the developer merged it.

3.4 A NOVEL STRATEGY FOR JSON STRINGS

Our fourth example is a case where DexFix proposes a fix just for comparing JSON strings. On the project Nutz [45], commit 97745dd, NonDex detected several tests that depend on some underdetermined specifications, including `JsonTest#test_enum`. According to the NonDex debugging output, this test also has `getDeclaredFields` as the root cause, but in a location that is in a third-party library not in the Nutz project, which means DexFix cannot make changes directly around where `getDeclaredFields` is used. However, the failed assertion (on line 1031 of `JsonTest.java`) just compares a JSON string to a hard-coded string literal, i.e., string `expected`. As such, DexFix has a general, automated fix strategy to utilize a Java library, JSONAssert [46], to change such comparisons. JSONAssert is a designated library to assist in writing tests on JSON strings, and it provides library methods like `JSONAssert#assertEquals` that compares two JSON strings in a smart way that ignores ordering of elements in the JSON structure. DexFix proposed the following change with `JSONAssert#assertEquals`, as shown in Figure 3.14.

```
+ import static org.junit.Assert.fail;
+ import org.json.JSONException;
+ import org.skyscreamer.jsonassert.JSONAssert;
  ...
- assertEquals(expected, Json.toJson(TT.T)); // former line 1031
+ try {
+   JSONAssert.assertEquals(expected, Json.toJson(TT.T), false);
+ } catch (JSONException jse) {
+   fail("Not comparing JSON strings.");
+ }
```

Figure 3.14: The new assertion proposed by DexFix for `test_enum`

The parameter `false` instructs the method to ignore the ordering of fields in the JSON object. The change also requires wrapping the call in a `try-catch` block to fail when the assertion does not compare JSON strings. Another required change is to modify `pom.xml` to add the `org.skyscreamer.jsonassert-1.5.0.jar` dependency. Because this same `JsonTest` class had 4 similar failures, we (manually) extracted all `try-catch` blocks in a helper method, called `assertJsonEqualsNonStrict`, and replaced calls to `assertEquals` with calls to the helper method. These changes make all the tests pass both with and without NonDex. We submitted this fix [47], and the developers merged it: “thank you very much ^_^”.

CHAPTER 4: TECHNIQUE

We next describe our DexFix technique for automated fixing of tests that fail due to underdetermined specifications. The input to our technique conceptually consists of (1) the project source code including the production and test code, and (2) the debugging output from (our extension of) NonDex for a failing test. In general, NonDex can detect multiple test failures, but DexFix fixes them one at a time. The output of our technique is a fix, consisting of one or more code changes, that makes the test pass when run with NonDex—if DexFix cannot propose such a fix, it may still provide some code changes that may help developers to manually fix the code. Inspired by ReAssert [12], DexFix proceeds by applying various fix strategies and checking if the test passes with NonDex.

Section 4.1 presents an overview of DexFix. Section 4.2 describes the novel strategies that DexFix uses for tests that fail due to underdetermined specifications. Section 4.3 presents how our prototype implementation of DexFix, which extends both NonDex and ReAssert, adds new strategies, and also provides scripts to automate running DexFix at scale.

4.1 OVERVIEW

Figure 4.1 presents the pseudo-code of the DexFix top-level `repair` function that modifies a project's source code. The specific inputs of DexFix are the test that fails due to underdetermined specifications, the failing assertion reported by JUnit, the root cause reported by NonDex, and the debug location reported by our extension of NonDex.

DexFix first calls `repair_location` to attempt to repair the code location itself. If the location is in a library dependency and not in the source code of the project being analyzed, then DexFix cannot change the source code at that location (but could still change the test code later). Otherwise, if DexFix can change the source code, it checks whether the root cause is `HashMap`/`HashSet` or `getDeclaredFields` for which it can apply an appropriate strategy (as discussed in Section 4.2.1 and 4.2.2). If DexFix modifies the code, it checks whether the test fails, and if so, DexFix tries to update the failing test assertion by applying the traditional ReAssert strategies [12] (as illustrated in Section 3.3).

If `repair_location` does not repair the test, DexFix applies its JSONAssertEquals strategy (as discussed in Section 4.2.3). This strategy is the last resort because it adds a dependency to the project, and developers tend to be cautious about adding more dependencies. After all these changes, if the test passes without NonDex, DexFix runs the test with NonDex (for a configurable number of rounds). If the test fails with NonDex, DexFix uses the potentially

```
input/output: project source code that gets modified

inputs: failing test t, failing assertion a, root cause c, debug location l
output: status REPAIRED/UNREPAIRED

def repair(t, a, c, l):
  result = repair_location(t, c, l)
  if result == UNREPAIRED:
    apply_strategy(JSONAssertEqualsStrategy, t, a)
    if compile_and_run(t) == FAIL:
      return UNREPAIRED
  # by now the test (with the fix) passes without NonDex
  result, an, cn, ln = run_NonDex(t, NUM_ROUNDS)
  if result == FAIL:
    return repair(t, an, cn, ln)
  else:
    return REPAIRED

def repair_location(t, c, l):
  if l in library:
    return UNREPAIRED
  if c is Hash*:
    apply_strategy(HashToLinkedHashStrategy, l)
  elif c is getDeclaredFields:
    apply_strategy(SortFieldsStrategy, l)
  else
    return UNREPAIRED
  if compile_and_run(t) == FAIL:
    apply_strategy(ReAssertStrategies, t, a)
    if compile_and_run(t) == FAIL:
      return UNREPAIRED
  return REPAIRED
```

Figure 4.1: Pseudo-code of DexFix repair process

new failing assertion `an`, root cause `cn`, and debug location `ln` to recursively apply `repair` again, e.g,. as illustrated in Section 3.2. However, if the test passes with NonDex, it is finally considered fixed. We then proceed to manually inspect the fix and prepare a pull request.

4.2 STRATEGIES

We develop three new strategies for DexFix. The first two strategies focus on the most common root causes for underdetermined specifications. These strategies can apply to both the production and test code, depending on where `HashMap`/`HashSet` objects are allocated or where `getDeclaredFields` is called. The third strategy focuses on comparisons of strings that encode JSON objects. This strategy applies only to the test code. In brief, the high-level goal of our strategies is to make the output deterministic or to make the test assertion order-agnostic.

4.2.1 HashToLinkedHash Strategy

This strategy simply replaces allocation sites `new HashMap`, respectively, `new HashSet`, with `new LinkedHashMap`, respectively, `new LinkedHashSet`. The strategy applies when the root cause of a test failure is a random choice of iteration of some `HashMap`/`HashSet` object. While NonDex can provide the stack trace at the iteration point, it did not provide the stack trace of the allocation until we extended NonDex. A minor issue for this strategy is that sometimes it needs to add an appropriate `import` statement to use `LinkedHashMap` or `LinkedHashSet`, i.e., `import java.util.LinkedHashMap;` or `import java.util.LinkedHashSet;`.

4.2.2 SortFields Strategy

This strategy adds sorting of fields arrays returned by the method `getDeclaredFields`. The strategy applies when the root cause of a test failure is a random choice of elements returned in such an array. NonDex already provides the stack trace at the point where the method is invoked. If the method invocation is the only expression in a statement, e.g., `field = clazz.getDeclaredFields()`, then adding sorting is easier.

A more challenging issue is handling method invocations that appear in more complex expressions, e.g., as illustrated in Section 3.1. Our solution is to use a fresh variable to store the array, then sort it, and finally replace the original invocation with the new variable name. A minor issue is that this strategy may need to add two appropriate `import` statements for comparing and sorting, i.e., `import java.util.Arrays;` and `import java.util.Comparator;`.

4.2.3 JSONAssertEquals Strategy

This strategy replaces invocations of the JUnit's standard `Assert.assertEquals` with invocations of `JSONAssert.assertEquals` taken from the Skyscreamer JSONassert library [48], specifically version 1.5.0. This strategy is illustrated in Section 3.4. Replacing `Assert` with `JSONAssert` would be easy, but the issues with this strategy involve the need to provide an additional argument (`false`) to the invocation, and to handle the potential `JSONException`. This strategy only applies when the strings being compared represent JSON objects.

4.3 IMPLEMENTATION

We implement our DexFix technique in a prototype tool, also called DexFix. Our current implementation is an amalgam of our extensions to publicly available NonDex [34] and ReAssert [36] codebases, with some newly written code.

Our key modification to NonDex is the collection of additional debugging info. Specifically, for every allocation of a `HashMap`/`HashSet` object, our extension records the stack trace at the allocation point. When NonDex reports the stack trace at the iteration point where it performs its random choice, our extension also reports the stack trace at the allocation point of the object being iterated. Our extension then finds the code location from this stack trace by looking for the first stack frame whose source code is in the project being analyzed (and not in a library, either the Java standard library or some third-party library).

Our key modifications to ReAssert implement the JSONAssertEquals strategy and upgrade ReAssert to work with most of Java 8. For the JSONAssertEquals strategy, we reuse the prior ReAssert code for its `AssertEquals` strategy (illustrated in Section 3.3). The prior code already instruments tests to capture the expected and actual values for a string comparison, and the (test) code location that invokes the comparison. Our extension checks whether the strings are likely JSON strings, by the presence of the '`{`' characters and the fact that the expected and actual strings when sorted (just by character ordering, not considering any of the JSON format) should be equal. Unlike `AssertEquals` that just replaces a literal, our extension has to perform somewhat elaborate changes to replace the invoked method, add an argument, and add a `try-catch` block. While working with existing ReAssert code, we find that it uses an old version of the Spoon library [49] for parsing Java files, which does not support most modern Java 8 features; we update the Spoon dependency and appropriately modify the ReAssert code. However, the Spoon version that we use still does not support all Java 8 features (e.g., lambda expressions). Spoon does have even newer versions, but they substantially broke backwards compatibility, so trying to use those latest versions would require substantial rewriting of the prior ReAssert code.

Our key new additions implement the HashToLinkedHash and SortFields fix strategies. We used the javaparser library [50] to parse the input Java files (whether production and test code), modify the code, and print the file content with modifications. The javaparser library is much more modern than Spoon and supports all the latest features of Java 8 (We analyze Java 8 projects as we use NonDex, and Java 8 is the most common among popular Java projects). Our implementation directly follows the descriptions in Section 4.2 and the examples in Chapter 3. This part of our prototype is more robust than our extension of ReAssert. Last but not least, our new code includes several scripts that connect NonDex, ReAssert, and our fix strategies, to implement the overall DexFix technique and to allow us to run DexFix at scale in our experiments.

CHAPTER 5: EXPERIMENTAL SETUP

In this chapter, we first describe how we select projects for our evaluation and how we use NonDex [34, 35] to detect tests within these projects that fail due to wrong assumptions on underdetermined specifications. We then describe how we use DexFix to propose fixes for these detected tests and how we prepare pull requests for these proposed fixes to developers.

5.1 SELECTING PROJECTS AND DETECTING TESTS WITH NONDEX

For our evaluation, we use open-source Java projects that build using the Maven build system [51]; our requirement for Maven stems from the fact that NonDex currently only supports running tests for Maven-based projects. We queried GitHub to find the top 1000 Java projects by number of stars, then we choose 200 from the 242 projects that have a top-level `pom.xml` file used to configure Maven. We ran NonDex on all these 200 projects.

For each project, we use the latest commit as of September 2019. We create a separate Docker image for each project. Each Docker image has the cloned project (including the production and test code), installed using `mvn install -DskipTests`, and an installed version of our modified version of NonDex. We build all Java code using Java 8, also installed in the Docker image.

For each Docker image, we start a Docker container where we run NonDex detection on all tests using `mvn nondex:nondex`. We configure NonDex to run 10 rounds (with varying random seeds) for each project. We also configure NonDex to run using the "`ONE`" mode [13], where NonDex randomizes the order for each method with an underdetermined specification *only once* for the first call and then reuses that randomized order for subsequent calls (with the same receiver object). We choose the "`ONE`" mode because tests that fail in this mode most likely indicate real problems due to wrong assumptions that developers tend to be interested in fixing. This mode puts a lower bound on the number of tests NonDex can detect on these projects; in the "`FULL`" mode, NonDex could find even more test failures by randomizing *all* calls for methods with underdetermined specifications. We collect all the tests that fail with NonDex randomization but pass when run normally without NonDex.

For each detected test, we run `mvn nondex:debug` to obtain debugging info (specifically a file that we copy outside of the Docker container). The prior NonDex debugging reports a single method-call location where NonDex random choice leads the test to fail [34]. When the call iterates over a `HashMap`/`HashSet`, our NonDex extension also reports the location where that object is allocated (as discussed in Section 4.3).

During this process NonDex can find that some tests are flaky [5], i.e., pass or fail even when rerun for the same random seed, so we remove such tests. Also the prior NonDex debugging occasionally crashes altogether and produces no output, so we also remove such tests. Because there is no info about any method-call location, our extension cannot report where the receiver object is allocated. These crashes are infrequent and hard to reproduce, so we have not yet reported them to NonDex developers but plan to do so in the future.

5.2 FIXING TESTS USING DEXFIX

DexFix uses the debugging info from (our extended) NonDex to fix each test individually. For each test, we start a new Docker container based on the Docker image for the test's project. We copy into this container the debug file from NonDex and then run DexFix for the test. This procedure ensures that DexFix proposes fixes for each test when run on the same version of code where NonDex detected the test. An alternative would have been to fix tests one after another in the same container, but an issue is that a later test would have been run on a different version of the code that contains the fix for a prior test.

When DexFix needs to check if its proposed fix works, we configure it to run NonDex on the test for 10 rounds to check if the test, after the applied change, can fail for any of the NonDex rounds. If the test does fail in any of these rounds, DexFix then has to again use the info collected from the NonDex run to propose additional changes to the code (Section 4.1). This process continues until either DexFix generates a fix, or it reports that it still cannot fully repair the test after potentially making some changes.

After collecting the fixes for tests that DexFix can fix for a project, we inspect the fixes and prepare GitHub pull requests with those fixes to the developers of that project.

The fixes for different tests can contain the same or similar code changes, because we use DexFix to fix each test individually on the same version of code where NonDex detected the test (without using changes from the fix of one test as a starting point for fixing another test). If fixes for multiple tests have some same changes to the production code or the test code (even if they still have separate changes to their test assertions), these fixes can be safely combined, because they all address the same cause of non-determinism. Moreover, all the changes to test assertions need to be combined together along with the changes to the production code; otherwise, the tests will fail when run without NonDex. Some pull requests we send to the developers fix multiple tests at once and are a combination of fixes for these related tests, with all the fixes sharing the same changes to the production and test code, modulo changes to the test assertions. Some other pull requests simply fix only one test.

As we prepare a pull request, we manually make stylistic changes to make the code changes match the coding style of the surrounding code. When we send pull requests, for a new project that we have not yet sent pull requests, we send one pull request to that project for review. As the pull request remains pending, we do not send more because we do not want to overwhelm developers with pull requests that they may not have time to review. We only send additional pull requests once developers accept the initial one. Also, if a pull request is rejected, we take that feedback into account and do not submit other similar pull requests to that same project. We describe more of our results concerning when we do not send pull requests to developers in Section 6.3.

CHAPTER 6: EXPERIMENTAL RESULTS

Our evaluation aims to answer the following research questions:

RQ1: What is the breakdown of the root causes and debug locations for tests that fail with NonDex due to wrong assumptions on underdetermined specifications?

RQ2: How many tests can DexFix fix, and which fix strategies are the most effective?

RQ3: How effective is DexFix at proposing fixes that developers actually accept?

We address RQ1 to better understand the causes for failing tests so that we and others can tailor fix toward the most common causes. We address RQ2 to evaluate how effective DexFix is at proposing fixes for such tests and to understand which strategies are effective at fixing which root causes. We address RQ3 to evaluate whether developers accept the fixes that DexFix proposes. Our dataset and pull requests (submitted using anonymous GitHub accounts) are publicly available [33].

6.1 RQ1: ROOT CAUSES OF DETECTED TESTS

Among the 200 projects on which we run NonDex, NonDex detects 275 tests that fail due to wrong assumptions in 37 projects. Table 6.1 lists these 37 projects. The "ID" column shows the brief ID we give to each project for later reference. The "Commit" column is the Git commit SHA on which we run NonDex for each project. The remaining columns show the breakdown of the root causes and debug locations that (our extension of) NonDex reports for test failures. The "`Hash*`" column shows the number of tests due to iteration over an unordered `HashMap`/`HashSet` collection, hence the header "`Hash*`". The "`gDF`" column shows the number of tests due to calling `getDeclaredFields`. The "Rest" column shows the remaining tests, of which 15 are due to `getMethods`, and the remaining due to six various causes: `getAnnotationsByType`, `getDeclaredClasses`, `getDeclaredConstructors`, `getFields`, unordered `ConcurrentHashMap` iteration, and unordered `PriorityQueue` iteration. Note that these columns, under "Root Cause", show the *single* cause from the debugging file that `mvn nondex:debug` outputs on the *first* run, but a test may have *multiple* root causes, e.g., as illustrated in Section 3.2. The next two columns, under "Source?", show whether the debug location reported by our NonDex extension is in the project's source code (in either production or test code) or in a third-party library. The final column shows the total number of tests detected per project.

While the NonDex tool implements random exploration for over 40 methods with underdetermined specifications, only a small number of these methods cause most test failures. From

Table 6.1: Projects used in the study and breakdown of root causes and their locations

			Root Causes			Source?		
ID	Project	Commit	`Hash*`	`gDF`	Rest	Y	N	Σ
P1	apache/flink	23c9b5a	31	1	9	34	7	41
P2	alibaba/fastjson	d4a6271	27	-	-	21	6	27
P3	apache/hive	90fa906	15	10	-	11	14	25
P4	Graylog2/graylog2-server	87d63f6	12	11	-	9	14	23
P5	apache/commons-lang	7c32e52	-	21	-	21	-	21
P6	flowable/flowable-engine	399ab58	3	-	14	17	-	17
P7	apache/incubator-shardingsphere	038232e	15	-	-	15	-	15
P8	dropwizard/dropwizard	616ed86	6	3	-	6	3	9
P9	square/retrofit	8c93b59	-	9	-	-	9	9
P10	rest-assured/rest-assured	d3602d9	3	5	-	3	5	8
P11	alibaba/jetcache	d280196	6	-	-	6	-	6
P12	apache/hadoop	14cd969	2	4	-	4	2	6
P13	graphhopper/graphhopper	91f1a89	6	-	-	6	-	6
P14	abel533/Mapper	1764748	-	5	-	5	-	5
P15	apache/pulsar	505e08a	-	5	-	5	-	5
P16	nutzam/nutz	97745dd	-	5	-	5	-	5
P17	stanfordnlp/CoreNLP	08f6dca	5	-	-	5	-	5
P18	apache/avro	bfbd2d1	-	2	2	4	-	4
P19	ctripcorp/apollo	24062ad	1	-	3	3	1	4
P20	liquibase/liquibase	31a2256	4	-	-	1	3	4
P21	apache/kylin	31ab936	3	-	-	3	-	3
P22	kiegroup/optaplanner	dff7457	-	3	-	2	1	3
P23	vipshop/vjtools	60c743d	-	3	-	-	3	3
P24	Alluxio/alluxio	e6d7680	-	2	-	-	2	2
P25	eclipse/jetty.project	9cede68	2	-	-	-	2	2
P26	elasticjob/elastic-job-lite	b022898	-	1	1	-	2	2
P27	intuit/karate	2ca51ac	2	-	-	2	-	2
P28	quarkusio/quarkus	84128ce	2	-	-	2	-	2
P29	querydsl/querydsl	2bf234c	2	-	-	2	-	2
P30	seata/seata	d334f85	1	-	1	1	1	2
P31	OpenFeign/feign	744fd72	1	-	-	1	-	1
P32	classgraph/classgraph	d3b5aeb	-	1	-	1	-	1
P33	hs-web/hsweb-framework	9eb96c4	-	1	-	1	-	1
P34	mybatis/mybatis-3	0ca4860	1	-	-	1	-	1
P35	pedrovgs/Algorithms	ed6f8a4	1	-	-	1	-	1
P36	spring-cloud/spring-cloud-config	922590e	1	-	-	1	-	1
P37	zhangxd1989/spring-boot-cloud	e3966d7	-	1	-	-	1	1
Σ	-	-	152	93	30	199	76	275

the table, the majority of the detected tests fail due to some `HashMap`/`HashSet` (152 out of the total 275 tests). The second most common root cause is due to calling `getDeclaredFields` (93 out of the total 275 tests). Prior reports from running NonDex on older versions of open-source projects also found these two causes to be the most common [13, 34], but interestingly, the ranking between these two is reversed in our findings compared to prior work, which found the most common cause to be from calling `getDeclaredFields` [13, 34]. The difference in ranking is due to the differences in projects and versions, but the fact that these two causes remain the most common among detected tests increases confidence that our fix strategies for DexFix will remain general for fixing tests detected by NonDex.

Compared to prior work, the number of tests that NonDex detects in our experiment—275 failing tests in 37 out of 200 open-source projects—is greater than the previously reported number, e.g., Shi et al. [13] detected 60 failing tests in 21 out of 195 open-source projects. While we run on some much larger (multi-module) Maven projects, it could be also that the problem of tests due to underdetermined specifications is growing, so it is timely to develop techniques and tools such as NonDex and DexFix.

In terms of debug locations, we see that the majority are in the production and test code as opposed to third-party libraries. These numbers also increase confidence that our fix strategies for DexFix can be effective, because they mostly work on production and test code. While we have a strategy (the JSONAssertEquals strategy) that can work even if the location is in library code, it applies only in some cases (for JSON strings).

6.2 RQ2: FIXED TESTS

Table 6.2 shows the number of detected tests for which DexFix automatically proposes a fix. Overall, DexFix fixes a total of 101 tests (out of 275). We discuss the cases that DexFix cannot fix in Section 7.1. The table shows the breakdown of fixed tests per root cause that NonDex reports and the strategy that DexFix uses: "`L1`" uses HashToLinkedHash at only one allocation site; "`LM`" uses HashToLinkedHash strategy at multiple allocation sites; "`LA`" uses HashToLinkedHash strategy (once) and also updates some test assertion(s); "`JA`" uses JSONAssertEquals strategy (which applies to both top root causes); "`SF`" uses SortFields at only one site (we never observe a case where the strategy needs to change more than one site) without updating any test assertion; and "`SA`" uses SortFields and also updates some test assertion(s). In the table, the breakdown of the strategies is grouped based on the root cause for the tests to fail, either `Hash*` or `gDF`, which are the most common causes as we discussed in Section 6.1. Note that JSONAssertEquals applies to both top root causes.

From the table, we see most fixes that DexFix proposes require using only one Hash-

ToLinkedHash strategy on a single allocation site (42 out of 69 for the cause Hash$*$) or only one SortFields strategy for a getDeclaredFields invocation (16 out of 32 for the cause gDF), *without changing any test assertion*. The tests can still assert on the same expected values as before, but now the outcomes are deterministic and will not be affected by evolution of the implementation of the library methods in the future.

The JSONAssertEquals strategy helps in a number of cases, with a total of 16 tests fixed for both causes. This total highlights that tests often make incorrect assumptions by relying on JSON serialization. We see that most of the tests fixed this way are caused by getDeclaredFields (13). DexFix cannot fix these tests with SortFields, because the call to getDeclaredFields is in library code. Our analysis shows that these cases often stem from the project using a library to serialize objects into the JSON format. As such, it is understandable why most of the JSONAssertEquals fixes are for tests that use getDeclaredFields instead of HashMap/HashSet.

Another interesting point to note is the relatively high number of tests (12 for "LM") whose fixes involve using HashToLinkedHash on multiple allocation sites. This number highlights that the fixes for these tests are not always straightforward, hence DexFix needs to make multiple passes, running the debugging support multiple times to find all the relevant code locations where non-determinism can happen to make tests to fail.

For repair cost, DexFix takes relatively little time to apply its strategies and change the code for each test, because DexFix applies only targeted changes and runs each test at most 5-6 times (unlike program repair that may need to explore thousands of changes and run many tests hundreds or thousands of times [24]). Our regression testing runs DexFix on all 275 tests and takes under 7 hours, for an average of $\sim$90 seconds per test, with minimum of 32 seconds and maximum of 388 seconds.

6.3 RQ3: PULL REQUESTS

We have submitted pull requests for 84 out of 101 tests for which DexFix proposes a fix. Table 6.2 also shows the status of these pull requests. The status of a pull request can be Accepted ("A"), Pending ("P"), or Rejected ("R"); we also show Unsubmitted ("U").

Overall, developers have accepted pull requests that we submitted for majority of the tests for which DexFix proposes a fix. This high ratio of accepted pull requests shows that the fixes that DexFix proposes are effective, and developers welcome the changes, e.g., consider the messages in Chapter 3. There are 25 tests that still have their pull requests open on GitHub, because our proposed changes are still pending review or final judgment from developers. When we look into the breakdown of the tests in the accepted pull requests across the two

Table 6.2: Tests repaired, strategy used, and PR statistics

	Hash∗				gDF			PR				Σ
ID	L1	LM	LA	JA	SF	SA	JA	A	P	R	U	
P1	11	6	-	-	-	-	-	5	8	-	4	17
P2	9	-	7	-	-	-	-	5	2	-	9	16
P3	1	-	-	-	1	-	-	-	2	-	-	2
P4	2	-	-	-	-	-	8	10	-	-	-	10
P5	-	-	-	-	6	-	-	6	-	-	-	6
P6	1	-	-	-	-	-	-	1	-	-	-	1
P7	1	4	-	-	-	-	-	5	-	-	-	5
P8	3	-	-	-	-	-	-	3	-	-	-	3
P10	-	-	-	3	-	-	2	4	1	-	-	5
P11	2	-	-	-	-	-	-	2	-	-	-	2
P12	-	-	-	-	2	-	-	2	-	-	-	2
P13	2	-	4	-	-	-	-	-	-	2	4	6
P14	-	-	-	-	3	2	-	-	5	-	-	5
P16	-	-	-	-	3	1	-	4	-	-	-	4
P18	-	-	-	-	1	-	-	-	1	-	-	1
P20	1	-	-	-	-	-	-	1	-	-	-	1
P21	3	-	-	-	-	-	-	3	-	-	-	3
P23	-	-	-	-	-	-	3	-	3	-	-	3
P28	-	2	-	-	-	-	-	2	-	-	-	2
P29	2	-	-	-	-	-	-	-	2	-	-	2
P30	1	-	-	-	-	-	-	1	-	-	-	1
P31	-	-	1	-	-	-	-	1	-	-	-	1
P34	1	-	-	-	-	-	-	1	-	-	-	1
P35	1	-	-	-	-	-	-	-	1	-	-	1
P36	1	-	-	-	-	-	-	1	-	-	-	1
Σ	42	12	12	3	16	3	13	57	25	2	17	101

root causes, the numbers are roughly similar: 36 of 69 for Hash∗, and 21 of 32 for gDF.

Unfortunately, we have 2 tests whose pull requests have been Rejected. The developers of this project, graphhopper/graphhopper (P13), rejected one of our pull requests because the fix uses JSONAssertEquals, which adds a dependency on the JSONassert library. The developers did not want to include that dependency, commenting that "We never had a problem with this test and so I would not want to change it. Especially when we need a big dependency for something small." Such feedback shows that, while JSONAssertEquals effectively modifies just the test code that compares JSON strings, the cost of including a dependency on a new library is too high for some developers. For the other pull request, the fix was from HashToLinkedHash, but the developers did not provide any feedback before rejecting, so we do not know their reason for rejection.

Concerning the Unsubmitted fixes, we have not submitted fixes for 17 tests for two reasons. First, two tests—`com.graphhopper.routing.ch.CHProfileSelectorTest#onlyEdgeBasedPresent` and `com.graphhopper.routing.ch.CHProfileSelectorTest#onlyNodeBasedPresent`—that DexFix fixes on an older code version (on which we started our experiments) in graphhopper/graphhopper (P13) do not even depend on underdetermined specifications in the latest code version. The developers substantially restructured their code in a commit [52] and (1) instead of using `HashMap` for a configuration, switched to use a different data structure, and (2) also changed the test assertion to no longer compare a string to a hard-coded constant, specifically, the test was failing with expected value "`{weighting=fastest, vehicle=car}`" that could get the actual value "`{vehicle=car, weighting=fastest}`".

Second, the remaining 15 fixes that DexFix proposes are unlikely to get positive responses, and we choose not to send and bother developers. For apache/flink (P1), we do not send pull requests for 4 fixes because the first one we sent is still pending review. For alibaba/fastjson (P2), our manual inspection shows that the 9 proposed fixes do not make sense in the context of how the test is run, so we do not feel comfortable sending out a pull request. Finally, for graphhopper/graphhopper (P13), where we have 2 rejected fixes, the remaining two fixes for the project are very similar to the rejected ones, so we do not send.

CHAPTER 7: DISCUSSION

We discuss limitations and overhead for DexFix. First, DexFix has limitations and cannot fix every test detected by NonDex. Second, the fix strategies that DexFix uses may introduce additional overhead into the production and test code.

7.1 LIMITATIONS

DexFix currently cannot fix 174 out of 275 tests detected by NonDex. We have inspected most of these cases, including at least one unfixed test from each project that has unfixed test(s). Some of the cases arise from engineering deficiencies of our current prototype, some are limitations on our fix strategies, and some are rare enough that they do not merit developing general strategies.

In terms of tool engineering, ReAssert crashes when run on 42 tests. ReAssert cannot run on 28 tests that use JUnit 5 or TestNG [53], as ReAssert currently supports only JUnit 3 and JUnit 4. Our attempt to upgrade ReAssert to support JUnit 5 revealed that this would require a major re-implemenation effort. Further, ReAssert does not work on 23 tests because they use assertions from the popular AssertJ library [54], which supports "fluent assertions" that use `assertThat` and are easier to read as English text than code. Again, the ReAssert code that we obtained (version 0.4.1) does not support this style of assertion; a published tool paper [36] claims that some JUnit's `assertThat` methods are supported by ReAssert, but ReAssert does not support AssertJ's `assertThat` methods. Finally, our toolset also does not handle source languages beyond Java; for example, 6 tests are written in Scala.

In terms of fix strategies, the current DexFix strategies focus on addressing the two top root causes for tests to fail due to wrong assumptions on underdetermined specifications—iterating over `HashMap`/`HashSet` and the order of fields returned in `getDeclaredFields`—as reported by prior work [13] and confirmed in our experiments (Section 6.1). We find 30 tests fail for 7 other root causes which form a small minority among all the detected tests, and it is generally not worthwhile to develop "one-off" strategies for each cause. The largest number of tests, 15, is for `getMethods`, but they come from only two projects. We inspect all 14 tests in flowable/flowable-engine (P6), and they all fail because a class has two methods called `equals` (one method declared in the class itself and another method inherited from `java.lang.Object`). We could easily build a new "strategy" to sort these methods by name, but it would not be widely applicable. In fact, the project has a comment "By convention, the implementing class should have one method with the same name" [55]. We do not want to add a specialized strategy to inflate the number of fixed tests. The remaining test for `getMethods` is interesting in that its root cause is not just one random choice but two random choices. However, the fix would again be specialized to this one case.

Two of DexFix's strategies only apply where the source code is accessible, and only the JSONAssertEquals strategy can apply if the cause is in a third-party library. There are 28 tests that have causes in a library but JSONAssertEquals does not apply (not a JSON string comparison). We do not currently handle these cases.

Finally, DexFix cannot handle 17 tests due to one-off instances, such as creating `HashMap` through reflection or using parts of the Java language that are not properly handled by the tooling DexFix relies on (javaparser or ReAssert). Again, developing a specific fix strategy for these one-off cases would be too specialized for the effort needed.

7.2 OVERHEAD

It is interesting to consider whether the three new strategies that DexFix provides (Section 4.2) should be applied in *all* cases, even if it currently does not fail any test: should all `Hash*` objects be `LinkedHash*`, should all arrays from `getDeclaredFields` be sorted, and should all JSON strings be compared with something like `JSONAssert.assertEquals`? In the limit, did Java standard library developers make a mistake by having some methods with underdetermined specifications?

While our strategies change code to make tests not fail due to wrong assumptions on underdetermined specifications, the changes can introduce other side-effects to the production and test code, in particular with introducing some extra overhead in execution.

Compared to `Hash*`, `LinkedHash*` objects provide a small overhead in both space (as `LinkedHash*` objects need to maintain a list in addition to a hashtable maintained by `Hash*`) and runtime (to manipulate the list) for most operations. However, that overhead is negligible for all applications outside microbenchmarks; and moreover some operations on `LinkedHash*` can be faster [56], including iteration, which becomes not only predictable but also faster, as well as resizing or `containsValue` method. In retrospect, the developers of the Java standard library could have specified that all hash maps behave like `LinkedHashMap`, but it would create a somewhat higher overhead even in cases where deterministic iteration is not required. In Python, all dictionaries since version 3.7 are guaranteed to behave similarly to a `LinkedHashMap`, while the older versions behaved like `HashMap` [57]. Some Java developers still raise concerns about these overheads, e.g., one of our pull requests had a discussion about it [58], but later the developer still accepted the fix.

Sorting `getDeclaredFields` provides an even smaller overhead as the library typically does this only once and then caches for later calls. In contrast to Java, Python's interface for reflection involves a `__dict__` attribute that returns a dictionary with attributes as keys, a global `dir` function that returns a list of members, and the `inspect` module whose `getmembers` function returns a list of members. However, since Python 3.7, the latter is more precisely specified: "`inspect.getmembers(object[...])` Return all the members of an object in a list of `(name, value)` pairs sorted by name."

Finally, using `JSONAssert.assertEquals` would affect only the test code and not the production code, so the overhead is even less important, while providing for deterministic failures and easier debugging.

CHAPTER 8: THREATS TO VALIDITY

In this chapter, we discuss why our overall results may not generalize to all projects, beyond those used in our evaluation. Our evaluation uses a diverse set of popular projects from GitHub but, due to limitations of the existing tooling on top of which we develop DexFix, all projects are written primarily in Java and build using the Maven build system. However, the 200 projects for our evaluation are among the most popular Java projects on GitHub, and we believe they are fairly representative of all Java projects.

DexFix itself may have bugs that affect our results. To reduce this threat, we have developed DexFix on top of existing tools, NonDex [34] and ReAssert [36], which have been used in prior research. Furthermore, several collaborators of the thesis author reviewed the new code and discussed the proposed fixes.

The tests for which NonDex detected failure due to wrong assumptions regarding underdetermined specifications are true positives since the tool actually observes these failures, and we can reproduce all 275 cases used in our evaluation. In fact, the number of tests detected in our evaluation is a lower-bound on the true number of tests that could fail due to underdetermined specifications in these projects. Most importantly, the key threat is the quality of the fixes proposed by DexFix. We confirm that these fixes actually do provide useful contributions by sending the fixes as pull requests to developers, allowing them to make the final judgment call.

CHAPTER 9: RELATED WORK

9.1 REPRODUCIBILITY AND RERUN/REPLAY

Differing library implementations can have consequential impacts on software due to the potential for unanticipated non-determinism in production code. In scientific applications, for example, there is a need to detect and fix non-determinism in order of operations [59, 60, 61]. Unanticipated changes in the results of code can cast doubt on the scientific accuracy or the utility of a result [62]. Some applications, e.g., n-body system simulations, may require bit-wise numerical reproducibility [59]. This work advances the state of the art in scientific reproducibility by introducing methods to reduce non-determinism in repeated code executions, a general interpretation of reproducibility in scientific research [63, 64, 65, 66]. In addition, efforts to advance scientific reproducibility have recently focused on the expanded use of software testing [67].

9.2 DETECTING FLAKY TESTS

Mora et al. [68] proposed the concept of client-specific equivalence, when two library versions are equivalent with respect to a specific client, to study how changes in upstream library code affect downstream clients. Shi et al. [13] studied when tests fail due to wrong assumptions on underdetermined specifications, and they developed NonDex to detect such tests. They focused on specifications for methods, from the Java standard library, whose actual implementation may not guarantee a deterministic outcome, e.g., the iteration order over a `HashSet/HashMap` is not guaranteed to be in any specific order. Our work focuses on an automated fix for such tests, which is carried out by modifying both the production and test code.

The tests NonDex detects can be considered as a type of flaky tests, which are tests that non-deterministically pass or fail on the same version of production code [5]. In the case of tests that fail due to wrong assumptions with underdetermined specifications, they can fail on the same version of code after the library they depend on (in this case the Java standard library) gets updated. By studying open-source projects, Luo et al. [5] found that one reason for flaky tests is reliance on unordered collections. NonDex detects flaky tests of this category among other tests that fail due to wrong assumptions on underdetermined specifications [13, 34]. Other prior work focused on detecting different types of flaky tests [6, 7, 8, 9, 10, 11, 69].

9.3 AUTOMATED PROGRAM REPAIR

Automated program repair aims to generate patches for fixing bugs in code in an automated manner [20, 21, 22, 23, 24, 25, 26]. Given that these techniques rely on test outcomes to guide them, they usually do not aim to fix the test code but focus on fixing the production code. In this thesis, we develop DexFix to fix tests that fail due to wrong assumptions, and our fix strategies change both the production and test code.

In contrast, there is prior work focused on fixing test code [12, 28, 29, 30, 31], specifically repairing tests that become outdated when code evolves. In our work, we develop DexFix to fix flaky tests, making sure they do not fail due to reasons unrelated to changes in the code in the future. To fix flaky tests, Shi et al. [27] proposed using a technique named iFixFlakies. iFixFlakies fixes order-dependent flaky tests, relying on the insight that the necessary code for such tests is often found in the test suite itself. Our technique DexFix aims to fix flaky tests with wrong assumptions on underdetermined specifications. Furthermore, our way of fixing such flaky tests is not restricted to changes to the test code but sometimes also involves making changes to the production code as well. We utilize ReAssert [12] to automatically repair assertions that have to be updated after changes are made to the production code to make the computations deterministic.

CHAPTER 10: FUTURE WORK

One potential line for future work is to speed up NonDex itself. To detect tests that have wrong assumptions on underdetermined specifications, we ran projects' regression test suites with the public version of NonDex tool [34] to conduct random shuffling for certain underdetermined methods. To fully utilize the features from NonDex, each test from the test suite needs to be run multiple times. According to the original NonDex study [13], a user of NonDex needs to run each test ten or more times with different random seeds from NonDex, to have high confidence of finding all the tests with wrong assumptions. The shuffling by NonDex also adds an overhead to each run. For example, for method `getDeclaredFields`, NonDex shuffles and caches the order of the returned array when `getDeclaredFields` is called the first time, if NonDex is in `ONE` mode. If NonDex is in `FULL` mode, NonDex could further increase the overhead because NonDex shuffles the order every time when the method is called. In addition, for large-scale projects, running entire test suite itself is already very costly [4], let alone running multiple times with random shuffles. So, although DexFix can propose fixes in a relatively short time (as discussed in Section 6.2), the high cost of detecting tests that have wrong assumptions may prevent developers from using NonDex to detect tests and then DexFix to fix flaky tests in their test suite. In the future, one could add a new module which performs test selection to select tests that indeed execute some random shuffles from NonDex in the first round of running each test, so that after the first round, tests that do not execute random shuffles should not be run because running them cannot help detect flaky tests that have wrong assumptions on underdetermined specifications.

In addition, even if many tests execute methods with underdetermined specifications, those tests may not produce different outcomes if the underlying implementation changes. For example, a user of `getDeclaredFields` may just use the `Field` objects acquired to check if a specific field name exists in the specific class, which means the order does not affect the test outcome. Sophisticated static analysis techniques can be implemented to reason if a test will be affected by a different implementation of underlying library methods, which hopefully reduces the size of test suite before running any test.

In the future, one could also add new strategies to handle other underdetermined methods, in addition to the existing strategies for `HashMap/HashSet` iterations and `getDeclaredFields`. Previously Shi et al. [13] found 31 methods that have underdetermined specifications, from 14 classes in the Java standard library. From a high-level perspective, our strategy to sort the `Field` array returned by `getDeclaredFields` should also apply to, for example, `getMethods`, but instead of sorting the array of `Field` objects, the strategy will sort the array of `Method` objects. Inspired by our JSONAssertEquals strategy (as described in Section 4.2.3), another interesting extension to DexFix's fix strategies can be made to fix tests that compare other popular string formats, e.g., XML [70].

CHAPTER 11: CONCLUSIONS

Software testing is an important practice to increase the quality of software by executing tests to evaluate if software behaves correctly according to the software specifications which guide the implementations. Regression testing is a popular software testing approach whose main assumption is that a test which fails after a new change but passes before the change indicates faults introduced by the change. In practice, however, this assumption may not always hold because test outcomes can be unreliable. A test that produces unreliable outcomes is often called flaky, i.e., passes or fails without changes made to the code, which makes the test less useful to detect regression. One cause of unreliable test output is wrong assumptions on underdetermined specifications. Such specifications, which allow software to produce different outputs given the same input, are important to provide library developers flexibility for their implementations, but library users may incorrectly assume specific deterministic behavior, e.g., the output of the underlying library is always the same given a specific input, which results in unreliable test outcomes. In the literature, a previous approach, NonDex, has been successful to detect regression tests that have wrong assumptions on underdetermined specifications but, to the best of our knowledge, there has not yet been tools that can generate automated fixes for those tests.

This thesis presents the DexFix approach to propose automated fixes for tests that fail due to wrong assumptions on underdetermined specifications. We run the public version of NonDex tool on 200 open-source Java projects and detect 275 tests that fail due to wrong assumptions, and where NonDex provides a specific root cause. We observe that the major root causes are from the `HashMap`/`HashSet` class iterations and the `getDeclaredFields` method. Additionally, we identify several tests that fail due to test assertions that compare JSON strings. Based on the observations, we have implemented a prototype, also named DexFix, which extends the work on program and test repair with novel, simple, yet effective fix strategies that can, in an automated way, fix wrong assumptions on underdetermined specifications. Unlike most prior work that focuses on fixing exclusively either the production code or test code, DexFix can fix both as necessary, sometimes needing to change both. We apply our prototype on the 275 tests, and find that DexFix can propose fixes for 101 tests.

In addtion, this thesis also studies how effective those fixes proposed by DexFix are, i.e., how often developers accept fixes by DexFix. The empirical results are encouraging: we have reported fixes proposed by DexFix as GitHub pull requests, and 57 have already been merged, with only 2 rejected, and the remaining pending.

REFERENCES

[1] J. M. McQuade, R. M. Murray, G. Louie, M. Medin, J. Pahlka, and T. Stephens, *Software is never done: Refactoring the acquisition code for competitive advantage.* Defense Innovation Board, 2019.

[2] P. Ammann and J. Offutt, *Introduction to software testing.* Cambridge University Press, 2008.

[3] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification & Reliability*, 2012.

[4] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *ESEC/FSE*, 2015.

[5] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *ESEC/FSE*, 2014.

[6] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *ICSE*, 2018.

[7] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing," in *ICSE*, 2017.

[8] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *ICSE*, 2015.

[9] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST*, 2019.

[10] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe Java test acceleration," in *ESEC/FSE*, 2015.

[11] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *ISSTA*, 2014.

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

[13] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *ICST*, 2016.

[14] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *ICSME*, 2015.

[15] B. Liskov and J. Guttag, *Program development in Java: Abstraction, specification, and object-oriented design.* Addison-Wesley Professional, 2000.

[16] "java.lang.Class#getDeclaredFields Javadoc," https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getDeclaredFields, 2020.

[17] "Java SE 7 and JDK 7 compatibility," https://www.oracle.com/technetwork/java/javase/compatibility-417013.html#jdk77, 2020.

[18] "JUnit and Java 7," http://intellijava.blogspot.com/2012/05/junit-and-java-7.html, 2012.

[19] "JUnit test method ordering," http://www.java-allandsundry.com/2013/01, 2013.

[20] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *ESEC/FSE*, 2017.

[21] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012.

[22] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009.

[23] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *ICSE*, 2013.

[24] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys*, 2018.

[25] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA*, 2019.

[26] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ICSE*, 2018.

[27] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *ESEC/FSE*, 2019.

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

[29] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Supporting test suite evolution through test case adaptation," in *ICST*, 2012.

[30] G. Yang, S. Khurshid, and M. Kim, "Specification-based test repair using a lightweight formal method," in *FM*, 2012.

[31] X. Li, M. d'Amorim, and A. Orso, "Intent-preserving test repair," in *ICST*, 2019.

[32] "Introducing JSON," https://www.json.org/json-en.html, 2020.

[33] "DexFix: Automated fixing of wrong assumptions on underdetermined specifications," https://sites.google.com/view/dexfix, 2020.

[34] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov, "NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications," in *FSE DEMO*, 2016.

[35] "TestingResearchIllinois/NonDex: A tool for finding assumptions on APIs with underdetermined specifications," https://github.com/TestingResearchIllinois/NonDex, 2020.

[36] B. Daniel, D. Dig, T. Gvero, V. Jagannath, J. Jiaa, D. Mitchell, J. Nogiec, S. H. Tan, and D. Marinov, "ReAssert: A tool for repairing broken unit tests," in *ICSE DEMO*, 2011.

[37] "apache/hadoop: Apache Hadoop," https://github.com/apache/hadoop, 2020.

[38] "HADOOP-16897. Sort fields in ReflectionUtils.java," https://github.com/apache/hadoop/pull/1868, 2020.

[39] "[HADOOP-16897] Sort fields in ReflectionUtils.java - ASF JIRA," https://issues.apache.org/jira/browse/HADOOP-16897, 2020.

[40] "quarkusio/quarkus: Quarkus: Supersonic subatomic Java," https://github.com/quarkusio/quarkus, 2020.

[41] "LinkedHashMap Javadoc," https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html#LinkedHashMap, 2020.

[42] "Make tests more stable by using LinkedHashSet for deterministic iterations," https://github.com/quarkusio/quarkus/pull/6839, 2020.

[43] "alibaba/fastjson: A fast JSON parser/generator for Java," https://github.com/alibaba/fastjson, 2020.

[44] "Use LinkedHashMap for deterministic iterations," https://github.com/alibaba/fastjson/pull/2996, 2020.

[45] "nutzam/nutz: Nutz – Web framework(mvc/ioc/aop/dao/json) for all Java developer," https://github.com/nutzam/nutz, 2020.

[46] "skyscreamer/JSONassert: Write JSON unit tests in less code. Great for testing REST interfaces," https://github.com/skyscreamer/JSONassert.

[47] "Make test more stable by using JSONAssert equals," https://github.com/nutzam/nutz/pull/1541, 2020.

[48] "JSONAssert - Write JSON unit tests with less code," http://jsonassert.skyscreamer.org, 2020.

[49] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of Java source code," *Software: Practice and Experience*, 2016.

[50] “JavaParser,” http://javaparser.org, 2020.

[51] “Maven,” https://maven.apache.org, 2020.

[52] “Select appropriate CH profile when no weighting or vehicle is given,” https://github.com/graphhopper/graphhopper/commit/524aaeab96bb778936120de957d787e91d51cf47, 2020.

[53] “TestNG,” https://testng.org/doc, 2020.

[54] “AssertJ - fluent assertions Java library,” https://assertj.github.io/doc, 2020.

[55] “AbstractFlowableShortHandExpressionFunction.java,” https://github.com/flowable/flowable-engine/blob/9b647e3c9a10ee28d8f290f6ea651aa478346860/modules/flowable-engine-common/src/main/java/org/flowable/common/engine/impl/el/function/AbstractFlowableShortHandExpressionFunction.java#L96, 2020.

[56] “How is the implementation of LinkedHashMap different from HashMap?” https://stackoverflow.com/questions/3020601, 2020.

[57] “Why is dictionary ordering non-deterministic?” https://stackoverflow.com/questions/14956313, 2020.

[58] “Use LinkedHashMap for deterministic iterations,” https://github.com/OpenFeign/feign/pull/1165, 2020.

[59] D. H. Bailey, R. Barrio, and J. M. Borwein, “High-precision computation: Mathematical physics and dynamics,” *Applied Mathematics and Computation*, 2012.

[60] N. Honarmand and J. Torrellas, “Replay debugging: Leveraging record and replay for program debugging,” *SIGARCH Computer Architecture News*, 2014.

[61] D. Chapp, K. Sato, D. H. Ahn, and M. Taufer, “Record-and-replay techniques for HPC systems: A survey,” *Supercomputing Frontiers and Innovations*, 2018.

[62] A. Dakkak, C. Li, J. Xiong, and W.-M. Hwu, “Frustrated with replicating claims of a shared model? A solution,” 2018.

[63] D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden, “Reproducible research in computational harmonic analysis,” *Computing in Science Engineering*, 2009.

[64] V. Stodden, J. M. Borwein, and D. H. Bailey, ““Setting the default to reproducible” in computational science research,” *SIAM News*, 2013.

[65] V. Stodden, M. McNutt, D. H. Bailey, E. Deelman, Y. Gil, B. Hanson, M. A. Heroux, J. P. Ioannidis, and M. Taufer, “Enhancing reproducibility for computational methods,” *Science*, 2016.

[66] V. Stodden, M. S. Krafczyk, and A. Bhaskar, "Enabling the verification of computational results: An empirical evaluation of computational reproducibility," in *P-RECS*, 2018.

[67] M. Krafczyk, A. Shi, A. Bhaskar, D. Marinov, and V. Stodden, "Scientific tests and continuous integration strategies to enhance reproducibility in the scientific software context," in *P-RECS*, 2019.

[68] F. Mora, Y. Li, J. Rubin, and M. Chechik, "Client-specific equivalence checking," in *ASE*, 2018.

[69] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *ICST*, 2018.

[70] "Extensible markup language (XML)," https://www.w3.org/XML, 2016.