Domain-Specific Fixes for Flaky Tests with Wrong Assumptions on Underdetermined Specifications

Peilun Zhang[1], Yanjie Jiang[2], Anjiang Wei[3], Victoria Stodden[1], Darko Marinov[1], August Shi[4]
[1]University of Illinois at Urbana-Champaign, USA [2]Beijing Institute of Technology, China
[3]Peking University, China, [4]The University of Texas at Austin, USA
Email: peilunz2@illinois.edu, yanjiejiang@bit.edu.cn, weianjiang@pku.edu.cn,
vcs@stodden.net, marinov@illinois.edu, august@utexas.edu

***Abstract*—Library developers can provide classes and methods with underdetermined specifications that allow flexibility in future implementations. 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. Prior work proposed the NonDex approach that detects such wrong assumptions.**

We present a novel approach, called *DexFix*, to repair wrong assumptions on underdetermined specifications in an automated way. We run the NonDex tool on 200 open-source Java projects and detect 275 tests that fail due to wrong assumptions. The majority of failures are from iterating over `HashMap`/`HashSet` collections and the `getDeclaredFields` method. We provide several new repair strategies that can fix these violations in both the test code and the main code. DexFix proposes fixes for 119 tests from the detected 275 tests. We have already reported fixes for 102 tests as GitHub pull requests: 74 have been merged, with only 5 rejected, and the remaining pending.

I. Introduction

Underdetermined specifications [51] 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 [23] 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."

Library developers sometimes provide methods with such underdetermined specifications to allow flexibility for future implementations. The latest 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 source code, but this ordering could change in the future.

If library users write code that relies on a specific (deterministic) *implementation* rather than on the (underdetermined) *specification* of a method from the library, the code can break when the library developers provide a new implementation of the same specification. For example, the Java standard library (from Sun 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`, which on several occasions broke substantial amounts of code [1], [2].

Shi et al. [59] developed NonDex, a technique to find the tests that fail due to making wrong, deterministic assumptions on underdetermined specifications. Lam et al. [48] recently used NonDex in a larger study of flaky tests, which can fail seemingly nondeterministically [53]. They reported that 190/684 of the flaky tests in their dataset are due to such wrong assumptions. They call these flaky tests *implementation-dependent (ID) tests*; we use the same name in this paper. Gyori et al. [41] provided a NonDex plugin for the Maven build system [28] to automate running tests with NonDex and also provided some *partial* support for manual debugging by locating one, or a few, random choice(s), which they call *root causes*, that can make each test fail. They wrote "In the future, we plan to explore [...] automated fixing" [41, p. 4]. However, no automated fixing technique has been developed before.

We present a novel technique, called *DexFix*, that can automatically propose fixes for ID code. Inspired by the growing body of work on program repair [40], [44], [49], [52], [57], [61]–[63] (including a survey paper [55]) and test repair [37], [38], [50], [54], [60], [64], we provide a set of novel, *domain-specific*, and simple (but effective!) repair strategies that can fix implementation dependency in *both* the test code and the main code[1]. To the best of our knowledge, no existing program or test repair tools can handle these cases.

We first perform a formative study. We run the NonDex tool on 200 open-source Java projects and detect 275 ID tests where NonDex provides a specific root cause. Our inspection of these root causes finds that the vast 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 different order of fields; the JSON specification [22] does not specify the order of fields.

We derive new, automated repair strategies that can fix the failing tests by changing the code to work properly 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

[1]Following Maven, we use the term "main code" for what may be called "production code" or "code under test".

because an assertion expects a particular order of elements in the array returned by `getDeclaredFields`. One repair 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 assertions may fail after sorting. We can then apply a test-repair technique to automatically repair these assertions that now fail when run on a more deterministic implementation. Another repair can be to change the assertions so that they are order-agnostic, e.g., treat fields as a set rather than an array.

We automate our strategies by building upon NonDex and ReAssert. The latter is a tool that can automatically repair failing test assertions [38]. We derive our strategies from the *most common* cases that we encounter in the process of inspecting tests detected by NonDex. The effort of adding a new DexFix strategy is usually about a day *if* the support already exists for the right technology; e.g., we did not have initial support for the AssertJ [19] style of assertions, so it took additional work to add that support.

This paper makes several contributions:

Dataset: We provide a novel dataset of 275 ID tests. This dataset is the largest for ID tests, showing the prevalence of this problem among open-source projects.

Debugging Support: We extend the existing NonDex tool to provide more info for debugging ID tests.

Repair Strategies: We derive novel strategies that can help to automatically repair the code exercised by ID tests. Our new strategies are complemented by the old ReAssert strategies for repairing tests [38]. We automate these strategies by building on top of NonDex and ReAssert.

Evaluation: We apply the DexFix technique on 275 tests, and we find that DexFix can propose fixes for 119 tests. After DexFix proposes code changes, we check that the fix indeed passes with NonDex. Inspired by the fixes proposed by DexFix, we have opened GitHub pull requests for 102 tests, and 74 have been already merged, with only 5 rejected, and the remaining still pending.

Our dataset and links to pull requests submitted from anonymous GitHub accounts are publicly available as a part of a larger dataset of flaky tests [21].

II. Background

A. Detecting ID Tests

Following Lam et al. [48], we call a flaky test that fails due to wrong assumptions on underdetermined specifications an *implementation-dependent (ID) test*. Such ID tests can be detected proactively by exploring different possible implementations of underdetermined specifications, finding one that makes the test fail. One specific technique and tool that detects ID tests due to wrong assumptions on specifications in the Java standard library is NonDex [29], [41], [59]. Another tool that can detect not only ID tests but also more kinds of flaky tests in C++ code is the Mozilla Chaos Mode [30]. In this paper, we use NonDex to detect ID tests in Java code.

NonDex detects ID tests by randomizing the output of several methods with underdetermined specifications [59]. NonDex is implemented as a Maven plugin [29], [41] that can be integrated into any Maven-based project that runs using Java 8. NonDex also provides a debugging feature (invoked through the command `mvn nondex:debug`) [41]. Given a detected test that fails for some random seed, NonDex attempts to find if one random choice location can make the test fail, e.g., a dynamic invocation of `getDeclaredFields`. 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 this single point.

B. Automatic Repair of Test Assertions

Test assertions can fail after developers make changes to the main code. Assuming the main code is correct, test assertion repair aims to update an assertion to pass when run on that main code, e.g., changing the assertion's expected value with the new actual value produced by running the test on the new main code, or even changing the failing assertion to a different kind that more properly captures the proper test behavior. Challenges in test assertion repair involve handling the numerous kinds of test assertions that developers use.

ReAssert [36]–[38] is a technique and tool for repairing test assertions in JUnit. Given a (failing) test, ReAssert instruments the code, executes the test, and records the failure message, including the expected and actual values for comparisons. ReAssert then applies several *repair strategies* to repair the assertion so that the test no longer fails. For example, the strategy `ReplaceLiteralInAssertion` works on assertions where the expected value is a literal (e.g., an integer or a string), and replaces that literal with the actual value it observes during the test execution [38].

III. Examples

We next discuss several example ID tests that NonDex detects and for which DexFix proposes a fix. The examples show a variety of root causes and repair strategies used to change the code. We introduce these examples in order of perceived "simplicity".

A. Simple Fix in Main Code

Apache Hadoop [3] is a widely used open-source project. NonDex detected several ID tests in Hadoop, which shows that even well-tested projects can have problems with underdetermined specifications. One such ID test was `TestMetricsSystemImpl#testInitFirstVerifyCallBacks`. This test passes when run normally but fails with NonDex, reporting an error message that appears challenging to debug:

```
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}>}
```

Fortunately, NonDex can provide debugging info for each failing test, and the prior NonDex debugging output [41] provides useful info for this case, specifically the "root cause":

```
java.lang.Class.getDeclaredFields(Class.java:1916)
org.apache.hadoop.util.ReflectionUtils.
  getDeclaredFieldsIncludingInherited(ReflectionUtils.java:353)
[...]
```

DexFix has a general, automatic strategy that sorts arrays of fields returned by `getDeclaredFields`, which makes the order of the elements in the array deterministic. Based on the NonDex output reporting the "root cause" in the `ReflectionUtils` class, DexFix proposes the following fix:

```
   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);
...
```

Rather than introducing its own sorting, DexFix uses the Java standard library classes `Arrays` and `Comparator`. Because `Arrays#sort` sorts the input array in place (and does not return the sorted array), DexFix introduces a fresh variable `sortedFields`, sorts the array (comparing fields by name), and uses `sortedFields` in the `for` loop. After DexFix makes a change, we (compile and) rerun the test with NonDex; in this case, the test passed after the above fix. We submitted this fix as a GitHub pull request, and the developer promptly accepted our fix with the message "+1, committing. we all hate flaky tests. thanks for this" [4].

B. Multiple Changes with the Same Strategy

In this example, DexFix proposes a fix with multiple changes in both main and test code, but all changes follow the same strategy. In the Quarkus project [5], NonDex detected several ID tests, including `CompilerFlagsTest#defaulting`. When the test failed with NonDex, it produced an error message that included the following:

```
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}>
```

While the prior debugging output from NonDex [41] provides some partial info for this failing test, unfortunately it does not provide enough info:

```
java.util.HashMap$HashIterator$HashIteratorShuffler.<init>(
  Unknown Source)
java.util.HashMap$HashIterator.<init>(HashMap.java:1435)
[...]
```

We can see that the failure stems from an iteration over a `HashMap`, but the NonDex output did not provide (1) the code location that allocated this object and (2) whether it indeed allocated a `HashMap` or `HashSet` (that internally uses `HashMap`). We extended NonDex to include the allocation location (Section IV-C), reporting that the object was a `HashSet` allocated on line 30 of the class `CompilerFlags`.

DexFix has another general automated strategy for replacing allocations of `HashMap`/`HashSet`, with `LinkedHashMap`/`LinkedHashSet`. The `LinkedHash*` classes have a precisely defined iteration order [27]: "Hash table and linked list implementation of the `Map` interface, with predictable iteration 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 following change:

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

Each `LinkedHash*` class is a subclass of its respective `Hash*` class, so the changed code can compile after importing the class, with no other changes.

After applying this change and rerunning the test with NonDex, unlike in the first example where the test passed after the first change, this test again failed after the change. The new failure was again due to iteration over a `HashMap`, and our extension reported that the object was allocated on line 41 of the same class `CompilerFlags`. DexFix proposed a similar fix (changing `HashSet` to `LinkedHashSet` on that line), but rerunning NonDex yet again resulted in a different failure due to iteration over another `HashMap`, this time allocated on line 88 of the class `CompilerFlagsTest`. Once again, DexFix proposed a similar change on that line.

After DexFix changed all these three lines (along with adding `import` statements), the test finally passed with NonDex. We submitted this fix, and the developer accepted it: "Merged, thanks!" [6].

C. Multiple Changes with Different Strategies

This example illustrates a case where DexFix proposes a fix that changes both main and test code, but uses different strategies. In the Alibaba Fastjson project [7], NonDex detected several ID tests, including `WriteDuplicateType#test_dupType2`. As in the previous example, the problem was with a `HashMap` iteration, specifically line 38 of `WriteDuplicateType`, and DexFix proposed the following change:

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

After this change, the test fails even without NonDex, indicating that the cause is not an underdetermined specification any more. In particular, this test fails on line 44 of the class `WriteDuplicateType`, which compares two strings.

At this point we run ReAssert [38] to repair 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 for the assertion:

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

These two changes now make the test pass both with and without NonDex. The developers merged our submitted fix [8].

D. A Novel Strategy for Comparing Collections

When a test has an assertion that compares some actual collection value against an expected collection whose iteration order is known to be underdetermined, we can change the assertion to a more suitable one that is order-agnostic. In the project Graylog2 Server [9], NonDex detected an ID test `V20161215163900_MoveIndexSetDefaultConfigTest#upgrade`. According to the NonDex debugging output, this test has a root cause in a third-party library, where the `HashSet` is initialized. The assertion that fails in this test is `containsExactly`, an assertion from the AssertJ library [19] that checks if a collection contains given elements, in the given order.

DexFix in this case changes the assertion to another AssertJ assertion, `containsExactlyInAnyOrder`, that allows any order of elements and is thus order-agnostic:

```
- assertThat(...).containsExactly("...0001", "...0003");
+ assertThat(...).containsExactlyInAnyOrder("...0001", "...0003");
```

We submitted this fix, and the developers merged it: "Yeah, that makes sense. Thank you very much" [10].

E. A Novel Strategy for JSON Strings

Our final example is a case where DexFix proposes a fix just for comparing JSON strings. In the project Nutz [11], NonDex detected several ID tests, including `JsonTest#test_enum`. According to the NonDex debugging output, this test also has a root cause in `getDeclaredFields`, but in a location that is in a third-party library, not in the Nutz project itself. However, the failed assertion (on line 1031 of `JsonTest`) just compares a JSON string to an expected value.

As such, DexFix has a general, automatic repair strategy to change such comparisons to use another assertion method, `JSONAssert#assertEquals`, from a specific library:

```
  import static org.junit.Assert.assertEquals;
+ import static org.junit.Assert.fail;
+ import org.json.JSONException;
+ import org.skyscreamer.jsonassert.JSONAssert;
  ...
  String expected = "{\n" + " \"name\":\"t\",\n" + " \"index
    \":1\n" + "}";
- 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.");
+ }
```

The parameter `false` instructs the assertion to ignore the ordering of fields in the JSON object, making this assertion order-agnostic. 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 four similar failures (all detected automatically using NonDex), we (manually) extracted all `try-catch` blocks in a helper method, called `assertJsonEqualsNonStrict`, and replaced calls to `assertEquals` with calls to `assertJsonEqualsNonStrict`. These changes make all the tests pass both without and with NonDex. The developers merged our submitted fix: "thank you very much ^_^" [12].

IV. Technique

The input to our technique DexFix conceptually consists of (1) the project source code including the main and test code, and (2) an ID test to be repaired. The output of our technique is a fix, consisting of one or more code changes, that makes the test pass when run with NonDex. Inspired by ReAssert [38], DexFix proceeds by applying various repair strategies on the code and checking if the test passes with NonDex.

A. Overview

Figure 1 presents the pseudo-code of the DexFix top-level `repair` function that changes a project's codebase. (We did not use this exact pseudo-code in our early experiments but over time developed this current pseudo-code based on our experience.) Specifically, `repair` takes as input the ID test to repair. It first runs NonDex to get the initial test result (which should be FAIL), the failing assertion `a` reported by JUnit, the root cause `c` reported by NonDex (e.g., that the failure is caused by use of `HashSet`, as obtained from the NonDex debug feature), and the code location `l` reported by our extension of NonDex. DexFix also keeps track of the locations already attempted for fixing (lines 5 and 11 in Figure 1).

DexFix first tries to apply the ChangeContainsExactly strategy (Section IV-B), which only changes test code assertions to make them order-agnostic. While the strategy does not introduce any new dependencies to the project, it only applies when the project already uses a specific assertion from the AssertJ library of so-called "fluent assertions" [19], commonly used to supplement standard assertions provided by JUnit. We first attempt this strategy that fixes only test code because developers are more likely to consider fixes to test code than fixes to the main code. Furthermore, nondeterminism used in the main code is not necessarily incorrect, so we believe it is preferable to reduce flakiness in tests by making the test code agnostic to the nondeterminism.

If the ChangeContainsExactly strategy does not apply, i.e., the test does not use the relevant assertion, DexFix then 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). 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 (Section IV-B). If DexFix changes the code, it also checks whether the test fails: while the change makes the order deterministic, the resulting deterministic order may not match the assumed order encoded in the current test assertions. DexFix then needs to update the failing test assertion by applying the traditional ReAssert strategies [38], e.g., as shown in Section III-C. If ReAssert cannot repair the assertion, then DexFix stops and reports unrepaired (line 33 in Figure 1).

If `repair_location` does not repair the test, DexFix tries to apply its JSONAssertion strategy (Section IV-B). Although this strategy (like the ChangeContainsExactly strategy) changes only test code, we use this strategy last because it can add new third-party dependencies to the project, and developers tend to be cautious about adding more dependencies. After all these changes,, we need to run the test again with NonDex (for a configurable number of rounds). If the test fails with NonDex, DexFix uses the potentially new failing assertion `a`, root cause `c`, and debug location `l` to continue the repair process again, e.g., as illustrated in Section III-B. If the new debug location has been attempted before (line 9 in Figure 1), then DexFix stops, reporting unrepaired, to avoid an infinite loop. (In our experiments, we never encountered a previous location showing up again.) By keeping track of all attempted repair locations and not allowing repeats, the overall loop eventually stops.

If the test passes with NonDex, DexFix considers the test repaired and exits the loop. We then manually inspect the fix to prepare a pull request. While Figure 1 shows how DexFix proposes one fix, we can easily adapt it to propose multiple possible fixes, so the user can choose the best fix.

```
1  # Input/Output: project source code that gets changed
2  # Input: failing test t
3  # Output: status REPAIRED/UNREPAIRED
4  def repair(t):
5    rl = [] # repaired locations
6    # failing assertion a, root cause c, debug location l
7    result, a, c, l = run_NonDex(t, NUM_ROUNDS)
8    while result == FAIL:
9      if l in rl: # stop if the location has been tried
10       return UNREPAIRED
11     rl.add(l)
12     applies = apply_strategy(ChangeContainsExactly, t, a)
13     if not applies:
14       status = repair_location(t, a, c, l)
15       if status == UNREPAIRED:
16         applies = apply_strategy(JSONAssertion, t, a)
17         if not applies:
18           return UNREPAIRED
19     # run NonDex again to see if there is more to handle
20     result, a, c, l = run_NonDex(t, NUM_ROUNDS)
21   return REPAIRED
22 def repair_location(t, a, c, l):
23   if l in library:
24     return UNREPAIRED
25   if c is Hash*:
26     apply_strategy(HashToLinkedHash, l)
27   elif c is getDeclaredFields:
28     apply_strategy(SortFields, l)
29   else:
30     return UNREPAIRED
31   if compile_and_run(t) == FAIL:
32     apply_strategy(ReAssertStrategies, t, a)
33     if compile_and_run(t) == FAIL:
34       return UNREPAIRED
35   return REPAIRED
```

Fig. 1. Pseudo-code of DexFix repair process

B. Repair Strategies

We develop four strategies for DexFix.

1) ChangeContainsExactly Strategy: This strategy changes an AssertJ assertion that uses `containsExactly`, which can check if a `Map` or a `Set` collection contains exactly the expected elements in the given order. For example, the assertion can be `assertThat(actual).containsExactly(expected)`. The strategy changes `containsExactly` to `containsOnly` if checking a `Map` or to `containsExactlyInAnyOrder` if checking a `Set`. These other assertions from AssertJ check if the collection contains the expected elements in *any* order. This strategy changes only test code, as illustrated in Section III-D.

2) HashToLinkedHash Strategy: This strategy replaces `new HashMap`, resp. `new HashSet`, with `new LinkedHashMap`, resp. `new LinkedHashSet`. The strategy applies when the root cause is iteration over some `HashMap`/`HashSet` object. While NonDex provided the stack trace at the iteration point, it did not provide the stack trace of the allocation until we extend NonDex. This strategy may need to add an appropriate `import` statement for some class. This strategy may change both main and test code, depending on the location of the allocation.

3) SortFields Strategy: This strategy adds sorting of field arrays returned by the method `getDeclaredFields`. The strategy applies when the root cause is the underdetermined order of the elements in such an array. NonDex already provided the stack trace at the point where the method is invoked. If the method invocation is the only expression in a statement, e.g., `fields = clazz.getDeclaredFields()`, then adding sorting is easier. A more challenging case is handling method invocations that appear in more complex expressions, e.g., as illustrated in Section III-A. Our solution is to use a fresh variable to store the array, sort it, and finally replace the original invocation with this variable. This strategy may need to add two appropriate `import` statements for comparing and sorting. This strategy may change both main and test code, depending on the location of the invocation.

4) JSONAssertion Strategy: This strategy repairs failing assertions that compare JSON strings.

If the assertion is JUnit's `Assert.assertEquals`, the strategy replaces the call with `JSONAssert.assertEquals` from the Skyscreamer JSONassert library [25] (specifically version 1.5.0), as illustrated in Section III-E; we call this substrategy JSONAssertion$_J$. Replacing the method call is conceptually easy, but there are some additional details. First, it needs to handle the potential `JSONException`. Second, it needs to provide a value for an additional boolean argument for the new assertion's `strict` parameter. Setting `strict` to `true` ignores the order of field-value pairs in the JSON string representation but does not ignore the order of elements in a JSON array. The JSON string representation for a `HashSet` is a JSON array, making it necessary to set `strict` to `false` to ignore the order of elements. However, setting `strict` to `false` is not ideal, because it also allows JSON strings that contain more elements than expected, as long as they contain all the expected elements. As such, JSONAssertion$_J$ first sets `strict` to `true`, and if the test still fails, then it sets the value to `false`.

If the failing assertion is from the AssertJ library, this strategy instead changes the AssertJ `assertThat` invocation to `assertThatJson` from the JsonUnit library [26] (specifically version 2.17.0); we call this substrategy JSONAssertion$_A$. Unlike JSONAssertion$_J$ that replaces the failing assertion method, JSONAssertion$_A$ changes the `assertThat` invocation that creates an `Assert` object from AssertJ. For example, consider `assertThat(actual).isEqualTo(expected)`; while it is the call to `isEqualTo` that fails, JSONAssertion$_A$ does not replace that call but instead changes the entire expression to `assertThatJson(actual).isEqualTo(expected)`.

JSONAssertion is similar to ChangeContainsExactly in the sense that both affect only test assertions, but JSONAssertion requires adding a new third-party dependency (either JSONassert or JsonUnit) to the project, if not already included. Adding a dependency is sometimes undesirable to developers.

C. Implementation

For evaluation, we implement DexFix through several modifications to the existing NonDex [41] and ReAssert [36] tools, along with automated and manual steps to connect everything together as per the overall DexFix process shown in Figure 1.

NonDex. 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. 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 standard or third-party).

ReAssert. Our key modifications to ReAssert are to implement the ChangeContainsExactly and JSONAssertion strategies. For the JSONAssertion$_J$ substrategy, we reuse the prior ReAssert code for its existing strategy that fixes `assertEquals` calls (Section III-C). 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 whether the expected and actual strings when sorted (purely character ordering, not considering any of the JSON format) end up equal. Unlike the existing ReAssert strategy that just replaces the expected string literal in `assertEquals`, our extension has to perform somewhat elaborate changes to replace the invoked method, add a new parameter, and add a `try-catch` block to handle the case when the actual value is not a JSON object.

For both ChangeContainsExactly and JSONAssertion$_A$, we also need to extend ReAssert to handle the so-called "fluent-assertion style" from AssertJ, which uses `assertThat`. Daniel et al. [36] supported related assertions in ReAssert but for the old, JUnit-style of `assertThat`, not for AssertJ. We modify ReAssert to specially capture failures stemming from AssertJ assertions. An example AssertJ assertion is of the form `assertThat(actual).method(expected)`, where the `assertThat` method wraps the `actual` value into an `Assert` object, which is the receiver for the `method` (e.g., `containsExactly` or `isEqualTo`); the `method` checks the value in `Assert` against `expected`. ReAssert captures the failure that comes from the `method` call. If the captured failure is from `containsExactly`, ChangeContainsExactly applies, and it replaces that `containsExactly` with `containsOnly` or `containsExactlyInAnyOrder`. For JSONAssertion$_A$, while the `isEqualTo` call fails, the strategy changes the receiver expression, namely change `assertThat` to `assertThatJson`.

The prior ReAssert code [36] parses Java files using an old version of the Spoon library [58], 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 newer versions, but they break backwards compatibility, and using them would require a substantial rewrite of the existing ReAssert code.

DexFix. Our key new additions, besides NonDex and ReAssert modifications, are to implement the HashToLinkedHash and SortFields repair strategies. We use the javaparser library [24] to parse the input Java files (main and test code), change the code, and output it. The javaparser library supports all latest features of Java 8 (we analyze Java 8 projects when running NonDex). Our implementation directly follows the descriptions in Section IV-B and the examples in Section III.

To support the overall DexFix process presented in Figure 1, we had to manually apply some of the steps in our experiments. While we automated the key `repair_location` function that fixes a location, we had to manually apply the steps in the top-level `repair` function that loops calling `repair_location` as long as NonDex detects the ID test. We also manually invoke the ChangeContainsExactly and JSONAssertion strategies (which themselves automatically change assertions), because these two strategies rely on ReAssert, which needs a failing test; integrating ReAssert's instrumentation for capturing failing test assertions and NonDex's instrumentation to run a test to trigger a failure is not straightforward.

V. Experimental Setup

We first describe how we select projects for our evaluation and how we use NonDex to detect ID tests within these projects. We then describe how we use DexFix to propose fixes for these detected tests and how we send pull requests based on these proposed fixes.

A. Selecting Projects and Detecting Tests with NonDex

For our evaluation, we select open-source Java projects that use the Maven build system [28] because NonDex was originally developed to support running tests for Maven-based projects [29], [41]. We queried GitHub to find the top 1,000 Java projects by the number of stars, then randomly chose 200 projects of the 242 that have a top-level `pom.xml` file used to configure Maven.

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

For each Docker image, we start a Docker container where we run NonDex on all tests using `mvn nondex:nondex`. We configure NonDex to run 10 rounds (with varying random seeds), using the "ONE" mode [59], 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). We choose the "ONE" mode because tests that fail in this mode most likely indicate real problems due to wrong assumptions that developers are motivated to fix. This mode puts a lower bound on the number of ID tests that NonDex detects; in the "FULL" mode, NonDex could detect even more test failures by randomizing *all* calls for methods with underdetermined specifications. We collect all the tests that pass when run without NonDex but fail with NonDex randomization.

For each detected ID test, we run `mvn nondex:debug` to obtain debugging info. The prior NonDex debugging reports a single method-call location where NonDex random choice leads the test to fail [41]. When the call iterates over a `HashMap`/`HashSet`, our NonDex extension also reports the location where that object is allocated (Section IV-C). NonDex debugs by rerunning the test while randomizing only a subset of method calls with underdetermined specifications. During this process NonDex can find that some tests pass or fail even when rerun for the same random seed, so we remove such flaky tests. Also, the prior NonDex debugging occasionally crashes altogether and produces no output, so we remove such tests as well. Because these tests have no info about any method-call location, our extension cannot report where the receiver object for that method is allocated. These crashes are infrequent and hard to reproduce, so we have not debugged them in NonDex.

B. Fixing Tests using DexFix and Opening Pull Requests

For each ID test, we start a new Docker container based on the Docker image for the test's project. We copy into this container the corresponding NonDex debug file and then run DexFix for the test. This procedure ensures that DexFix proposes a fix for each test when run on the same code version where NonDex detected the test. An alternative to fix multiple tests in the same container would have run a later test on a different code version that contains the fix for a prior test.

When DexFix needs to rerun NonDex, we configure it to run 10 rounds to check if the test, after the applied change, can still fail. If the test fails in any of these rounds, DexFix has to use the info collected from the NonDex run to propose additional changes to the code (Section IV-A). This process loops until either DexFix generates a fix or reports that it cannot fully repair the test after potentially making some changes.

We inspect all proposed fixes, potentially modifying them to prepare GitHub pull requests to the developers of each 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 code version where NonDex detected the test. If fixes for multiple tests share some changes to the main code or the test code (even if they still have separate changes to their test assertions), these fixes can be safely combined, because they address the same root cause. Moreover, all the changes to test assertions need to be combined together along with the changes to the main code; otherwise, the tests will fail when run without NonDex. Some pull requests we send fix multiple tests at once and combine fixes for these related tests, with all the fixes sharing the same changes to the main and test code, modulo changes to the test assertions. Most pull requests simply fix only one test.

As we prepare a pull request, we manually make stylistic changes so that our code patch matches the coding style that the project uses. For each new project for which we have not yet sent pull requests, we send one pull request to that project for review. While the pull request remains pending, we do not send more to not "spam" developers with pull requests that they may not have time to review. We send additional pull requests only after developers accept the initial one. Also, if a pull request is rejected, we do not submit other similar pull requests to that same project. We describe more of our cases when we do not send pull requests in Section VI-C.

VI. Results

Our evaluation addresses the following research questions:
RQ1: What is the breakdown of the root causes and debug locations for ID tests detected by NonDex?
RQ2: How many tests can DexFix fix, and which repair strategies propose the fixes?
RQ3: How effective is DexFix at proposing fixes that developers actually accept?

Our dataset and pull requests are publicly available [21]. Note that we sent some pull requests before finalizing the pseudo-code presented in Figure 1. Our results accumulate our experience from sending pull requests while refining DexFix to create fixes that developers are more likely to accept.

A. *RQ1*: Detected Tests

After we run NonDex on 200 projects, it detects 275 ID tests in 37 projects. Table I lists these 37 projects. The "PID"

column shows the short id we use for later reference. The "Commit" column is the Git commit SHA on which we run NonDex. The remaining columns show the breakdown of the root causes and debug locations for test failures. The "`Hash*`" column is the number of tests due to iteration over a `HashMap`/`HashSet` collection. The "`gDF`" column is the number of tests due to `getDeclaredFields`. The "Rest" column shows the remaining tests, of which 15 are due to `getMethods`, and the others due to six various causes. These columns show the one cause from the debugging file that `mvn nondex:debug` outputs on the *first* run, but a test may have multiple root causes (Section III-B). The columns under "Source?" show whether the debug location reported by our NonDex extension is in the project's source code or in a library. The final column shows the total number of ID tests detected per project.

While NonDex implements random exploration for over 40 methods with underdetermined specifications, only a few of them cause most test failures. The majority of the detected tests fail due to some `HashMap`/`HashSet` (152 out of 275 tests). The second most common root cause is `getDeclaredFields` (93 out of 275 tests). Together, these two causes lead to 89% of all detected tests. Prior reports from running NonDex also found these two causes to be the most common [41], [59], but interestingly, the ranking between these two is reversed in our findings compared to prior work. 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 repair strategies for DexFix can apply broadly for fixing tests detected by NonDex.

Compared to prior work, the number of tests that NonDex detects in our experiment—275 tests in 37 out of 200 projects—is greater than the previously reported proportion, e.g., Shi et al. [59] detected 60 tests in 21 out of 195 projects. While we run on some much larger Maven projects, it could be also that the problem of ID tests continues to increase.

In terms of debug locations, the majority (207/275) are in the main and test code rather than in a library. This ratio also increases confidence that our repair strategies for DexFix could be effective, because they mostly work on main and test code. Two strategies (ChangeContainsExactly and JSONAssertion) can work even if the location is in library code, but they apply only in some cases (for certain assertions or for JSON strings).

TABLE I
PROJECTS USED IN THE STUDY AND BREAKDOWN OF ROOT CAUSES AND THEIR LOCATIONS FOR 275 ID TESTS DETECTED BY NONDEX

PID	Project Slug on GitHub	Commit	Root Causes			Source?		Σ
			`Hash*`	`gDF`	Rest	Y	N	
P1	apache/flink	23c9b5a	31	1	9	39	2	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	-	11	12	23
P5	apache/commons-lang	7c32e52	-	21	-	21	-	21
P6	flowable/flowable-engine	399ab58	3	-	14	17	-	17
P7	apache/incubator-shard... [13]	038232e	15	-	-	15	-	15
P8	dropwizard/dropwizard	616ed86	6	3	-	7	2	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	-	-	4	-	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	-	-	2	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-... [14]	922590e	1	-	-	1	-	1
P37	zhangxd1989/spring-... [15]	e3966d7	-	1	-	-	1	1
Σ	-	-	152	93	30	207	68	275

B. *RQ2: Fixed Tests*

Table II shows statistics about the fixes that DexFix proposes, overall for 119 out of 275 tests. The table shows the breakdown of fixed tests per root cause and the strategy DexFix uses: "`CC`" for ChangeContainsExactly; "`L1`", "`LM`", and "`LA`" are for HashToLinkedHash, corresponding to only one allocation site, multiple allocation sites, and one allocation site but also updating test assertion(s); "`JA`" for JSONAssertion; "`SF`" for SortFields at only one site (we never observe the need to change more than one site); and "`SA`" for SortFields with some updates of test assertion(s). Note that ChangeContainsExactly and JSONAssertion apply to both top root causes.

For the `Hash*` cause, nearly half the fixes that DexFix proposes use HashToLinkedHash only once (41 out of 83 tests), *without changing any test assertion*. The tests can still assert on the same expected values as before, but now the values are deterministic and cannot be affected by evolution of the implementation of the library methods in the future.

JSONAssertion helps in a number of cases, with a total of 31 tests fixed for both causes, highlighting that tests often make incorrect assumptions on JSON serialization. Most of the tests fixed this way are caused by `getDeclaredFields` (19); in fact, JSONAssertion fixes the highest number of tests for this cause. DexFix cannot fix most of these tests with SortFields, because the call to `getDeclaredFields` is in library code.

ChangeContainsExactly, the first strategy that just changes the test code without introducing new dependencies to the project, applies to few places, fixing only 7 tests, all in one project, P4. We note that one test in P8 that we fix using HashToLinkedHash also could have been fixed using ChangeContainsExactly (marked with "†" under column `L1`). We initially developed HashToLinkedHash first before ChangeContainsExactly, and the developers accepted our fix using HashToLinkedHash. As such, we count the test under HashToLinkedHash. Regardless, the number of tests Change-

TABLE II
STATISTICS FOR TESTS REPAIRED, STRATEGIES USED PER ROOT CAUSE, ASSERTION CHANGES, REPAIR LOCATIONS, AND PULL REQUESTS

PID	Hash* (total 83)					gDF (total 36)				Assert.Changes		Repair Locations			Pull Requests				Σ
	CC	L1	LM	LA	JA	CC	SF	SA	JA	Yes	No	Test	Main	Both	A	P	R	U	
P1	-	11	6	-	-	-	-	-	-	-	17	6	10	1	5	8	-	4	17
P2	-	8	-	9	6	-	-	-	-	15	8	17	2	4	7	*7	-	9	23
P3	-	1	-	-	-	-	1	-	1	1	2	2	1	-	1	2	-	-	3
P4	5	2	-	-	-	2	-	-	7	14	2	14	2	-	15	1	-	-	16
P5	-	-	-	-	-	-	1	5	-	5	1	-	1	5	6	-	-	-	6
P6	-	1	-	-	-	-	-	-	-	-	1	-	1	-	1	-	-	-	1
P7	-	1	4	-	-	-	-	-	-	-	5	1	4	-	5	-	-	-	5
P8	-	†3	-	-	2	-	-	-	1	3	3	3	3	-	3	-	§3	-	6
P10	-	-	-	-	3	-	-	-	2	5	-	5	-	-	5	-	-	-	5
P11	-	2	-	-	-	-	-	-	-	-	2	-	2	-	2	-	-	-	2
P12	-	-	-	-	-	-	2	-	-	-	2	-	2	-	2	-	-	-	2
P13	-	2	-	3	1	-	-	-	-	4	2	1	2	3	-	-	2	4	6
P14	-	-	-	-	-	-	4	1	-	1	4	-	4	1	-	5	-	-	5
P16	-	-	-	-	-	-	-	-	5	5	-	5	-	-	5	-	-	-	5
P18	-	-	-	-	-	-	-	1	-	1	-	-	-	1	1	-	-	-	1
P20	-	1	-	-	-	-	-	-	-	-	1	-	1	-	1	-	-	-	1
P21	-	3	-	-	-	-	-	-	-	-	3	2	1	-	3	-	-	-	3
P23	-	-	-	-	-	-	-	-	3	3	-	3	-	-	3	-	-	-	3
P28	-	-	2	-	-	-	-	-	-	-	2	-	-	2	2	-	-	-	2
P29	-	2	-	-	-	-	-	-	-	-	2	-	2	-	2	-	-	-	2
P30	-	1	-	-	-	-	-	-	-	-	1	1	-	-	1	-	-	-	1
P31	-	-	-	1	-	-	-	-	-	1	-	-	-	1	1	-	-	-	1
P34	-	1	-	-	-	-	-	-	-	-	1	1	-	-	1	-	-	-	1
P35	-	1	-	-	-	-	-	-	-	-	1	-	1	-	1	-	-	-	1
P36	-	1	-	-	-	-	-	-	-	-	1	-	1	-	1	-	-	-	1
Σ	5	41	12	13	12	2	8	7	19	58	61	61	40	18	74	23	5	17	119

ContainsExactly fixes is relatively small, suggesting that developers tend not to directly assert upon these collections.

Table II also shows whether DexFix changes assertions for proposed fixes: "Yes" means it does; "No" means it does not. Almost half of the fixes (58 out of 119) involve assertions, showing the importance of using ReAssert. Table II also shows where the repair locations are: only in the test code, only in the main code, or in both. We see a variety, demonstrating the importance of considering both main and test code.

For repair cost, DexFix strategies take relatively little time to change the code, because the strategies apply only targeted changes and run each test at most 5–6 times (unlike automated program repair that may explore thousands of changes and run many tests hundreds or thousands of times [55]). Running DexFix on all 275 tests takes under 7 hours, an average of ∼90sec per test, ranging from 32sec to 388sec.

C. *RQ3: Pull Requests*

We have sent pull requests for 102 out of 119 tests for which DexFix proposes a fix. Table II also shows the pull request status for the tests: Accepted ("A"), Pending ("P"), Rejected ("R"), or Unsubmitted ("U").

Overall, developers have accepted pull requests that we submitted for a majority of the fixed tests (74 tests). This high ratio of accepted pull requests shows the effectiveness of the fixes that DexFix proposes, and developers welcome the changes, e.g., recall the messages in Section III. We have 23 tests whose pull requests are still open on GitHub, pending review or final judgment from developers. Overall, of the tests in the accepted pull requests across the two root causes, 47 of 83 are for Hash*, and 27 of 36 are for gDF.

We next discuss 22 tests whose pull requests were rejected or for which we did not send pull requests.

Rejected. We have 5 tests whose pull requests have been Rejected, across two projects, P8 and P13. For the 3 tests in P8, the developer did not accept our pull request because the fix involves JSONAssertion and adds a new dependency on the JsonUnit library to the project. However, the developer acknowledged the potential problem with the tests and fixed all these tests in another way through a combination of project-specific annotations for specifying order and using some sorted maps. The developer closed this pull request with the message "I've changed the tests accordingly. Thanks for the hint!" [16]. We mark "§" on the cell for P8 under "R" in Table II to indicate that the developer found our report useful.

In the other project, P13, the developer rejected the pull request for one test for a similar reason, because it adds a new dependency (on the JSONassert library). The developer commented "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." Based on the feedback from these two projects, we see that while JSONAssertion effectively changes just the test code when comparing JSON strings, the cost of including a new library dependency is too high for some developers. For the final rejected pull request in P13, the fix was from HashToLinkedHash, but the developers did not provide any feedback before rejecting, so we do not know their reason for rejection.

Unsubmitted. We did not submit fixes for 17 tests, spread across three projects. For P1, we have not sent pull requests for 4 tests yet because they are quite similar to some initial ones we sent that are still pending review, so we do not want to bother developers with additional similar fixes until we get proper feedback from the initial ones.

For P2, we have 9 tests with unsubmitted pull requests. For 4 of these tests, the fixes are similar to an initial pull request we sent. That initial pull request received positive feedback from developers but has yet to be officially merged into the codebase; we plan to send the followup pull requests for these other 4 tests after the merge. For the remaining 5 tests, we ourselves "rejected" the fixes after our manual inspection. We believe the developers truly want to use a `HashMap` (no deterministic ordering) at the location where DexFix proposes to use a `LinkedHashMap`. In our inspection, we find the code has a flag to determine if a `LinkedHashMap` should be used. This flag does not help to fix the 5 unsubmitted tests. However, we manually change the test code in P2 to set the flag, which fixes 7 other tests, marked "$*$" under "P" in Table II.

Finally, for P13, of the 4 tests with unsubmitted pull requests, 2 tests that DexFix automatically fixes on an older code version (on which we started our experiments) are no longer ID tests in the latest code version. The remaining 2 tests have fixes very similar to previously rejected ones we sent, so we do not send these additional ones.

VII. Discussion

A. Limitations

DexFix currently cannot propose a fix for 156 of 275 tests detected by NonDex. We inspected most of these tests, at least one unfixed test from each project with some unfixed test(s). Some of the cases are limitations of our general repair strategies, and some are so rare that they do not merit developing general strategies. We next show a breakdown of the reasons why DexFix could not fix the tests.

Tool Engineering (42+27+6). ReAssert crashes altogether when run on 42 tests. Also, ReAssert cannot run on 27 tests that use JUnit 5 or TestNG [31], as ReAssert currently supports only JUnit 4 and JUnit 3. Our attempt to upgrade ReAssert to support JUnit 5 revealed that it would require a major re-implementation effort. Finally, our toolset cannot handle source languages beyond Java; 6 tests are written in Scala (but NonDex can still detect their failures as its instrumentation works at the level of bytecode).

Unsupported Root Cause (30). The current DexFix strategies focus on addressing the two top root causes—iterating over `HashMap`/`HashSet` and the order of fields returned by `getDeclaredFields`—as reported by prior work [59] and confirmed in our experiments (Section VI-A). 30 tests fail for 7 other root causes. The largest number of tests, 15, is for `getMethods`, but they come from only two projects. We inspect 14 such tests in P6, and they all fail because a class has two methods named `equals` (one declared in the class itself and another inherited). 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" [17]. The remaining test for `getMethods` is interesting in that its root cause is not just one random choice but two random choices. The fix would again be overly specialized to this one case. In the future, we believe it worthwhile to develop more strategies only for more general cases.

Library Location (29). Only two DexFix strategies—ChangeContainsExactly and JSONAssertion—can apply if the root cause is in a library, i.e., the source code is not accessible for DexFix to change. 29 tests have such root causes, and the two strategies do not apply because the tests are not using `containsExactly` or comparing JSON strings.

Others (22). Finally, DexFix cannot handle 22 tests due to one-off instances, such as creating `HashMap` through reflection or using Java language features in repair locations not handled by our prototype implementation of DexFix strategies.

B. Making All Code Deterministic

It is interesting to consider whether the four new strategies that DexFix provides (Section IV-B) should be applied in *all* cases, i.e., should all sets/maps be compared using `containsExactly`, should all `Hash*` objects be `LinkedHash*`, should all arrays from `getDeclaredFields` be sorted, and should all JSON strings be compared with something like `JSONAssert`? In the limit, should the Java standard library have *any* underdetermined specification? Developers of the Java standard library could have specified that `HashMap` behaves like `LinkedHashMap`, even when deterministic iteration is not required. For example, in Python, all dictionaries since version 3.7 are specified [32] to behave similar to `LinkedHashMap`.

Considering that the Java standard library has *not* made the underdetermined specifications deterministic like Python did, we believe that underdetermined specifications remain valuable. We approach fixing of ID tests with the goal to first prioritize changing the test code. Our assumption is that the developers are fine with an underdetermined specification and do not require determinism in their main code. The problem is that the tests are overly specific to the current implementation and would fail if the implementation changes. Thus, our strategies first help fix the test code before changing the main code. We do have strategies that change main code, but DexFix attempts them only if it cannot repair only test code without adding dependencies. Ultimately, we send fixes to developers so they can decide how to best address the problem.

C. Overhead

While DexFix aims to make tests no longer implementation-dependent, the changes could introduce other side-effects to the main and test code, such as extra execution overhead.

Compared to `Hash*`, `LinkedHash*` objects provide a small overhead in both space (`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 but microbenchmarks, and some operations on `LinkedHash*` can

be even faster [20], [27], including resizing, `containsValue`, and ironically, iteration, which becomes not only predictable but also faster in certain situations. Some Java developers still raise concerns about the overhead, e.g., one of our pull requests had a discussion about it [18], but later the developer still accepted the fix.

Sorting `getDeclaredFields` provides an even smaller overhead as a library usually sorts only once and caches for later calls. In contrast to Java, Python's interface for reflection includes the `inspect` module's `getmembers` function that returns a list of members and since Python 3.7, is precisely specified: "Return all the members of an object in a list of `(name, value)` pairs sorted by name."

Finally, changing `containsExactly` or using `JSONAssert` affects only the test code, not the main code, so the overhead is less important but still almost negligible.

VIII. Threats to Validity

Our overall results may not generalize to all projects. Our evaluation uses a diverse set of popular projects from GitHub, and due to our use of existing tooling, our evaluation uses Java projects that build with Maven. However, the 200 projects we use are among the most popular Java projects on GitHub, so we believe they are fairly representative of all Java projects.

Our fix strategies may be overfitted [40] towards the ID tests we use in our evaluation. We developed our strategies based on the most common root causes for ID tests we found; these common root causes match findings in prior work [59]. We believe the common root causes should still be relevant for ID tests in projects beyond those in our dataset, so our strategies could still be effective in repairing those ID tests.

Our implementation of DexFix strategies may have bugs that affect our results. To reduce this threat, we build on top of existing tools, NonDex [41], ReAssert [36], and javaparser [24], which have been used in past research. Furthermore, multiple authors reviewed some of our new code and discussed the proposed fixes. The ID tests that NonDex detected are true positives, and we can reproduce the failures from all 275 ID tests. The number of tests detected in our evaluation is a lower bound on the true number of ID tests in these projects. Finally, the key threat is the quality of the fixes DexFix proposed. We confirm that these fixes are useful by sending pull requests to developers, so they make the final judgment call.

IX. Related Work

Mora et al. [56] 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. [59] studied when tests fail due to wrong assumptions on underdetermined specifications, developing NonDex to detect such tests. Our work focuses on automated fixing for such tests by modifying both the main and test code as needed.

The ID tests NonDex detects can be considered a type of flaky tests, which nondeterministically pass or fail on the main code [45], [46], [53]. Luo et al. [53] reported unordered collections as one reason for flaky tests. Lam et al. [48] used NonDex in a longitudinal study of flaky tests, finding 190/684 of the flaky tests in their study to be the ID tests that NonDex detects. Other prior work focused on detecting different types of flaky tests [34], [35], [39], [42], [43], [47], [65].

Automatic program repair (APR) aims to automatically generate patches to fix bugs in main code [40], [44], [49], [52], [55], [57], [61]–[63]. APR techniques often generate patches by searching and mutating existing code, e.g., applying pattern-based transformations learned from prior fixed bugs, or through symbolic execution. These techniques rely on test failures to indicate that the bug still exists, and the aim is to make all tests pass. DexFix is most similar to pattern-based repair [44], [61], which uses transformations inspired by existing bugs. However, these techniques rely on test outcomes to guide them and do not aim to fix the test code. Also, their general transformations do not apply to fixing the ID tests.

In contrast, there is prior work on fixing test code [37], [38], [50], [54], [64], repairing tests that become outdated when main code evolves. For flaky tests, Shi et al. proposed iFixFlakies [60] to fix specifically order-dependent flaky tests. Our technique DexFix aims to fix ID flaky tests. Our approach of fixing ID tests is not restricted to changes to the test code but sometimes also involves making changes to the main code. We utilize ReAssert [38] to automatically repair assertions that have to be updated after changes are made to the main code.

X. Conclusions and Future Work

We present the DexFix approach for automatically fixing ID tests that fail due to wrong assumptions on underdetermined specifications. DexFix extends the work on program and test repair with novel, *domain-specific* and simple, yet effective, automated repair strategies that can propose fixes for ID code. Unlike most prior work that focuses on fixing exclusively either the main code or test code, DexFix can fix either or both as necessary. The empirical results are encouraging: of 275 ID tests, DexFix proposed fixes for 119 tests; we have opened pull requests for 102 tests, and 74 have already been merged, with only 5 rejected, and the rest pending.

In the future, as more root causes for ID tests are found, we envision the set of domain-specific strategies for these causes growing into a general solution that handles a large fraction of ID tests. We believe domain-specific program repair is highly effective at fixing ID tests. We hope DexFix inspires more research into domain-specific repair not just for ID tests but also for other types of flaky tests and bugs in main code.

XI. Data Availability

Our input data and links to pull requests are archived and available [33]. Per email from the Open Science Chair, we do not include the Git histories of projects used in this paper.

Acknowledgments

We thank Lingming Zhang for discussion about this work. This work was partially supported by NSF grants CNS-1646305, CCF-1763788, and OAC-1839010. We acknowledge support for research on flaky tests from Facebook and Google.

REFERENCES

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

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

[3] https://github.com/apache/hadoop, 2020.

[4] https://github.com/apache/hadoop/pull/1868, 2020.

[5] https://github.com/quarkusio/quarkus, 2020.

[6] https://github.com/quarkusio/quarkus/pull/6839, 2020.

[7] https://github.com/alibaba/fastjson, 2020.

[8] https://github.com/alibaba/fastjson/pull/2996, 2020.

[9] https://github.com/Graylog2/graylog2-server, 2020.

[10] https://github.com/Graylog2/graylog2-server/pull/8087, 2020.

[11] https://github.com/nutzam/nutz, 2020.

[12] https://github.com/nutzam/nutz/pull/1541, 2020.

[13] https://github.com/apache/incubator-shardingsphere, 2020.

[14] https://github.com/spring-cloud/spring-cloud-config, 2020.

[15] https://github.com/zhangxd1989/spring-boot-cloud, 2020.

[16] https://github.com/dropwizard/dropwizard/pull/3284, 2020.

[17] 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.

[18] https://github.com/OpenFeign/feign/pull/1165, 2020.

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

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

[21] Illinois Dataset of Flaky Tests (IDoFT). http://mir.cs.illinois.edu/flakytests, 2020.

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

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

[24] Javaparser. http://javaparser.org, 2020.

[25] JSONassert. http://jsonassert.skyscreamer.org, 2020.

[26] JsonUnit. https://github.com/lukas-krecan/JsonUnit, 2020.

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

[28] Maven. https://maven.apache.org, 2020.

[29] NonDex. https://github.com/TestingResearchIllinois/NonDex, 2020.

[30] Test verification. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification, 2020.

[31] TestNG. https://testng.org/doc, 2020.

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

[33] Dataset for "Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications". https://doi.org/10.5281/zenodo.4539907, 2021.

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

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

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

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

[39] A. Gambi, J. Bell, and A. Zeller. Practical test dependency detection. In *ICST*, 2018.

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

[41] 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.

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

[43] 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.

[44] D. Kim, J. Nami, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE*, 2013.

[45] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *ISSTA*, 2019.

[46] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta. A study on the lifecycle of flaky tests. In *ICSE*, 2020.

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

[48] W. Lam, S. Winter, W. Anjiang, T. Xie, D. Marinov, and J. Bell. A large-scale longitudinal study of flaky tests. *PACMPL*, 4(OOPSLA), 2020.

[49] 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.

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

[51] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Professional, 2000.

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

[53] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *FSE*, 2014.

[54] M. Mirzaaghaei, F. Pastore, and M. Pezzè. Supporting test suite evolution through test case adaptation. In *ICST*, 2012.

[55] M. Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, 2018.

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

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

[58] 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*, 46(9), 2016.

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

[60] 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.

[61] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury. Repairing crashes in Android apps. In *ICSE*, 2018.

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

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

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

[65] 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.