Finding Polluter Tests Using Java PathFinder

Pu Yi
Peking University
Beijing, China
lukeyi@pku.edu.cn

Anjiang Wei
Peking University
Beijing, China
weianjiang@pku.edu.cn

Wing Lam
University of Illinois
Urbana, IL, USA
winglam2@illinois.edu

Tao Xie
Peking University
Beijing, China
taoxie@pku.edu.cn

Darko Marinov
University of Illinois
Urbana, IL, USA
marinov@illinois.edu

ABSTRACT

Tests that modify (i.e., "pollute") the state shared among tests in a test suite are called "polluter tests". Finding these tests is important because they could result in different test outcomes based on the order of the tests in the test suite. Prior work has proposed the PolDet technique for finding polluter tests in runs of JUnit tests on a regular Java Virtual Machine (JVM). Given that Java PathFinder (JPF) provides desirable infrastructure support, such as systematically exploring thread schedules, it is a worthwhile attempt to re-implement techniques such as PolDet in JPF. We present a new implementation of PolDet for finding polluter tests in runs of JUnit tests in JPF. We customize the existing state comparison in JPF to support the so-called "common-root isomorphism" required by PolDet. We find that our implementation is simple, requiring only ∼200 lines of code, demonstrating that JPF is a sophisticated infrastructure for rapid exploration of research ideas on software testing. We evaluate our implementation on 187 test classes from 13 Java projects and find 26 polluter tests. Our results show that the runtime overhead of PolDet@JPF compared to base JPF is relatively low, on average 1.43x. However, our experiments also show some potential challenges with JPF.

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

Keywords: Polluter tests, flaky tests, PolDet, Java PathFinder

1. INTRODUCTION

Flaky tests [9] can nondeterministically pass or fail for the same code under test. Finding flaky tests proactively is important because their failures can mislead developers to debug their recent code changes [5]. Specifically, developers would assume the cause of the failures is in the code changes if the tests passed before the code changes but fail after the code changes. Some flaky tests are *order dependent* [17], i.e., they depend on the test-suite order and can pass in one order but fail in another order. These order-dependent tests most commonly [13] involve a pair of a *polluter test*, which modifies (i.e., "pollutes") the state shared among tests, and a *victim test*, which fails when run after the polluter test but passes otherwise. Strictly speaking, only the victim tests are flaky tests, because they can pass or fail, but finding polluter tests is important to prevent victim tests from failing.

Prior work [4] has proposed a technique, called PolDet, to find polluter tests during regular testing on a Java Virtual Machine (JVM). The idea of PolDet is to find the polluter tests "by definition": run each test from the test suite, capture the shared pre-state (before the test starts running) and the post-state (after the test finishes), and compare these two states. In PolDet's original implementation, where the running of JUnit tests is on a regular JVM, a shared state consists of the part of the heap reachable from the static (class) fields.

Implementing PolDet requires a few key features. Specifically, the original PolDet implementation uses the XStream library [16] for XML serialization to traverse the relevant part of the heap and serialize it into XML for later comparison. The serialization starts from a set of roots, i.e., from a map whose keys are fully qualified names of the static fields and whose values are either primitive values or the references to the actual heap objects pointed to by these fields. PolDet uses a Java agent to track all loaded classes to identify the static fields. PolDet also uses a modified JUnit runner to call the logic for capturing and comparing states.

PolDet's comparison of Java states requires handling an important technical challenge, namely, lazy class loading, which could cause false alarms for state differences. Java programs do not load all of the classes necessary for program execution at the start of the execution but dynamically discover what classes are needed and load them only when needed. As a result, the pre-state and the post-state for a test can often trivially differ because they have different static fields whenever the test execution loads a new class (that has at least one static field). Reporting such state differences would be undesirable and create false alarms in PolDet.

To avoid reporting these false alarms, PolDet defines the notion of *common-root isomorphism* [4]. It views pre-states and post-states as multi-rooted graphs whose nodes represent heap objects (including arrays) and primitive values, and whose edges represent object fields (including array indices). The graph roots correspond to the static fields. PolDet finds the set of common roots for the two graphs and compares whether the subgraphs reachable from these common roots are isomorphic (up to the node identity). Precise definitions are in the original PolDet paper [4].

Given that Java PathFinder (JPF) [15] provides desirable infrastructure for systematically testing Java programs, e.g., for exploring thread schedules, it is worthwhile to re-implement techniques such as PolDet in JPF, as is the focus of our work in this paper. Our re-implementation of PolDet in JPF is relatively simple, demonstrating JPF's high extensibility to support testing techniques such as PolDet. In particular, we develop a new, customized state comparison in JPF to support common-root isomorphism required by PolDet. We also write a JUnit listener to call our code when a test starts (to capture the pre-state) and when a test finishes (to capture the post-state and to compare its appropriate parts with the pre-state). In total, our implementation has ∼200 lines of code. It is pending as a pull request to JPF (`https://github.com/javapathfinder/jpf-core/pull/285`). We refer to our implementation as *PolDet@JPF* and the original implementation as *PolDet@JVM*.

We evaluate our PolDet@JPF implementation on 187 test classes from 13 open-source Java projects used in the original PolDet

```java
// in PotionTest class
@Test // the polluter test
public void setExtended() {
  PotionEffectType.registerPotionEffectType(new
    PotionEffectType(19) {
    /* other methods */
    @Override
    public String getName() {
      return "Poison";
    }
  });
  /* some checking */
}

// in PotionEffectType class
private static final Map<String, PotionEffectType>
  byName = new HashMap<String, PotionEffectType>();
public static void registerPotionEffectType(
  PotionEffectType type) {
  /* check that the argument is valid */
  byName.put(type.getName().toLowerCase(), type);
}
public static PotionEffectType getByName(String name) {
  Validate.notNull(name, "name cannot be null");
  return byName.get(name.toLowerCase());
}
```

Figure 1: A polluter test `setExtended` found with PolDet@JPF

evaluation [4]. These projects have a total of 991 test classes, but unfortunately, JPF (even *without* our extensions) could not run more than 187 classes. We find 26 polluter tests. (We use the term "test" to refer to one JUnit test method.) We also measure the overhead that PolDet@JPF has over base JPF. The average (geometric mean) overhead is fairly low, only 1.43x, and is quite stable across projects, ranging from 1.12x to 1.86x. In contrast, PolDet@JVM reports the average overhead of 4.50x, but ranging much more widely, from 1.07x to 1029.57x [4, Fig. 4]. In summary, when JPF can execute a test class, our PolDet@JPF works fairly well and can search for polluter tests relatively fast.

2. EXAMPLE

We next show one example of a polluter test found by PolDet@JPF in our experiments. Figure 1 shows the relevant snippets of the test code and code under test. This test is from the `Bukkit` project that provides an extension for the popular Minecraft game. The test is `PotionTest.setExtended`, which registers a new subclass object of an abstract class `PotionEffectType`. The test assigns the name for an object by overriding the `getName` method. The abstract class `PotionEffectType` contains a static field whose type is `Map` and whose name is `byName`. This field supports getting a registered subclass by its name using the `getByName` method. When registering this new subclass object, the `setExtended` test eventually adds an entry to the map in `byName`. Therefore, the pre-state and post-state differ due to this additional entry in the map.

This change of shared state may not be obvious to the developers who wrote this test. However, this change could be easily observed by another test that would try to use the `getByName` method from the `PotionEffectType` class. Accordingly, this polluter test could potentially cause some newly added victim test to fail if the victim test runs after this test and relies on the content of the map. Using the state-serialization feature in our PolDet@JPF, we can find the change of the program state by comparing the state serialization results in the test pre-state and post-state. This test was also found and reported in the original PolDet paper [4].

3. IMPLEMENTATION

The PolDet technique finds polluters by comparing states before and after test runs, i.e., test pre-state and post-state. According to the original implementation [4], the key features required for implementing PolDet in Java are (1) finding the set of all loaded classes (by the JVM) from executing the tests to get the set of all static fields from these classes; (2) capturing the shared heap state reachable from static fields to enable state comparison; (3) comparing the states using the "common-root-isomorphism" technique to handle dynamic class loading [4]; and (4) extending JUnit to make appropriate calls to the core system that captures and compares states. We implement our PolDet@JPF tool based on the jpf-core code [7]. Before we describe how we implement each of the key features, we first provide a high-level overview.

3.1 Overview

JPF implements a JVM that runs on the host JVM and interprets the application code. JPF has two execution layers: the native JVM level in which JPF runs and the JPF level in which the application code runs. JVM and JPF load classes only on demand.

Our PolDet@JPF implementation runs JUnit tests at the JPF level but captures and compares the states in the native JVM level. PolDet@JPF extends the existing JPF state serialization for our purpose. Before each test starts and after it finishes, our JUnit listener calls our serialization to capture and compare the states. To enable these calls, we expose a new *native peer* that can be called from the Java code interpreted by JPF to jump into the native JVM that executes JPF.

3.2 Finding loaded classes

At the native JVM level, it is easy to find the set of all classes loaded by the Java code interpreted by JPF. (In contrast, finding classes loaded by JVM requires using a Java agent as done by the original PolDet@JVM [4, §4.3].) Our JPF state serialization finds loaded classes while capturing the shared state.

3.3 Capturing shared state

The key of our implementation is to capture the shared state (the pre-stare and post-state for a test). We leverage the existing JPF state serialization, specifically the `FilteringSerializer` class and write a sub-class of it called `PolDetSerializer`. Traditionally, JPF calls state serialization at "choice points", where it matches the current state with the previously encountered states, performing stateful search and stopping the current execution path if it matches a previously encountered state. Instead, our code calls into state serialization before and after executing each test.

The `FilteringSerializer` produces an integer array that serializes (almost) the entire state of the JVM interpreted by JPF, including the static area (loaded classes), thread information, stack frames, and the heap reachable from all of the roots. Our `PolDetSerializer` ignores two kinds of fields. First, we ignore all of the fields from JUnit, i.e., all instance fields in classes starting with `org.junit`. Because we run the tests and JUnit in JPF, the JVM interpreted by JPF has the entire JUnit state. For example, one of the JUnit fields, named `org.junit.runner.Result.count`, stores the number of executed tests. This field changes for each test, and we do not want to label every test as a polluter simply because JUnit changes this counter field. Second, we ignore all of the fields whose class or field names contain `cache` (case insensitive). For example, JPF keeps some cached objects in `gov.nasa.jpf.vm.BoxObjectCacheManager` to speed up execution. Again, these objects can change for many (albeit not all) test executions, but their change does not indicate that the test is truly a polluter test. As a result, PolDet@JPF could have false negatives for some of JPF's test classes.

```
class PolDetListener extends RunListener {
  // native method declarations for JPF
  public native static void capturePreState();
  public native static boolean compareStates();
  public void testStarted(Description description) {
    capturePreState(); // also collect loaded classes
  }
  public void testFinished(Description description) {
    if (!compareStates()) { // compare pre- & post-state
      /* print "polluter found" for the method */
    }
  }
  /* testRunStarted and testRunFinished methods collect
    and print statistics */
}
```

Figure 2: JUnit listener to capture the pre-state and post-state

In addition to capturing the state, our code also (1) at the start of each test records the set of loaded classes, and (2) at the end of each test calls our `PolDetSerializer` to capture, as the roots for serialization, only the static fields from the classes that are loaded before the test starts (in other words, our code ignores the static fields from the classes newly loaded during the test execution). Thus, we ensure that the pre-state and post-state have the same set of roots, based on the classes that are loaded in the pre-state, effectively providing the "common-root isomorphism" [4, §4.4]. This comparison can have false negatives, e.g., a polluter test cannot be found if it is checked first by PolDet@JPF.

3.3.1 Debugging support

To compare serialized states more easily, and inspired by the existing `DebugCFSerializer` class in JPF, we use a feature that is not necessary to detect polluter tests but greatly aids in debugging why a test is a polluter. Namely, the `FilteringSerializer` (as every other state-serialization class) in JPF returns an integer array that compactly encodes the entire state. While such an array is good for performance (both space and time) of state comparison, the array makes it rather challenging to determine which part of the shared state is polluted.

In addition to the integer array, our debugging feature can also print a more human-readable graph representation of the state. Each edge in the graph can be a field on the heap (reachable from the root static fields), e.g., if `objRef1` and `objRef2` are two object references used by JPF, and the field named `f` of `objRef1` has value `objRef2`, our debug output has a triple `objRef1, f, objRef2`. Our implementation also handles primitive values, arrays (whose elements are serialized with array indices instead of field names for objects), and various kinds of state graph roots: static fields, stack frames, and thread information. This feature makes it easier to find which parts of the pre-state and post-state differ for a polluter test. We can traverse from the difference back to the root in the state graph to understand how the changes happen. We use this feature to print the states only after a test is reported as a polluter, i.e., when we inspect the pollution. We do not print the states while determining whether some test is a polluter, because printing this debug information would add a substantial overhead. The debugging feature requires ∼50 more lines of code.

3.4 Comparing shared states

State comparison is straightforward because of how shared states are captured, i.e., ignoring irrelevant parts of the state and traversing only the heap reachable from the common roots. PolDet@JPF simply compares the two integer arrays, for pre-state and post-state, and reports a test as a polluter if the arrays differ.

```
// implementation of native methods at the JVM level
public class JPF_PolDetListener extends NativePeer {
  static int[] preState; // store for later comparison
  static PolDetSerializer serializer = new
    PolDetSerializer(); // stores loaded classes
  @MJI
  public static void capturePreState____V(MJIEnv env,
    int classRef) {
    serializer.attach(env.getVM());
    preState = serializer.getState(PRESTATE);
  }
  @MJI
  public static boolean compareStates____Z(MJIEnv env,
    int classRef) {
    serializer.attach(env.getVM());
    int[] postState = serializer.getState(POSTSTATE);
    return Arrays.equals(preState, postState);
  }
}
```

Figure 3: Native peer showing the key methods

3.5 Extending JUnit

Our current PolDet@JPF implementation supports JUnit 4, because it is still the most widely used JUnit version, although JUnit 5 is the latest version and is becoming widely used. We do not need to change the JUnit 4 core itself but just implement a JUnit listener, as shown in Figure 2, to call our methods for capturing and comparing shared states. In particular, before each test, we capture the pre-state (including the set of loaded classes), and after each test, we (1) capture the post-state (reachable from the previously loaded classes) and (2) compare the states and print that the test is a polluter if the states differ, as shown in Figure 3. The implementation for capturing states could be further optimized to reuse the post-state of one test for the pre-state of the next test; we do not currently do so because the overhead of PolDet@JPF is already quite low compared to base JPF.

4. EXPERIMENTS

We evaluate our PolDet@JPF on a subset of projects (13 out of 26) used in the original PolDet@JVM evaluation [4, Fig. 3]. Our initial plan was to repeat the exact experiments from PolDet@JVM. However, we encounter two problems. First, some of the code versions used in PolDet@JVM evaluation are rather old and cannot compile "out-of-the-box", e.g., due to missing library dependencies. As a result, we decide to use the latest versions of all these projects. Second, even when projects could compile, JPF could not run a large number of test classes from these projects.

To determine which test classes to use in our experiments, we proceed as follows. We first clone the latest version of the project from its GitHub repository and discard projects that are not Maven-based or that cannot compile with Maven. At this point, we have a total of 991 test classes. We then run *each* test class by itself on JPF and discard classes that JPF could not run, e.g., due to missing native peers or incorrect native peers that return the wrong values. (Note that these issues are *not* due to our PolDet@JPF extensions of JPF.) We did initially try to add some native peers, but we found the effort rather futile as we had dozens of such problems, e.g., with code calling into graphic interfaces (even when running fully on the command line), making network calls, or using other I/O. In the end, we are left with 187 (out of 991) test classes belonging to 13 projects that JPF could run.

Table 1 shows some statistics of the projects used in our experiments. For each project, we tabulate the name of the repository, the exact commit, and the number of test classes and test methods that we could run on JPF. For each project, we collect all of

GitHub project slug	commit SHA	# test classes	# test methods	polluters	time [s] PolDet @JPF	time [s] base JPF	overhead of PolDet	time [s] JVM	overhead of JPF
ahorn/android-rss	4f0bd7cd	2	20	0	0.268	0.144	1.86	0.040	3.60
apache/httpcomponents-client	918ac153	33	240	* 7	4.516	3.225	1.40	0.649	4.97
Athou/commafeed	b597c655	2	7	0	0.142	0.085	1.67	0.016	5.31
Bukkit/Bukkit	f210234e	31	271	* 5	5.088	3.627	1.40	1.025	3.54
caelum/vraptor4	593ce9ad	26	129	4	7.351	6.552	1.12	1.498	4.37
fakemongo/fongo	7301aa1f	1	8	0	0.100	0.054	1.85	0.009	6.00
github/maven-plugins	8d6d4939	2	5	0	0.440	0.339	1.30	0.186	1.82
jopt-simple/jopt-simple	81e6a674	60	384	* 1	7.632	5.843	1.31	1.753	3.33
nurkiewicz/spring...repository	fafe7dc8	1	13	0	0.182	0.105	1.73	0.051	2.06
perwendel/spark	5ca2a0a6	14	53	2	0.967	0.624	1.55	0.138	4.52
qos-ch/slf4j	62309486	5	37	1	1.133	1.003	1.13	0.072	13.93
tbruyelle/spring-test-mvc	31530307	6	53	2	2.785	2.472	1.13	0.331	7.47
twitter/hbc	b3c73e60	4	22	4	0.587	0.417	1.41	0.126	3.31
sum (geomean for overhead)		187	1,242	**26**	31.191	24.490	**1.43**	5.894	**4.30**

Table 1: Key statistics of our experiments for finding polluter tests using our PolDet@JPF implementation; '*' denotes that one of the polluter tests is a parameterized unit test that has multiple runs that pollute the state, as discussed in Section 4.1.3

the test classes that JPF could run individually, and then we run these classes all at once (1) in our PolDet@JPF to find polluters and measure runtime, (2) in JPF to measure the runtime for base JPF, and (3) in a regular JVM to measure the overhead of base JPF over JVM. All timing experiments are performed on a server with 32 Physical CPUs (Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz) using Java OracleJDK version 1.8.0_261.

Table 1 also summarizes the results that we obtain in our experiments. For each project, we tabulate the number of polluters, the time for running all test classes that could run, the overhead of our PolDet@JPF over base JPF, as well as the time for running all (187) test classes in JVM, and the overhead of base JPF over JVM. The overhead of our PolDet@JPF stems from capturing and comparing states before and after each test. (The timing experiments do *not* run our debugging that prints states in a format easier for comparison.) We find the overhead of PolDet@JPF over base JPF quite acceptable, on average (geometric mean) just 1.43x, and ranging from 1.12x to 1.86x across projects. For these test classes, we also find that the overhead of JPF over JVM to be quite acceptable, on average just 4.30x, and ranging from 1.82x to 13.93x across projects. In comparison, PolDet@JVM was reported to have an average overhead of 4.50x, but ranging much more widely, from 1.07x to 1029.57x [4, Fig. 4]. Note that we do *not* directly compare PolDet@JPF and PolDet@JVM as they were run on different project versions and tests.

4.1 Analysis of polluter tests

Our experiments find a total of 26 polluter tests. These polluter tests are in 8 projects, i.e., more than half of the studied 13 projects. This result already shows that polluters may be widely present across various projects. The overall ratio of polluter tests that our experiments find among all of the tests is 2.08% (26 out of 1,242 tests). This ratio is lower than reported for PolDet@JVM, 5.31% (324 polluters out of 6105 tests). The reason could be that more *complex* tests, which manipulate larger portions of the state and involve more extensive operations, are both more likely to be polluters and also less likely to be able to run in JPF.

We have inspected all of the tests that PolDet@JPF reports as polluters. Our initial attempt to simply inspect the test code (and potentially directly invoked code under test) proved to be rather challenging because the pollution can often be deep in the heap. Therefore, we develop our debugging support (Section 3.3.1) to make it easier to locate the state difference, as well as the static field that is a root from which the difference can be reached. Section 2 has already discussed one example test. We next discuss several more selected example tests. Admittedly, many of the state differences would be hard to observe with other "victim" tests; this result is again in contrast to the original PolDet@JVM evaluation [4] presumably for the same reason of test complexity. Nevertheless, we still find some interesting state differences.

4.1.1 Real pollution

The test `ResourceUtilsTest.testGetFile...` from the `spark` project modifies the static field `java.net.URL.handlers` from the Java standard library. The field points to an object of the type `Hashtable`. This test adds an element to the hashtable. A potential victim test could easily observe such a change by invoking the static method `URL.getURLStreamHandler` that looks up the hashtable `handlers`.

4.1.2 Pollutions due to caching

The four tests from the `HttpHostsTest` class in the `hbc` project all modify the same part of the state that happens to be in the standard library. Specifically, these tests involve a list of host addresses whose order is randomized using the standard library method `Collections.shuffle`. This method uses a pseudorandom number generator (PRNG) stored in a static field. Calling `shuffle` changes the internal state of the PRNG. Strictly speaking, this change is a pollution because another test could observe it by checking the result of `shuffle` or by using reflection to directly access the internal state of the PRNG. However, it is unlikely to have a realistic test that depends on such an observation.

The test `XpathRequestMatchersTests.testStringNoMatch` from the `spring-test-mvc` project checks an operation from the XPath query language for XML documents. The test executes code that involves serializing object graphs into XML strings. This serialization uses the `StringBufferPool` class to cache string buffers that can often be reused in serialization. The test execution pollutes this internal cache; while this pollution is indeed a true modification of the shared state, it is unlikely that any other test would be a "victim" that would observe the internal cache state and fail when run after this polluter but pass when run before it.

The test `OptionException...Test.givesCorrectExceptionMessage` from the `jopt-simple` project uses the locale feature to set the user's language preferences. The execution of this test modifies the shared state reachable from the field `Locale.LOCALECACHE` in the standard library. This field stores a cache that is lazily initialized. The test execution makes this cache bigger, thus polluting the state. It is unlikely that any other test would be a "victim" that would observe such cache state.

The test `ExceptionMapperTest.testGetInstance...` from the `spark` project modifies a part of the state that is not too far from the test code. The test is for the class `ExceptionMapper` and modifies its static field `ExceptionMapper.servletInstance`. This field points to an object of the type `HashMap`. This test replaces one empty map with another. According to the common-root isomorphism [4], the pre-state and post-state *are* isomorphic, but `FilteringSerializer` does not declare these states as equivalent (in other words, `FilteringSerializer` does not fully break heap symmetry).

4.1.3 Parameterized unit tests

We have also found some interesting cases of parameterized unit tests [14] that are polluters. While 26 *tests* are polluters, there are actually more test *runs* that pollute. For example, the test `TestClassicHttpRequests.testCreateFromString` from the project `httpcomponents-client` is a parameterized unit test, and it has 8 sets of parameters that all pollute the shared state. The test `OptionException...Test.givesCorrectExceptionMessage` from `jopt-simple` also has 8 sets of parameters but only pollutes for the first set. The test `DyeColorTest.getWoolDyeColor` from `Bukkit` has 16 sets of parameters but only one set pollutes the shared state.

5. RELATED WORK

There is a growing body of research on flaky tests. Luo et al. [9] presented a characterization of flaky tests, identifying a dozen kinds of flaky tests based on the root causes of nondeterminism. Some of the earliest work [10, 17] considered flaky tests that depend on the order of the tests in the test suite, and this topic continues to garner attention [2, 8, 13]. Specifically, the work on iFixFlakies [13] proposed a technique to fix one kind of flaky tests and also named tests related to flaky tests due to test-suite order, including "polluters" addressed in this paper, as well as "victims" and "brittles" that can fail due to the shared state.

The most related work by Huo and Clause [6] proposed the notion of "brittle assertions", i.e., test assertions that depend on the shared state that is read by the test but not written by the test. Thus, tests with such brittle assertions can fail if run in a wrong pre-state, even if the code under test has no faults. In particular, victim tests pass when run in isolation (starting from the default JVM state) but fail when run after other (polluter) tests; in contrast, brittle tests fail when run in isolation but pass when run after other tests [13]. Moreover, Huo and Clause proposed finding tests with brittle assertions via taint tracking, and they implemented a sophisticated system in JPF [6]. Our work is complementary to theirs because PolDet@JPF finds polluter tests.

Most prior and ongoing work on flaky tests has been on open-source Java projects, e.g., Alshammari et al. [1] use machine learning to predict which tests are flaky. However, other domains have also been analyzed, e.g., Gruber et al. [3] report on thousands of flaky tests in Python, and Romano et al. [11] report on hundreds of flaky tests in Android and web applications. Besides academic research, various companies have published papers about flaky tests, reporting the importance of the problem, with Harman and O'Hearn presenting a compelling overview [5].

6. CONCLUSIONS AND FUTURE WORK

We have presented a novel implementation, called PolDet@JPF, of the previously proposed PolDet [4] technique to find polluter tests. Our work highlights some positive aspects of JPF: (1) JPF enables our implementation to be fairly simple, with just $\sim$200 lines of code; (2) the runtime overhead of PolDet@JPF over base JPF is relatively low, with 1.43x on average; and (3) the runtime overhead of base JPF over JVM is relatively low, with 4.30x. Our experiments also show a negative aspect of JPF: JPF currently cannot handle a lot of real code, e.g., it could run only 187 out of 991 test classes that we have tried.

Future work could, in general, help to increase JPF's applicability to more code, e.g., implementing more peer methods or advancing jpf-nhandler [12]. Specifically for PolDet@JPF, providing visualization or better output information could help developers spot the root of pollution more easily. Another interesting topic would be automatically removing state pollution from tests.

Acknowledgments

We thank August Shi and Zhengxi Li for engaging discussions about flaky tests in general and polluter tests in particular. This work was partially supported by NSF grants CNS-1564274, CCF-1763788, CCF-1816615, and CCF-1956374. We thank Facebook and Google for supporting research on flaky tests. Wing Lam is supported by a Google – CMD-IT LEAP Dissertation Fellowship. Tao Xie is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China, and is the corresponding author.

7. REFERENCES

[1] A. Alshammari, C. Morris, M. Hilton, and J. Bell. FlakeFlagger: Predicting flakiness without rerunning tests. In *ICSE*, 2021.

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

[3] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser. An empirical study of flaky tests in Python. In *ICST*, 2021.

[4] A. Gyori, A. Shi, F. Hariri, and D. Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*, 2015.

[5] M. Harman and P. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *SCAM*, 2018.

[6] C. Huo and J. Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*, 2014.

[7] JPF Core. `https://github.com/javapathfinder/jpf-core`.

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

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

[10] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. In *ESEC/FSE*, 2011.

[11] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang. An empirical analysis of UI-based flaky tests. In *ICSE*, 2021.

[12] N. Shafiei and F. v. Breugel. Automatic handling of native methods in Java PathFinder. In *SPIN*, 2014.

[13] 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.

[14] N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/SIGSOFT FSE*, 2005.

[15] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE*, 2000.

[16] XStream. `https://x-stream.github.io`.

[17] 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.