Practical Test Dependency Detection

Alessio Gambi
Passau University
Passau, Germany
alessio.gambi@uni-passau.de

Jonathan Bell
George Mason University
Fairfax, VA USA
bellj@gmu.edu

Andreas Zeller
Saarland University and CISPA
Saarbrücken, Germany
zeller@cs.uni-saarland.de

***Abstract*—Regression tests should consistently produce the same outcome when executed against the same version of the system under test. Recent studies, however, show a different picture: in many cases simply changing the order in which tests execute is enough to produce different test outcomes. These studies also identify the presence of dependencies between tests as one likely cause of this behavior. Test dependencies affect the quality of tests and of the correlated development activities, like regression test selection, prioritization, and parallelization, which assume that tests are independent. Therefore, developers must promptly identify and resolve problematic test dependencies.**

This paper presents PRADET, a novel approach for detecting problematic dependencies that is both effective and efficient. PRADET uses a systematic, data-driven process to detect problematic test dependencies significantly faster and more precisely than prior work. PRADET scales to analyze large projects with thousands of tests that existing tools cannot analyze in reasonable amount of time, and found 27 previously unknown dependencies.

***Index Terms*—Test dependence, detection algorithm, empirical study, flaky tests, data-flow**

I. INTRODUCTION

A fundamental property of good regression tests is reproducibility: executing the same test suite on the same version of a system should always produce the same result. Rerunning tests on the same code, or running them in a different order should not cause the outcome of any test to change. However, in practice this is not always the case, and tests may be *flaky*, passing and failing nondeterministically. Flaky tests disrupt regression testing [1], and are generally considered as bugs [2]. Recent studies suggest that one possible source for this apparently inexplicable behavior is the presence of ordering dependencies between tests [3], [4], [5]. When dependencies are ignored and tests are run in a different order (or only a subset of tests run), then tests may unexpectedly fail [6].

Test dependencies may arise when there is a *read-after-write* (RAW) data-flow dependency between several tests: For example, if test T_1 writes some value V, this introduces a data dependency if test T_2 then reads that value V. If test T_1 were not run before T_2 (for instance, in the context of test selection [7], test prioritization [8] or test parallelization [9]), then T_2 might fail. However, simply observing a data dependency between the two tests is not sufficient to decide that T_2 will fail if this dependency is violated. For instance, T_2 may be able to initialize that value V itself, or V might not actually impact the outcome of the test. If the outcome of T_2 changes when T_1 is not run before it, then we say that there is a *manifest dependency* between T_2 and T_1. Manifest dependencies are problematic dependencies that developers need to identify.

In some cases, data-flow between tests is a feature, and not a bug (e.g., to cache shared state and avoid repeated setup for each test), but it is still necessary for developers to be aware of these dependencies so that they can be respected. To that ends, several techniques have been proposed to detect test dependencies and warn developers of their presence [9], [10].

Unfortunately, the problem of finding manifest test dependencies is hard (NP-complete [3]). Zhang et al.'s DTDETECTOR tool finds these manifest dependencies by executing tests in all possible combinations, a solution often not scalable in practice, or only in some combinations using unsound heuristics [3]. To avoid running tests multiple times, we previously proposed ElectricTest, a test dependency detector based on data-flow analysis [9]. While ElectricTest is generally much faster than DTDETECTOR, ElectricTest reports *all* tests with data dependencies to developers, many of which may be benign and unimportant to the outcome of the tests.

In this paper, we present PRADET, a practical bug-hunting tool that detects manifest test dependencies in a reasonable amount of time and also scales to large projects with thousands of tests. PRADET combines the precision of DTDETECTOR and the speed of ElectricTest. PRADET builds upon two key ideas: *(i)* compute an over-approximation of the manifest dependencies using a lightweight data-flow analysis in the context of a reference tests execution; that is, PRADET can identify all the manifest dependencies that are rooted in the observed data dependencies. And, *(ii)* it refines this over-approximate solution by testing the data dependencies and removing the non-manifest ones.

By relying on information about data dependencies, PRADET can identify the few tests that are involved in these data dependencies, and executes them out of order to possibly uncover manifest dependencies. Compared to testing all the possible test execution orders, PRADET drastically reduces the amount of test executions required to expose manifest dependences. Compared to simpler heuristics for re-ordering tests to expose manifest dependencies, PRADET is more effective.

Figure 1 illustrates PRADET's approach: it starts by executing the tests according to a reference order and uses dynamic data-flow analysis to uncover the dependencies between tests; next, it iteratively selects a data dependency and checks if that corresponds to a manifest dependency; this is done by

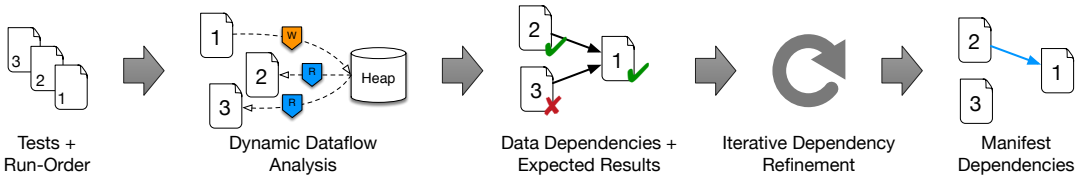

Figure 1. Overview of PRADET's approach. The input is a test suite and a reference run-order of the tests. PRADET then performs a dynamic data-flow analysis, resulting in data dependency information, and knowledge about the expected results of each test. Next, PRADET runs its iterative dependency refinement algorithm, which filters out all the unproblematic data dependencies.

executing *only* the relevant tests out of order. If the execution leads to a test outcome different than the one of the reference execution, PRADET reports a manifest dependency; otherwise, it removes the data dependency from consideration.

PRADET is based on dynamic analysis and test executions, hence is not – and as a dynamic tool cannot be – complete. It relies on data dependencies that are discovered via dynamic analysis after a single execution of the tests; hence, it might miss data dependencies that occur non-deterministically. We assume that test executions are *deterministic modulo run-order* during the refinement, which might not be always the case. There is presently no alternative approach to solving this problem which is sound: even a naïve approach of rerunning all tests in all possible orders is unsound in the presence of nondeterminism.

Empirically, we have found that PRADET is effective in identifying manifest dependencies, which are otherwise hard to find in practice, especially for projects with extensive test suites. As our evaluation shows, on projects featuring small test suites (less than 500 tests), PRADET is as effective as state-of-art approaches; however, on projects featuring large test suites (more than 500 tests), PRADET is more effective: in 75% of the cases it discovered more manifest dependencies than state-of-art approaches. Additionally, PRADET can analyze large test suites with more than 3,800 tests involving more than 37,000 data dependencies in a reasonable amount of time (approximately 4 hours).

The primary contributions of this paper are:

1) the presentation of PRADET, a novel approach to efficiently detect manifest test dependencies. PRADET uses a systematic, data-driven process to discover test dependencies which is different than state-of-art solutions based on executing tests according to predefined and random orders.
2) the evaluation of PRADET on nineteen open-source projects that shows how it can effectively analyze test suites of different sizes and find manifest dependencies that prior state-of-art approaches could not find;
3) a historical evaluation which shows that almost all these manifest dependencies existed when at least one test involved in the dependency was written, suggesting to run PRADET only when new tests are written.

II. MOTIVATING EXAMPLE

Software tests are typically assumed to be *isolated* and *independent*. That is to say, they should not make persistent changes to the environment or leak data among tests.

```
1  public class DataSourceTest {
2    public static DataSource data;
3    @Test
4    public void testSetField() {
5      String short_name = "Repository";
6      RepoKind repo_kind = RepoKind.HG;
7      data = new DataSource(short_name, "path", repo_kind);
8      assertEquals(short_name, data.getShortName());
9      assertEquals(repo_kind, data.getKind());
10   }
11   @Test
12   public void testSetCloneString() {
13     assertTrue(data.getCloneString().equals("path"));
14     data.setCloneString("path_2");
15     assertTrue(data.getCloneString().equals("path_2"));
16   }
17   @Test
18   public void testToString() {
19     String short_name = "short_name";
20     RepoKind kind = RepoKind.HG;
21     String cloneString = "clone_string";
22     data.setShortName(short_name);
23     data.setKind(kind);
24     data.setCloneString(cloneString);
25     assertTrue(data.toString().equals(short_name + "_" +
        kind + "_" + cloneString));
26   }
27 }
```

Figure 2. Example of tests with data dependencies that result in manifest dependencies. The tests depend on each other because they write and read the same static field `data`. These tests are taken from Crystal, version 1.0.20111015.

Additionally, they should not depend on assumptions about the state of the environment before their execution. This is key for a variety of techniques that enhance regression testing such as regression test selection [7], minimization [11], prioritization [8] or parallelization [9]. This assumption is part of the greater *controlled regression testing assumption*, which states that as software is developed and regression test suites are repeatedly executed, the only factor that should change the outcome of the tests is a change to the code [12]. While this assumption may seem intuitive and trivial, upon closer inspection it can be very difficult to enforce. Tests might be controlled by a variety of other factors that developers may or may not be aware of, such as nondeterminism due to concurrency, reliance on external systems, or reliance on some state created by other tests.

In this paper, we consider the problem caused by *state polluting* tests: tests that leave the environment in a different state than they found it in [13]. State pollution might cause tests to change their behavior when re-ordered, because data from one test execution may flow into, and possibly interfere with, the execution of other tests. For example, a test might change the value of a global variable that another test later uses without overwriting or resetting it first. This creates a data-flow between the tests and possibly results in undesired effects, for

example causing the dependent tests to fail instead of pass. The impact of these dependent tests might be mitigated by enforcing a total ordering on the execution of each test, but, again, this approach is contradictory to existing techniques for test acceleration such as regression test selection [7], minimization [11], prioritization [8] or parallelization [9].

The code shown in Figure 2 is an example of a real test that is state polluting and results in a manifest dependency. The test method `testSetField` initializes the static field `data`, which is **`null`** at the start of the test, and not **`null`** at the end. Other tests, then, might come to depend on `testSetField` running before or after them, depending on their own assumptions about the state of the static field `data`. Indeed, the other two methods shown, `testSetCloneString` and `testToString`, implicitly depend on `testSetField` executing before them: they both assume that `data` is not **`null`** and are intended to pass.

State pollution might lead to data dependencies, which might lead to manifest dependencies, and hence, flaky test failures. All manifest test order dependencies co-occur with data dependencies, and all data dependencies co-occur with state pollution; hence, it is possible to use data dependencies and state pollution as proxies to indicate the presence of manifest dependencies. Note, however, that the presence of state pollution does not imply that there will be data dependencies between tests, and the presence of data dependencies does not imply that tests will fail if the tests are re-ordered.

Our goal is to automatically and practically detect these manifest dependencies, reporting these results to developers and allowing them to be aware of and respect that dependency. In contrast, prior work has focused on practical detection of state polluting tests (PolDet [13]), or practical detection of data dependencies between tests (ElectricTest [9]). Simply detecting tests that *could* cause data dependencies, or tests that have data dependencies between each other and *could* manifest as flaky test failures is insufficient to provide clear, actionable guidance to developers.

Developers need a tool that can report *precisely* dependent tests, allowing them to react accordingly. While our example test dependency described above and in Figure 2 is intentionally very easy to recognize, many test dependencies are far more complicated and difficult to debug. For example, manifest dependencies can stem from hidden, and more subtle, interactions through objects through aliasing. A careful inspection of the code reported in Figure 2 illustrates this case: In line 14, `testSetCloneString` asserts that the value of `data.cloneString` is the one previously set by `testSetField` (line 8); however, in line 25, `testToString` changes that value. As a consequence, running `testToString` before `testSetCloneString` results in a new read-after-write dependency, which, as before, turns out to be a manifest one.[1]

[1]The manifest dependency is revealed by running the tests in the order {`testSetField`, `testToString`, `testSetCloneString`}, which causes the assertion at line 14 to fail.

By detecting and reporting these dependencies to developers we can allow them to reduce the time spent debugging test failures caused by test dependencies, focusing on other development tasks. For instance, once a developer is aware of the dependencies in Figure 2, she might choose to refactor the tests to have an `@Before` method, which performs pre-test setup for each test, correctly initializing the `data` field.

III. Practical Test Dependency Detection

To detect manifest dependencies between tests, PRADET first performs a dynamic data-flow analysis. The result of this analysis is a set of tests that have data dependencies between each other. However, these data dependencies *do not* all represent manifest dependencies: many of the data dependencies may be benign (i.e., they can be ignored, and tests will still have the same outcome). This list of data dependencies is used to search for manifest dependencies between tests — that is, those data dependencies that can lead to flaky tests if tests are executed in a different order. Improving on prior work about test dependency detection, PRADET then iteratively refines the set of tests with data dependencies into a set of tests with manifest dependencies by re-executing them in different orders.

This high level approach is comparable to Zhang et al's "Dependence-Aware Bounded Algorithm" for detecting test order dependencies [3], with a key distinction in how we detect dependencies. Zhang's dependence-aware algorithm considered dependencies at the very coarse granularity of entire static fields: if two tests accessed the same static field, then they were considered dependent — an approach that is both unsound and incomplete in the presence of aliasing. In contrast, we leverage a precise, alias-aware data dependency detection approach.

A. Uncovering Test Data Dependencies

PRADET detects data dependencies by monitoring access to application state that is shared *within the memory of the process executing the test cases*, hence it focuses on detecting dependencies caused by global variables in Java applications; therefore, we do not claim PRADET to be complete. Nevertheless, PRADET aims to be precise, reporting only test dependencies that can truly manifest as flaky tests.

The approach to detect test data dependencies which PRADET implements is inspired by *ElectricTest* [9]; hence, in the remainder of this section, we summarize ElectricTest's key ideas. Then, we discuss important distinctions in PRADET.

ElectricTest implements two key ideas for uncovering data dependencies between tests: i) it utilizes dynamic data-flow analysis to identify conflicting writes and reads over objects in the heap; and, ii) it navigates the heap to find *all* conflicting objects. Dynamic data flow analysis is implemented by annotating each object using a data structure that stores information about the last test writing them. For primitive fields, which are not regular objects and cannot be directly annotated with this information, the data structure is attached to the objects which contain them. PRADET associates these data structures with

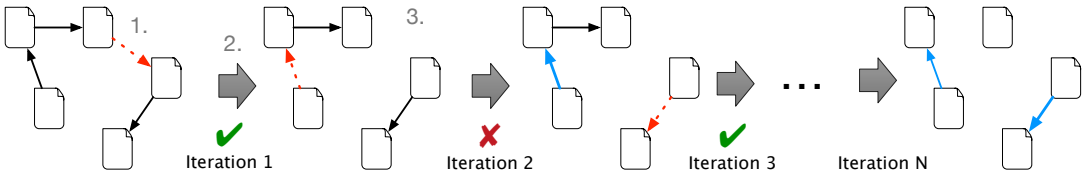

Figure 3. Data dependency refinement. At every iteration, PRADET tests a data dependency edge (red), and either it removes the edge (iteration 1) or identifies a manifest dependency (iteration 2). Eventually, all the data dependencies are tested and only the manifest dependencies remain.

variables in a way very similar to the Java taint tracking tool, Phosphor [14].

Before the execution of each test, all information about conflicts (stored in each of these structures) is cleared. Then, during the test execution, upon writing a value or setting an object, the data structure associated to the corresponding instance is updated and the currently executing test is recorded as the one doing the write. Upon reading a value or accessing an object, the data structure is checked for previous writes: If the object was written by a test different than the currently executing one, a conflict on the object is detected.

Note that, even though we are interested in data dependencies related to static references, it is not sufficient to store information about conflicts only for static object fields. As we discussed in Section II, other objects might be reachable through these static fields, and those can cause data dependencies as well. Such aliasing causes the simple "Dependence-Aware Bounded Algorithm" presented by Zhang et al. to be unsound. Therefore, after test execution, *all* objects that can be reached by navigating the heap are inspected for conflicts. Conflicting objects imply data flowing from previously executed tests to the current one, hence, they uncover test data dependencies. While it is slower to collect dependencies at this fine, per-field granularity than at a coarser level, we found that this upfront analysis time pays off in a significantly smaller set of data dependencies, reducing the number of tests that need to be checked in the following phase.

Two main distinctions separate PRADET and ElectricTest, and make ElectricTest less suitable then PRADET to discover manifest dependencies between tests. Compared to ElectricTest, PRADET precisely handles dependencies which involve String objects and enumerations, but it does not handle external data dependencies. Because of this, ElectricTest finds a much larger amount of data dependencies than PRADET; however, only a small fraction of those dependencies are true test dependencies, while the remaining dependencies are just an artifact of the dynamic data flow analysis implemented by ElectricTest. The basic approach implemented by ElectricTest cannot precisely handle the way Java implements *one-time* initializations of String objects and enumerations. In Java, Strings are *pooled* and *immutable*; therefore, setting two or more String objects to the same literal does not result in the actual instantiation of multiple String instances. Instead, only one string is instantiated and referenced by all the objects with the same content. A similar situation arises with `Enum` classes, since Java internally implements them with arrays of Strings. As a consequence, during the execution of tests, ElectricTest marks String objects and enumerations as written by the tests which firstly instantiate them. This causes spurious data dependencies if the same String value or enumeration is used in multiple tests. PRADET solves these two issues at once by tracking dependencies on String literals by reference. Instead of attaching the dependency and conflict information to the actual String objects, PRADET associates the data about reads and writes to the objects which contain the Strings. This way, the dependency information is decoupled from the way Java handles Strings or implements `Enum`, and the basic heap walking remains almost the same while its accuracy improves.

Regarding tracking external data dependencies, PRADET does not to follow ElectricTest example for the following two reasons. First, previous work by Bell et al. [9], [15], Zhang et al. [3] and Gyori et al. [13] show that the largest amount of data dependencies between tests are caused by (mis)using data *within the JVM*, while only a small fraction of data dependencies is caused by tests which *pollutes* the external environment. Second, ElectricTest can handle only elementary external test dependencies, through files and network, that can be tracked by an *in-process monitor*. Consider the case where the file `/tmp/f1` is written by `t1` and `/tmp/f2` is read by `t2`: if those two files are the same (e.g., symbolic links), then a data dependency exists between the tests, but ElectricTest cannot report them.

In summary, to detect data dependencies, PRADET executes the tests once and observes conflicting accesses to objects that can be reached by the tests through static references. If tests are data dependent there is the possibility of observing unexpected behaviors, i.e., the manifest dependencies, when they are executed out of order. Efficiently identifying these cases is the goal of iterative data dependency refinement which we describe in the next section.

B. Refining Data Dependencies to Manifest Dependencies

The main idea of the data dependency refinement is to check if a data dependency results in a manifest dependency when tests are re-executed out of order.

Given a set of data dependencies, PRADET discovers manifest dependencies following an iterative process (see Figure 3). First, it *selects* a target data dependency to check (highlighted as dotted-red arrow in the figure). Then, it *schedules* the execution of the tests such that all dependencies, except for the target one, are satisfied. Next, it *checks* that tests produce the expected outcome albeit executed out-of-order. If the outcome of the tests involved in the target data dependency does not change, PRADET removes it from the set of data dependencies

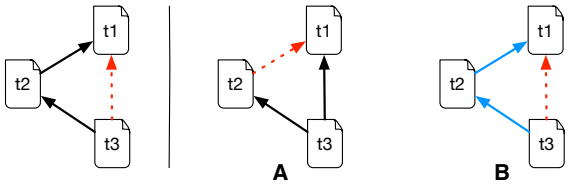

Figure 4. A dependency graph with an edge, t3→t1, that would lead to a cycle when inverted (left side), along with the possible outcomes of testing the other edges instead (right side).

to check; otherwise, the target dependency becomes a manifest dependency (highlighted in light-blue in the figure) and is not removed. This process is repeated until all the data dependencies are removed or become manifest.

PRADET represents tests as vertices and dependencies as directed edges in a test dependency graph. The test dependency graph contains an edge from a test t2 to another test t1 iff t2 is dependent on t1. This captures the fact that t2 might require the execution of t1 to produce the expected result. As a consequence, the test dependency graph is acyclic.

To check if a given data dependency leads to a manifest dependency (i.e., to a test outcome changing), PRADET selects the corresponding edge in the test dependency graph and inverts its direction. Then, it computes a schedule for test execution through topological sorting. This way, inverting the direction of that edge leads to schedule the tests in a way that precisely violates the target data dependency but not the other dependencies, which are maintained during the execution.

Selecting which edge in the graph to test first is crucial to effectively discover manifest dependencies. A naive approach to randomly select and invert dependency edges is not always possible or effective. For instance during the refinement, it is possible that inverting an edge introduces a cycle in the test dependency graph. Figure 4 illustrates this case over an example with three tests; in this example, inverting the edge t3→t1 results in a circular dependency between the tests. In the presence of circular dependencies, it is impossible to linearize the dependency graph, and PRADET cannot compute a test schedule that satisfies all the dependencies between the tests; hence, the data dependency is *untestable*. PRADET detects the presence of such cases and defers the analysis of the corresponding data dependencies. Temporarily skipping such untestable edges is not a problem because only one of the following two situations can occur: either the graph contains another data dependency that can be tested, i.e., it can break the cycle, or there is already a number of manifest dependencies large enough to let PRADET deduce that the untestable edges are indeed *un-interesting* to check. In this case, no schedule exists such that the test dependency can be checked reliably. The former situation is labelled as A in Figure 4. In this case, t2→t1 can be inverted, hence tested. In case inverting t2→t1 does not result in a manifest dependence, the corresponding edge is removed and the cycle does not appear anymore when the direction of t3→t1 is inverted. The latter situation is labelled as B in Figure 4. In this case, testing t2→t1 reveals a manifest dependency; at this

```
1  public class Test {
2    private static Foo foo = null;
3    private static boolean initialized = false;
4    private void initialize(){
5        if ( ! initialized ) initialized = true;
6    }
7    @Test
8    public void t1(){
9      initialize()
10   }
11   @Test
12   public void t2(){
13     initialize()
14     foo=new Foo();
15   }
16   @Test
17   public void t3(){
18     assertTrue( initialized );
19     assertNotNull( foo );
20   }
21 }
```

Figure 5. Example of a test, t3, with multiple data dependencies.

point, t3→t1 is still not testable because inverting it results in a cycle, but t3→t2 is testable. If t3→t2 corresponds to a manifest dependency, then testing t3→t1 is un-interesting because there is no valid run-order which can execute t3 before t1.

Tests that have *joint* data dependencies represent another interesting situation where the basic approach of randomly checking data dependencies might be not effective. In these cases, the act of verifying one data dependency might violate other dependencies' preconditions. This, in turns, might lead to misjudge data dependencies as manifest dependencies, or mask manifest dependencies as benign, hence miss them.

We illustrate this case on the tests reported in Figure 5, which is inspired by a real case that we found during our evaluation. The execution of those tests in the order {t1,t2,t3} results in a test dependency graph similar to the one reported in Figure 4. In this example, tests t2 and t3 depend on t1 because they access initialized, which was last written by t1; test t3 also depends on t2 because it accesses foo, which was last written by t2. At the stage of refinement pictured in Figure 4-A, t2→t1 is already broken, hence inverting t3→t1 is possible. PRADET inverts t3→t1 to check if the dependency between t3 and t1 on the initialized field is manifest which results in executing t3 without initializing that field. In this case, the assertion on line 21 in Figure 5 triggers, and PRADET concludes that the dependency between t3 and t1 is indeed a manifest one. Testing t3→t1 is done by executing {t3,t1} which does not include t2 . In fact, t2 is not directly involved in that dependencies, and, most importantly, executing t2 would mask the manifest dependency on initialized since t2 initializes that field. However, t2 is a precondition of t3, hence it must be executed before t3. And, this is an impossible situation.

PRADET handles such cases in a conservative way, according to a *source-first* selection strategy, which selects first test dependencies between tests that were executed later during the collection. Those tests correspond to source nodes in the dependency graph with larger index value. In the example of Figure 5, t3 is the source node and source-first would choose t3→t2 instead of t3→t1 as first dependency to

test, because `t2` was executed after `t1`. Compared to a basic random selection, source-first comes at the cost of running more tests during the refinement. However, it guarantees that either all the test preconditions are met or the dependency is skipped because it cannot be tested reliably.

Dependency refinement optimizations

Despite its conservative test selection strategy, PRADET drastically reduces the number of test executions required to expose manifest dependencies between n tests, from $n!$ (in a simple implementation with no awareness of data dependencies) to $n*d$, where d is the number of data dependencies. This makes PRADET usable in practice; however, for projects with large test suites and many data dependencies, it might be still prohibitive. Therefore, to further reduce the number of test executions, PRADET skips the execution of *irrelevant* tests. Tests that are irrelevant for a given data dependency fall in two categories: tests that do not belong to the weakly connected component that contains the data dependency [16], and tests that are scheduled *after* the verification of the data dependency. Weakly connected components contain all the nodes that are reachable from every other node in the symmetric closure of the graph. Hence, tests outside the weakly connected component of a data dependency are irrelevant because they cannot reach the data dependency by any means, hence influence its verification. Similarly, tests that are executed after the verification of a data dependency cannot have an influence on it, thus there is no point in running them for the purpose of verifying that data dependency.

IV. PRADET IMPLEMENTATION

We implemented PRADET as a stand-alone Java tool which integrates with JUnit [17], the testing automation framework. Unfortunately, due to intellectual property restrictions, we were not able to use the ElectricTest tool itself to collect the data dependencies [9]. Instead, we used an open-source version of that tool implemented independently from the original ElectricTest implementation [18]. The data dependency detector utilizes static and dynamic bytecode instrumentation to inject the analysis code. The Java core API classes (e.g., the rt.jar file) are instrumented statically before executing the tests, while the application code is instrumented dynamically at load time, using Java Agents.

V. LIMITATIONS

PRADET has the following conceptual limitations which might affect the quality of its results: it requires a test execution order which produces the reference outcome; and, it cannot observe all data dependencies if tests fail during the collection. Developers must specify a test execution order for collecting data dependencies. Ideally, such execution order satisfies all the test dependencies, and tests execution produces the *expected* outcome.[2] PRADET relies on dynamic data-flow analysis to uncover data dependencies; therefore, it can find only those data dependencies that are observed during the test execution. As consequence, if developers specify test execution orders which do not expose some data dependencies, PRADET might ignore dependencies during the refinement.

For the same reason, if during the execution of a test an exception is raised or an assertion fails, the regular flow of the execution is broken, and all writes and reads that come after that do not execute. Hence, PRADET cannot observe them and possibly misses relevant data dependencies. As we show in the next section, despite these limitations that make our approach incomplete, PRADET remains effective in practice.

[2]Note that expected outcome does not necessarily imply that all the tests pass.

VI. EVALUATION

Our evaluation has the main goals of *(i)* assessing the ability of PRADET to find manifest dependencies in large test suites in reasonable amount of time; and, *(ii)* understanding when test dependencies are introduced. To contextualize our finding we also compare PRADET against DTDETECTOR.

We conducted our evaluation on nineteen open source projects as follows: first, we ran PRADET to collect the data dependencies in each project's test suite; for the collection we used either the test execution order provided by developers or the default as computed by JUnit. Then, for each project, we ran PRADET to refine data dependencies and discover manifest dependencies. Similarly, given the test execution order we ran DTDETECTOR in its basic configurations: *reverse*, which executes the tests in the opposite order than the given one; *isolate*, which executes each test in a separate JVM; and *exhaustive*, which executes all the possible (pairwise) combinations of test executions.

A. Test Subjects

We organize our evaluation around two groups of test subjects. The first group contains four projects, namely Crystal, XML Security, synoptic, and jodatime, which Zhang and co-authors investigated in a previous study about test dependencies [3]. We choose these projects for two reasons: information about the amount of manifest dependencies in their test suites are available; and, we can directly compare the results of PRADET and DTDETECTOR. In the following, we refer to these four test subjects as the DTDETECTOR dataset, and report the results of our analysis on them in Table I.

The second group contains fifteen projects, namely photoplatform-sdf, DiskLruCache, indextank-engine, Bateman, dspot, webbit, stream-lib, http-request, okio, togglz, Bukkit, jackson-core, jsoup, dynjs, jfreechart, selected among the most active projects on github. We choose these projects for two reasons: they were investigated in related work [9], [6]; and, they have a wide range of tests (ranging from 31 tests in photoplatform-sdf to 2,234 in jfreechart) and data dependencies (ranging from 48 dependencies in stream-lib to almost 44,000 dependencies in dynjs). In the following, we refer to the application of PRADET to these test subjects as PRADET in the wild, and report the results of our analysis on them in Table II.

Table I
RESULTS OF THE EVALUATION OF PRADET AND DTDETECTOR ON THE "DTDETECTOR DATASET."

Project			Data Deps	Manifest Deps					Analysis Cost (Seconds)			
Name	Revision	Tests	(#)	PRADET	Reverse	Isolate	Exhaustive	Tot.	PRADET	Reverse	Isolate	Exhaustive
Crystal	`1a11279`	76	93	8	8	8	8	8	46	109	182	13,523
XML Security	v 1.0.4	108	118	4	0	4	4	4	146	7	570	17,201
Synoptic	`d5ea6fb`	118	204	2	1	0	2	4	106	9	121	13,832
jodatime	`b609d7d`	3,861	37,418	4	1	1	—	4	14,914	31	3,895	—
4 Total		4,163	37,833	18	10	13	14	20	15,212	156	4,768	44,106

We mark cases with (—) when DTDETECTOR did not finish in 2 days, and highlight the setups which found the highest amount of dependencies. We verified that all the reported dependencies can occur in normal test executions.

The test subjects from the DTDETECTOR dataset are well studied and the published results about their manifest dependencies can be considered to some extent as ground truth. Therefore, the application of PRADET on them can be interpreted as an *experimental* study. Results from this study support our claims that PRADET can effectively discover manifest dependencies, and can do so fast enough to be usable in practice. Differently than the test subjects in the DTDETECTOR dataset, the other subjects had never been studied before for manifest test order dependencies. Hence, the use of PRADET to analyze them can be interpreted as a novel *empirical* study on the subjects. Results from this study complement and strengthen the conclusion of previous studies, that is, tests are not always independent. Additionally, this study improves the generality of this claim because PRADET enables the analysis of more and larger test subjects.

B. Results

In this section, we report the results obtained by applying PRADET and DTDETECTOR on the DTDETECTOR dataset (see Table I), and on their execution in the wild (see Table II). We also report the results of our historical evaluation on the manifest dependencies that we detected (see Table III). For each project, we report name, version, and number of tests, or alternatively name, commit hash, and number of tests (col. *Project*). Regarding data dependency collection, we report the number of data dependencies discovered (col. *Data Deps*).

The number of unique manifest dependencies lets us draw conclusions about the relative capability of both tools to find new manifest dependencies, while the amount of manifest dependencies possible currently quantifies how many of the discovered dependencies can be actually in a *normal, regular test run* given each project's configuration. For manifest dependencies, we report the number of manifest dependencies discovered by PRADET and by DTDETECTOR in its different configurations (exhaustive, and its simple heuristics of isolating tests and running tests in the reverse order), the total number of unique manifest dependencies.

Most projects that we examined specified their unit tests with JUnit, and ran them using the Maven build system. JUnit organizes tests into methods and classes: a single test is a test method, and many test methods are grouped into a test class. Maven determines the order of test classes, and then within each test class, JUnit determines the order of each test method. By default, Maven will order test classes in the order returned by the filesystem [19], which may be largely alphabetical, but has some degree of non-determinism. By default, JUnit will then order test methods in the order returned by `Class.getDeclaredMethods()`, which is entirely non-deterministic. Hence, simply running each project's test suite normally (e.g. using `mvn test`), it is possible for tests to be shuffled within these constraints (re-ordering test methods within a test class or re-ordering test classes).

In practice, we consider manifest dependencies that can occur from any re-ordering, including those not permitted by these constraints, such as interleaving test methods from multiple test classes in a new ordering. As a consequence, some of the manifest dependencies identified by the tools, despite being valid, might not be possible to occur in a normal test execution, although they may be possible in the event of test selection or parallelization. Despite this theoretical concern, we verified that all the reported dependencies can actually occur in normal test executions, hence they are instances real dependencies.

Finally, regarding the cost of discovering manifest dependencies, we report the execution time (in seconds) to complete the analyses (col. *Analysis Cost*). In particular, for PRADET, this value includes both the time to collect and to refine data dependencies; instead for DTDETECTOR, this value includes the time to discover manifest dependencies and to minimize them using delta debugging [20].

Experimental Study: Regarding the ability to find manifest dependencies Table I shows that PRADET discovered a total of 18 manifest dependencies over the four projects, while DTDETECTOR discovered at most only 14 manifest dependencies using exhaustive search, its best and foremost expensive configuration. Thanks to its ability to track data-flow across multiple tests, PRADET exposed four new manifest dependencies in test subjects that were extensively tested in previous work.

Regarding the efficiency of the approach, the results show that the analysis cost of PRADET increases as the number of tests and data dependencies increase. Nevertheless, the runtime remains extremely small for projects with relatively few dependencies and tests. For projects featuring large test suites and many dependencies, PRADET's execution time is comparable with standard strategies for software quality assur-

Table II
RESULTS OF THE EVALUATION OF PRADET AND DTDETECTOR IN THE WILD.

Project			Data Deps	Manifest Deps					Analysis Cost (Seconds)			
Name	Revision	Tests	(#)	PRADET	Reverse	Isolate	Exhaustive	Total	PRADET	Reverse	Isolate	Exhaustive
photoplatform-sdf	3a7d9e7	31	332	13	0	0	0	13	3,533	2	32	933
DiskLruCache	3aa6286	61	75	0	0	0	0	0	47	2	62	3,666
indextank-engine	f2354fe	61	1,686	7	0	0	19*	22	4,210	15	73	4,309*
Bateman	08db4a6	76	77	0	0	0	0	0	102	3	77	5,708
dspot	fe82256	85	215	0	1	0	2	3	3,582	206	297	94,195
webbit	f628a7a	131	116	0	0	1	0	1	990	21	365	18,543
stream-lib	5868141	139	48	0	0	0	0	0	2,364	230	390	52,822
http-request	2d62a3e	163	1,129	0	27	0	28	28	2,267	5,160	445	72,536
okio	20e259c	234	307	0	0	0	0	0	4,523	252	488	108,981
togglz	20e259c	262	733	0	0	0	0	0	729	29	276	27,313
Bukkit	574f7a8	276	195	0	0	0	0	0	455	2	277	76,029
jackson-core	d04bea9	284	311	0	0	0	0	0	202	6	2,926	80,619
jsoup	f28c024	526	566	2	0	1	—	2	1,418	3	1,142	—
dynjs	4bc6715	865	43,969	0	0	0	—	0	67,385	15	875	—
jfreechart	v 1.0.15	2,234	1,508	1	0	0	—	1	10,256	164	2,241	—
15 Total		5,428	51,267	23	28	2	49	70	102,063	6,110	9,966	545,654

We mark cases with (—) when DTDETECTOR did not finish in 2 days, and highlight the setups which found the highest amount of dependencies. An asterisk (*) identifies cases when DTDETECTOR failed; for those, we report the results achieved by DTDETECTOR without dependency minimization, hence obtained much faster. We verified that all the reported dependencies can occur in normal test executions.

ance. For instance, the execution of integration and acceptance tests, common quality assurance activities, might take many hours to run [21]. In particular, PRADET took about 4 hours to collect and refine some 37,400 dependencies for 3,861 tests for jodatime. On the same project, DTDETECTOR did not finish its exhaustive search within two days, and in its other configurations, reverse and isolation, which are faster, it did not find as many manifest dependencies as PRADET. Interestingly, in all of the cases in which PRADET found the same amount of manifest dependencies as DTDETECTOR, PRADET was faster than DTDETECTOR. In those cases, no matter which DTDETECTOR configuration we used, PRADET was from 2.3 times to 130.5 times faster.

Empirical Study: Table II reports the results of our empirical evaluation. Regarding the efficiency of PRADET, we observe that it can analyze projects with large test suites in a reasonable amount of time. For eleven projects the analysis cost ranged from less than a minute to about one hour; for three projects the analysis took between one and three hours; and only for one project PRADET took more than three hours. In contrast, DTDETECTOR's pair-wise exhaustive combination took much longer. For one project, the analysis took less than one hour to complete; for three projects, it took less than three hours; and, for the remaining projects it took more than five hours, when it finished in the allotted time. In fact, for all the projects with more than 500 tests, DTDETECTOR did not finish within two days. Moreover, for indextank-engine, DTDETECTOR failed to run the test dependencies minimization; in this case, we report in Table II the partial, and very optimistic results that DTDETECTOR achieved without running delta debugging. In comparison to a tool that *only* detects data dependencies (like ElectricTest), PRADET found only true, manifest test order dependencies. In summary, these results support our previous observation that PRADET is efficient, and generalizes to larger test subjects.

We note, however, that for extremely large projects, i.e, dynjs and jfreechart, the usability of PRADET as a day-to-day tool becomes questionable. For example, in the case of dynjs, PRADET collected and analyzed 43,969 data dependencies in about 19 hours. While PRADET should clearly not be executed for every change to the program under test, it is certainly performant enough to execute in the background regularly.

We must note that we obtained the results reported in this study by using single machine and a serial implementation of our approach. In practice, however, different refinement runs can be parallelized; for example, data dependencies that belong to different weakly connected components (see Section III-B) can be analyzed in parallel. As the refinement progresses, more dependencies are broken; this further increases the level of parallelism of the analysis and might improve the overall efficiency of the refinement to a larger extent.

Regarding the effectiveness of the approach, the results in Table II support our previous observation of the ability of PRADET to find new manifest dependencies. In particular, PRADET found 23 manifest dependencies across four projects. In contrast, DTDETECTOR found between 2 and 49 manifest dependencies across six projects. The higher discovery rate of DTDETECTOR can be explained with the following considerations: First, DTDETECTOR's exhaustive combination strategy is much more expensive than PRADET. Second, the results for indextank-engine are optimistic since the reported manifest dependencies are not minimized. Finally, the manifest dependencies in http-request are all rooted in a single problematic test which affects the tests which follow it; in this case, PRADET's results are extremely dependent on the run-order as we discussed in Section V. For http-request in particular, we verified that by collecting data dependencies using the reverse

Table III
HISTORICAL EVALUATION OF DEPENDENCY INTRODUCTION

	Manifest Dependencies	
Project Name	**Detected**	**Existed at test creation**
photoplatform-sdf	13	13
dspot	3	2
joda-time	4	4
webbit	1	0
jsoup	2	2
http-request	28	28
jfreechart	1	1
7 Total	**52**	**50**

order of execution, PRADET identified at least 27 out 28 manifest dependencies. Considering the sheer analysis cost of DTDETECTOR's exhaustive combination, this result suggests that one might run PRADET two times, one following the normal run-order and one following its reverse, and still be more cost-effective than DTDETECTOR. In general, picking an optimal scheduling for tests is non-trivial, and we leave this to future work.

Interestingly, different experimental setups led to the discovery of manifest dependencies across almost disjoint sets of projects: in only two projects, indextank-engine and jsoup, both PRADET and DTDETECTOR discovered common manifest dependencies; and in only two additional projects, dspot and http-request, DTDETECTOR discovered common manifest dependencies using different configurations. This suggests that PRADET and DTDETECTOR have complementary strengths, and might be successfully combined together.

Historical Evaluation of Test Dependencies: In order to further understand how developers should use PRADET, we performed a historical evaluation on the manifest dependencies that we detected. Specifically, for each project that used the git version control system and the maven build system, we investigated whether the manifest dependencies existed *when a test involved in the dependency was written*. In other words, we wanted to see if it would be a reasonable suggestion to run PRADET only when new tests were written. Table III shows the results of this investigation: 96% of the total manifest dependencies that we found through any means existed since at least one of the tests involved in the dependency were created. We took this study one step further, and looked to see how often new tests were introduced. We examined all commits since each of those test dependencies were introduced and found that only 18% of them introduced new tests.

C. Discussion

The results presented in the previous section let us draw conclusions about the efficiency and effectiveness of PRADET, its practical applicability on large projects, as well as its usefulness for developers.

Efficiency of the approach: Our experiments show that PRADET reduces the cost to detect manifest test order dependencies compared to basic strategies, such as reverse execution and isolated execution. Compared to a pairwise exhaustive search over the possible test execution orders, the speed up which PRADET achieves is drastic. All in all, PRADET can analyze projects with large test suites (i.e., more than 500 tests) and many data dependencies (i.e., more than 1,500 dependencies) in a reasonable amount of time. Further, we found that most manifest dependencies were created when the tests were — they did not develop later in the projects history — suggesting that it is reasonable to run PRADET only when new tests are created. As a consequence, we can conclude that:

> PRADET *dramatically improves the efficiency of manifest dependency discovery, by speeding up the data dependency refinement up to 97%. Running* PRADET *only when new tests are added is a good heuristic for detecting dependencies before they can become problematic.*

Additional investigations show that a considerable amount of time in each refinement step is spent in setting up isolated executions environments, i.e., spawning the additional JVMs which host the test executions. As explained in Section III-B, this is necessary to prevent data from subtly flowing from tests executed in one refinement step to tests executed in subsequent refinement steps. In theory, this overhead could be reduced by either parallelizing and distributing the refinement, for example using tools like CUT [22], or by employing techniques which can reset the state of JVMs to the default one without restarting them, like VMVM [23].

Effectiveness of the approach: Our approach relies on dynamic analysis to collect data dependencies between tests in the context of a reference execution order. Similarly to DTDETECTOR, PRADET relies on the available oracles in the test suite, hence empirical evidence, to identify manifest dependencies; and, it assumes that test suites have no flaky tests or other nondeterministic behaviors. For these reasons, PRADET is not complete. Despite this, our evaluation shows that PRADET can successfully confirm the presence of both known and previously unknown manifest test order dependencies. This is the case for both test subjects that are well studied in the domain of investigation, and large projects that could not be analyzed with state-of-the-art approaches. As a consequence, we can conclude that:

> PRADET *can effectively identify manifest dependencies in both small and large test suites.*

VII. RELATED WORK

Test dependencies represent a subset of the greater problem of flaky tests. Recent studies conducted at Microsoft [24], at Google [25], on TravisCI [26], and at Pivotal labs [27] have shown that flaky tests exist in practice and lead to many broken builds. Luo et al. [4] investigated possible root-causes of flaky tests and defined a taxonomy of ten common root-causes. They found that test order dependency is one of the top three common causes of flakiness. In a more recent study, Palomba and Zaidman [5] correlated test smells to flaky tests and showed how smells can help locate flaky tests. For example, the authors identified a strong correlations between

test order dependency issues and the *indirect testing smell* [28]. Other recent work has considered other causes of flaky tests, such as reliance on non-deterministic APIs [29], or general nondeterminism [30]. Our work here focuses specifically on finding test order dependencies.

Our prior approach, ELECTRICTEST, uses dynamic analysis to detect data dependencies between tests [9]. PRADET and ElectricTest share the basic approach to detect data dependencies; however, ElectricTest is less precise than PRADET, hence it identifies more data dependencies (see discussion in Section III-A). Further, ElectricTest uses dependency information to soundly parallelize the execution of tests, hence it has a different goal than PRADET.

Manifest dependencies are caused by poorly isolated test executions. On this topic, Muşlu et al. [31] identified tests that fail when executed in complete isolation. This indicates that the tests require a specific state to be in place before their execution; however, Muşlu et al. do not extract concrete dependencies. Therefore, differently than PRADET, the approach proposed by Muşlu et al. does not provide relevant information to the developers to address the problematic tests, such as the missing data dependencies and the tests which might set them. In our prior work, VMVM [23], we instead proposed to isolate tests by resetting the static state of the application to its default before each test execution, preventing data flowing between tests via internal resources (see Section III-A) by design. This approach effectively masks the *effects* of poorly designed tests, i.e., the manifest dependencies. This, in turns, makes the job of developers which must identify and fix such poorly designed tests harder.

Test dependencies can also be caused by external resources, such as files and sockets that are shared between the tests. Identifying how tests interact with those external resources might give developers a better view of what causes tests to behave differently in different executions. Gyori et al. [13] proposed POLDET, which uses dynamically identified state polluting tests, i.e., tests whose execution results in persistent changes to the testing environment and that might influence the behavior of other tests. Although Gyori et al.'s work shares the same vision as PRADET, it focuses on finding the tests which *might* introduce manifest dependencies by leaking data, but does not consider the tests which are *actually* affected by that. As a consequence, PRADET and POLDET do not identify exactly the same problematic tests. PRADET identifies state polluting tests as causing data dependencies only if they pollute within the JVM limits (see Section III-A) and when the state pollution results in actual data flows. POLDET detects all state polluting tests.

Focusing on manifest dependency detection, Zhang et al. [3] developed DTDETECTOR, which detects manifest dependencies by exhaustively exploring every test execution order. However, even considering various heuristics (including a weakly dependence-aware technique, similar to PRADET), DTDETECTOR cannot scale to large test suites. As we found in our evaluation, focusing only on manifest dependencies that can be traced back to data dependencies allows PRADET is more efficient and effective than DTDetector.

VIII. CONCLUSIONS AND FUTURE WORK

To effectively address the problem of finding poorly designed test cases that result in problematic, manifest test order dependencies, we developed a novel approach called PRADET. PRADET follows a systematic, data-driven process to discover manifest dependencies which drastically reduces the amount of time needed to expose such problematic cases. Compared to state-of-art solutions, PRADET is significantly faster and more precise.

Our evaluation on nineteen open source projects of different sizes shows that PRADET is effective and efficient enough to be used in practice: PRADET confirmed the presence of known manifest dependencies and identified new manifest dependencies in large projects that existing tools cannot analyze. Notably, PRADET also exposed new manifest dependencies in test subjects that were extensively tested in previous studies.

While developing PRADET, we identified several interesting opportunities for further improving the efficiency and effectiveness of our approach. Our future work will focus on the following subjects:

Parallelizing data dependency refinement.
During the refinement PRADET tests one data dependency at a time. This limits the overall efficiency of the approach and can be improved by parallelizing the testing of the different weakly connected components.

Define better heuristics for dependency selection.
PRADET selects data dependencies to test randomly. This guarantees an unbiased selection, but it might miss opportunities to further improve the efficiency of the approach. For example, by preferring data dependencies that, if removed, break large weakly connected into smaller ones, we improve the efficiency of the refinement.

Force application states to break dependencies.
Instead of repeating the execution of tests while checking a data dependency, one might reduce the number of tests to execute by *carving* the relevant application state after each test execution during the collection [32], and then force that state in memory on-demand during refinement [33].

Using static analysis.
PRADET uncovers data dependencies via dynamic analysis. By employing static analysis, we can compute a set of potential data dependencies to refine, which increases the chances to trigger manifest dependencies.

To facilitate replication of this study and extension of PRADET, the code implementing the approach and the data used for the evaluation are available at: http://www.jonbell.net/software/pradet/.

IX. ACKNOWLEDGMENT

The authors would like to thank Sebastian Kappler for providing feedback about the work, and Alex Gyori for assistance implementing the open-source version of ElectricTest. This work was funded in part by the European Research Council Advanced Grant "SPECMATE".

References

[1] J. Micco, "The state of continuous integration testing at google," 2017, international Conference on Software Testing, Verification, and Validation (ICST'17).

[2] ——. (2016) Flaky tests at Google and how we mitigate them. [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[3] S. Zhang, D. Jalali, J. Wuttke, K. .Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA 14, 2014, pp. 385–396.

[4] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the International Symposium on Foundations of Software Engineering*, ser. FSE '14, 2014, pp. 643–653.

[5] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *Proceedings of the International Conference on Software Maintenance*, ser. ICSME '17, 2017.

[6] W. Lam, S. Zhang, and M. D. Ernst, "When tests collide: Evaluating and coping with the impact of test dependence," University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-15-03-01, Mar. 2015.

[7] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '15, 2015, pp. 211–222.

[8] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, Oct 2001.

[9] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in *Proceedings of the Join Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '15, 2015.

[10] G. Denaro, A. Margara, M. Pezzè, and M. Vivanti, "Dynamic data flow testing of object oriented systems," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 947–958.

[11] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Softw. Test. Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[12] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Trans. Software Eng.*, vol. 22, no. 8, pp. 529–551, 1996.

[13] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '15, 2015, pp. 223–233.

[14] J. Bell and G. Kaiser, "Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs," in *ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14, 2014, pp. 83–101.

[15] J. S. Bell, "Making software more reliable by uncovering hidden dependencies," Ph.D. dissertation, Columbia University, 2016.

[16] S. Kappler, "Finding and breaking test dependencies to speed up test execution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE-SRC '16, 2016, pp. 1136–1138.

[17] JUnit, "Junit," http://junit.org/, Mar 2016.

[18] J. S. Bell, "Data dependency detector for java," https://github.com/gmu-swe/datadep-detector, Dec 2016.

[19] "Maven Surefire plugin," http://maven.apache.org/surefire/index.html.

[20] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Proceedings of the European Software Engineering Conference Held Jointly with the International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE '99, 1999, pp. 253–267.

[21] M. Fowler, "Continuous integration," https://www.martinfowler.com/articles/continuousIntegration.html, Feb 2017.

[22] A. Gambi, S. Kappler, J. Lampel, and A. Zeller, "CUT: automatic unit testing in the cloud," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '17, 2017, pp. 364–367.

[23] J. Bell and G. Kaiser, "Unit test virtualization with vmvm," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '14, 2014, pp. 550–561.

[24] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice Track*, ser. ICSE-SEIP '15, 2015.

[25] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, ser. ICSE-SEIP '17, 2017, pp. 233–242.

[26] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the Cost of Regression Testing in Practice: A Study of Java Projects Using Continuous Integration," in *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '17, 2017, pp. 821–830.

[27] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '17, 2017, pp. 197–207.

[28] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok, "Refactoring test code," CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 2001.

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

[30] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *Proceedings of the 2018 International Conference on Software Engineering*, ser. ICSE 2018, 2018.

[31] K. Muşlu, B. Soran, and J. Wuttke, "Finding bugs by isolating unit tests," in *Proceedings of the joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011, pp. 496–499.

[32] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proceedings of the International Symposium on Foundations of Software Engineering*, ser. FSE '14, 2006, pp. 253–264.

[33] M. Gligoric, D. Marinov, and S. Kamin, "Codese: Fast deserialization via code generation," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011, pp. 298–308.