Empirically Revisiting the Test Independence Assumption

Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam,
Michael D. Ernst, David Notkin
Department of Computer Science & Engineering, University of Washington, Seattle, USA
{szhang, darioush, wuttke, kivanc, winglam, mernst}@cs.washington.edu

ABSTRACT

In a test suite, all the test cases should be independent: no test should affect any other test's result, and running the tests in any order should produce the same test results. Techniques such as test prioritization generally assume that the tests in a suite are independent. Test dependence is a little-studied phenomenon. This paper presents five results related to test dependence.

First, we characterize the test dependence that arises in practice. We studied 96 real-world dependent tests from 5 issue tracking systems. Our study shows that test dependence can be hard for programmers to identify. It also shows that test dependence can cause non-trivial consequences, such as masking program faults and leading to spurious bug reports.

Second, we formally define test dependence in terms of test suites as ordered sequences of tests along with explicit environments in which these tests are executed. We formulate the problem of detecting dependent tests and prove that a useful special case is NP-complete.

Third, guided by the study of real-world dependent tests, we propose and compare four algorithms to detect dependent tests in a test suite.

Fourth, we applied our dependent test detection algorithms to 4 real-world programs and found dependent tests in each human-written and automatically-generated test suite.

Fifth, we empirically assessed the impact of dependent tests on five test prioritization techniques. Dependent tests affect the output of all five techniques; that is, the reordered suite fails even though the original suite did not.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging.
General Terms: Reliability, Experimentation.
Keywords: Test dependence, detection algorithms, empirical studies.

1. INTRODUCTION

Consider a test suite containing two tests `A` and `B`, where running `A` and then `B` leads to `A` passing, while running `B` and then `A` leads to `A` failing. We call `A` an *order-dependent* test (in the context of this test suite), since its result depends on whether it runs after `B` or not.

In a test suite, all the test cases should be independent: no test should affect any other test's result, and running the tests in any order should produce the same test results.[1] The assumption of test independence is important so that testing techniques behave as designed. Consider a test prioritization or selection algorithm. By design, if its input is a passing test suite, then its output should be a passing test suite. If the suite contains an order-dependent test, then prioritization or selection can introduce test failures, which violates the design requirement.

Many techniques assume test independence, including test prioritization [20, 31, 50, 54], test selection [7, 24, 27, 41, 42, 67], test execution [38, 39], test factoring [51, 63], test carving [19], and experimental debugging techniques [55, 65, 68]. However, this critical assumption is rarely questioned, investigated, or even mentioned: none of the above papers explicitly mentions the assumption as a limitation or a threat to validity. Between 2000 and 2013, 31 papers on test prioritization were published in the research track of ICSE, FSE, ISSTA, ASE, or ICST or in TOSEM or TSE [70]. Of these, 27 papers explicitly or implicitly assumed test independence, 3 papers acknowledged that the potential dependences between tests may affect the execution result of a prioritized test suite [37, 44, 47], and only 1 paper considered test dependence in the design of test prioritization algorithms [23].

Anecdotally, a number of researchers have told us that they believe test dependence is not a significant concern in practice. We investigated the validity of this unverified conventional wisdom, in order to understand whether test dependence arises in practice, the repercussions of dependent tests, and how to detect dependent tests.

1.1 Manifest Test Dependence

This paper focuses on test dependence that manifests as a difference in test result (i.e., passing or failing) as determined by the testing oracle. We adopt the results of the default order of execution of a test suite as the expected results; these are the results that a developer sees when running the suite in the standard way. A test is dependent when there exists a possibly reordered subsequence of the original test suite, in which the test's pass/fail result (determined by its existing testing oracles) differs from its expected result in the original test suite. That is, manifest test dependence requires a concrete order of the test suite that produces different results than expected.

[1] Test dependence is undesirable in the context of regression testing. On the other hand, for tasks like generating test cases or bug-finding, code fragments with dependences between them can be useful. By combining such fragments, new behaviors of a system can be exercised. Such behaviors can be encapsulated into independent tests to be included in a regression test suite.

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2610404

This paper uses *dependent test* as a shorthand for *manifest order-dependent test* unless otherwise noted. A single test may consist of setup and teardown code, multiple statements, and multiple assertions distributed through the test.

1.2 Causes and Repercussions

Test dependence results from interactions with other tests, as reflected in the execution environment. Tests may make *implicit* assumptions about their execution environment — values of global variables, contents of files, etc. A dependent test manifests when another test alters the execution environment in a way that invalidates those assumptions.

Why does this happen? Each test ought to initialize (or mock) the execution environment and/or any resources it will use. Likewise, after test execution, it should reset the execution environment and external resources to avoid affecting other tests' execution. However, developers sometimes make mistakes when writing tests. Even though frameworks such as JUnit provide ways to set up the environment for a test execution and clean up the environment afterward, they cannot ensure that it is done properly. This means that tests, like other code, sometimes have unintended and unexpected behaviors.

Here are three consequences of the fact that a dependent test gives different results depending on when it is executed during testing.

(1) Dependent tests can *mask faults in a program*. Specifically, executing a test suite in the default order does not expose the fault, whereas executing the same test suite in a different order does. One bug [11] in the Apache CLI library [10] was masked by two dependent tests for 3 years (Section 2.2.2).

(2) Test dependences can lead to *spurious bug reports*. When a dependent test fails, it usually represents a weakness in the test suite (such as failure to perform proper initialization) rather than a bug in the program. When a test should pass but it fails after reordering due to the dependence, people who are not aware of the dependence can get confused and might report bugs. As an example, the Eclipse developers investigated a bug report [18] in SWT for more than a month before realizing that the bug report was invalid and was caused by test dependences (i.e., a test should pass, but it failed when a user ran tests in a different order).

(3) Dependent tests can *interfere with downstream testing techniques* that change a test suite and thereby change a test's execution environment. Examples of such techniques include test selection techniques (which identify a subset of the input test suite to run during regression testing) [7, 24, 27, 41, 42, 67], test prioritization techniques (which reorder the input to discover defects sooner) [20, 31, 37, 50, 54], test execution techniques [38], test factoring [51, 63] and test carving [19] (which convert large system tests into smaller unit tests), experimental debugging techniques (such as Delta Debugging [55,65,68] and mutation analysis [52,66,67], which run a set of tests repeatedly), etc. Most of these techniques implicitly assume that there are no test dependences in the input test suite. Violation of this assumption, as we show happens in practice, can cause unexpected output. As an example, test prioritization may produce a reordered sequence of tests that do not return the same results as they do when executed in the default order. Section 6.3.3 provides empirical evidence to show that dependent tests do affect the output of five test prioritization techniques.

1.3 Contributions

This paper addresses and questions conventional wisdom about the test independence assumption. This paper makes the following contributions:

- **Study.** We describe a study of 96 real-world dependent tests from 5 software issue tracking systems to characterize dependent tests that arise in practice. Test dependence can have potentially non-trivial repercussions and can be hard to identify (Section 2).
- **Formalization.** We formalize test dependence in terms of test suites as ordered sequences of tests and explicit execution environments for test suites. The formalization enables reasoning about test dependence as well as a proof that finding manifest dependent tests is an NP-complete problem (Section 3).
- **Algorithms.** We present four algorithms to detect dependent tests: reversal, randomized, exhaustive bounded, and dependence-aware exhaustive bounded. All four algorithms are *sound* but *incomplete*: every dependent test they identify is real, but the algorithms do not guarantee to find all dependent tests (Section 4).
- **Evaluation.** We implemented our algorithms in a tool called DTDetector (Section 5)[2]. DTDetector detected 27 previously-unknown dependent tests in human-written unit tests in 4 real-world subject programs. The developers confirmed all of these as undesired (Section 6).
- **Impact Assessment.** We implemented five test prioritization techniques and evaluated them on 4 subject programs that contain dependent tests. The results show that all five test prioritization techniques are affected by dependent tests, that is, the prioritized test suite fails even though the original suite did not (Section 6).

Implications.

Our findings are of utility to practitioners and researchers. Both can learn that test dependence is a real problem that should not be ignored any longer, because it leads to false positive and false negative test results. Practitioners can adjust their practice based on what code patterns most often lead to test dependence, and they can use our tool to find dependent tests. Researchers are posed important but challenging new problems, such as how to adapt testing methodologies to account for dependent tests and how to detect and correct all dependent tests.

2. REAL-WORLD DEPENDENT TESTS

Little is known about the characteristics of dependent tests. This section qualitatively studies concrete examples of test dependence found in well-known open source software.

2.1 Sources and Study Methodology

We examined five software issue tracking systems: Apache [1], Eclipse [18], JBoss [30], Hibernate [25], and Codehaus [12]. Each issue tracking system serves tens of projects.

For each issue tracking system, we searched for four phrases ("dependent test", "test dependence", "test execution order", "different test outcome") and manually examined the matched results. For each match, we read the description of the issue report, the discussions between reporters and developers,

[2]DTDetector "exceeded expectations" of the ISSTA 2014 artifact evaluation committee. It is publicly available at `https://testisolation.googlecode.com/`.

Table 1: Real-world dependent tests. Column "Severity" is the developers' assessment of the importance of the test dependence. Column "# Involved Tests for Manifestation" is the number of tests needed to manifest the dependence. Column "Self" shows the number of tests that depend on themselves. Column "Days" is the average days taken by developers to resolve a dependent test. Column "Patch Location" shows how developers resolved the dependent tests: by modifying program code, by modifying test code, by adding code comments, or not fixed.

Issue Tracking System	Dependent Tests				# Involved Tests for Manifestation					Resolution					Root Cause			
	Total	Severity									Patch Location				Static	File	Data-	Un-
	number	Major	Minor	Trivial	Self	1 test	2 tests	3 tests	Unknown	Days	Code	Test	Doc	Unfixed	variable	system	base	known
Apache	26	22	3	1	0	5	18	1	2	93	5	20	0	1	9	3	8	6
Eclipse	59	0	59	0	0	0	49	1	9	48	1	8	49	1	49	0	0	10
JBoss	6	6	0	0	0	0	3	0	3	44	0	2	0	4	1	0	0	5
Hibernate	3	1	1	1	0	0	3	0	0	6	0	1	0	2	0	0	2	1
Codehaus	2	2	0	0	1	1	0	0	0	3	0	1	0	1	0	1	0	1
Total	96	31	63	2	1	6	73	2	14	194	6	32	49	9	59	4	10	23

and the fixing patches (if available). This information helped us understand whether the report is about test dependence. Each dependent test candidate was examined by at least two people and the whole process consisted of several rounds of (re-)study and cross checking. We ignored reports that are described vaguely, and we excluded tests whose results are affected by non-determinism (e.g., multi-threading). In total, we examined the first 450 matched reports, of which 53 reports are about test dependence (some reports contain multiple dependent tests). All collected dependent tests are publicly available at: `http://homes.cs.washington.edu/~szhang/dependent_tests.html`

2.2 Findings

Table 1 summarizes the dependent tests.

2.2.1 Characteristics

We summarize three characteristics of dependent tests: manifestation, root cause, and developer actions.

Manifestation: at least 82% of the dependent tests in the study can be manifested by 2 or fewer tests. A dependent test is manifested if there exists a possibly reordered subsequence of the original test suite, such that the test produces a different result than when run in the original suite. We measure the size of the reported subsequence in the issue report. If the test produces a different result when run in isolation, the number of tests to manifest the dependent test is 1. If the test produces a different result when run after one other test (often, the subsequence is running these two tests in the opposite order as the full original test suite), then the number of tests to manifest the dependent test is 2. Among the 96 studied dependent tests, we found only 2 of them require 3 tests to manifest the dependence. One other test depends on itself: running the test twice produces different results than running it once, because this test side-effects a database it reads. We count this special case separately in the "Self" column of Table 1.

For the remaining 14 dependent tests, the number of involved tests is unknown, since the relevant information is missing or vaguely described in the issue tracking systems. For example, some reports simply stated that "running *all* tests in one class before test t makes t fail" or "randomizing the test execution order makes test t fail".

Root cause: at least 61% of the dependent tests in the study arise because of side-effecting access to shared static variables. Among 96 dependent tests: 59 (61%) of them arise due to access to shared static variables, 10 (10%) of them arise due to access to a database, and 4 (4%) of them arise due to access to the file system. The root cause for the remaining 23 (25%) tests is not apparent in the issue tracking system.

Table 2: Repercussions of the 96 dependent tests.

Issue tracking system	False alarm	Missed alarm
Apache	24	2
Eclipse	59	0
JBoss	6	0
Hibernate	3	0
Codehaus	2	0
Total	94	2

Developer actions: dependent tests often indicate flaws in the test code, and developers usually modify the test code to remove them. Among 96 dependent tests, developers considered 94 (98%) to be major or minor problems, and the developers' discussions showed that the developers thought that the test dependence should be removed. Nonetheless, developers fixed only 38 (40%) of the 96 dependent tests. Another 49 (51%) were "fixed" by adding comments to the test code to document the existing dependence. For the remaining 9 (9%) unfixed tests, developers thought they were not important enough given the limited development time, so they simply closed the issue report without taking any action.

A dependent test usually reveals a flaw in the test code rather than the program code: only 16% of the code fixes (6 out of 38) are on the program code. In all 6 cases, the developers changed code that performs static variable initialization, which ensures that the tests will not read an uninitialized value. Section 2.2.2 gives an example. The other 32 code fixes were in the test code: 28 (87%) of the dependent tests were fixed by manually specifying the test execution order in a test script or a configuration file, 3 (10%) of them were simply deleted by developers from the test suite, and the remaining 1 (3%) test was merged with its initializing test.

2.2.2 Repercussions of Dependent Tests

A dependent test may manifest as a false alarm or a missed alarm (Table 2).

False alarm. Most of the dependent tests (94 out of 96) result in false alarms: the test should pass but fails after reordering due to the dependence. The test dependence arises due to incorrect initialization of program state by one or more tests. Typically, one test initializes a global variable or the execution environment, and another test does not

```
public final class OptionBuilder {
  private static String argName = null;
  private static void reset() {
    ...
    argName = "arg";
    ...
  }
}
```

Figure 1: Simplified fault-related code in CLI [10] (revision 661513). The fault was masked by two dependent tests for over 3 years.

perform any initialization, but relies on the program state after the first test's execution. Such dependence in the test code is often masked because the initializing test always executes before other tests in the default execution order. The dependent tests are not revealed until the initializing test is reordered to execute after other tests.

Sometimes developers introduce dependent tests intentionally because it is more efficient or convenient [35, 61]. Even though the developers are aware of these dependences when they create tests, this knowledge can get lost. Other people who are not aware of these dependences can get confused when they run a subset of the test suite that manifests the dependence, so they might report bugs about the failing tests even though this is exactly the intended behavior. If the dependence is not documented clearly and correctly, it can take a considerable amount of time to work out that these reported failures are spurious. The Eclipse project contains at least 49 such dependent tests. In September 2003, a user filed a bug report in SWT [56] [18], stating that 49 tests were failing unexpectedly if she ran any other test before `TestDisplay` — a test suite that creates a new `Display` object and tests it. However, this bug report was spurious and was caused by undocumented test dependence. All 49 failing tests are dependent tests with the same root cause: in SWT, only one global `Display` object is allowed; the user ran tests that create but do not dispose of a `Display` object, while the tests in `TestDisplay` attempt to create a new `Display` object, which fails, as one is already created. This is the desired behavior of SWT, and points to a weakness in the test suite.

Missed alarm. In rare cases, dependent tests can hide a fault in the program, exactly when the test suite is executed in its default order. Masking occurs when a test case t *should* reveal a fault, but tests executed before t in a test suite always generate environments in which t passes accidentally and does not reveal the fault. Tests in this category result in *missed alarms* — a test should fail but passes due to the dependence.

We found two such dependent tests in the Apache CLI library [10,40]. Figure 1 shows the simplified fault-related code. The fault is due to side-effecting initialization of the static variable `argName`. CLI should set value `"arg"` to the static variable `argName` before its clients instantiate an `OptionBuilder` object. However, the CLI implementation in Figure 1 only sets the value after the clients call method `reset()`. In CLI, two test cases `BugsTest.test13666` and `BugsTest.test27635` can reveal this fault by directly instantiating a `OptionBuilder` object without calling `reset()`. These two tests fail when run in isolation, but both pass when run in the default order. This is because in the default order, tests running *before* these two tests call `reset()` at least once, which sets the value of `argName` and masks the fault.

This fault was reported in the bug database several times [11], starting on March 13, 2004 (CLI-26). The report was marked as resolved *three years* later on March 15, 2007 when developers realized the test dependence. The developers fixed this fault by adding a static initialization block which calls `reset()` in class `OptionBuilder`.

2.2.3 Implications for Dependent Test Detection

We summarize the main implications of our findings.

Dependent tests exist in practice, but they are not easy to identify. None of the dependent tests we studied can be identified by running the existing test suite in the default order. Every dependent test was reported when the test suite was reordered, either accidentally by a user or by a testing tool. This indicates the need for a tool to detect dependent tests.

Dependent test detection techniques can bound the search space to a small number of tests. In theory, a technique needs to exhaustively execute all $n!$ permutations of a n-sized test suite to detect all dependent tests. This is not feasible for realistic n. Our study shows that most dependent tests can be manifested by executing no more than 2 tests together (Section 2.2.1). Thus, a practical technique can focus on running only short subsequences (whose length is bounded by a parameter k) of a test suite. This reduces the permutation number to $O(n^k)$, which is tractable for small k and n.

Dependent test detection techniques should focus on analyzing accesses to global variables. Dependent tests can result from many interactions with the execution environment, including global variables, databases, the file system, network, etc. However, in our study, most of the real-world dependent tests are caused by side-effecting static variable accesses. This implies that a dependent test detection technique may find most dependent tests by focusing on global variables.

2.3 Threats to Validity

Our findings apply in the context of our study and methodology and may not apply to arbitrary programs. The applications we studied are all written in Java and have JUnit test suites.

We accepted the developers' judgment regarding which tests are dependent, the severity of each dependent test, and how many tests are needed to manifest the dependence. We are unlikely to have found all the dependent tests in those projects. We did not intentionally ignore any test dependence in the issue tracking system. However, a limitation is that the developers might have made a mistake or might not have marked a test dependence in a way we found it (different search terms might discover additional dependent tests).

3. FORMALIZING TEST DEPENDENCE

The result of a test depends not only on its input data but also on its *execution conditions*. To characterize the relevant execution conditions, our formalism represents (a) the order in which test cases are executed and (b) the environment in which a test suite is executed.

3.1 Definitions

We express test dependences through the results of executing *ordered* sequences of tests in a given *environment*.

DEFINITION 1 (TEST). *A test is a sequence of executable program statements and an oracle — a Boolean predicate that decides whether the test passes or fails.*

DEFINITION 2 (TEST SUITE). *A test suite T is an n-tuple (i.e., ordered sequence) of tests $\langle t_1, t_2, \ldots, t_n \rangle$.*

DEFINITION 3 (ENVIRONMENT). *An* environment $\mathbf{E}$ *for the execution of a test consists of all values of global variables, files, operating system services, etc. that can be accessed by the test and program code exercised by the test case.*

We use $\mathbf{E}_0$ to represent the initial environment, such as a fresh JVM initialized by frameworks like JUnit before executing any test.

DEFINITION 4 (TEST EXECUTION). *Let $\mathcal{T}$ be the set of all possible tests and $\mathcal{E}$ the set of all possible environments. The function $exec : \mathcal{T} \times \mathcal{E} \rightarrow \mathcal{E}$ represents test execution. $exec$ maps a test $t \in \mathcal{T}$ and an environment $\mathbf{E} \in \mathcal{E}$ to a (potentially updated) environment $\mathbf{E}' \in \mathcal{E}$.*

Given a test suite $T = \langle t_1, t_2, \ldots, t_n \rangle$, we use the shorthand $exec(T, \mathbf{E})$ for $exec(t_n, exec(t_{n-1}, \ldots exec(t_1, \mathbf{E}) \ldots))$, to represent its execution.

Tests call into a program, but our definitions leave the program implicit, since it is always clear from context.

DEFINITION 5 (TEST RESULT). *The result of a test t executed in an environment $\mathbf{E}$, denoted $R(t|\mathbf{E})$, is defined by the test's oracle and is either PASS or FAIL.*

The result of a test suite $T = \langle t_1, \ldots, t_n \rangle$, executed in an environment $\mathbf{E}$, denoted $R(\langle t_1, \ldots, t_n \rangle|\mathbf{E})$, is a sequence of results $\langle o_1, \ldots, o_n \rangle$ with $o_i \in \{PASS, FAIL\}$. We use $R(T|\mathbf{E})[t]$ to denote the result of a specific test $t \in T$.

For example, $R(\langle t_1, t_2 \rangle|\mathbf{E}_1) = \langle FAIL, PASS \rangle$ represents that if t_1 then t_2 are run, starting with the environment $\mathbf{E}_1$, then t_1 fails and t_2 passes.

A manifest order-dependent test (for short, dependent test) is one that can be exposed by reordering existing test cases. A dependent test t manifests only if there are two test suites S_1 and S_2 which are two permutations of the original test suite T, in which t exhibits a different result in the execution $exec(S_1, \mathbf{E}_0)$ than in the execution $exec(S_2, \mathbf{E}_0)$.

DEFINITION 6 (MANIFEST ORDER-DEPENDENT TEST). *Given a test suite T, a test $t \in T$ is a manifest order-dependent test in T if $\exists$ two test suites $S_1, S_2 \in permutations(T)$: $R(S_1|\mathbf{E}_0)[t] \neq R(S_2|\mathbf{E}_0)[t]$.*

It would be possible to consider a test dependent if reordering could affect any internal computation or heap value (non-manifest dependence); but these internal details, such as order of elements in a hash table, might never affect any test result: they could be false dependences. Another alternative would be to ask whether it is possible to write a new dependent test for an existing test suite; but the answer to this question is trivially "yes". This paper focuses on manifest dependence and works with real, existing test suites to determine the practical impact and prevalence of dependent tests.

3.2 The Dependent Test Detection Problem

We prove that the problem of detecting dependent tests is NP-complete.

DEFINITION 7 (DEPENDENT TEST DETECTION PROBLEM). *Given a set suite $T = \langle t_1, \ldots, t_n \rangle$ and an initial environment $\mathbf{E}_0$, is $t \in T$ a dependent test in T?*

We prove that this problem is NP-hard by reducing the NP-complete Exact Cover problem to the Dependent Test Detection problem [36]. Then we provide a linear-time algorithm to verify any answer to the question. Together these two parts prove that the Dependent Test Detection Problem is NP-complete.

THEOREM 1. *The problem of determining whether a test is a dependent test for a test suite is NP-complete.*

PROOF. Due to space limits, we omit the proof. Interested readers can refer to [69] for details. □

3.3 Discussion

For the sake of simplicity, our definition does not consider non-deterministic tests, non-terminating tests, and tests aborting the JVM. Our formalism only considers deterministic tests, and excludes tests whose results might be affected by non-determinism such as thread scheduling and timing issues. Our formalism excludes self-dependence, when executing the same test twice may lead to different results. Our empirical study in Section 2.2.1 indicates that self-dependent tests are rare in practice. In addition, typical downstream testing techniques such as test selection and prioritization do not usually execute a test twice within the same JVM.

4. DETECTING DEPENDENT TESTS

Since the general form of the dependent test detection problem is NP-complete, we do not expect to find an efficient algorithm that fully solves it.

To approximate the exact solution, this section presents four algorithms that find a *subset* of all dependent tests. Section 4.1 describes a heuristic algorithm that executes all the tests of a suite in the reverse order. Section 4.2 describes a randomized algorithm that repeatedly executes all the tests of a suite in random order. Section 4.3 describes an exhaustive bounded algorithm that executes all possible sequences of k tests for a bounding parameter k (specified by the user). Section 4.4 describes a dependence-aware k-bounded algorithm. The dependence-aware algorithm dynamically collects the static fields that each test reads or writes, and uses the collected information to reduce the search space. All four algorithms are *sound* but *incomplete*: every dependent test they find is real, but they do not guarantee to find every dependent test (unless the bound is n, the size of the test suite).

4.1 Reversal Algorithm

Figure 2 shows the base algorithm. Given a test suite $T = \langle t_1, t_2, \ldots, t_n \rangle$, the base algorithm first executes T with its default order to obtain the *expected result* of each test (line 2). It chooses some set of test suites (line 3) and then executes each test suite to observe its results (line 4). The algorithm checks whether the result of any test differs from the expected result (lines 5–9).

Figure 3 instantiates the base algorithm (Figure 2) by reversing the original test execution order.

4.2 Randomized Algorithm

Figure 4 instantiates the base algorithm (Figure 2) by randomizing the original test execution order (line 2).

Input: a test suite T
Output: a set of dependent tests *dependentTests*
1: *dependentTests* $\leftarrow \emptyset$
2: *expectedResults* $\leftarrow R(T|\mathbf{E}_0)$
3: **for** each *ts* in getPossibleExecOrder(T) **do**
4: *execResults* $\leftarrow R(ts|\mathbf{E}_0)$
5: **for** each test t in *ts* **do**
6: **if** *execResults*[t] $\neq$ *expectedResults*[t] **then**
7: *dependentTests* $\leftarrow$ *dependentTests* $\cup$ t
8: **end if**
9: **end for**
10: **end for**
11: **return** *dependentTests*

Figure 2: The base algorithm to detect dependent tests. The getPossibleExecOrder function is instantiated by different algorithms in Figures 3, 4, 5, and 6.

getPossibleExecOrder(T):
1: **yield** reverse(T)

Figure 3: The reversal algorithm to detect dependent tests. It instantiates the algorithm of Figure 2, defining the getPossibleExecOrder function.

4.3 Exhaustive Bounded Algorithm

This algorithm uses the findings of our study (Section 2) that most dependent tests can be found by running only short subsequences of test suites. For example, in our study, at least 82% of the real-world dependent tests can be found by running no more than 2 distinct tests together. Instead of executing all permutations of the whole test suite, our algorithm (Figure 5) executes all k-permutations for a bounding parameter k. By doing so, the algorithm reduces the number of permutations to execute to $O(n^k)$, which is tractable for small k and n.

Figure 5 shows the algorithm.

An advantage of this algorithm is that it produces the shortest possible test suite as a witness that a specific test is dependent. By contrast, the reversal and randomized algorithms produce a large test suite, which the user must inspect and/or minimize in order to understand why a specific test is dependent.

4.4 Dependence-Aware Bounded Algorithm

The key idea of the dependence-aware k-bounded algorithm is to avoid permutations that cannot reveal a dependent test. Suppose that all tests in suite S_1 pass. The algorithm determines, for every field (or other external resource such as a file) read by a test in S_1, which test previously wrote that field. If suite S_2 has the same relationships, then suite S_2 also passes (and, therefore, it need not be run). Thus, the dependence-aware k-bounded algorithm detects the same number of dependent tests as the exhaustive k-bounded algorithm does (when using the same k), but it prunes the search space.

As a special case, suppose that for each test, every global field (and other resources from the execution environment) it reads is *not* written by any test executed before it; then each test in the permutation produces the same result as when executed in isolation. Dependent tests whose isolation execution results are different from the results in the default execution order can be cheaply detected. Thus, the permutation can be safely ignored.

We give two cases for the algorithm: an optimized version for k=1 using the original suite as S_1, and a general version for $k\geq 2$ using isolated execution (one test at a time) as S_1.

getPossibleExecOrder(T):
1: **for** i in 1..*numtrials* **do**
2: **yield** shuffle(T)
3: **end for**

Figure 4: The randomized algorithm to detect dependent tests. It instantiates the algorithm of Figure 2, defining the getPossibleExecOrder function. Our experiments use $numtrials = 10, 100, 1000$**.**

Auxiliary methods:
kPermutations(T, k): returns all k-permutations of T; that is, all sequences of k distinct elements selected from T

getPossibleExecOrder(T):
1: **return** kPermutations(T, k)

Figure 5: The exhaustive k-bounded algorithm to detect dependent tests. It instantiates the algorithm of Figure 2, defining the getPossibleExecOrder function. Our experiments use $k = 1$ and $k = 2$.

In the case of k=1, the algorithm executes all tests in the default order within the same JVM. Any test that does *not* access (read or write) any global fields or other external resources such as a file is not a dependent test. The algorithm executes each of the remaining tests in isolation (i.e., in a fresh JVM); a test is dependent if its result is different than when executed in the default order.

In the case of $k\geq 2$, the algorithm is shown in Figure 6. The defined getPossibleExecOrder function first executes each test in *isolation*, and records the fields that each test reads and writes (lines 1–3). It uses the isolation execution result of each test as a comparison baseline. When generating all possible test permutations of length k, the algorithm checks whether *all* global fields that *each* test (in the generated permutation) may read are not written by *any* test executed before it (lines 6–11). If so, all tests in the permutation must produce the same results as executed in isolation, and the algorithm can safely discard this permutation without executing it. Otherwise, the algorithm adds the generated permutation to the result set (line 9). Finally, the algorithm adds all 1-permutations to the result set (line 13) to find all dependent tests that exhibit different results when executed in isolation. The algorithm in Figure 2 takes the returned result set (line 14) and identifies dependent tests. We have proved the dependence-aware k-bounded algorithm to be correct. Interested readers can refer to [69] for the proof.

The given algorithm uses isolated execution results as a baseline and avoids executing permutations that are redundant with them. It has two major benefits. First, it clusters tests by the fields they read and write. Only tests reading or writing the same global field(s), rather than *all* tests in a suite, are treated as potentially dependent. Second, for tests reading or writing the same global field(s), some permutations are ignored by checking the global fields each test may access (lines 6–11 in Figure 6).

5. TOOL IMPLEMENTATION

We implemented our four dependent test detection algorithms in a tool called DTDetector. DTDetector supports JUnit 3.x/4.x tests.

To ensure there is no interaction between different runs, DTDetector launches a fresh JVM when executing a test permutation, and after a run it resets resources, such as deleting any temporary files that were created. When comparing the observed result of a test in a permutation with its expected

Auxiliary methods:
recordFieldAccess(t): executes test t in a fresh JVM and returns the fields it reads and writes.

getPossibleExecOrder(T):
1: **for** each t in T **do**
2: $\langle reads_t, writes_t \rangle \leftarrow$ recordFieldAccess(t)
3: **end for**
4: $result \leftarrow \emptyset$
5: **for** each ts in kPermutations(T, k) **do**
6: **for** i in 1 ... k **do**
7: $previousWrites \leftarrow \bigcup_{j<i} writes_{ts[j]}$
8: **if** $previousWrites \cap reads_{ts[i]} \neq \emptyset$ **then**
9: $result \leftarrow result \cup ts$
10: **end if**
11: **end for**
12: **end for**
13: $result \leftarrow result \cup$ kPermutations(T, 1)
14: **return** $result$

Figure 6: The dependence-aware k-bounded algorithm to detect dependent tests, for $k \geq 2$. It instantiates the algorithm of Figure 2, defining the getPossibleExecOrder function. For $k=1$, see Section 4.4. The kPermutations auxiliary method is defined in Figure 5.

result, DTDetector considers two JUnit test results to be the same when both tests pass, or when both tests exhibit exactly the same exception or assertion violation, from the same line of code.

To implement the dependence-aware k-bounded algorithm, DTDetector uses ASM [3] to perform load-time bytecode instrumentation. DTDetector inserts code to monitor each static field access (including read and write), and monitors each file access by installing a Java `SecurityManager` that provides file-level read/write information. Each test produces a trace file containing both field and file access information, after being executed on a DTDetector-instrumented program. The dependence-aware k-bounded algorithm uses the recorded read/write information to detect test dependence (Figure 6).

DTDetector conservatively treats both read and write to a mutable static field as a write effect. DTDetector assumes that the JDK is stateless, and thus does not track field access in JDK classes. DTDetector does not perform any sophisticated points-to or shape analyses. It uses the side-effect annotations provided by Javari [45] to determine the immutable classes. That is, if a method is annotated as side-effect-free in Javari, DTDetector ignores all fields accessed by this method.

Optionally, a user can specify a list of "dependence-free" fields (e.g., a static field for logging or counting), which will not be considered as the root cause of manifest test dependence by DTDetector.

6. EMPIRICAL EVALUATION

Our evaluation answers the following research questions:

1. How many dependent tests can each detection algorithm detect in real-world programs (Section 6.3.1)?
2. How long does each algorithm take to detect dependent tests (Section 6.3.2)?
3. Can dependent tests interfere with downstream testing techniques such as test prioritization (Section 6.3.3)?

Table 3: Subject programs used in our evaluation. Column "Tests" shows the number of human-written unit tests. Column "Auto Tests" shows the number of unit tests generated by Randoop [43].

Program	LOC	Tests	Auto Tests	Revision
Joda-Time	27183	3875	–	b609d7d66d
XML Security	18302	108	665	version 1.0.4
Crystal	4676	75	3198	1a11279d3d6c
Synoptic	28872	118	2467	d5ea6fb3157e

6.1 Subject Programs

Table 3 lists the programs and tests used in our evaluation. Each of the programs includes a well-written unit test suite.

Joda-Time [32] is an open source date and time library. XML Security [64] is a component library implementing XML signature and encryption standards. Crystal [8, 13] is a tool that pro-actively examines developers' code and identifies textual, compilation, and behavioral conflicts. Synoptic [6, 57] is a tool to mine a finite state machine model representation of a system from logs. All of the subject programs' test suites are designed to be executed in a single JVM, rather than requiring separate processes per test case [4].

Given the increasing importance of automated test generation tools [15, 21, 43, 71], we also want to investigate dependent tests in automatically-generated test suites. For each subject program, we used Randoop [43], an automated test generation tool, to create a suite of 5,000 tests. Randoop discards redundant tests [46, §III.E]; Table 3 shows how many non-redundant tests Randoop output.

We discarded the automatically-generated test suite of Joda-Time, since many tests in it are non-deterministic — they depend on the current time.

6.2 Evaluation Procedure

We evaluated each algorithm on both the human-written test suite and the automatically-generated test suite of each subject program in Table 3.

We ran the randomized algorithm 10, 100, and 1000 times on each test suite, and recorded the total number of detected dependent tests and time cost for each setting. The choice of 1000 times is based on a practical guideline for using randomized algorithms in software engineering, as summarized in [2]. For the exhaustive k-bounded algorithm and the dependence-aware k-bounded algorithm, we use isolated execution ($k = 1$) and pairwise execution ($k = 2$). The choice of k is based on the results of our empirical study (Section 2) that a small k can find most realistic dependent tests.

We provided DTDetector with a list of 39 "dependence-free" fields for the 4 subject programs. This manual step required about 30 minutes in total.

We examined each output dependent test manually to make sure the test dependence is not caused by non-deterministic factors, such as multi-threading.

Our experiments were run on a 2.67GHz Intel Core PC with 4GB physical memory (2GB was allocated for the JVM), running Windows 7.

6.3 Results

Table 4 summarizes the number of detected dependent tests and the time cost for each algorithm in DTDetector.

6.3.1 Detected Dependent Tests

DTDetector detected 29 human-written dependent tests (among which 27 dependent tests were previously unknown)

Table 4: Experimental results. Column "# Tests" shows the total number of tests, taken from Table 3. Column "# Detected Dependent Tests" shows the number of detected dependent tests in each test suite. When evaluating the randomized algorithm, we used *numtrials* = 10, 100, and 1000 (Figure 4). "—" means the test suite is not evaluated due to its non-determinism. An asterisk (*) means the algorithm did not finish within 1 day: the number of dependent tests is those discovered before timing out, and the time estimation methodology is described in Section 6.3.2.

Subject Programs	# Tests	# Detected Dependent Tests								Analysis Cost (seconds)							
		Rev	Randomized			Exhaustive		Dep-Aware		Rev	Randomized			Exhaustive		Dep-Aware	
			10	100	1000	k=1	k=2	k=1	k=2		10	100	1000	k=1	k=2	k=1	k=2
Human-written unit tests																	
Joda-Time	3875	2	1	1	6	2	≥2 *	2	≥2 *	11	57	528	5538	1265	4×10^6 *	291	5×10^5 *
XML Security	108	0	1	4	4	4	4	4	4	11	65	594	5977	106	11927	93	3322
Crystal	75	18	18	18	18	17	18	17	18	2	14	131	1304	166	7323	95	4155
Synoptic	118	1	1	1	1	0	1	0	1	1	7	67	760	25	3372	24	1797
Total	4176	21	21	24	**29**	23	≥24	23	≥25	26	143	1320	13579	1562	4×10^6 *	503	5×10^5 *
Automatically-generated unit tests																	
Joda-Time	2639	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
XML Security	665	138	167	171	171	129	≥129 *	128	≥128 *	6	50	430	4174	133	1×10^5 *	128	5×10^4 *
Crystal	3198	75	159	162	164	55	≥55 *	55	≥55 *	18	103	949	9436	2477	8×10^6 *	2297	1×10^6 *
Synoptic	2467	3	3	7	10	2	≥2 *	2	≥2 *	12	81	770	6311	454	1×10^6 *	454	2×10^4 *
Total	8969	216	329	340	**345**	186	≥186	185	≥185	36	234	2149	19921	3064	1×10^7 *	2879	1×10^6 *

and 1311 automatically-generated dependent tests. A larger percentage (20% vs. 0.7%) of automatically-generated tests are dependent. Developers' understanding of the code, and their goals when writing the tests, help them build well-structured tests that carefully initialize and destroy the shared objects they may use. By contrast, most automated test generation tools are not "state-aware": the generated tests often "misuse" APIs, such as not setting up the environment correctly. This misuse may indicate that the tests are invalid; it may indicate weaknesses, poor design, or fragility of the APIs; or it may indicate that the human-written tests have failed to exercise some functionality.

The root cause of all the detected dependent tests is improper access to static fields. The XML Security and Crystal developers use more static fields in the test code, so those projects have relatively more dependent tests.

The randomized algorithm is surprisingly effective in detecting dependent tests. In our experiments, when run 1000 times, it found every dependent test identified by the other algorithms, plus 4 more human-written dependent tests in Joda-Time. These 4 tests only manifest when a sequence of 3 tests is run in a specified, non-default order. Both exhaustive and dependence-aware k-bounded algorithms fail to detect these tests, because they cannot scale to k=3 for Joda-Time. The randomized algorithm also detects more dependent tests in the automatically-generated test suites.

The dependence-aware bounded algorithm found the same number of dependent tests as the exhaustive bounded algorithm except that it missed one dependent test in XML Security's automatically-generated test suite. The dependent test was missed because DTDetector did not track static field accesses in dynamically-loaded classes.

6.3.2 Performance of DTDetector

The time cost of the reversal algorithm is very low, and the time cost of the randomized algorithm is proportional to the run time of the suite and the number of runs. Overall, the time cost is acceptable for practical use. For example, the randomized algorithm took around 1.5 hours to finish 1000 runs, for Joda-Time's human-written test suite (3875 tests).

Table 5: Five test prioritization techniques used to assess the impact of dependent tests. These five techniques are introduced in Table 1 of [20]. (We use the same labels as in [20]. We did not implement the other 9 test prioritization techniques introduced in [20], since they require a fault history that is not available for our subject programs.)

Label	Technique Description
T1	Randomized ordering
T3	Prioritize on coverage of statements
T4	Prioritize on coverage of statements not yet covered
T5	Prioritize on coverage of methods
T7	Prioritize on coverage of functions not yet covered

The time cost of running the exhaustive k-bounded algorithm is prohibitive. The JVM initialization time is the main cost. The exhaustive algorithm failed to scale to one human-written test suite and all four automatically-generated test suites when k=2, and failed to scale to all test suites when k=3. The primary reason is the large number of possible test permutations. For example, there are 15,011,750 size-2 permutations for Joda-Time's human-written test suite (3875 tests), which would take approximately 58 days to execute.

Table 4 gives an estimated time cost for each test suite that an algorithm failed to scale to. For each test suite, we randomly chose 1000 permutations from all test permutations, executed them, and measured the average time cost per permutation. Then, we multiplied the average cost by the total number of permutations to estimate the time cost.

The dependence-aware k-bounded algorithm ran about an order of magnitude faster than the exhaustive k-bounded algorithm, when k=2. The dependence-aware algorithm helps most when there are relatively many tests, each one of them relatively small.

Table 6: Differences in test results between original and prioritized human-written unit test suites. Each cell shows the number of tests that do not return the same results as they do when executed in the default, unprioritized order.

Subject Program	T1	T3	T4	T5	T7
Joda-Time	0	0	1	0	0
XML Security	0	0	0	0	0
Crystal	12	11	16	11	12
Synoptic	0	0	0	0	0
Total	12	11	17	11	12

6.3.3 The Impact on Test Prioritization

We implemented five test prioritization techniques [20] (summarized in Table 5) and applied them to the human-written test suites of our subject programs.

For each test prioritization algorithm, we counted the number of dependent tests that return different results (pass or fail) in the prioritized order than they do when executed in the unprioritized order. Table 6 summarizes the results.

The dependent tests in our subject programs interfere with *all* five test prioritization techniques in Table 5. This is because all these techniques implicitly assume that there are no test dependences in the input test suite. Violation of this assumption, as happened in real-world test suites, can cause the prioritized suite to fail even though the original suite passed.

We did not evaluate the effect of test dependence on metrics such as APFD [49]; there is no point optimizing such a metric at the cost of false positives or false negatives.

6.4 Discussion

Developers' Reactions to Dependent Tests. We sent the identified human-written dependent tests to the subject program developers, asking for their feedback.

One dependent test in Joda-Time was previously known and had already been fixed. Joda-Time's developers confirmed the other new dependent tests, and thought that they are due to unintended interactions in the library. The Crystal developers confirmed that all dependent tests found in Crystal were unintentional and happened because of dependence through global variables. The developers considered the dependent tests undesirable and opened a bug report for this issue [14]. The dependent test in Synoptic was previously known. The developers merged two related tests to fix the dependent test. The SIR [53] maintainers confirmed our reported dependent tests in XML-Security, and accepted our suggested patch to fix them. They also highlighted the practice that tests should always "stand alone" without dependency on other tests, and characterized that as "test engineering 101".

Threats to Validity There are several threats to the validity of our evaluation. First, the 4 open-source programs and their test suites may not be representative enough. However, these are the first 4 subject programs we tried, and the fact that we found dependent tests in all of them is suggestive. Second, in this evaluation, we focus specifically on the manifest dependence between *unit tests*. JUnit executes many unit tests in a single JVM. Integration or system tests, or tests written using a different testing framework, might have less dependence if each one is run in its own environment. Third, due to the computational complexity of the general dependent test detection problem, we do not yet have empirical data regarding DTDetector's recall and how many dependent tests exist in a test suite. Fourth, we only assessed the impact of dependent tests on five test prioritization techniques. Using other test prioritization techniques might yield different results.

Experimental Conclusions We have four chief findings. **(1)** Dependent tests do exist in practice, both in human-written and automatically-generated test suites. **(2)** These dependent tests reveal weakness in a test suite rather than defects in the tested code. **(3)** Dependent tests can interfere with test prioritization techniques and cause unexpected test failures. **(4)** The randomized algorithm is the most effective in detecting dependent tests.

7. RELATED WORK

Treating test suites explicitly as *mathematical sets* of tests dates at least to Howden [26, p. 554] and remains common in the literature. The execution order of tests in a suite is usually not considered: that is, test independence is assumed. Nonetheless, some research has considered it. We next discuss some existing definitions of test dependence, techniques that assume test dependence, and tools that support specifying test dependence.

7.1 Test Dependence

Definitions in the testing literature are generally clear that the conditions under which a test is executed may affect its result. The importance of context in testing has been explored in databases [9, 22, 35], with results about test generation, test adequacy criteria, etc., and mobile applications [60]. For the database domain, Kapfhammer and Soffa formally define independent test suites and distinguish them from other suites that "can capture more of an application's interaction with a database while requiring the constant monitoring of database state and the potentially frequent re-computations of test adequacy" [35, p. 101]. By contrast, our definition differs from that of Kapfhammer and Soffa by considering test results rather than program and database states (which may not affect the test results).

The IEEE Standard for Software and System Test Documentation (829-1998) §11.2.7, "Intercase Dependencies," says in its entirety: "List the identifiers of test cases that must be executed prior to this test case. Summarize the nature of the dependences" [28]. The succeeding version of this standard (829-2008) adds a single sentence: "If test cases are documented (in a tool or otherwise) in the order in which they need to be executed, the Intercase Dependencies for most or all of the cases may not be needed" [29].

Bergelson and Exman characterize a form of test dependence informally: given two tests that each pass, the composite execution of these tests may still fail [5, p. 38]. However, they do not provide any empirical evidence of test dependence nor any detection algorithms.

The C2 wiki acknowledges test dependence as undesirable [59]:

> Unit testing ... requires that we test the unit in isolation. That is, we want to be able to say, *to a very high degree of confidence* [emphasis added], that any actual results obtained from the execution of test cases are purely the result of the unit under test. The introduction of other units may color our results.

They further note that other tests, as well as stubs and drivers, may "interfere with the straightforward execution of one or more test cases."

Compared with these informal definitions, we formalize test dependence and characterize test dependence in practice.

7.2 Techniques Assuming Test Independence

The assumption of test independence lies at the heart of most techniques for automated regression test selection [7, 24, 41, 42, 67], test case prioritization [20, 31, 37, 50, 54], coverage-based fault localization [33, 55, 68], etc.

Test prioritization seeks to reorder a test suite to detect software defects more quickly. Early work in test prioritization [49, 62] laid the foundation for the most commonly used problem definition: consider the set of all permutations of a test suite and find the best award value for an objective function over that set [20]. The most common objective functions favor permutations where higher code coverage is achieved and more faults in the underlying program are found with running fewer tests. Test independence is a requirement for most test selection and prioritization work (e.g., [50, p. 1500]). Evaluations of selection and prioritization techniques are based in part on the test independence assumption as well as the assumption that the set of faults in the underlying program is known beforehand [17, 49, *et alia*]; the possibility that test dependence may interfere with these techniques is not studied.

Coverage-based fault localization techniques often treat a test suite as a collection of test cases whose result is *independent* of the order of their execution [33]. They can also be impacted by test dependence. In a recent evaluation of several coverage-based fault locators, Steimann et al. found that fault locators' accuracy is affected by tests that fail due to violation of the test independence assumption [55]. Compared to our work, Steimann et al.'s work focuses on identifying possible threats to validity in evaluating coverage-based fault locators, and does not present any formalism, study, or detection algorithms for dependent tests.

Test independence is different than determinism. Nondeterminism does not imply dependence: a program may execute non-deterministically, but its tests may deterministically succeed. Further, a test may non-deterministically pass/fail without being affected by any other test, including its own previous executions. Determinism does not imply independence: a program may have no sources of nondeterminism, but two of its tests can be dependent. The testing community sometimes mentions determinism (such as multithreading) and execution environment (such as library versions), without considering test dependence [42]. A stronger assumption than determinism is the Controlled Regression Testing Assumption (CRTA) [48]. It forbids porting to another system, nondeterminism, time-dependencies, and interactions with the external environment. It also forbids test dependence, though the authors did not mention test dependence explicitly. The authors state that CRTA is "not necessarily impossible" to employ. We have a practical focus on the often-overlooked issue of test dependence.

As shown in Sections 2 and 6, the test independence assumption often does not hold for either human-written or automatically-generated tests; and the dependent tests identified in our subject programs interfere with existing test prioritization techniques. Thus, techniques that rely on this assumption may need to be reformulated.

7.3 Tools Supporting Test Dependence

Testing frameworks provide mechanisms for developers to define the context for tests. JUnit 4.11 supports executing tests in lexicographic order by test method name [34]. DepUnit [16] allows developers to define dependences between two unit tests. TestNG [58] allows dependence annotations and supports a variety of execution policies that respect these dependences. What distinguishes our work from these testing frameworks is that, while they allow dependences to be made explicit and respected during execution, they do not help developers *identify* dependences.

Haidry and Miller proposed a set of test prioritization techniques that consider test dependence [23]. Their work assumes that dependencies between tests are known, and improves existing test prioritization techniques to make them produce a test ordering that preserves the test dependencies. By contrast, our work formally defines test dependence, studies the characteristics of real-world test dependence, shows how to detect dependent tests, and empirically evaluates whether dependent tests exist in real-world programs and their impact on test prioritization techniques.

Our previous work proposed an algorithm to find bugs by executing each unit test in isolation [40]. With a different focus, this work investigates the validity of the test independence assumption rather than finding new bugs, and it presents five new results. Further, as indicated by our study and experiments, most dependent tests reveal weakness in the test code rather than bugs in the program. Thus, using test dependence may not achieve a high return in bug finding.

A simple way to eliminate test dependence is starting a new process or otherwise completely re-initializing the environment (variables, heap, files, etc.) before executing each test; JCrasher does this [15], as do some SIR applications [53] and some database or GUI testing tools [9, 22, 35]. However, such an approach is computationally expensive: Table 4 shows that executing each test in a separate JVM introduces 10–138× slowdown (compare the "Exhaustive $k = 1$" column to the "Rev" column).

8. CONCLUSION

Test independence is widely assumed but rarely addressed, and test dependence has largely been ignored in previous research on software testing. We formalized the dependent test detection problem. To detect dependent tests, we designed and implemented four algorithms to identify manifest test dependence in a test suite. We showed that test dependence does arise in practice: our experiments revealed dependent tests in every subject program we studied, from both human-written and automatically-generated test suites. The dependent tests cause real-world prioritized test suites to fail, for five existing test prioritization techniques.

9. ACKNOWLEDGMENTS

Bilge Soran participated in the project that led to the initial result. Reid Holmes and Laura Inozemtseva identified a Joda-Time dependence. Cheng Zhang suggested exploring software issue tracking systems to study dependent tests. Yuriy Brun, Colin Gordon, Mark Grechanik, Adam Porter, Michal Young, Reid Holmes, and anonymous reviewers provided insightful comments on a draft. This work was supported in part by NSF grants CCF-1016701 and CCF-0963757.

10. REFERENCES

[1] Apache issue tracker. `https://issues.apache.org/jira`.

[2] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.

[3] ASM. `http://asm.ow2.org/`.

[4] J. Bell and G. Kaiser. Unit Test Virtualization with VMVM. In *ICSE*, pages 550–561, 2014.

[5] B. Bergelson and I. Exman. Dynamic test composition in hierarchical software testing. In *2006 IEEE 24th Convention of Electrical and Electronics Engineers in Israel*, pages 37–41, 2006.

[6] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE*, pages 267–277, 2011.

[7] L. C. Briand, Y. Labiche, and S. He. Automating regression test selection based on uml designs. *Inf. Softw. Technol.*, 51(1):16–30, Jan. 2009.

[8] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *ESEC/FSE*, pages 168–178, 2011.

[9] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weber. A framework for testing database applications. In *ISSTA*, pages 147–157, 2000.

[10] Apache CLI. `http://commons.apache.org/cli/`.

[11] A CLI bug masked by dependent tests. `https://issues.apache.org/jira/browse/CLI-26` `https://issues.apache.org/jira/browse/CLI-186` `https://issues.apache.org/jira/browse/CLI-187`.

[12] Codehaus issue tracker. `http://jira.codehaus.org`.

[13] Crystal VC. `http://crystalvc.googlecode.com`.

[14] A bug report in Crystal about dependent tests. `https://code.google.com/p/crystalvc/issues/detail?id=57`.

[15] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exper.*, 34(11):1025–1050, Sept. 2004.

[16] DepUnit. `https://code.google.com/p/depunit/`.

[17] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering*, 36(5):593–617, 2010.

[18] Eclipse issue tracker. `https://bugs.eclipse.org`.

[19] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *FSE*, pages 253–264, 2006.

[20] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, pages 102–112, 2000.

[21] G. Fraser and A. Zeller. Generating parameterized unit tests. In *ISSTA*, pages 364–374, 2011.

[22] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Rec.*, 23(2):243–252, 1994.

[23] S. Z. Haidry and T. Miller. Using dependency structures for prioritization of functional test suites. *IEEE Transactions on Software Engineering*, 39(2):258–275, 2013.

[24] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *OOPSLA*, pages 312–326, 2001.

[25] Hibernate issue tracker. `https://hibernate.atlassian.net`.

[26] W. Howden. Methodology for the generation of program test data. *IEEE Transactions on Computers*, C-24(5):554–560, 1975.

[27] H.-Y. Hsu and A. Orso. MINTS: A General Framework and Tool for Supporting Test-suite Minimization. In *ICSE*, May 2009.

[28] IEEE. IEEE standard for software test documentation. *IEEE Std 829-1998*, 1998.

[29] IEEE. IEEE standard for software and system test documentation. *IEEE Std 829-2008*, pages 1–118, 2008.

[30] JBoss issue tracker. `https://issues.jboss.org/`.

[31] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *ASE*, pages 233–244, 2009.

[32] Joda Time. `http://joda-time.sourceforge.net/`.

[33] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[34] Test Execution Order in JUnit. `https://github.com/junit-team/junit/blob/master/doc/ReleaseNotes4.11.md#test-execution-order`.

[35] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *ESEC/FSE*, pages 98–107, 2003.

[36] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum, 1972.

[37] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*, pages 119–129, 2002.

[38] T. Kim, R. Chandra, and N. Zeldovich. Optimizing unit test execution in large software programs using dependency analysis. In *APSys*, pages 19:1–19:6, 2013.

[39] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with Korat. In *FSE*, pages 135–144, 2007.

[40] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. In *FSE (NIER Track)*, pages 496–499, 2011.

[41] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *ICST*, pages 21–30, 2011.

[42] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *FSE*, pages 241–251, 2004.

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

[44] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *ISSTA*, pages 75–86, 2008.

[45] J. Quinonez, M. S. Tschantz, and M. D. Ernst.

Inference of reference immutability. In *ECOOP*, pages 616–641, July 9–11, 2008.

[46] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *ASE*, pages 23–32, 2011.

[47] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol.*, 13(3):277–331, July 2004.

[48] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug. 1996.

[49] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *ICSM*, pages 179–188, 1999.

[50] M. J. Rummel, G. M. Kapfhammer, and A. Thall. Towards the prioritization of regression test suites with data flow information. In *SAC*, pages 1499–1504, 2005.

[51] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE*, 2005.

[52] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA*, pages 69–80, 2009.

[53] Software-artifact Infrastructure Repository. `http://sir.unl.edu`.

[54] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA*, pages 97–106, 2002.

[55] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*, pages 314–324, 2013.

[56] SWT. `http://eclipse.org/swt/`.

[57] Synoptic. `http://synoptic.sourceforge.net/`.

[58] TestNG. `http://testng.org/`.

[59] Standard definition of unit test. `http://c2.com/cgi/wiki?StandardDefinitionOfUnitTest`. Accessed: 2012/03/16.

[60] Z. Wang, S. Elbaum, and D. S. Rosenblum. Automated generation of context-aware tests. In *ICSE*, pages 406–415, 2007.

[61] J. A. Whittaker, J. Arbon, and J. Carollo. *How Google Tests Software*. Addison-Wesley Professional, 2012.

[62] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore. A study of effective regression testing in practice. In *ISSRE*, pages 264–274, 1997.

[63] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang. Language-based replay via data flow cut. In *FSE*, pages 197–206, 2010.

[64] XML Security. `http://projects.apache.org/projects/xml_security_java.html`.

[65] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28:183–200, February 2002.

[66] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *ISSTA*, pages 235–245, 2013.

[67] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *ISSTA*, pages 331–341, 2012.

[68] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*, pages 765–784, 2013.

[69] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. Technical Report UW-CSE-14-01-01, Department of Computer Science & Engineering, University of Washington, 2014. Available at: `ftp://ftp.cs.washington.edu/tr/2014/01/UW-CSE-14-01-01.PDF`.

[70] S. Zhang and K. Muşlu. List of papers on test prioritization techniques from major software engineering conferences and journals, 2000–2013. `http://homes.cs.washington.edu/~szhang/test_prioritization_paper_list_2000-2013.txt`.

[71] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363, 2011.