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. Test independence is important so that tests behave consistently as intended by the developers. In addition, techniques such as test prioritization assume that the tests in a suite are independent, but they do not justify that assumption. 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 three 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 and found that dependent tests affect the output of *all* five techniques.

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. The assumption of test independence is important so that tests behave consistently as designed. In addition, many techniques assume test independence, including test prioritization [19,29,47,51], test selection [6,22,25,39,40,64], test execution [36,37], test factoring [48,60], test carving [18], and experimental debugging techniques [52,62,65]. However, this critical assumption is rarely questioned, investigated, or even mentioned: none of the above papers mentions the assumption as a limitation or a threat to validity. A total of 31 papers on test prioritization have been published in the research track of five major software engineering conferences (ICSE, FSE, ISSTA, ASE, and ICST) or in two major software engineering journals (TOSEM and TSE) between 2000 and 2013 [68]. Of these, 27 papers explicitly or implicitly assumed test independence, 3 papers acknowledged that the potential dependences between tests may affect the prioritization output [35,42,45], and only 1 paper considered test dependence in the design of test prioritization algorithms [16]. Anecdotally, researchers have told us that test dependence is not a significant concern in practice. We wish to investigate 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 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.

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 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 as when they are writing other code. 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, will have unintended and unexpected behaviors in some cases.

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 [9] in the Apache CLI library [8] 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 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 [17] 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 (that identify a subset of the input test suite to run during regression testing) [6, 22, 25, 39, 40, 64], test prioritization techniques (that reorder the input to discover defects sooner) [19, 29, 35, 47, 51], test parallelization techniques (that schedule the input tests for execution across multiple CPUs) [37], test execution techniques [36], test factoring [48, 60] and test carving [18] (which convert large system tests into smaller unit tests), experimental debugging techniques (such as Delta Debugging [52, 62, 65] and mutation analysis [49, 63, 64], which run a set of tests repeatedly), etc. Most of these downstream testing 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 6.3.4 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 three algorithms to detect dependent tests: one randomized, one exhaustive bounded, and one that prunes the search space using dynamic analyses. All three 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 prototype tool, called DTDetector (Section 5). 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 (Section 6).

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 [17], JBoss [28], Hibernate [23], and Codehaus [10]. 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, 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

Issue Tracking System	Dependent Tests				# Involved Tests for Manifestation					Resolution					Root Cause			
	Total Number	Severity									Patch Location				Static Variable	File System	Data-base	Unknown
		Major	Minor	Trivial	Self	1 test	2 tests	3 tests	Unknown	Days	Code	Test	Doc	Unfixed				
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

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.

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 improper access to shared static variables. Among 96 dependent tests, 59 (61%) of them arise due to inappropriate access to shared static variables; 4 (4%) of them arise due to inappropriate access to the file system, and 10 (10%) of them arise due to inappropriate access to a database. The root cause for the remaining 23 (25%) tests is not apparent in the issue tracking system.

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 each dependent test will not read an undesired 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 *Manifestation 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 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.

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

Table 2: Manifestation of the 96 dependent tests.

Sometimes developers introduce dependent tests intentionally because it is more efficient or convenient [33, 58]. 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 dependent tests, and 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 issue tracking system contains at least 49 such dependent tests. In September 2003, a user filed a bug report in SWT [53] [17], 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

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

Figure 1: Simplified fault-related code in CLI [8] (revision 661513). The fault was masked by two dependent tests for over 3 years.

always generate environments in which t passes accidently 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 [8]. Figure 1 shows the simplified fault-related code. The fault is due to improper initialization of the static variable `argName`. The static variable `argName` should be set to its default value `"arg"` by CLI's clients via calling method `reset()`. Otherwise, `argName`'s default value remains `null` and should *not* be used in creating an `OptionBuilder` object. In CLI, two test cases `BugsTest.test13666` and `BugsTest.test27635` can reveal this potential fault by directly initializing 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.

Such dependent tests have a non-trivial impact in practice. This fault was reported in the bug database several times [9], 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. Thus, a practical technique can focus on running only short subsequences (whose length is bounded by a parameter k) of a test suite. This will reduce the number of permutations 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, file systems, databases, network, etc. However, as reflected by our study, more than half of the real-world dependent tests are caused by improper static variable accesses. This implies that a technique may achieve a high return 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 did not intentionally ignore any test dependence in the issue tracking system. However, a limitation is that the developers might have made a mistake, might not have marked a test dependence in a way we found it (different search terms might discover additional dependent tests), and are unlikely to have found all the dependent tests in those projects.

3. FORMALIZING TEST DEPENDENCE

The result of a test not only depends on its input data but also 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 (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 2 (Test). *A test is a sequence of executable program statements, and an oracle — a Boolean predicate that decides whether the test passes or fails.*

For simplicity, our definition does not consider non-deterministic tests, non-terminating tests, and tests aborting the JVM.

Definition 3 (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 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 the execution of a test $t \in \mathcal{T}$ and an environment $\mathbf{E} \in \mathcal{E}$ to a new (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.

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 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 for T?*

We prove that this problem is NP-hard by reducing the NP-complete Exact Cover problem to the Dependent Test Detection problem [34]. 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. In the Exact Cover problem, we are given a set $X = \{x_1, x_2, x_3, \ldots, x_m\}$ and a collection S of subsets of X. The goal is to identify a sub-collection S^* of S such that *each* element in X is contained in *exactly* one subset in S^*.

Assume a set $V = \{v_1, v_2, v_3, \ldots, v_m\}$ of variables, and a set $S = \{S_1, S_2, \ldots, S_n\}$ with $S_i \subseteq V$ for $1 \leq i \leq n$.

We now construct a tested program P, and a test suite $T = \langle t_1, t_2, \ldots t_n, t_{n+1} \rangle$ as follows:

- P consists of m global variables $v_1, v_2, \ldots, v_m$, each with initial value 1.
- For $1 \leq i \leq n$, t_i is constructed as follows: for $1 \leq j \leq m$, if $v_j \in S_i$, then add a single assignment statement `v`$_j$ `= v`$_j$ `- 1` to t_i.

  t_{n+1} consists only of the oracle `assert(`v_1 `!= 0 ||` v_2 `!= 0 ...||` v_m `!=0)`.

In the above construction, the tests t_i for $1 \leq i \leq n$ will always pass. The only test that may fail and thus exhibit different behavior is t_{n+1}, which *only* fails when each variable v_i appears exactly once in a test case.

For the given test t_{n+1}, if we can find a sequence $\langle t_{i_1}, t_{i_2}, \ldots, t_{i_j} \rangle$ that makes t_{n+1} fail, the subsets S^* corresponding to each t_{i_j} are an exact cover of V.

```
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, and 5.

In practice, the structure of the proof directly translates to the structure of test suites. t_{n+1} is the dependent test, S is defined by the tests that write variables used by t_{n+1}, and every exact cover of S represents an independent shortest test suite that is a manifest dependency of t_{n+1}. □

To complete the proof that Dependent Test Detection is NP-complete, we provide an algorithm to verify a solution to the problem, that is linear in the size of the test suite. Given a test suite T, a test $t \in T$ and a sequence $S \subseteq T$ that manifests a dependency on t, we first execute T, then S, and compare the result for t in both executions. If the results differ the solution is correct; if they do not differ, the solution is rejected. Since in the worst case we have to execute $2n$ tests, the complexity of this algorithm is linear.

3.3 Discussion

For the sake of simplicity, 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 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 for it.

To approximate the exact solution, this section presents three approximate algorithms that find a *subset* of all dependent tests. Section 4.1 describes a randomized algorithm that repeatedly executes all the tests of a suite in random order. Section 4.2 describes an exhaustive bounded algorithm that executes all possible sequences of k tests for a bounding parameter k (specified by the user). Section 4.3 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 three 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 Randomized Algorithm

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

Figure 3: 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 4: The exhaustive k-bounded algorithm to detect dependent tests. It instantiates the algorithm of Figure 2, defining the getPossibleExecOrder function.

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 it for the randomized algorithm by randomizing the original test execution order (line 2).

4.2 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, 82% of the real-world dependent tests can be found by running no more than 2 tests together. Instead of executing all permutations of the whole test suite, our algorithm (Figure 4) 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 4 shows the algorithm.

4.3 Dependence-Aware Bounded Algorithm

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 uses dynamic analyses to prune the search space.

Its key idea is to estimate which tests can interact through which fields and to only run permutations in which the interactions may be different. The algorithm determines, for every field read by a test, which test previously wrote that field. If, in a permutation, all of those relationships are unchanged from the default test order, then all the tests in that permutation give the same result as in the default order (and, therefore, that permutation need not be run). 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, and a general version for $k \geq 2$.

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 (including read and 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) and reports the tests whose results are different than when executed in the default order.

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 each $t_i$ in $ts$ { $i$ is the index of $t_i$ in $ts$ } do
 7:         $previousWrites \leftarrow \bigcup_{j<i} writes_{t_j}$
 8:         if $previousWrites \cap reads_{t_i} \neq \emptyset$ then
 9:             $result \leftarrow results \cup ts$
10:         end if
11:     end for
12: end for
13: return $result$
```

Figure 5: 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 4.3.

In the case of $k \geq 2$, the algorithm first runs the case of k=1 (described above) to find all dependent tests that exhibit different results when executed in the default order and when executed in isolation. Then, it runs the algorithm shown in Figure 5. 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 6–10). 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), and the algorithm in Figure 2 identifies dependent tests.

We have proved the dependence-aware k-bounded algorithm to be correct. Interested readers can refer to [66] for the proof.

The given algorithm uses isolated execution results as a baseline and avoids executing permutations that are redundant with them. It would be possible to optimize the algorithm by adding each executed permutation to the baseline and avoiding all redundant executions. Such an algorithm is more complex and we do not show it.

The dependence-aware k-bounded algorithm 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 can be ignored by checking the global fields each test may access.

5. TOOL IMPLEMENTATION

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

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

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 [41].

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

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` which 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.

DTDetector conservatively treats both read and write to a mutable static field as a write effect, since a read access to a static field may mutate objects reachable from the field in the heap. 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 [43] to determine the immutable classes.

Optionally, users can also specify a list of "dependence-free" fields (e.g., a static field for logging or counting), which can never be the root cause of manifest test dependence. DTDetector ignores accesses to these fields.

The source code of DTDetector is available at: `http://testisolation.googlecode.com`.

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 in DTDetector take to detect dependent tests (Section 6.3.2)?

3. Which algorithm is the most cost-effective in detecting dependent tests (Section 6.3.3)?

4. Can dependent tests interfere with downstream testing techniques such as test prioritization (Section 6.3.4)?

6.1 Subject Programs

Table 3 lists the programs and tests used in our evaluation. We used these subject programs because they have been developed for a considerable amount of time (3–10 years) and each of them includes a well-written unit test suite.

Joda-Time [30] is an open source date and time library. It is a mature project that has been under active development for ten years. XML Security [61] is a component library implementing XML signature and encryption standards. XML Security is included in the SIR repository [50] and has been used widely as a subject program in the software testing community. Crystal [11] is a tool that pro-actively examines developers' code and identifies textual, compilation, and behavioral conflicts. Synoptic [54] 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 [13,20,41,69], we also want to investigate dependent tests in automatically-generated test suites. For each subject program, we use Randoop [41], a state-of-the-art automated test generation tool, to create a suite of 5,000 tests. Randoop automatically drops textually-redundant tests and outputs a subset of the generated tests as shown in Table 3.

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 cost 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) and 1311 automatically-generated dependent tests. A larger percentage (15%) 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

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

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. k-bounded algorithm, respectively. When evaluating the randomized algorithm, we used *numtrials* = 10, 100, 1000 (Figure 3). "—" means the test suite is not evaluated due to its non-determinism. An asterisk (*) means the algorithm did not finish (i.e., requiring more than 1 day to execute all test permutations): the number of dependent tests is those discovered before timing out, and the time estimation methodology is described in Section 6.3.2.

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 there are relatively more dependent tests detected in them.

The randomized algorithm is surprisingly effective in detecting dependent tests. In our experiments, when run 1000 times, it identifies *more* dependent tests and found all dependent tests identified by the other two algorithms. For the human-written test suites, the randomized algorithm detects 4 more dependent tests in Joda-Time. These tests only manifest when a sequence of three tests are 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. Related, the randomized algorithm detects more dependent tests in the automatically-generated test suites, because both the exhaustive and dependence-aware k-bounded failed to scale to k=2 for all 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 access in the `java.security` package of the JDK, and Javari did not provide annotations for APIs in that package.

6.3.2 Performance of DTDetector

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).

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 finish.

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 multiple 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.

6.3.3 Comparison of Algorithms

We next discuss the tradeoffs between choosing different detection algorithms in DTDetector. Although the randomized algorithm detects the most dependent tests in our subject programs, it has several limitations. First, there is no guarantee of how many dependent tests the randomized algorithm can detect. A randomized algorithm might even produce different results across different runs. Second, there is no clear stopping criterion for running the randomized algorithm in practice. Thus, it can be hard for users to know how many runs would be enough to find all dependent tests in a test suite. Third, given an identified dependent test, users need to inspect the tests executed before it and isolate a minimized subsequence of tests (either manually or using an assisting tool [62]) to understand the dependence root cause.

By contrast, both the exhaustive k-bounded and the dependence-aware k-bounded algorithms systematically search for dependent tests of a given size and do not suffer from the above limitations. However, the major limitation that prevents them being applied to a large test suite is the time cost to explore all possible test permutations.

6.3.4 The Impact on Test Prioritization

We implemented five test prioritization techniques [19]

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

Table 5: Five test prioritization techniques used to assess the impact of dependent tests. These five techniques are introduced in Table 1 of [19]. (We use the same labels as in [19]. We did not implement the other 9 test prioritization techniques introduced in [19], since they require a fault history that is not available for our subject programs.)

Subject Program	T1	T3	T4	T5	T7
Joda-Time	0	0	0	0	0
XML Security	0	0	0	0	0
Crystal	6	0	2	1	1
Synoptic	0	1	0	0	0
Total	6	1	2	1	1

Table 6: Results of evaluating the five test prioritization techniques in Table 5 on four human-written unit test suites. Each cell shows the number of dependent tests that do not return the same results as they do when executed in the default, unprioritized order.

(summarized in Table 5) and evaluated them on 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 in the prioritized order as they do when executed in the unprioritized order. Table 6 summarizes the results.

The dependent tests in our subject programs interfere with *all* the 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 unit test suites, causes undesired output.

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 design of the library. The Crystal developers confirmed that all dependent tests found in Crystal were not intentional and happened because of dependence through global variables. The developers considered the dependent tests undesirable and opened a bug report for this issue [12]. The dependent test in Synoptic was previously known. The developers merged two related tests to fix the dependent test. The SIR [50] 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. Thus, we cannot claim the results can be generalized to an arbitrary program. 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*. We did not investigate possible test dependence that may arise in other types of tests, such as integration tests. 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 achieve 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)** Like the dependent tests studied in Section 2, the identified dependent tests in our subject programs generally 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 output. **(4)** The randomized algorithm is the most cost-effective in detecting dependent tests, but it has no guarantee of the number of dependent tests it can detect.

7. RELATED WORK

Treating test suites explicitly as *mathematical sets* of tests dates at least to Howden [24, 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 [7, 21, 33], with results about test generation, test adequacy criteria, etc., and mobile applications [57]. 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" [33, 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" [26]. 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" [27].

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]. That is, if t_1 executed by itself passes and t_2 executed by itself passes, executing the sequence $\langle t_1, t_2 \rangle$ in the same context may fail. However, they do not provide any empirical evidence of test dependence nor any detection algorithms.

The C2 wiki acknowledges test dependence as undesirable [56]:

> 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 [6, 22, 39, 40, 64], test case prioritization [19, 29, 35, 47, 51], and coverage-based fault localization [31, 52, 65], etc.

Test prioritization seeks to reorder a test suite to detect software defects more quickly. Early work in test prioritization [46, 59] 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 [19]. 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., [47, p. 1500]). Evaluations of selection and prioritization techniques [15, 46, *et alia*] 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; the possibility that test dependence may interfere with these techniques is not studied.

Coverage-based fault localization techniques [31] often treat a test suite as a collection of test cases whose result is *independent* of the order of their execution. They can also be impacted by test dependence. In a recent evaluation of several coverage-based fault locators, Steimann et al. found fault locators' accuracy has been affected by tests failed due to the violation of the test independence assumption [52]. 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.

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.

Most automated test generation techniques [41, 57, 69] do not take test dependence into consideration. As shown in our experiments (Section 6) and previous work [44], a large number of tests generated by Randoop are dependent. We speculate that these dependences arise because automated test generators generally create new tests based on the program state after executing the previous test, for the sake of test diversity and efficiency. When Randoop generates a nondeterministic test, it can disable the test but leave it in the suite where it is executed in order to prevent other tests that are dependent on it from beginning to fail [44]. Exploring how to incorporate test dependence into the design of an automated test generator is future work.

7.3 Tools Supporting Test Dependence

Testing frameworks provide mechanisms for developers to define the context for tests. JUnit, for example, provides means to automatically execute setup and clean-up tasks (`setUp()` and `tearDown()` in JUnit 3.x, and annotations `@Before` and `@After` in JUnit 4.x). The latest release 4.11 of JUnit supports executing tests in lexicographic order by test method name [32]. However, ensuring that these mechanisms are used properly is beyond the scope and capability of any framework. Further, our empirical study and experimental results indicate that programmers often do not use them properly and introduce dependent tests.

Only a few tools explicitly allow developers to annotate dependent tests and provides supporting mechanisms to ensure that the test execution framework respects those annotations. DepUnit [14] allows developers to define soft and hard dependences. Soft dependences control test ordering, while hard dependences in addition control whether specific tests are run at all. TestNG [55] allows dependence annotations and supports a variety of execution policies that respect these dependences such as sequential execution in a single thread, execution of a single test class per thread, etc. What distinguishes our work from these approaches is that, while they allow dependences to be made explicit and respected during execution, they do not help developers *identify* dependences. A tool that finds dependences (Section 5) could co-exist with such frameworks by generating annotations for them.

Haidry and Miller [16] proposed a set of test prioritization techniques that consider test dependence. Their work aims to improve existing test prioritization techniques to make them produce a test ordering that preserves the test dependencies. Their work assumes that dependencies between tests are known (and are represented as partial orderings, such as that one test should be executed before another) without providing any empirical evidence of whether dependent tests exist in practice. 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 existing test prioritization techniques.

Our previous work [38] proposed an algorithm to find bugs by executing each unit test in isolation. With a different focus, this work investigates the validity of the test independence assumption rather than finding new bugs, and 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 [13] does this, as do some SIR applications [50] and some database or GUI testing tools [7, 21, 33]. However, such an approach is computationally expensive: Table 4 shows that executing each test in a separate JVM introduces 16–240× slowdown (compare the "Exhaustive $k = 1$" column to 1/10 of the "Randomized 10" column).

8. CONCLUSION AND FUTURE WORK

Test independence is broadly assumed but rarely addressed, and test dependence has largely been ignored in previous research on software testing. To understand dependent tests, we described one of the first studies on real-world dependent tests. We showed that test dependence *does* arise in practice, and could have non-trivial repercussions. We also formalized the dependent test detection problem. To detect dependent tests, we designed and implemented three algorithms to identify manifest test dependence in a test suite. Our experiments revealed dependent tests in every subject program we studied, from both human-written and automatically-generated test suites. The revealed dependent tests interfere with five existing test prioritization techniques. Our tool is publicly available at `https://testisolation.googlecode.com/`.

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 how to detect and correct all dependent tests.

As future work, we plan to study the impact of dependent tests on other downstream testing techniques, such as test selection and test parallelization. We also plan to develop a general methodology to eliminate dependent tests. In addition, we are interested in investigating how to prevent dependent tests. One possible way is encouraging developers to use advanced testing frameworks that support test dependence [55], so that developers can explicitly specify test dependence when writing tests. Stylized coding patterns may also be useful. Developers should be encouraged to write tests "defensively" by specifying necessary test execution pre-conditions and using less (or properly mocking) global variables or shared resources.

9. 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, New York, NY, USA, 2011. ACM.

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

[4] J. Bell and G. Kaiser. Unit Test Virtualization with VMVM. *University of Columbia Technical Report CUCS-021-13*, 2013.

[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] 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.

[7] 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.

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

[9] 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`.

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

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

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

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

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

[15] 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.

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

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

[18] 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.

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

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

[21] 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.

[22] 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.

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

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

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

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

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

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

[29] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *ASE*, pages 233–244, Washington, DC, USA, 2009. IEEE Computer Society.

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

[31] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, New York, NY, USA, 2002. ACM.

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

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

[34] 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.

[35] 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.

[36] T. Kim, R. Chandra, and N. Zeldovich. Optimizing unit test execution in large software programs using dependency analysis. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 19:1–19:6, New York, NY, USA, 2013. ACM.

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

[38] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. In *FSE NIER*, SIGSOFT/FSE '11, pages 496–499, 2011.

[39] 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, Washington, DC, USA, 2011. IEEE Computer Society.

[40] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *FSE*, pages 241–251, New York, NY, USA, 2004. ACM.

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

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

[43] J. Quinonez, M. S. Tschantz, and M. D. Ernst. Inference of reference immutability. In *ECOOP*, pages 616–641, Paphos, Cyprus, July 9–11, 2008.

[44] 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*, Nov. 2011.

[45] 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.

[46] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 179–188, 1999.

[47] M. J. Rummel, G. M. Kapfhammer, and A. Thall. Towards the prioritization of regression test suites with data flow information. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 1499–1504, 2005.

[48] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE*, pages 114–123, 2005.

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

[50] SIR. `http://sir.unl.edu`.

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

[52] 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.

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

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

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

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

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

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

[59] 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.

[60] 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, New York, NY, USA, 2010. ACM.

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

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

[63] L. Zhang, D. Marinov, and S. Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *ISSTA*, pages 235–245, New York, NY, USA, 2013. ACM.

[64] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *ISSTA*, pages 331–341, New York, NY, USA, 2012. ACM.

[65] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*, pages 765–784, New York, NY, USA, 2013. ACM.

[66] S. Zhang. Proof of the correctness of the dependence-aware k-bounded algorithm, Sept. 2013. `http://homes.cs.washington.edu/~szhang/pdf/dependence-aware-algorithm-correctness-proof.pdf`.

[67] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically Revisiting the Test Independence Assumption. *University of Washington Technical Report UW-CSE-14-01-01. Available at: http://homes.cs.washington.edu/~szhang/pdf/testdependence.pdf*, 2014.

[68] 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`.

[69] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363, Toronto, Canada, July 19–21, 2011.