When Tests Collide: Evaluating and Coping with the Impact of Test Dependence

Wing Lam   Sai Zhang   Michael D. Ernst
Department of Computer Science & Engineering
University of Washington
{winglam, szhang, 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. The assumption of test independence is important so that tests behave consistently as designed. In addition, many downstream testing techniques, including test prioritization, test selection, and test parallelization, assume test independence. However, this critical assumption often does not hold in practice.

This paper empirically investigates the impact of test dependence on three downstream testing techniques (test prioritization, selection, and parallelization) and proposes a general approach to cope with such impact. It presents two sets of results.

First, we describe an empirical study to assess the impact of test dependence on 4 test prioritization, 6 test selection, and 2 test parallelization algorithms. Test dependence negatively affects the results of all these downstream testing algorithms. For example, an automatically-generated test suite for the XML-Security program contains 665 tests, and 111 of those tests yield a different test result (success vs. fail) if the suite is parallelized to run on 16 CPUs.

Second, we present an approach that enhances each test prioritization, selection, and parallelization algorithm to respect test dependence, so that each test in a suite yields the same result before and after applying the downstream testing technique. In an experimental evaluation, the enhanced testing algorithms worked as intended: the test results were consistent even in the presence of test dependence, and they did not substantially compromise the effectiveness of the original testing algorithms.

1. INTRODUCTION

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 [11, 14]. This assumption is important so that tests behave consistently as designed. However, the test independence assumption often does not hold in practice [52]: a test's result may depend on whether it runs after some other tests.

Test dependence causes inconsistent test results in different execution orders. This leads to *missed alarms* by masking 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 [7]. It also leads to *false alarms*. 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 report bugs [1].

Previous work showed that test dependence exists: there exist reordering of real-world test suites that cause false alarms and missed alarms [52]. Even if test dependence is widespread, it is possible that testing techniques such as test prioritization, selection, and parallelization do not give rise to the problematic test reordering in practice, How often do false alarms and missed alarms occur in practice? How can we modify testing techniques so that they no longer suffer from false alarms and missed alarms, and they are safe to run even on test suites that contain dependent tests?

This paper answers the above questions by empirically investigating the impact of test dependence on 12 downstream testing algorithms and presenting enhanced algorithms that cope with the impacts.

1.1 Evaluating the Impact

Test dependence can affect downstream testing techniques that change a test suite and thereby change a test's execution environment. Examples of downstream testing techniques include test prioritization (reorder the input to discover defects sooner) [11, 18, 23, 36, 39], test selection (identify a subset of the input test suite to run during regression testing) [5, 14, 16, 27, 28, 50], and test parallelization (schedule the input tests for execution across multiple CPUs) [24, 26]. 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 yield the same results as they do when executed in the default order.

To evaluate whether and how test dependence can impact the downstream testing techniques, we implemented 4 test prioritization, 6 test selection, and 2 test parallelization algorithms, and evaluated each algorithm on 5 real-world subject programs that contain dependent tests. We measured the number of tests that yield different results before and after applying a testing technique. Our empirical results indicate that all 12 downstream testing algorithms are affected by test dependence in a non-ignorable manner. For example, an automatically-generated test suite for XML-Security contains 665 tests, and 111 of those tests yield a different test result (success vs. fail) if the suite is parallelized to run on 16 CPUs (Figure 9).

Our findings provide empirical evidence that test dependence should no longer be ignored in designing a downstream testing algorithm that may change the original test execution order.

1.2 Coping with the Impact

Downstream testing techniques should respect existing test dependences and keep the test results consistent.

In this paper, we propose a general approach to enhance existing testing algorithms so that they will become dependence-aware. The enhanced algorithms take as input a test suite and its test dependences, possibly detected by a tool like DTDetector [52]. When creating a new test suite, the enhanced testing algorithm ensures that for each dependent test in a test suite, all other tests it depends on will be executed before it. The key idea of the enhanced algorithm is to use Delta Debugging [48] to minimize the test set that causes a dependent test to yield a different result in the new test suite from in the original test suite, and then modify the new test suite to make the exhibited dependent test yield the same result as in the original suite.

We enhanced each of the 12 testing algorithms to form a dependence-aware version. We evaluated each dependence-aware testing algorithm on real-world programs with dependent tests. All enhanced algorithms produce consistent and correct results without substantially compromising the original testing technique's effectiveness. In the case of test prioritization, 4 out of the 8 results we collected was noticeably affected by our enhanced algorithms. For test selection, our enhanced algorithm increased the size of the selected subset by 2% or less for all 31 results. Our enhanced test parallelization results ranged from having no noticeable differences to up to 4 times slower than the original test parallelization results. However even with the compromises, all three enhanced algorithms are still significantly more effective than running the original test suite. In summary, our results indicate that the enhanced algorithms do compromise the original testing technique's effectiveness but they produce consistent and correct results while the benefits of applying the testing techniques are still significant.

1.3 Contributions

This paper makes the following contributions.

- **Assess the Impact of Test Dependence.** We present an empirical study to assess the impact of test dependence on 4 test prioritization, 6 test selection, and 2 test parallelization algorithms. We evaluated these testing algorithms on 5 subject programs with a real-world test suite containing dependent tests. The results indicate that all of the downstream testing algorithms we studied are affected by test dependence. This also suggests that test dependence should no longer be overlooked in designing testing algorithms (Section 2).
- **Cope with Test Dependence.** We propose an approach to cope with test dependence. Our approach enhances existing test prioritization, selection, and parallelization algorithms to make them respect test dependence (Section 3). We evaluated the enhanced algorithms on 5 subject programs containing test dependence. The enhanced testing algorithms are aware of test dependence, and the suites they produce yield the same results as the default order without completely compromising their effectiveness (Section 3.4).

2. EVALUATING THE IMPACT

This section describes our empirical evaluation of the impact of test dependence on three well-known downstream testing techniques. We present background for each downstream testing technique (Section 2.1), describe the evaluation methodology (Section 2.3), and present the results (Section 2.4).

2.1 Downstream Testing Techniques

Test prioritization, selection, and parallelization are three important downstream testing techniques. These three techniques change the execution order of an existing test suite to make it detect faults earlier or complete faster.

2.1.1 Test prioritization

Test prioritization schedules test cases for execution in an order that attempts to produce useful results more quickly — notably test failures that indicate a code defect, but also test successes to yield confidence in the software. When no test fails, the entire test suite will be executed, so test prioritization does not reduce the overall cost of test execution.

Label	Algorithm Description
T3	Prioritize on coverage of statements
T4	Prioritize on coverage of statements not yet covered
T5	Prioritize on coverage of functions
T7	Prioritize on coverage of functions not yet covered

Figure 1: Four test prioritization algorithms used to assess the impact of dependent tests. We use the same labels as in Table 1 of [11].

Test prioritization algorithms developed in the literature fall into three major categories: (1) algorithms that order test cases based on their total coverage of code components; (2) algorithms that order test cases based on their coverage of code components not previously covered; (3) algorithms that order test cases based on their estimated ability to reveal faults in the code components that they cover. In addition to the test execution information, the third category requires a comprehensive history of known faults, which is often absent in practice.

This paper evaluates 4 well-known test prioritization algorithms from the first and second categories (Figure 1). We did not implement algorithms from the third category, including the other 10 test prioritization algorithms introduced in [11], since they require a fault history that is not available for our subject programs. The 4 evaluated test prioritization algorithms monitor the execution of each test, record the exercised statements or functions at runtime, and then uses the recorded runtime information to reorder the test execution. Two techniques (T3 and T5 in Figure 1) prefer to execute tests that cover more statements or functions first, and the other two techniques (T4 and T7 in Figure 1) prefer to execute tests that cover more *uncovered* statements or functions first.

2.1.2 Test selection

Regression test selection (for short, test selection) algorithms create a smaller test suite that contains only a subset of the original test cases in a program. The smaller test suite runs faster. Test selection aims to reduce the worst-case execution time of test execution without reducing the effectiveness of the test suite.

Test selection algorithms developed in the literature fall into two major categories: (1) algorithms that employ program analysis to select tests based on the coverage of modified program fragments [5, 14]; and (2) algorithms that select tests based on the historical data *without* analyzing the tested program [27]. The second category requires a comprehensive set of historical testing activities which are often absent in practice.

This paper evaluates 6 test selection algorithms developed in [14] (Figure 2) in the first category. All of the algorithms use program analysis to select every test case in a test suite that may be affected in the modified software. They share the same idea based on comparisons of program control-flow graphs (CFGs) but work at a different granularity level and how the tests are ordered once they are selected. The granularity levels are statement-level and function-level. The statement-level selection algorithms are more precise and selects fewer tests than the function-level selection algorithms, but is more expensive. Likewise, depending on the way tests are ordered by, the algorithm may achieve higher test coverage faster but it will be more expensive. Ordering by number of uncovered elements tests cover will be the fastest way to achieve high test coverage, followed by number of elements tests cover and then by test id. Ordering by test id essentially does no reordering while the other two orderings is a combination of test selection followed by test prioritization. Given a program P, each algorithm executes regression tests to build an edge-coverage matrix which maps each test to the set of CFG edges

Label	Granularity	Ordered by
S1	Statement	Test id
S2	Statement	Number of elements tests cover
S3	Statement	Number of uncovered elements tests cover
S4	Function	Test id
S5	Function	Number of elements tests cover
S6	Function	Number of uncovered elements tests cover

Figure 2: Six test selection algorithms used to assess the impact of dependent tests. The algorithms select a test if it covers new, deleted, or modified coverage elements. Each algorithm's coverage element is listed under column "Granularity". The selected tests are then ordered by the method described under column "Ordered by". These algorithms are introduced in [14]. These two algorithms form the basis of many other popular test selection algorithms [5,27,28,34,50].

Label	Algorithm Description
P1	Parallelize on test id
P2	Parallelize on test execution time

Figure 3: Two test parallelization algorithms used to assess the impact of dependent tests. These algorithms are supported in industrial-strength tools [43].

Program	LOC	Tests	Auto Tests	Revision
Crystal	88010	78	3198	version 1.0.20111015
JFreechart	418057	2234	2438	version 1.0.15
Joda-Time	351745	3875	2234	b609d7d66d
Synoptic	160662	118	2467	d5ea6fb3157e
XML Security	66248	108	665	version 1.0.4

Figure 4: Subject programs used in our evaluation. Column "LOC" represents the number of lines the subject program's source, manual and automated tests contains. Column "Tests" shows the number of human-written unit tests. Column "Auto Tests" shows the number of unit tests generated by Randoop [29].

Program	Dependent Tests	Auto Tests
Crystal	18 (23.1%)	164 (5.1%)
JFreechart	8 (0.4%)	6 (0.2%)
Joda-Time	6 (0.2%)	257 (9.7%)
Synoptic	1 (0.8%)	10 (0.4%)
XML Security	4 (3.4%)	171 (25.8%)

Figure 5: Number of dependent tests found in our subject programs. Column "Dependent Tests" represents a lower bound amount of dependent tests in the program's human-written test suite. Column "Dependent Auto Tests" represents a lower bound amount of dependent tests in the program's automatically-generated test suite. The percentage within the columns represents the percent of tests exposed to be dependent within the test suite.

exercised by the test. For a subsequent modified program version P', the CFGs of P and P' are compared to identify "dangerous" edges in the CFG for P. These edges represent program points at which P and P' differ. All and only test cases for P which cover dangerous edges are selected for testing of P'.

2.1.3 Test parallelization

Test execution parallelization (for short, test parallelization) schedules the input tests for execution across multiple CPUs to reduce the test latency — the time it takes to run all of the tests.

Test parallelization techniques are widely adopted in industry. For example, Visual Studio 2010 (and later) supports a model of executing tests in parallel on a multi-CPU/core machine [43]. Popular open-source frameworks such as JUnit [30], TestNG [32], and Maven [31] provides annotations to execute a test suite in parallel. In general, existing approaches for test parallelization fall into three categories: (1) parallelize a test suite based on user annotations, (2) parallelize a test suite based on test id (or randomly parallelize a test suite), and (3) parallelize a test suite based on the execution time. The first category requires substantial manual effort, while the second and third categories are fully automated.

This paper evaluates 2 test parallelization algorithms (Figure 3) belonging to the second and third categories. We chose them because both algorithms are fully automated. The first algorithm parallelizes a test suite purely based on its id. Given the i-th test, the algorithm simply schedules its execution on the $i \bmod n$ (n is the number of available machines/CPUs) machines/CPUs. The second algorithm (P1 in Figure 3) parallelizes a test suite based on the execution time of each test takes. Given a test, it schedules its execution on the machine/CPU that completes earliest based on the current load and thus minimizes the execution time of all tests.

2.2 Subject Programs

Figure 4 lists the subject programs used in our evaluation.

Crystal [8] is a tool that pro-actively examines developers' code and identifies textual, compilation, and behavioral conflicts. JFreechart is a chart library [17]. Joda-Time [19] is an open source date and time library. It is a mature project that has been under active development for ten years. Synoptic [41] mines a finite state machine model representation of a system from logs. XML Security [47] is a component library implementing XML signature and encryption standards. It has been used widely as a subject program in the software testing community.

Each of them includes a well-written unit test suite All of the subject programs' test suites are designed to be executed in a single JVM, rather than requiring separate processes per test case [3].

Given the increasing importance of automated test generation tools [9, 12, 29, 53], we also want to investigate dependent tests in automatically-generated test suites. For each subject program, we used Randoop [29], 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 Figure 4.

Figure 5 lists the number of dependent tests found in our subject programs. The number of dependent tests were exposed by the tool DTDetector and copied from Table 4 in [52]. Since JFreechart's number of dependent tests were not in Table 4 of [52], we collected JFreechart's results ourselves by utilizing the DTDetector. Since the general form of the dependent test detection problem is NP-complete, the DTDetector we used was sound but incomplete. This means that every dependent test it found is real, but it is not guaranteed to find every dependent test. Therefore the number reported in Figure 5 is a lower bound of a test suite's number of dependent tests.

2.3 Methodology

We implemented the testing algorithms listed in Figures 1, 2, and 3, and evaluated each algorithm on the subject programs in Figure 4.

For each subject program, we first executed its test suite in the *default* order and recorded the execution result of each test. We adopt the results of the default order of execution of a test suite as

Program	Revision	Lines Different
Crystal	trunk-2013-12-12	2730 (3.1%)
JFreechart	version 1.0.16	2894 (0.7%)
Joda-Time	d6791cb5f9	165368 (47.0%)
Synoptic	ea407ba0a750	388 (0.2%)
XML Security	version 1.2.0	36900 (55.7%)

Figure 6: Changes to the subject programs between two versions used in our test selection evaluation. Column "Revision" represents a revision of the program that was created later in date than the revision of the program in Figure 4. Column "Lines Different" shows the number of lines added, deleted and edited in the program's source code from its Figure 4 revision. The percentage in this column represents the percent of lines different out of the total number of lines this subject program has.

the expected results; these are the results that a developer sees when running the suite.

A **test prioritization** algorithm outputs a reordered test suite. We executed the reordered test suite and counted the number of tests that yielded different results in the prioritized order compared to the expected results from the default order.

A **test selection** algorithm identifies a subset of the input test suite to run during regression testing, based on changes to the program's source code. In order to obtain the subset of the test suite that should run, we supplied the test selection algorithm with the changes between the revision in Figure 4 and the next stable revision in Figure 6. We executed the selected subset of tests and counted the number of tests that yielded different results compared to the expected results from the default order. Even though we only studied one specific change for each subject program, we selected revisions where the number of lines different ranged from 0.2%-55.7%.

A **test parallelization** algorithm divides the input test suite into multiple subsets, and schedules each subset for execution on a different processor. We ran each subset in a separate JVM to simulate separate processors. We counted the number of tests that yielded different results compared to the expected results from the default order. We parameterized each parallelization algorithm by the number of available machines: k, and we evaluated each algorithm with $k = 2, 4, 8$, and 16.

2.4 Results

This section shows the evaluation results.

2.4.1 Impact on Test Prioritization

The dependent tests in our subject programs interfere with all of the test prioritization algorithms in Figure 1. This is because all of these algorithms 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, caused undesired output. Furthermore, test prioritization algorithms usually do not take the potential test dependence into consideration when reordering the test suite.

The prioritization algorithm that affected the most amount of tests' execution result is T5, prioritization on coverage of functions. This prioritization algorithm made 388 tests yield different execution results than they did when these tests were executed in its default order. The test prioritization algorithm that affected the least of amount of tests is T4, prioritization on coverage of statements not yet covered. In comparison to T5, this algorithm exposed 48 tests less, resulting in a total of 340 tests. The remaining prioritization algorithms, prioritization on statements (T3) and prioritization on functions not yet covered (T7) affected 377 and 363 tests, respectively.

Subject Program	T3	T4	T5	T7
Human-written Test Suites				
Crystal	2	5	2	5
Joda-Time	0	1	0	0
JFreechart	3	3	3	0
Synoptic	0	0	0	0
XML Security	0	0	2	1
Total	5	9	7	6
Automatically-generated Test Suites				
Crystal	59	71	67	64
JFreechart	5	5	6	5
Joda-Time	224	193	224	226
Synoptic	2	3	3	3
XML Security	82	59	81	59
Total	372	331	381	357

Figure 7: Dependent tests that are exposed by 4 test prioritization algorithms. Each cell shows the number of dependent tests that yield different results in the prioritized suite as they do when executed in the default, unprioritized order.

Subject Program	S1	S2	S3	S4	S5	S6
Human-written Test Suites						
Crystal	1	1	1	1	1	1
JFreechart	0	0	0	0	0	0
Joda-Time	0	0	0	0	0	1
Synoptic	0	0	0	0	0	0
XML Security	0	0	0	0	0	0
Total	1	1	1	1	1	2
Automatically-generated Test Suites						
Crystal	17	25	26	21	35	39
JFreechart	0	0	0	3	3	3
Joda-Time	54	54	54	218	230	225
Synoptic	0	0	0	0	0	0
XML Security	24	30	29	24	26	28
Total	95	109	109	266	294	297

Figure 8: Dependent tests that are exposed by 6 test selection algorithms. Each cell shows the number of dependent tests that do not yield the same results as they do when executed in the original test suite.

Although not one prioritization algorithm exposed dependent tests in every subject program's human-written test suite, dependent tests were exposed by each algorithm in at least 2 of the 5 subject programs. In addition, all prioritization algorithms exposed dependent tests in every subject program's automatically-generated test suite. The evaluation results suggest that test dependence has a high chance of affecting test prioritization results particularly on automatically-generated test suites.

2.4.2 Impact on Test Selection

The dependent tests in our subject programs interfere with all of the test selection algorithms in Figure 2. Similar to the test prioritization algorithms in Figure 1, these algorithms implicitly assume that there are no test dependences in the input test suite. Even though the test selection algorithms we implemented may not necessarily reorder a test suite (S1 and S4), we exposed dependent

Subject Program	P1 (Original Order)				P2 (Time-Minimized)			
	k=2	k=4	k=8	k=16	k=2	k=4	k=8	k=16
Human-written Test Suites								
Crystal	2	2	2	8	9	9	9	9
JFreechart	0	0	0	1	2	2	2	0
Joda-Time	0	0	0	0	2	3	1	2
Synoptic	0	0	0	0	0	0	0	0
XML Security	0	0	4	4	1	1	2	3
Total	2	2	6	13	14	15	14	14
Automatically-generated Test Suites								
Crystal	0	0	1	4	72	76	74	81
JFreechart	1	3	3	3	3	3	3	3
Joda-Time	126	135	184	229	223	228	224	233
Synoptic	1	1	2	2	2	2	2	2
XML Security	66	78	104	111	60	91	110	114
Total	194	217	294	349	360	400	413	433

Figure 9: Dependent tests that are exposed by 2 test parallelization algorithms. Each cell shows the number of dependent tests that do not yield the same results as they do when executed in the original test suite.

tests when our algorithms selected them but not the tests they depend on.

The selection algorithm that affected the most amount of tests' execution result is S6, selecting tests covering new, deleted or modified functions ordered by the number of functions not yet covered. This selection algorithm made 299 tests yield different execution results than they did when these tests were executed in its default order. In comparison to S1, selecting tests covering new, deleted or modified statements ordered by test id, this algorithm exposed 203 tests less, resulting in a total of 96 tests.

Although all selection algorithms only exposed dependent tests in one subject program's human-written test suite, all algorithms exposed dependent tests in at least 3 of the 5 subject program's automatically-generated test suites. The evaluation results suggest that test dependence has a minimal chance of affecting test selection results on human-written test suites and a moderate chance of affecting automatically-generated test suites.

2.4.3 *Impact on Test Parallelization*

The dependent tests in our subject programs interfere with all of the test parallelization algorithms in Figure 3. Similar to the other two downstream testing techniques we studied, test parallelization implicitly assumes that there are no test dependences in the input test suite. When test parallelization is applied to a test suite, tests may not only be reordered but they may no longer be executed on the same machine anymore. This exposes dependent tests because our algorithms may schedule a dependent test on a particular machine but the test it depends on another machine.

The parallelization algorithm that affected the most amount of tests' execution result is P2, parallelize on test execution time with 16 machines. This parallelization algorithm made 447 tests yield a different execution result than it did when these tests were executed in its default order on one machine. In comparison to P2, the remaining parallelization algorithm, P1, parallelize on test id at most exposed 85 tests less with 16 machines, resulting in a total of 362 tests.

Although P1 only exposed dependent tests in at most 2 of the 5 subject program's human-written test suite, P2 consistently exposed dependent tests in 4 of the 5. Additionally, P1 exposed dependent tests in at least 4 of the 5 subject program's automatically-generated test suite while P2 consistently exposed dependent tests in 5 of the 5. The evaluation results suggest that test dependence has a high chance of affecting test parallelization results particularly on automatically-generated test suites.

2.5 Discussion

2.5.1 *How Significant is the Impact?*

As shown in Section 2, all of the testing techniques exposed some dependent tests in one or more of the subject programs we studied. Synoptic did not reveal any dependent tests for its human-written test suite in all of the algorithms we studied because only 1 (0.8%) of its tests were dependent. For the other programs, our results indicated that the impact of test dependence on downstream testing techniques ranged from minimal to high.

2.5.2 *Root Causes of Test Dependence*

Essentially, 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.

Concurring with our previous study [52], our experiments suggested three common root causes of test dependence: (1) improper access to shared global variables (i.e., static variables in Java), (2) improper access to the file systems, and (3) improper access to other external resources (e.g., databases, networks). Improper access to shared global variables is the most common root causes, accounting for at least 61% of all dependent tests we have studied.

The downstream testing techniques we have evaluated have indirectly changed the execution environment a test may implicitly assume, such as alternating the test execution order (as test prioritization does) and selecting a subset of test to execute (as test selection and parallelization do).

2.5.3 *Threats to Validity*

There are three main reasons why our findings apply in the context of our study and methodology and may not apply to arbitrary programs. First, the applications we studied are all written in Java and have JUnit test suites. Second, the 5 open-source programs and their test suites may not be representative enough. Third, in our study, we only evaluated 4 test prioritization, 6 test selection, and 2 test parallelization algorithms. Although these reasons threaten the validity of the results presented, we do believe that if we evaluated test dependence on other testing techniques and subject programs, we will continue to find more conclusive proof that these techniques should not ignore test dependence.

2.5.4 *Experimental Conclusion*

We have two chief findings. **(1)** Dependent tests in both human-written and automatically-generated test suites can affect the results of test prioritization, test selection, and test parallelization algorithms. **(2)** The degree of impact varies across different testing techniques. In general automatically-generated test suites across all techniques seem to be significantly affected by test dependence. For human written test suites, test prioritization and test parallelization seems to have a higher chance of being affected by test dependence while test selection has a lower chance.

3. COPING WITH THE IMPACT

Auxiliary routines:
exec(T): executes a test suite T (i.e., an ordered sequence of tests) in a fresh environment and returns a tuple containing the results of all tests.
getFirstDifferentTest(T_{new}, T_{orig}): returns the first test $t \in T_{new} \cap T_{orig}$ that yields different results in T_{orig} and T_{new}, or null if no such test exists. Uses the ordering of T_{new} to determine which test is first.
ddmin(T, t, r): minimizes the test suite T while maintaining that a given test t yields result r. t is the last test in the result suite. Requires $t \in T$ and $\text{exec}(T)[t] = r$.
reorderTests(T, $T_{ordering}$): returns T, reordered with tests in the same order as they were in $T_{ordering}$. Requires $T \subseteq T_{ordering}$.
merge(T_1, T_2, T_{ref}): merges T_1 and T_2 based on the test ordering in T_{ref}. Retains the relative test order of T_1 and T_2.

nullifyTestDependence(T_{new}, T_{orig})
Input: T_{new} is a possibly reordered subsequence of T_{orig}, the original test suite. Without loss of generality, we assume that every test in T_{orig} passes.
Output: an ordered subsequence of T_{orig} that includes every test in T_{new}, and every test in it yields the same result as in T_{orig}.

```
1  $R_{orig} \leftarrow \text{exec}(T_{orig})$
2  while $\exists t \in T_{new} : \text{exec}(T_{new})[t] \neq R_{orig}[t]$ do
3      $t \leftarrow \text{getFirstDifferentTest}(T_{new}, T_{orig})$
4      $r_{t\_isolated} \leftarrow \text{exec}([t])[t]$
5      $T_{t\_isolated\_diff} \leftarrow$ (if $r_{t\_isolated} == PASS$ then $T_{new}$ else $T_{orig}$)
6      $T_{t\_isolated\_same} \leftarrow$ (if $r_{t\_isolated} == PASS$ then $T_{orig}$ else $T_{new}$)
7      $T_{min} \leftarrow \text{ddmin}(T_{t\_isolated\_diff}, t, r_{t\_isolated})$
8      $T \leftarrow \text{reorderTests}(T_{min}, T_{orig})$
9      $r_{t\_min} \leftarrow \text{exec}(T)[t]$
10     if $r_{t\_min} \neq PASS$ then
11         $T_{min} \leftarrow \text{ddmin}(T_{t\_isolated\_same}, t, r_{t\_min})$
12         $T \leftarrow \text{reorderTests}(T_{min}, T_{orig})$
13     end if
14     $T_{new} \leftarrow \text{merge}(T, T_{new} \setminus T, T_{new})$
15 end while
16 return $T_{new}$
```

Figure 10: A general approach to remove test dependence. This algorithm is instantiated in Figures 11, 12, and 13 to cope with test dependence in test prioritization, test selection, and test parallelization, respectively.

Downstream testing techniques should be cognizant of test dependence, accounting for it to keep the test execution results consistent. 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 — even if the input suite contains dependent tests. A prioritization or selection algorithm that assumes its input contains no dependent tests can introduce test failures, which violates the design requirement.

To cope with the impact of test dependence on existing test prioritization, selection, and parallelization algorithms, this section presents a general approach and shows how to enhance existing algorithms to ensure they produce an order in which each test yields the same result as it did in the original test suite.

3.1 Enhancing downstream testing techniques

3.1.1 General Approach

Figure 10 gives a general approach that "nullifies" the impact of test dependence in a reordered subsequence of the original test suite. Given a sequence of tests, this algorithm converts it into a different, possibly longer sequence of tests. The resulting sequence includes every test in the input sequence, and each test in it yields the same result as in the original test suite.

The basic idea of this algorithm is to use Delta Debugging [48] to minimize the test set that causes a dependent test to yield different result (line 7), and then modify the test sequence (by adding necessary tests or reordering it) to make the exhibited dependent test yield the same result as in the original suite (line 8). Each iteration of the loop in the algorithm nullifies one dependent test but the minimized test suite from delta debugging may introduce other dependent tests. These other dependent tests are nullified by eventually becoming t (line 3) in subsequent loop iterations.

We identified two causes for test dependence. (1) A dependent test is reordered so that the tests it depends on are no longer executing before it. (2) A dependent test is created because tests that did not execute before the dependent test in the original test suite is now reordered to execute before it. Line 4 executes t by itself to determine whether t passed in the original test suite because some tests it depended on was executing before it or if t naturally passes. Our use of delta debugging is dependent on the execution result from line 4. If we are to find the tests that are missing to nullify the test dependence (1), then the first argument passed to delta debugging (line 7) is the original test suite. If we are to reorder the tests that are causing t to attain a different result (2), then the first argument passed is the reordered subsequence of the original test suite (line 5). Lines 9 to 13 addresses the scenario when a dependent test is exposed because of both causes. By rerunning delta debugging this time with the other test order we ensure that when we merge (line 14) t will no longer be affected by either causes of test dependence.

It is possible that the modified sequence may yield a different result when executed with the rest of the tests (line 10). In this case, the algorithm employs Delta debugging again to identify the minimal set of tests from the *original* test suite that must be executed in the output sequence (lines 10–14).

The algorithm repeats the above steps until all dependent tests yield the same results as in the original test suite (line 2).

3.1.2 Termination and Complexity

The algorithm in Figure 10 is guaranteed to terminate, and each test in the output test sequence must yield the same result as in the original test sequence. Intuitively, after each iteration in Figure 10 (lines 2–15), at least one more test in the new ordering yields the same result as in the original ordering. In the worst case, the T_{new} has the same size of T_{old} and every test in T_{new} yields a different result as in the original suite, the algorithm simply returns the original test sequence after n iterations, where n is the size of the original test suite. Therefore, the worst-case time complexity of the general algorithm is $O(n)$. For brevity, we omit the termination and complexity proofs here.

3.1.3 Enhancing Downstream Testing Algorithms

Figures 11, 12, and 13 show the enhanced dependence-aware test prioritization, selection, and parallelization algorithms. In each case, the enhanced algorithm first invokes the original downstream testing algorithm to produce a possibly-reordered subsequence of the original test suite. Then, the enhanced algorithm employs the general approach in Figure 10 to nullify the impact of potential test dependence in it and outputs a test execution order in which every test yields the same results as in the original test suite.

3.1.4 Discussion

The enhanced algorithms may reorder tests compared to the base-

Auxiliary routines:
prioritize(T): the target prioritization algorithm to enhance.

depAwareTestPrioritization(T)
Input: a test suite (i.e., an ordered sequence of tests) T
Output: a reordered sequence of T in which every test yields the same result as it does in T

```
1 $T_{prioritized} \leftarrow$ proritized($T$)
2 return nullifyTestDependence($T_{prioritized}$, $T$)
```

Figure 11: A dependence-aware test prioritization algorithm. When invoking nullifyTestDependence (Figure 10), this algorithm overrides the merge routine. See Section 3.1.3 for details.

Auxiliary routines:
select(T): the target test selection algorithm to enhance.

depAwareTestSelection(T)
Input: a test suite (i.e., an ordered sequence of tests) T
Output: a subsequence of T in which every test yields the same result as in T

```
1 $T_{selected} \leftarrow$ select($T$)
2 return nullifyTestDependence($T_{selected}$, $T$)
```

Figure 12: A dependence-aware test selection algorithm.

Auxiliary routines:
parallelize(T): the target test parallelization algorithm to enhance. It returns a set of test suites, each of which is scheduled to run a different CPU.

depAwareTestParallelization(T)
Input: a test suite (i.e., an ordered sequence of tests) T
Output: a set of test suites, each of which is a subsequence of T and every test in it yields the same result as in T

```
1 suites $\leftarrow$ parallelize($T$)
2 for each $T_{par}$ in suites do
3     yield nullifyTestDependence($T_{par}$, $T$)
4 end for
```

Figure 13: A dependence-aware test parallelization algorithm.

line algorithms. The reordered test suite has exactly the same code coverage as the suite before reordering. However, for test prioritization, the reordered suite may score lower on a metric such as time to find the first fault or APFD [35]. This is an acceptable cost, because the enhanced algorithms are correct whereas the original algorithms produce test suites in which tests spuriously fail. The charts in Figures 14 and 15 depicts the cost in coverage. From our results only 1 out of 3 human-written test suites really suffered a noticeable cost in coverage while 3 out of 5 automatically-generated test suites were noticeably affected.

The enhanced algorithms may add tests compared to the baseline algorithms. The resulting test suite achieves at least as much coverage as the original algorithms. However, the resulting suite may take longer to run and it may also score lower on other metrics that take execution time into account, such as time to first fault. Again, this is an acceptable tradeoff. Our selection algorithm guarantees that each test is ran only once since the worst case is we run the same number of tests as the original test suite. Yet as evident in Figure 16, none of our subject program's test suites needed 100% of its tests. In fact the tradeoff to nullify test dependence only cost 2% or less for all of the subject programs. On the other hand, our enhanced parallelization algorithms may execute the same test more than once but in different machines. Although the total number of tests executed across all machines may be more than the number of tests in the original test suite, the time for the machines to finish should be faster, if not, the same. From our results in Figure 17, it is evident that our enhanced algorithms are generally slower than the original ones but they are still faster than running the original test suite.

3.2 Methodology

Section 2 identified human-written and automatically-generated test suites that yield different results after applying a downstream testing technique. We applied the corresponding enhanced testing technique to the same test suite and measured these two results:

- **correctness:** whether the test dependence has been properly handled. Every test should yield the same result as it did in the original test suite in the default order.
- **effectiveness** whether the enhanced testing technique has compromised the effectiveness of the original testing technique. Specifically, for test prioritization techniques, we measure the test coverage of the prioritized test suite; for test selection techniques, we measure the size of the selected test subset; and for test parallelization techniques, we measure the execution time of the whole test suite after parallelization.

3.3 Correctness

For every test suite and every enhanced testing algorithm, every test yields the same result as it did in the original test suite in the default order. This result is as expected, because the enhanced algorithms explicitly check for and correct test dependence, ensuring that whenever a test is executed, any other tests that it depends on are executed before it. Therefore, test dependence is preserved.

This result validates both our algorithms and our implementation.

3.4 Effectiveness

3.4.1 Enhanced Test Prioritization Algorithms

We evaluate the effectiveness of the enhanced prioritized test suite by measuring its coverage. Figures 14 and 15 show the results. Our enhanced test prioritization algorithms nullifies the impact of test dependence by reordering the sequence of tests in a suite. In other words, the algorithms execute the same number of tests as the original test prioritization algorithms. Yet some of the test orders produced by the enhanced test prioritization algorithms can achieve *higher* coverage *faster* than the unenhanced test orders. This was surprising because we expected a test order to achieve higher coverage slower when we have to reorder it to preserve test dependence.

Unlike the enhanced orders, dependent tests in the unenhanced orders actually do not provide the coverage it does in the unprioritized test suite. As an example, test *a* may covers 50 statements when executed in the original test suite before test *b*. Because test *a* is dependent on test *b* running after it, if a test prioritization order has test *b* running before test *a*, then test *a* may only cover a subset of the 50 statements it is expected to cover. Common reasons for why our dependent tests achieved higher coverage slower in our unenhanced orders was because of the occurrences of events that disrupts the normal flow of instructions and the different paths of execution taken for branches.

3.4.2 Enhanced Test Selection Algorithms

We measure the size of the selected subset. Figure 16 shows the results. Our enhanced test selection algorithms nullifies the impact of test dependence by adding the necessary tests to the subsequence of tests created by the original test selection algorithms. In other words, our enhanced algorithms can increment the subsequence

Subject Program	S1	S2	S3	S4	S5	S6
Human-written Test Suites						
Crystal	$24\% \rightarrow 26\%$	$24\% \rightarrow 26\%$	$24\% \rightarrow 26\%$	$29\% \rightarrow 30\%$	$29\% \rightarrow 30.0\%$	$29\% \rightarrow 30.0\%$
JFreechart	-	-	-	-	-	-
Joda-Time	-	-	-	-	-	$84\% \rightarrow 84\%$
Synoptic	-	-	-	-	-	-
XML-Security	-	-	-	-	-	-
Automatically-generated Test Suites						
Crystal	$10\% \rightarrow 11\%$	$10\% \rightarrow 11\%$	$10\% \rightarrow 11\%$	$33\% \rightarrow 34\%$	$33\% \rightarrow 33\%$	$33\% \rightarrow 33\%$
JFreechart	$33\% \rightarrow 33\%$	$33\% \rightarrow 33\%$	$33\% \rightarrow 33\%$	$85\% \rightarrow 85\%$	$85\% \rightarrow 85\%$	$85\% \rightarrow 85\%$
Joda-Time	$13\% \rightarrow 13\%$	$13\% \rightarrow 13\%$	$13\% \rightarrow 13\%$	$78\% \rightarrow 78\%)$	$78\% \rightarrow 78\%$	$78\% \rightarrow 78\%$
Synoptic	-	-	-	-	-	-
XML Security	$45\% \rightarrow 46\%$	$45\% \rightarrow 46\%$	$45\% \rightarrow 46\%$	$52\% \rightarrow 53\%$	$52\% \rightarrow 53\%$	$52\% \rightarrow 53\%$

Figure 16: The 6 enhanced test selection algorithms may reorder the selected test suite, increase the number of selected tests (the size of the result suite) or do both. In each cell, the data before $\rightarrow$ shows the percentage of tests selected by the original selection algorithm and the data after $\rightarrow$ shows the percentage of tests selected by the enhanced, dependence-aware selection algorithm. A "-" in the cell means that no dependent tests were exposed by the program and algorithm, and thus the before and after values are identical.

of tests up to the same number of tests as the original test suite. Yet in the data we collected, our enhanced algorithms only slightly increased the selected test subset by 2% or less.

3.4.3 *Enhanced Test Parallelization Algorithms*

The major purpose of using test parallelization algorithms is to shorten the total test execution time. We measure the total test execution time by recording the time taken by the slowest machine. Figure 17 shows the speedup of each enhanced parallelization algorithm. The data shown in the figure is calculated by the time for the slowest machine to run / the time to run the original test suite. A number greater than 1 would imply that the time to run the test suite with the enhanced algorithm is greater, while the number 1 or a number less than 1 would imply the time is equal to or less than (respectively) the time to run the test suite without the application of any testing algorithms. Our enhanced test parallelization algorithms nullifies the impact of test dependence by adding the necessary tests to the subsequences of tests created by the original test parallelization algorithms. In other words, our enhanced algorithms can increment the total number of tests executed across all machines up to $\sum_{i=0}^{k-1}(n-i)$ where n is the number of tests in the original test suite and k is the number of machines. The maximum size in all of the subsequences of tests would be the same as size of the original test suite. The slowdown between the original and enhanced parallelization algorithm is a result of the increased amount of tests machines have to execute in order to preserve test dependence. To elaborate, say test b depends on test a to come before it yet they were both scheduled to be executed in different machines. In order for test b to exhibit the same behavior as it did in the unparallelized test suite, whichever machine test b is contained in will have to have test a be added to it.

Certain results appears to be the same with the original or enhanced parallelization algorithm. This is because the subject programs contains tests that relies on one another to setup UI components, files or a combination of both. The negligence of test dependence for the original parallelization order resulted in its tests waiting for a particular UI event or file system state to trigger. Most of these tests end up waiting a set amount of time before terminating. Despite how these enhanced testing algorithms may require the user to execute more tests, this example illustrates that the consequences of executing more tests may not always be detrimental to the user.

3.5 Discussion

3.5.1 *Threats to Validity*

There are several threats to the validity of our experiments. First, the subject programs and the test suites used to evaluate the enhanced testing algorithms may not be representative enough. We chose these subject programs because they contain developer-confirmed dependent tests, and some of them, such as XML-Security, are widely used to evaluate downstream testing technique in the software engineering community. Second, Section 3.4 evaluated effectiveness based on proxy measures of test suite quality rather than on detection of real faults. Our future work plans to evaluate the enhanced algorithms on more subject programs and employ other metrics to evaluate the effectiveness.

3.5.2 *Experimental Conclusions*

We have two chief findings. **(1)** All enhanced downstream testing techniques output consistent results on test suites containing dependent tests. And **(2)** The enhanced testing techniques are more effective than executing the test suites without any testing techniques.

4. RELATED WORK

We next discuss three lines of closely-related work on: (1) definitions and studies of test dependence, (2) testing techniques affected by test dependence, and (3) techniques to cope with test dependence.

4.1 Test Dependence Definitions and Studies

Treating test suites explicitly as mathematical sets of tests [15] and assuming test independence are common practice in the testing literature [5,11,14,18,20,23,27,28,36,39,40,50,51]. Nonetheless, the conditions under which a test is executed may affect its result.

Bergelson and Exman describe a form of test dependence informally: given two tests that each pass, the composite execution of these tests may still fail [4]. 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.

Subject Program	P1 (Original Order)				P2 (Time-Minimized)			
	k=2	k=4	k=8	k=16	k=2	k=4	k=8	k=16
Human-written Test Suites								
Crystal	0.72 → 0.77	0.58 → 0.58	0.53 → 0.56	0.54 → 0.56	0.74 → 0.74	0.67 → 0.67	0.67 → 0.67	0.68 → 0.68
JFreechart	-	-	-	0.46 → 0.48	0.54 → 0.98	0.47 → 0.82	0.43 → 0.53	-
Joda-Time	-	-	-	-	0.66 → 0.66	0.65 → 0.65	0.40 → 0.41	0.32 → 0.32
Synoptic	-	-	-	-	-	-	-	-
XML Security	-	-	0.12 → 0.28	0.12 → 0.27	0.53 → 0.66	0.46 → 0.69	0.15 → 0.22	0.12 → 0.19
Automatically-generated Test Suites								
Crystal	-	-	0.13 → 0.41	0.11 → 0.35	0.61 → 0.81	0.68 → 0.93	0.25 → 0.80	0.22 → 0.58
JFreechart	0.54 → 0.85	0.53 → 0.66	0.45 → 0.56	0.34 → 0.49	0.63 → 0.74	0.55 → 0.64	0.40 → 0.44	0.33 → 0.54
Joda-Time	0.54 → 0.58	0.42 → 0.44	0.28 → 0.62	0.27 → 0.33	0.52 → 0.62	0.50 → 0.53	0.23 → 0.32	0.26 → 0.28
Synoptic	0.59 → 0.70	0.24 → 0.47	0.17 → 0.17	0.13 → 0.14	0.54 → 0.75	0.25 → 0.77	0.18 → 0.57	0.14 → 0.40
XML Security	0.52 → 0.53	0.39 → 0.41	0.18 → 0.48	0.16 → 0.22	0.83 → 0.83	0.54 → 0.80	0.28 → 0.41	0.26 → 0.37

Figure 17: Results of evaluating the enhanced test parallelization algorithms. In each cell, the data before → shows the speedup (time to run parallelized / time to run the original test suite) for the original parallelization algorithm and the data after → shows the speedup for the enhanced, dependence-aware parallelization algorithm. A "-" in the cell means that no dependent tests were exposed by the program and algorithm.

In the context of databases, 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" [22]. In addition to results about test generation and test adequacy criteria for database testing [6,13,22], the effect of context in testing has also been explored in mobile applications [44]. Such test dependence definitions focus on program and database states that may not affect the actual test results as well as the downstream testing techniques.

Our previous work [52] gave a formal definition for test dependence based on test execution results. Our definition differs from related work [2, 3, 22] by considering test results rather than program and database states (which may not affect the test results). Our previous work also showed there *exist* reordering of real-world test suites that cause the same test to yield different results, but it was unclear whether downstream testing techniques such as test prioritization, selection, and parallelization give rise to the problematic test reordering in practice. This paper empirically evaluates the impact of test dependence on real-world subject programs, and it presents a general approach to cope with the impact by enhancing existing algorithm to respect test dependence and keep the test results consistent.

Some prior work [25] studied the impact of flaky tests — tests that have non-deterministic outcomes — on regression testing. However, non-determinism is different than test dependence. Test dependence does not imply non-determinism: a test may non-deterministically pass/fail without being affected by any other test, including its own previous executions. Non-determinism does not imply test dependence: a program may have no sources of non-determinism, but two of its tests can be dependent. Furthermore, a test may execute non-deterministically, but it may deterministically pass/fail.

4.2 Testing Techniques Affected by Test Dependence

The assumption of test independence lies at the heart of most techniques for automated regression test selection [5, 14, 27, 28, 50], test suite minimization [16, 45], test case prioritization [11, 18, 23, 36, 39], coverage-based fault localization [20, 40, 51], and test generation [29, 44, 53], etc. Section 2 evaluated the impact of test dependence on test prioritization, test selection, and test parallelization. Here we discuss the impacts of test dependence on two additional techniques.

Coverage-based fault localization techniques [20] 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. Steimann et al. found that the accuracy of coverage-based fault locators is reduced when tests fail due to the violation of the test independence assumption [40]. Compared to our work, Steimann et al. focused on identifying possible threats to validity in evaluating coverage-based fault locators, and do not present any results of the impact of test dependence on test selection, prioritization, and parallelization techniques.

Most state-of-the-art test generation techniques [29, 44, 53] do not take test dependence into consideration. For example, the execution outcomes of tests generated by Randoop can depend on the execution of other tests [33, 52]. Such test dependences arise because current 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. Randoop has a mechanism to preserve test dependence: when it discovers that a test execution is nondeterministic, it disables the test's assertions but continues to execute it in the suite, so that other tests that are dependent on it do not begin to fail [33]. Exploring how to incorporate test dependence into the design of an automated test generator is future work.

4.3 Techniques to Cope with Test Dependence

Only a few techniques and tools have been developed to prevent or alleviate the impact of test dependence. Some 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). Release 4.11 of JUnit supports executing tests in lexicographic order by test method name [21]. DepUnit [10] 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 [42] 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.

However, no tool exists to ensure that programmers use these

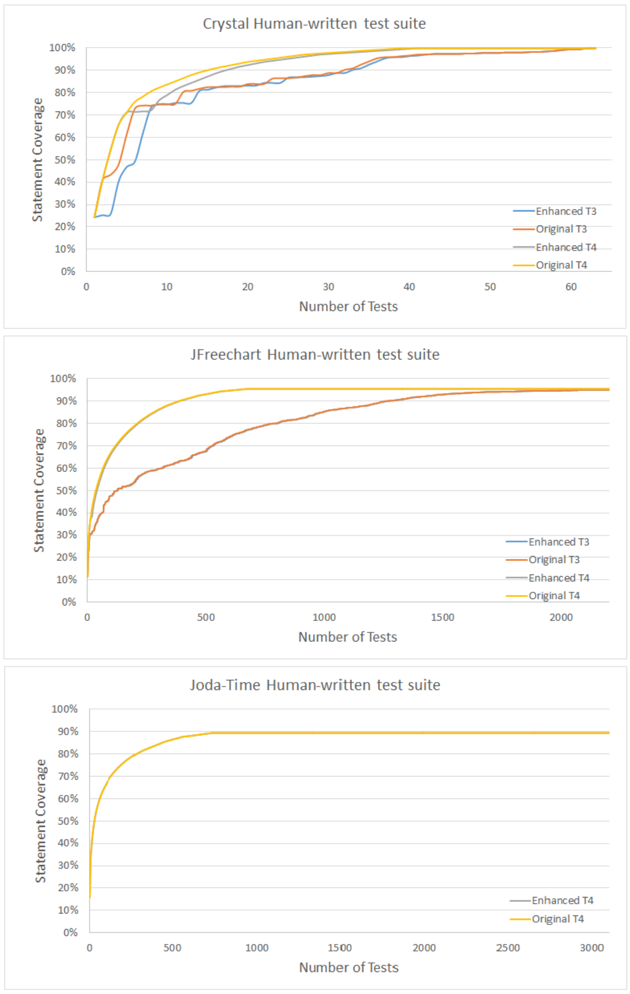

Figure 14: Reduced efficiency of the enhanced versions of test prioritization techniques T3 and T4 from Figure 1 on human-written test suites. The subject programs Synoptic and XML-Security are omitted, as is T3 for Joda-Time, because those combinations do not cause test failures due to dependent tests.

advanced mechanisms properly, and often they do not [25]. By contrast, this paper proposed techniques to cope with the impact of dependent tests. Our techniques could co-exist with such frameworks by generating annotations rather than dictating a single order of test execution.

More recently, Bell and Kaiser [3] developed an approach, called Unit Test Virtualization, to speed up unit test execution. Unit Test Virtualization dynamically tracks the memory access of each unit test, identifies code segments that may create side-effects, and executes them in an isolated container (similar to a lightweight virtual machine) with significantly lower cost than executing in a normal JVM. Similar to Kapfhammer and Soffa's work [22], Unit Test Virtualization detects a test dependence if a test accesses a memory location that has been written by another test, which is neither necessary nor sufficient to affect the test result; by contrast, our definition focuses on the test result. Because the test suite is executed on multiple virtual machines, it is unclear how to integrate Unit Test Virtualization with downstream testing techniques. By contrast, our work enhances existing testing techniques to make them respect test dependence and produce test orderings that produce consistent results when running on a single, standard JVM.

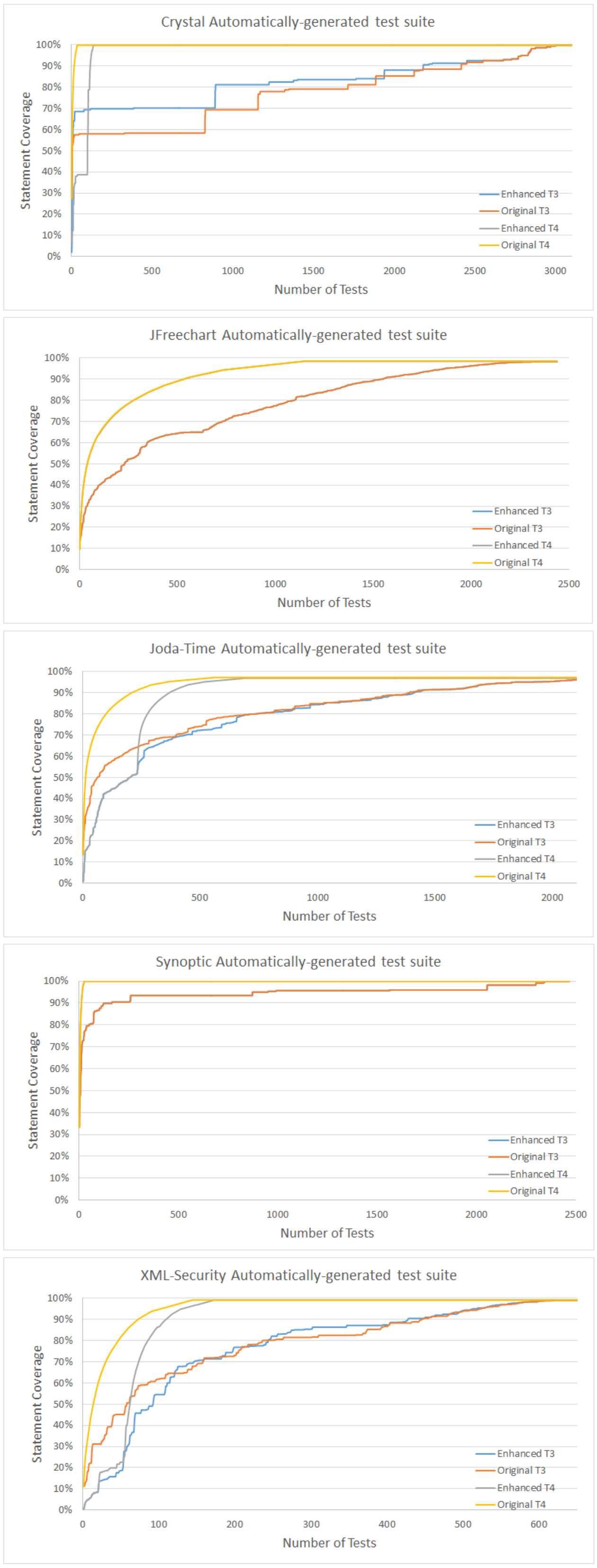

Figure 15: Reduced efficiency of the enhanced versions of test prioritization techniques T3 and T4 from Figure 1 on automatically-generated test suites.

5. CONCLUSION AND FUTURE WORK

Test dependence often arises in a test suite, but its impact is unclear and has largely been ignored in previous software testing research. This paper addresses this important issue by empirically investigating the impact of test dependence on 12 downstream testing algorithms. Our experimental results show that test dependence *does* affect the results of well-known test prioritization, test selection, and test parallelization algorithms *negatively* and *frequently*. This paper also proposes an approach to cope with test dependence. The proposed approach enhances existing test prioritization, selection, and parallelization algorithms to make them respect test dependence

Our findings are useful to practitioners and researchers. Both can learn that the impact of test dependence should no longer be ignored when deploying and designing new testing techniques. Practitioners can adjust their practice based on test patterns most often lead to test dependence, and they can use our approach to cope with the impact the test dependence. Researchers are posed interesting new problems, such as how to adapt existing testing methodologies in the presence of test dependence.

Future work should address the following issues:

Other downstream testing techniques. It would be interesting to measure the impact of dependent tests on other downstream testing techniques, such as mutation testing [38, 49, 50], test factoring [37, 46], and experimental debugging techniques [40, 48, 51]. We are also interested in enhancing these testing algorithms to respect test dependence.

Eliminating dependent tests. Another way to cope with the impact of test dependence is developing algorithms to eliminate dependent tests before their impact arises. However, the practice of eliminating dependent tests remains mostly manual and ad hoc — software developers usually manually hard-code test execution orders in a configuration file or simply merge or remove tests. A more flexible and robust methodology for dependent test elimination should be developed. This question also applies to automated test generators, and some work has been developed to alleviate this problem [3, 12, 33].

6. REFERENCES

[1] Invalid Thread Access Error Caused by Test Dependence. https://bugs.eclipse.org/bugs/show_bug.cgi?id=43500.

[2] J. Bell. Detecting, isolating, and enforcing dependencies among and within test cases. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 799–802, 2014.

[3] J. Bell and G. Kaiser. Unit test virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 550–561, 2014.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Crystal VC. http://crystalvc.googlecode.com.

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

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

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

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

[13] 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.

[14] 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.

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

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

[17] JFreechart. http://www.jfree.org/jfreechart/.

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

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

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

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

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

[23] 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.

[24] 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, 2013.

[25] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, 2014.

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

[27] 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.

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

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

[30] Parallel test execution api in junit. http://junit.org/apidocs/org/junit/experimental/ParallelComputer.html.

[31] Fork options and parallel test execution in maven. http://maven.apache.org/surefire/maven-surefire-plugin/examples/fork-options-and-parallel-execution.html.

[32] Parallel execution of test methods in testng. http://seleniumeasy.com/testng-tutorials/parallel-execution-of-test-methods-in-testng.

[33] 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.

[34] 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.

[35] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test case prioritization: an empirical study. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 179–188, 1999.

[36] 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.

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

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

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

[40] 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.

[41] Synoptic. http://synoptic.sourceforge.net/.

[42] TestNG. http://testng.org/.

[43] Executing Unit Tests in parallel on a multi-CPU/core machine in Visual Studio. http://blogs.msdn.com/b/vstsqualitytools/archive/2009/12/01/executing-unit-tests-in-parallel-on-a-multi-cpu-core-machine.aspx.

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

[45] 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.

[46] 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.

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

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

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

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

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

[52] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 385–396, San Jose, CA, USA, July 23–25, 2014.

[53] 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.