Dependent-Test-Aware Regression Testing Techniques

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

August Shi
University of Illinois
Urbana, Illinois, USA
awshi2@illinois.edu

Reed Oei
University of Illinois
Urbana, Illinois, USA
reedoei2@illinois.edu

Sai Zhang
Google*
Kirkland, Washington, USA
saizhang@google.com

Michael D. Ernst
University of Washington
Seattle, Washington, USA
mernst@cs.washington.edu

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

ABSTRACT

Developers typically rely on regression testing techniques to ensure that their changes do not break existing functionality. Unfortunately, regression testing techniques suffer from flaky tests, which can both pass and fail when run multiple times on the same version of code and tests. One prominent type of flaky tests is order-dependent (OD) tests, which are tests that pass when run in one order but fail when run in another order. Although OD tests may cause flaky-test failures, OD tests can help developers run their tests faster by allowing them to share resources. We propose to make regression testing techniques dependent-test-aware to reduce flaky-test failures.

To understand the necessity of dependent-test-aware regression testing techniques, we conduct the first study on the impact of OD tests on three regression testing techniques: test prioritization, test selection, and test parallelization. In particular, we implement 4 test prioritization, 6 test selection, and 2 test parallelization algorithms, and we evaluate them on 11 Java modules with OD tests. When we run the orders produced by the traditional, dependent-test-unaware regression testing algorithms, 82% of human-written test suites and 100% of automatically-generated test suites with OD tests have at least one flaky-test failure.

We develop a general approach for enhancing regression testing algorithms to make them dependent-test-aware, and we apply our approach to enhance 12 algorithms. Compared to traditional, unenhanced regression testing algorithms, the enhanced algorithms use provided test dependencies to produce orders with different permutations or extra tests. Our evaluation shows that, in comparison to the orders produced by unenhanced algorithms, the orders produced by enhanced algorithms (1) have overall 80% fewer flaky-test failures due to OD tests, and (2) may add extra tests but run only 1% slower on average. Our results suggest that enhancing regression testing algorithms to be dependent-test-aware can substantially reduce flaky-test failures with only a minor slowdown to run the tests.

* Most of Sai Zhang's work was done when he was at the University of Washington.

ISSTA '20, July 18–22, 2020, Los Angeles/Virtual, CA, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397364

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

flaky test, regression testing, order-dependent test

ACM Reference Format:
Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. 2020. Dependent-Test-Aware Regression Testing Techniques. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Los Angeles/Virtual, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3395363.3397364

1 INTRODUCTION

Developers rely on *regression testing*, the practice of running tests after every change, to check that the changes do not break existing functionalities. Researchers have proposed a variety of regression testing techniques to improve regression testing. These regression testing techniques produce an order (a permutation of a subset of tests in the test suite) in which to run tests. Examples of such traditional regression testing techniques include test prioritization (run all tests in a different order with the goal of finding failures sooner) [37, 40, 44, 55, 56, 61], test selection (run only a subset of tests whose outcome can change due to the code changes) [20, 32, 35, 50, 51, 69], and test parallelization (schedule tests to run across multiple machines) [38, 41, 49, 62].

Unfortunately, regression testing techniques suffer from *flaky tests*, which are tests that can both pass and fail when run multiple times on the same version of code and tests [42, 46–48]. Flaky-test failures mislead developers into thinking that their changes break existing functionalities, wasting the developers' productivity as they search for non-existent faults in their changes. In fact, Herzig et al. [33] reported that test result inspections, which include verifying whether a test failure is a flaky-test failure, can cost about $7.2 million per year for products such as Microsoft Dynamics. Flaky-test failures can also lead developers to start ignoring test failures during builds. Specifically, a recent study [63] has reported how developers ignore flaky-test failures. Another study [53] found that when developers ignored flaky-test failures during a build, the deployed build experienced more crashes than builds that did not contain flaky-test failures. Harman and O'Hearn [31] have even suggested that all tests should be assumed flaky and called for more attention on improving regression testing techniques to reduce flaky-test failures.

One prominent type of flaky tests is *order-dependent (OD) tests* [43, 46, 71]. An OD test is a test that passes or fails depending only on the order in which the test is run. Typically, the order in which OD tests pass is the order in which developers prefer to run the tests; we refer to this order as the *original order*. Running the tests in orders other than the original order may cause flaky-test failures from OD tests, which we refer to as *OD-test failures*.

Although OD tests may cause OD-test failures, OD tests can help developers run their tests faster by allowing the tests to share resources [6]. For example, one test may create a database that another test uses, allowing the latter test to run faster as an OD test instead of recreating the database. Every popular Java testing framework already permits developers to specify dependencies among tests, including JUnit [2–4], TestNG [13], DepUnit [10], Cucumber [9], and Spock [1]. In fact, as of May 2020, the JUnit annotations `@FixMethodOrder` and `@TestMethodOrder`, and the TestNG attributes `dependsOnMethods` and `dependsOnGroups` appear in over 197k Java files on GitHub.

Regression testing techniques should not assume that all tests are independent, since developers may still want OD tests. Specifically, these techniques should produce only test orders where each test has the same test outcome as it has when run in the original order. For example, if all of the tests pass in the original order, they should also pass in the order produced by a regression testing technique. However, traditional regression testing techniques may not achieve this goal if the test suite contains OD tests.

To understand how problematic the assumption of independent tests is to regression testing techniques as well as how neccessary it is for these techniques to be dependent-test-aware, we conduct the first study on the impact of OD tests on traditional regression testing techniques. Based on prior literature, we implement 4 test prioritization, 6 test selection, and 2 test parallelization algorithms[1]. We apply each algorithm to 11 Java modules from 8 projects, obtained from a prior dataset of OD tests [43]. Each module contains tests written by developers, which we refer to as *human-written* tests, and at least one test in the module is an OD test. Due to the attractiveness of automatic test generation tools to reduce developers' testing efforts, we also use Randoop [52], a state-of-the-art test generation tool, to obtain *automatically-generated* tests for each module. For both types of tests, all 12 regression testing algorithms produce orders that cause OD-test failures from tests that pass in the original order. Our findings provide empirical evidence that these regression testing algorithms should not ignore test dependencies.

We propose a new, general approach to enhance traditional regression testing algorithms to make them dependent-test-aware. Similar to unenhanced algorithms, our enhanced algorithms take as input the necessary metadata for the algorithms (e.g., coverage information for a test prioritization algorithm) and the original order. Besides the metadata, our enhanced algorithms also take as input a set of test dependencies (e.g., test $t1$ should be run only after running test $t2$). Our general approach enhances traditional algorithms by first using the traditional algorithms to produce an order, and then reordering or adding tests to the order such that any OD test is ordered and selected while satisfying the test dependencies.

[1] We refer to an algorithm as a specific implementation of a regression testing technique.

We evaluate our general approach by applying it to 12 regression testing algorithms and comparing the orders produced by the enhanced algorithms to those produced by the unenhanced ones. Specifically, we run both the enhanced and unenhanced algorithms on multiple versions of our evaluation projects. To evaluate our enhanced algorithms, we use DTDetector [71] to automatically compute a set of test dependencies. Our use of an automated tool to compute test dependencies demonstrates that even if developers are not manually specifying test dependencies in their projects now, they can still likely achieve the results of our work by using automated tools such as DTDetector. Ideally, one would automatically compute test dependencies frequently, so that they are up-to-date and orders produced with them would not cause OD-test failures. However, such an ideal is often infeasible because automatically computing these test dependencies can take substantial time. To closely imitate how developers may infrequently recompute test dependencies, for each of our evaluation projects we compute a set of test dependencies on one version and use them for many future versions.

Our evaluation finds that the orders from the enhanced algorithms cause substantially fewer OD-test failures and run only marginally slower than the orders from the unenhanced algorithms. Although our enhanced algorithms may not be using the most up-to-date test dependencies, our evaluation still finds that the algorithms produce orders that have 80% fewer OD-test failures than the orders produced by the unenhanced algorithms. Furthermore, although our enhanced algorithms may add tests for test selection and parallelization, we find that the orders with extra tests run only 1% slower on average. Our results suggest that making regression testing algorithms be dependent-test-aware can substantially reduce flaky-test failures with only a minor slowdown to run the tests.

This paper makes the following main contributions:

Study. A study of how OD tests affect traditional regression testing techniques such as test prioritization, test selection, and test parallelization. When we apply regression testing techniques to test suites containing OD tests, 82% of the human-written and 100% of the automatically-generated test suites have at least one OD-test failure.

Approach. A general approach to enhance traditional regression testing techniques to be dependent-test-aware. We apply our general approach to 12 traditional, regression testing algorithms, and make them and our approach publicly available [8].

Evaluation. An evaluation of 12 traditional, regression testing algorithms enhanced with our approach, showing that the orders produced by the enhanced algorithms can have 80% fewer OD-test failures, while being only 1% slower than the orders produced by the unenhanced algorithms.

2 IMPACT OF DEPENDENT TESTS

To understand how often traditional regression testing techniques lead to flaky-test failures due to OD tests, denoted as OD-test failures, we evaluate a total of 12 algorithms from three well-known regression testing techniques on 11 Java modules from 8 real-world projects with test suites that contain OD tests.

2.1 Traditional Regression Testing Techniques

Test prioritization, selection, and parallelization are traditional regression testing techniques that aim to detect faults faster than

Table 1: Four evaluated test prioritization algorithms from prior work [24].

Label	Ordered by
T1	Total statement coverage of each test
T2	Additional statement coverage of each test
T3	Total method coverage of each test
T4	Additional method coverage of each test

Table 2: Six evaluated test selection algorithms. The algorithms select a test if it covers new, deleted, or modified coverage elements. We also consider reordering the tests after selection. These algorithms form the basis of many other test selection algorithms [20, 50, 51, 54, 69].

Label	Selection granularity	Ordered by
S1	Statement	Test ID (no reordering)
S2	Statement	Total statement coverage of each test
S3	Statement	Additional statement coverage of each test
S4	Method	Test ID (no reordering)
S5	Method	Total method coverage of each test
S6	Method	Additional method coverage of each test

Table 3: Two evaluated test parallelization algorithms. These algorithms are supported in industrial-strength tools [7].

Label	Algorithm description
P1	Parallelize on test ID
P2	Parallelize on test execution time

simply running all of the tests in the given test suite. We refer to the order in which developers typically run all of these tests as the *original order*. These traditional regression testing techniques produce orders (permutations of a subset of tests from the original order) that may not satisfy test dependencies.

2.1.1 Test Prioritization. Test prioritization aims to produce an order for running tests that would fail and indicate a fault sooner than later [66]. Prior work [24] proposed test prioritization algorithms that reorder tests based on their (1) total coverage of code components (e.g., statements, methods) and (2) additional coverage of code components *not* previously covered. These algorithms typically take as input coverage information from a prior version of the code and test suite, and they use that information to reorder the tests on future versions[2]. We evaluate 4 test prioritization algorithms proposed in prior work [24]. Table 1 gives a concise description of each algorithm. Namely, the algorithms reorder tests such that the ones with more total coverage of code components (statements or methods) are run earlier, or reorder tests with more additional coverage of code components not previously covered to run earlier.

2.1.2 Test Selection. Test selection aims to select and run a subsuite of a program's tests after every change, but detect the same faults as if the full test suite is run [66]. We evaluate 6 test selection algorithms that select tests based on their coverage of modified code components [20, 32]; Table 2 gives a concise description of each algorithm. The algorithms use program analysis to select every test that may be affected by recent code modifications [32]. Each algorithm first builds a control-flow graph (CFG) for the then-current version of the program P_{old}, runs P_{old}'s test suite, and maps each test to the set of CFG edges covered by the test. When the program is modified to P_{new}, the algorithm builds P_{new}'s CFG and then selects the tests that cover "dangerous" edges: program points where P_{old} and P_{new}'s CFGs differ. We choose to select based on two levels of code-component granularity traditionally evaluated before, namely statements and methods [66]. We then order the selected tests. Ordering by test ID (an integer representing the position of the test in the original order) essentially does no reordering, while the other orderings make the algorithm a combination of test selection followed by test prioritization.

2.1.3 Test Parallelization. Test parallelization schedules the input tests for execution across multiple machines to reduce test latency — the time to run all tests. Two popular automated approaches for test parallelization are to parallelize a test suite based on (1) test ID and (2) execution time from prior runs [7]. A test ID is an integer representing the position of the test in the original order. We evaluate one test parallelization algorithm based on each approach (as described in Table 3). The algorithm that parallelizes based on test ID schedules the i^{th} test on machine i mod k, where k is the number of available machines and i is the test ID. The algorithm that parallelizes based on the tests' execution time (obtained from a prior execution) iteratively schedules each test on the machine that is expected to complete the earliest based on the tests already scheduled so far on that machine. We evaluate each test parallelization algorithm with $k = 2$, 4, 8, and 16 machines.

2.2 Evaluation Projects

Our evaluation projects consist of 11 modules from 8 Maven-based Java projects. These 11 modules are a subset of modules from the comprehensive version of a published dataset of flaky tests [43]. We include all of the modules (from the dataset) that contain OD tests, except for eight modules that we exclude because they are either incompatible with the tool that we use to compute coverage information or they contain OD tests where the developers have already specified test orderings in which the OD tests should run in. A list of which modules that we exclude from the dataset and the reasons for why we exclude them are on our website [8].

In addition to the existing human-written tests, we also evaluate automatically-generated tests. Automated test generation tools [21, 22, 25, 52, 72] are attractive because they reduce developers' testing efforts. These tools typically generate tests by creating sequences of method calls into the code under test. Although the tests are meant to be generated independently from one another, these tools often do not enforce test independence because doing so can substantially increase the runtime of the tests (e.g., restarting the VM between each generated test). This optimization results in automatically-generated test suites occasionally containing OD tests. Given the increasing importance of automatically-generated tests in both research and industrial use, we also investigate them for OD tests. Specifically, we use Randoop [52] version 3.1.5, a state-of-the-art random test generation tool. We configure Randoop to generate at most 5,000 tests for each evaluation project, and to drop tests that are subsumed by other tests (tests whose sequence of method calls is a subsequence of those in other tests).

[2]There is typically no point collecting coverage information on a future version to reorder and run tests on that version, because collecting coverage information requires one to run the tests already.

Table 4: Statistics of the projects used in our evaluation.

ID	Project	LOC		# Tests		# OD tests				# Evaluation versions	Days between versions	
		Source	Tests	Human	Auto	Human		Auto			Average	Median
M1	apache/incubator-dubbo - m1	2394	2994	101	353	2	(2%)	0	(0%)	10	4	5
M2	- m2	167	1496	40	2857	3	(8%)	0	(0%)	10	25	18
M3	- m3	2716	1932	65	945	8	(12%)	0	(0%)	10	10	8
M4	- m4	198	1817	72	2210	14	(19%)	0	(0%)	6	32	43
M5	apache/struts	3015	1721	61	4190	4	(7%)	1	(0%)	6	61	22
M6	dropwizard/dropwizard	1718	1489	70	639	1	(1%)	2	(0%)	10	11	6
M7	elasticjob/elastic-job-lite	5323	7235	500	566	9	(2%)	4	(1%)	2	99	99
M8	jfree/jfreechart	93915	39944	2176	1233	1	(0%)	0	(0%)	5	13	2
M9	kevinsawicki/http-request	1358	2647	160	4537	21	(13%)	0	(0%)	10	42	4
M10	undertow-io/undertow	4977	3325	49	967	6	(12%)	1	(0%)	10	22	15
M11	wildfly/wildfly	7022	1931	78	140	42	(54%)	20	(14%)	10	37	19
Total/Average/Median		**122803**	**66531**	**3372**	**18637**	**111**	**(3%)**	**28**	**(0%)**	**89**	**25**	**8**

2.3 Methodology

Regression testing algorithms analyze one version of code to obtain metadata such as coverage or time information for every test so that they can compute specific orders for future versions. For each of our evaluation projects, we treat the version of the project used in the published dataset [43] as the latest version in a sequence of versions. In our evaluation, we define a "version" for a project as a particular commit that has a change for the module containing the OD test, and the change consists of code changes to a Java file. Furthermore, the code must compile and all tests must pass through Maven. We go back at most 10 versions from this latest version to obtain the First Version (denoted as *firstVer*) of each project. We may not obtain 10 versions for an evaluation project if it does not have enough commits that satisfy our requirements, e.g., commits that are old may not be compilable anymore due to missing dependencies. We refer to each subsequent version after *firstVer* as a *subseqVer*. For our evaluation, we use *firstVer* to obtain the metadata for the regression testing algorithms, and we evaluate the use of such information on the *subseqVer*s. For automatically-generated test suites, we generate the tests on *firstVer* and copy the tests to *subseqVer*s. Any copied test that does not compile on a *subseqVer* is dropped from the test suite when run on that version.

Table 4 summarizes the information of each evaluation project. Column "LOC" is the number of non-comment, non-blank lines in the project's source code and human-written tests as reported by `sloc` [11] for *firstVer* of each evaluation project. Column "# Tests" shows the number of human-written tests and those generated by Randoop [52] for *firstVer* of each evaluation project. Column "# Evaluation versions" shows the number of versions from *firstVer* to latest version that we use for our evaluation, and column "Days between versions" shows the average and median number of days between the versions that we use for our evaluation.

To evaluate how often OD tests fail when using test prioritization and test parallelization algorithms, we execute these algorithms on the version immediately following *firstVer*, called the Second Version (denoted as *secondVer*). For test selection, we execute the algorithms on all versions after *firstVer* up to the latest version (the version from the dataset [43]). The versions that we use for each of our evaluation projects are available online [8]. For all of the algorithms, they may rank multiple tests the same (e.g., two tests cover the same statements or two tests take the same amount of time to run). To break ties, the algorithms deterministically sort tests based on their ordering in the original order. Therefore, with the same metadata for the tests, our the algorithms would always produce the same order.

For the evaluation of test prioritization algorithms, we count the number of OD tests that fail in the prioritized order on *secondVer*. For test selection, given the change between *firstVer* and the future versions, we count the number of unique OD tests that fail from the possibly reordered selected tests on all future versions. For test parallelization, we count the number of OD tests that fail in the parallelized order on any of the machines where tests are run on *secondVer*. All test orders are run three times and a test is counted as OD only if it consistently fails for all three runs. Note that we can just count failed tests as OD tests because we ensure that all tests pass in the original order of each version that we use.

Note that the general form of the OD test detection problem is NP-complete [71]. To get an approximation for the maximum number of OD-test failures with which the orders produced by regression testing algorithms can cause, we apply DTDetector [71] to randomize the test ordering for 100 times on *firstVer* and all *subseqVer*s. We choose randomization because prior work [43, 71] found it to be the most effective strategy in terms of time cost when finding OD tests. The DTDetector tool is sound but incomplete, i.e., every OD test that DTDetector finds is a real OD test, but DTDetector is not guaranteed to find every OD test in the test suite. Thus, the reported number is a lower bound of the total number of OD tests. Column "# OD tests" in Table 4 reports the number of OD tests that DTDetector finds when it is run on all versions of our evaluation projects.

There are flaky tests that can pass or fail on the same version of code but are not OD tests (e.g., flaky tests due to concurrency). For all of the test suites, we run each test suite 100 times in its original order and record the tests that fail as flaky but not as OD tests. We use this set of non-order-dependent flaky tests to ensure that the tests that fail on versions after *firstVer* are likely OD tests and not other types of flaky tests.

2.4 Results

Table 5 and Table 6 summarize our results (parallelization is averaged across k = 2, 4, 8, and 16). The exact number of OD-test failures for each regression testing algorithm is available on our website [8]. In Table 5, each cell shows the percentage of unique OD tests that fail in all of the orders produced by the algorithms of a technique over the number of known OD tests for that specific evaluation project. Cells with a "n/a" represent test suites (of evaluation projects) that do not contain any OD tests according to DTDetector

Table 5: Percentage of OD tests that fail in orders produced by different regression testing techniques, per evaluation project.

	OD tests that fail (per evaluation project)					
	Prioritization		Selection		Parallelization	
ID	Human	Auto	Human	Auto	Human	Auto
M1	50%	n/a	100%	n/a	50%	n/a
M2	67%	n/a	67%	n/a	0%	n/a
M3	12%	n/a	50%	n/a	0%	n/a
M4	7%	n/a	14%	n/a	14%	n/a
M5	0%	100%	25%	100%	75%	100%
M6	100%	0%	0%	0%	0%	50%
M7	44%	50%	0%	0%	0%	25%
M8	0%	n/a	0%	n/a	0%	n/a
M9	71%	n/a	71%	n/a	0%	n/a
M10	17%	0%	17%	0%	0%	100%
M11	0%	60%	0%	0%	0%	30%
Total	23%	54%	24%	4%	5%	36%

and the regression testing algorithms. The "Total" row shows the percentage of OD tests that fail across all evaluation projects per technique over all OD tests found by DTDetector. In Table 6, each cell shows the percentage of OD tests across all evaluation projects that fail per algorithm. The dependent tests that fail in any two cells of Table 6 may not be distinct from one another.

On average, 3% of human-written tests and 0% of automatically-generated tests are OD tests. Although the percentage of OD tests may be low, the effect that these tests have on regression testing algorithms is substantial. More specifically, almost every project's human-written test suite has at least one OD-test failure in an order produced by one or more regression testing algorithms (the only exception is jfree/jfreechart (M8), for which DTDetector found only one OD test across all versions). These OD-test failures waste developers' time to investigate or delay the discovery of a real fault.

According to Table 6, it may seem that algorithms that order tests by Total coverage (T1, T3, S2, S5) always have fewer OD-test failures than their respective algorithms that order tests by Additional coverage (T2, T4, S3, S6), particularly for test prioritization algorithms. However, when we investigate the algorithm and module that best exhibit this difference, namely kevinsawicki/http-request's (M9) test prioritization results, we find that this one case is largely responsible for the discrepancies that we see for the test prioritization algorithms. Specifically, M9 contains 0 OD-test failures for T1 and T3, but 14 and 24 OD-test failures for T2 and T4, respectively. All OD tests that fail for M9's T2 and T4 would fail when one particular test is run before them; we refer to this test that runs before as the dependee test. The dependent tests all have similar coverage, so in the Additional orders, these tests are not consecutive and many of them come later in the orders. The one dependee test then comes inbetween some dependent tests, causing the ones that come later than the dependee test to fail. In the Total orders, the dependee test has lower coverage than the dependent tests and is always later in the orders. If we omit M9's prioritization results, we no longer observe any substantial difference in the number of OD-test failures between the Total coverage algorithms and Additional coverage algorithms.

2.4.1 Impact on Test Prioritization. Test prioritization algorithms produce orders that cause OD-test failures for the human-written test suites in eight evaluation projects. For automatically-generated test suites, test prioritization algorithms produce orders that cause OD-test failures in three projects. Our findings suggest that orders produced by test prioritization algorithms are more likely to cause OD-test failures in human-written test suites than automatically-generated test suites. Overall, we find that test prioritization algorithms produce orders that cause 23% of the human-written OD tests to fail and 54% of the automatically-generated OD tests to fail.

Table 6: Percentage of OD tests that fail in orders produced by individual regression testing algorithms.

	OD tests that fail (per algorithm)											
	Prioritization				Selection						Parallelization	
Type	T1	T2	T3	T4	S1	S2	S3	S4	S5	S6	P1	P2
Human	5%	25%	5%	20%	1%	28%	31%	1%	5%	9%	2%	5%
Auto	36%	43%	71%	25%	4%	4%	4%	4%	4%	4%	21%	64%

2.4.2 Impact on Test Selection. Test selection algorithms produce orders that cause OD-test failures for the human-written test suites in seven evaluation projects and for the automatically-generated test suites in one project. These test selection algorithms produce orders that cause 24% of the human-written OD tests and 4% of the automatically-generated OD tests to fail. The algorithms that do not reorder a test suite (S1 and S4) produce orders that cause fewer OD-test failures than the algorithms that do reorder. This finding suggests that while selecting tests itself is a factor, reordering tests is generally a more important factor that leads to OD-test failures.

2.4.3 Impact on Test Parallelization. Test parallelization algorithms produce orders that cause OD-test failures because the algorithms may schedule an OD test on a different machine than the test(s) that it depends on. The parallelization algorithm that reorders tests, P2, produces orders that cause substantially more OD-test failures than P1. This result reaffirms our finding from Section 2.4.2 that reordering tests has a greater impact on OD-test failures than selecting tests. The percentages reported in Table 5 and Table 6 are calculated from the combined set of OD tests that fail due to parallelization algorithms for k = 2, 4, 8, and 16 machines. The orders of these algorithms cause 5% of the human-written OD tests and 36% of the automatically-generated OD tests to fail on average.

2.5 Findings

Our study suggests the following two main findings.

(1) Regression testing algorithms that reorder tests in the given test suite are more likely to experience OD test failures than algorithms that do not reorder tests in the test suite. We see this effect both by comparing test selection algorithms that do not reorder tests (S1 and S4) to those that do, as well as comparing a test parallelization algorithm, P2, which does reorder tests, to P1, which does not. Developers using algorithms that reorder tests would especially benefit from our dependent-test-aware algorithms described in Section 3.

(2) Human-written and automatically-generated test suites are likely to fail due to test dependencies. As shown in Table 5, we find that regression testing algorithms produce orders that cause OD-test failures in 82% (9 / 11) of human-written test suites with OD tests, compared to the 100% (5 / 5) of automatically-generated test suites. Both percentages are substantial and showcase the likelihood of OD-test failures when using traditional regression testing algorithms that are unaware of OD tests.

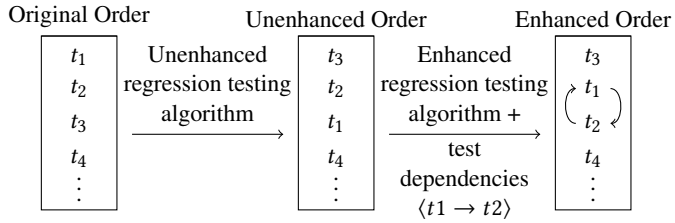

Figure 1: Example of an unenhanced and its enhanced, dependent-test-aware regression testing algorithms.

3 DEPENDENT-TEST-AWARE REGRESSION TESTING TECHNIQUES

When a developer conducts regression testing, there are two options: running the tests in the original order or running the tests in the order produced by a regression testing algorithm. When the developer runs the tests in the original order, a test might fail because either (1) there is a fault somewhere in the program under test, or (2) the test is flaky but it is not a OD test (e.g., flaky due to concurrency). However, when using a traditional regression testing algorithm, there is a third reason for why tests might fail: the produced orders may not be satisfying the test dependencies. Algorithms that are susceptible to this third reason do not adhere to a primary design goal of regression testing algorithms. Specifically, the orders produced by these algorithms should not cause OD-test failures: if the tests all pass in the original order, then the algorithms should produce only orders in which all of the tests pass (and dually, if tests fail in the original order, then the algorithms should produce only orders in which these tests fail).[3] Since many real-world test suites contain OD tests, a regression testing algorithm that assumes its input contains no OD tests can produce orders that cause OD-test failures, violating this primary design goal (which we see from our results in Section 2).

We propose that regression testing techniques should be dependent-test-aware to remove OD-test failures. Our general approach completely removes all possible OD-test failures with respect to the input test dependencies. In practice, such test dependencies may or may not always be complete (all test dependencies that prevent OD-test failures are provided) or minimal (only test dependencies that are needed to prevent OD-test failures are provided). Nevertheless, our general approach requires as input an original order, a set of test dependencies, and an order outputted by a traditional regression testing algorithm to output an updated, enhanced order that satisfies the given test dependencies.

3.1 Example

Figure 1 shows an illustrated example of the orders produced by an unenhanced algorithm and its corresponding enhanced algorithm. The unenhanced algorithm does not take test dependencies into account, and therefore may produce orders that cause OD-test failures. Specifically, the unenhanced algorithm produces the Unenhanced Order. On the other hand, the enhanced algorithm produces the Enhanced Order, a test order that satisfies the provided test dependencies of the test suite. The enhanced algorithm does so by first using the unenhanced algorithm to produce the Unenhanced Order and then enforcing the test dependencies on to the order by reordering or adding tests.

[3] Another design goal is to maximize fault-finding ability over time, i.e., efficiency. The efficiency goal is a trade-off against the correctness goal.

enhanceOrder(T_u, D, T_{orig}):

```
T_e ← []
P ← {⟨p → d⟩ ∈ D} // Positive test dependencies
N ← {⟨n ↛ d⟩ ∈ D} // Negative test dependencies
T_a ← T_u ∘ (P^-1)* // Get all transitive positive dependee tests
for t : T_u do // Iterate T_u sequence in order
    if t ∈ T_e then continue end if
    T_e ← addTest(t, T_e, T_u, T_orig, T_a, P, N)
end for
return T_e
```

addTest$(t, T_e, T_u, T_{orig}, T_a, P, N)$:

```
B ← {t' ∈ T_u ∪ T_a | ⟨t' → t⟩ ∈ P ∨ ⟨t ↛ t'⟩ ∈ N}
L ← sort(B ∩ T_u, orderBy(T_u)) ⊕ sort(B \ Tu, orderBy(T_orig))
for b : L do // Iterate the before tests in order of L
    if b ∈ T_e then continue end if
    T_e ← addTest(b, T_e, T_u, T_orig, T_a, P, N)
end for
return T_e ⊕ [t]
```

Figure 2: General approach to enhance an order from traditional regression testing algorithms.

We define two different types of test dependencies, positive and negative dependencies. A *positive test dependency* $\langle p \rightarrow d \rangle$ denotes that for OD test d to pass, it should be run only after running test p, the test that d depends on. A *negative test dependency* $\langle n \nrightarrow d \rangle$ denotes that for OD test d to pass, it should *not* be run after test n. Previous work [60] refers to test p as a state-setter and test d in a positive dependency as a brittle. It also refers to test n as a polluter and d in a negative dependency as a victim. For simplicity, we refer to the tests that OD tests depend on (i.e., p and n) as *dependee tests*. For both types of dependencies, the dependee and OD test do not need to be run consecutively, but merely in an order that adheres to the specified dependencies.

In Figure 1, there is a single, positive test dependency $\langle t_1 \rightarrow t_2 \rangle$ in the input test dependencies. $\langle t_1 \rightarrow t_2 \rangle$ denotes that the OD test t_2 should be run only after running test t_1. In the Unenhanced Order, the positive test dependency $\langle t_1 \rightarrow t_2 \rangle$ is not satisfied and t_2 will fail. Our enhanced algorithm prevents the OD-test failure of t_2 by modifying the outputted order of the unenhanced algorithm so that the test dependency (t_2 should be run only after running t_1) is satisfied in the Enhanced Order.

3.2 General Approach for Enhancing Regression Testing Algorithms

Figure 2 shows our general algorithm, enhanceOrder, for enhancing an order produced by a traditional regression testing algorithm to become dependent-test-aware. enhanceOrder takes as input T_u, which is the order produced by the traditional unenhanced algorithm that enhanceOrder is enhancing (this order can be a different permutation or subset of T_{orig}), the set of test dependencies D, and the original test suite T_{orig}, which is an ordered list of tests in the original order. While in theory one could provide test dependencies that are

not linearizable (e.g., both $\langle t_1 \rightarrow t_2 \rangle$ and $\langle t_2 \rightarrow t_1 \rangle$ in D), we assume that the provided test dependencies are linearizable, and T_{orig} is the tests' one total order that satisfies all of the test dependencies in D. For this reason, enhanceOrder does not check for cycles within D. Based on T_u, enhanceOrder uses the described inputs to output a new order (T_e) that satisfies the provided test dependencies D.

enhanceOrder starts with an empty enhanced order T_e and then adds each test in the unenhanced order T_u into T_e using the addTest function (Line 7). To do so, enhanceOrder first computes T_a, the set of tests that the tests in T_u transitively depend on from the positive test dependencies (Line 4). T_a represents the tests that the traditional test selection or parallelization algorithms did not include in T_u, but they are needed for OD tests in T_u to pass. enhanceOrder then iterates through T_u in order (Line 5) to minimize the perturbations that it makes to the optimal order found by the traditional unenhanced algorithm. The addTest function adds a test t into the current T_e while ensuring that all of the provided test dependencies are satisfied. Once all of T_u's tests are added into T_e, enhanceOrder returns T_e.

On a high-level, the function addTest has the precondition that all of the tests in the current enhanced order T_e have their test dependencies satisfied in T_e, and addTest has the postcondition that test t is added to the end of T_e (Line 16) and all tests in T_e still have their test dependencies satisfied. To satisfy these conditions, addTest starts by obtaining all of the tests that need to run before the input test t (Line 10), represented as the set of tests B.

The tests in B are all of the dependee tests within the positive test dependencies P for t, i.e., all tests p where $\langle p \rightarrow t \rangle$ are in P. Note that these additional tests must come from either T_u or T_a (the additional tests that the traditional algorithm does not add to T_u). Line 10 just includes into B the direct dependee tests of t and not those that it indirectly depends on; these indirect dependee tests are added to T_e through the recursive call to addTest (Line 14). The tests in B also include the dependent tests within the negative test dependencies N whose dependee test is t, i.e., all tests d where $\langle t \nrightarrow d \rangle$ are in N. addTest does not include test d that depends on t from the negative test dependencies if d is not in T_u or T_a. Conceptually, these are tests that the unenhanced algorithm originally did not find necessary to include (i.e., for test selection the test is not affected by the change, or for test parallelization the test is scheduled on another machine), and they are also not needed to prevent any OD tests already included in T_u from failing.

Once addTest obtains all of the tests that need to run before t, it then adds all of these tests into the enhanced order T_e, which addTest accomplishes by recursively calling addTest on each of these tests (Line 14). Line 11 first sorts these tests based on their order in the unenhanced order T_u. This sorting is to minimize the perturbations that it makes to the optimal order found by the unenhanced algorithm. For any additional tests not in T_u (tests added through T_a), they are sorted to appear at the end and based on their order in the original order, providing a deterministic ordering for our evaluation (in principle, one can use any topological order). Once all of the tests that must run before t are included into T_e, addTest adds t (Line 16).

OD-test failures may still arise even when using an enhanced algorithm because the provided test dependencies (D) may not be complete. For example, a developer may forget to manually specify some test dependencies, and even if the developer uses an automatic tool for computing test dependencies, such tool may not find all dependencies as prior work [71] has shown that computing *all* test dependencies is an NP-complete problem. Also, a developer may have made changes that invalidate some of the existing test dependencies or introduce new test dependencies, but the developer does not properly update the input test dependencies.

4 EVALUATION OF GENERAL APPROACH

Section 2 shows how both human-written and automatically-generated test suites with OD tests have OD-test failures when developers apply traditional, unenhanced regression testing algorithms on these test suites. To address this issue, we apply our general approach described in Section 3 to enhance 12 regression testing algorithms and evaluate them with the following metrics.

- **Effectiveness of reducing OD-test failures:** the reduction in the number of OD-test failures after using the enhanced algorithms. Ideally, every test should pass, because we confirm that all tests pass in the original order on the versions that we evaluate on (the same projects and versions from Section 2.2). This metric is the most important desideratum.
- **Efficiency of orders:** how much longer-running are orders produced by the enhanced regression testing algorithms than those produced by the unenhanced algorithms.

4.1 Methodology

To evaluate 12 enhanced regression testing algorithms, we start with *firstVer* for each of the evaluation projects described in Section 2.2. Compared to the unenhanced algorithms, the only additional input that the enhanced algorithms require is test dependencies D. These test dependencies D and the other metadata needed by the regression testing algorithms are computed on *firstVer* of each evaluation project. The details on how we compute test dependencies for our evaluation are in Section 4.2. With D and the other inputs required by the unenhanced and enhanced algorithms, we then evaluate these algorithms on *subseqVer*s of the projects.

In between *firstVer* and *subseqVer*s there may be tests that exist in one but not the other. We refer to tests that exist in *firstVer* as *old tests* and for tests that are introduced by developers in a future version as *new tests*. When running the *old tests* on future versions, we use the enhanced algorithms, which use coverage or time information from *firstVer* (also needed by the unenhanced algorithms) and D. Specifically, for all of the algorithms, we use the following procedure to handle changes in tests between *firstVer* and a *subseqVer*.

(1) The test in *firstVer* is skipped if *subseqVer* no longer contains the corresponding test.
(2) Similar to most traditional regression testing algorithms, we treat tests with the same fully-qualified name in *firstVer* and *subseqVer* as the same test.
(3) We ignore *new tests* (tests in *subseqVer* but not in *firstVer*), because both unenhanced and enhanced regression testing algorithms would treat these tests the same (i.e., run all of these tests before or after *old tests*).

4.2 Computing Test Dependencies

Developers can obtain test dependencies for a test suite by (1) manually specifying the test dependencies, or (2) using automatic tools

to compute the test dependencies [26, 29, 43, 64, 71]. For our evaluation, we obtain test dependencies through the latter approach, using automatic tools, because we want to evaluate the scenario of how any developer can benefit from our enhanced algorithms without having to manually specify test dependencies. Among the automatic tools to compute test dependencies, both DTDetector [71] and iDFlakies [43] suggest that randomizing a test suite many times is the most cost effective way to compute test dependencies. For our evaluation, we choose to use DTDetector since it is the more widely cited work on computing test dependencies. Before we compute test dependencies, we first filter out tests that are flaky but are not OD tests (e.g., tests that are flaky due to concurrency [43, 46]) for each of our evaluation projects. We filter these tests by running each test suite 100 times in its original order and removing all tests that had test failures. We remove these tests since they can fail for other reasons and would have the same chance of affecting unenhanced and enhanced algorithms. To simulate how developers would compute test dependencies on a current version to use on future versions, we compute test dependencies on a prior version (*firstVer*) of a project's test suite and use them with our enhanced algorithms on future versions (*subseqVer*s).

4.2.1 DTDetector. DTDetector [71] is a tool that detects test dependencies by running a test suite in a variety of different orders and observing the changes in the test outcomes. DTDetector outputs a test dependency if it observes a test t to pass in one order (denoted as po) and fail in a different order (denoted as fo). Typically, t depends on either some tests that run before t in po to be dependee tests in a positive test dependency (some tests must always run before t), or some tests that run before t in fo to be dependee tests in a negative test dependency (some tests must always run after t). t can also have both a positive dependee test and a negative dependee test; this case is rather rare, and we do not observe such a case in our evaluation projects. DTDetector outputs the minimal set of test dependencies by delta-debugging [28, 67] the list of tests coming before t in po and fo to remove as many tests as possible, while still causing t to output the same test outcome.

When we use DTDetector directly as its authors intended, we find that DTDetector's reordering strategies require many hours to run and are designed to search for test dependencies by randomizing test orders; however, we are interested in test dependencies only in the orders that arise from regression testing algorithms. To address this problem, we extend DTDetector to compute test dependencies using the output of the regression testing algorithms. This strategy is in contrast to DTDetector's default strategies, which compute test dependencies for a variety of orders that may not resemble the outputs of regression testing algorithms. Specifically, for test prioritization or test parallelization, we use the orders produced by their unenhanced algorithms. DTDetector will find test dependencies for any test that fails in these orders, since these tests now have a passing order (all tests must have passed in the original order) and a failing order. Using these two orders, DTDetector will minimize the list of tests before an OD test, and we would then use the minimized list as test dependencies for the enhanced algorithms. If the new test dependencies with the enhanced algorithms cause new failing OD tests, then we repeat this process again until the orders for the enhanced algorithms no longer cause any failing OD test.

Table 7: Average time in seconds to run the test suite and average time to compute test dependencies for an OD test. "Prioritization" and "Parallelization" show the average time per algorithm, while "All 6" shows the average time across *all* 6 algorithms. "-" denotes cases that have no OD test for all algorithms of a particular technique.

	Suite run time		Time to precompute test dependencies					
			Prioritization		Parallelization		All 6	
ID	Human	Auto	Human	Auto	Human	Auto	Human	Auto
M1	7.3	0.3	64	-	24	-	44	-
M2	0.4	0.2	95	-	-	-	95	-
M3	184.2	0.2	1769	-	-	-	1769	-
M4	0.1	0.2	19	-	13	-	17	-
M5	2.4	0.4	-	396	31	215	31	275
M6	4.1	0.3	75	-	-	29	75	29
M7	20.4	45.6	241	242	-	176	241	216
M8	1.2	1.3	-	-	-	-	-	-
M9	1.2	0.1	28	-	-	-	28	-
M10	19.9	0.8	157	-	-	39	157	39
M11	2.3	0.4	-	484	-	210	-	438

For test selection, we simply combine and use all of the test dependencies that we find for the test prioritization and test parallelization algorithms. We use the other algorithms' orders because it is difficult to predict test selection orders on future versions, i.e., the tests selected in one version will likely be different than the tests selected in another version. This methodology simulates what developers would do: they know what regression testing algorithm to use but do not know what code changes they will make in the future.

4.2.2 Time to precompute dependencies. Developers should not compute test dependencies as they are performing regression testing. Instead, as we show in our evaluation, test dependencies can be collected on the current version and be reused later.

Developers can compute test dependencies infrequently and offline. Recomputing test dependencies can be beneficial if new test dependencies are needed or if existing test dependencies are no longer needed because of the developers' recent changes. While the developers are working between versions v_i and v_{i+1}, they can use that time to compute test dependencies. Table 7 shows the time in seconds to compute the test dependencies that we use in our evaluation. The table shows the average time to compute test dependencies per OD test across all test prioritization or parallelization algorithms (for the columns under "Prioritization" and "Parallelization", respectively), and the time under "All 6" is the average time to compute dependencies per OD test across all six test prioritization and parallelization algorithms (as explained in Section 4.2.1, the test dependencies that test selection uses are the combination of those from test prioritization and parallelization). The reported time includes the time for checks such as rerunning failing tests to ensure that it is actually an OD test (i.e., it is not flaky for other reasons) as to avoid the computation of non-existent test dependencies.

Although the time to compute test dependencies is substantially more than the time to run the test suite, we can see from Tables 4 and 7 that the time between versions is still much more than the time to compute test dependencies. For example, while it takes about half an hour, on average, to compute test dependencies per OD test in M3's human-written test suite, the average number of days between the versions of M3 is about 10 days, thus still giving

Table 8: Percentage of how many fewer OD-test failures occur in the test suites produced by the enhanced algorithms compared to those produced by the unenhanced algorithms. Higher percentages indicate that the test suites by the enhanced algorithms have fewer OD-test failures.

	% Reduction in OD-test failures					
	Prioritization		Selection		Parallelization	
ID	Human	Auto	Human	Auto	Human	Auto
M1	100%	n/a	50%	n/a	100%	n/a
M2	100%	n/a	100%	n/a	-	n/a
M3	100%	n/a	-25%	n/a	-	n/a
M4	100%	n/a	60%	n/a	100%	n/a
M5	-	60%	0%	100%	100%	82%
M6	100%	-	-	-	-	100%
M7	100%	57%	-	-	-	12%
M8	-	n/a	-	n/a	-	n/a
M9	100%	n/a	92%	n/a	-	n/a
M10	100%	-	100%	-	-	100%
M11	-	56%	-	-	-	29%
Total	100%	57%	79%	100%	100%	66%

developers substantial time in between versions to compute test dependencies. Although such a case does not occur in our evaluation, even if the time between v_i and v_{i+1} is less than the time to compute test dependencies, the computation can start running on v_i while traditional regression testing algorithms (that may waste developers' time due to OD-test failures) can still run on v_{i+1}. Once computation finishes, the enhanced regression testing algorithms can start using the computed test dependencies starting at the current version of the code (e.g., version v_{i+n} when the computation starts on v_i). As we show in Section 4.3, these test dependencies are still beneficial many versions after they are computed.

4.3 Reducing Failures

Table 8 shows the reduction in the number of OD-test failures from the orders produced by the enhanced algorithms compared to those produced by the unenhanced algorithms. We denote cases that have no OD tests (as we find in Section 2.4) as "n/a", and cases where the unenhanced algorithms do not produce an order that causes any OD-test failures as "-". Higher percentages indicate that the enhanced algorithms are more effective than the unenhanced algorithms at reducing OD-test failures.

Concerning human-written tests, we see that enhanced prioritization and parallelization algorithms are very effective at reducing the number of OD-test failures. In fact, the enhanced prioritization and parallelization algorithms reduce the number of OD-test failures by 100% across all of the evaluation projects. For test selection, the algorithms reduce the number of OD-test failures by 79%. This percentage is largely influenced by M3, where the enhanced selection orders surprisingly lead to *more* failures than the unenhanced orders as indicated by the negative number (-25%) in the table.

There are two main reasons for why an enhanced order can still have OD-test failures: (1) changes from later versions introduce new tests that are OD tests and are not in *firstVer*, and (2) the computed test dependencies from *firstVer* are incomplete; as such, the regression testing algorithms would have OD-test failures due to the missing test dependencies. In the case of (1), if this case were to happen, then both enhanced and unenhanced orders would be equally affected, and for our evaluation, we simply ignored all newly added tests. In the case of (2), it is possible that the test dependencies computed on *firstVer* are incomplete for the same OD tests on a new version, either because the test dependencies are not captured on *firstVer* or because the test initially is not an OD test in *firstVer* but becomes one due to introduced test dependencies in the new version. In fact, for M3's human-written test selection results, we see that the reason for why the enhanced orders have more OD-test failures is that the enhanced orders in later versions expose a test dependency that is missing from what is computed on *firstVer*. As we describe in Section 4.2, to efficiently compute test dependencies on *firstVer*, we use only the orders of test prioritization and test parallelization instead of many random orders as done in some previous work [43, 71]. It is important to note that such a case occurs in our evaluation only for M3, but nonetheless, this case does demonstrate the challenge of computing test dependencies effectively and efficiently.

Table 9: Percentage slower that orders produced by the enhanced algorithms run compared to those produced by the unenhanced algorithms. Higher percentages indicate that orders produced by the enhanced algorithms are slower.

	% Time Slowdown			
	Selection		Parallelization	
ID	Human	Auto	Human	Auto
M1	9%	=	16%	7%
M2	-1%	=	9%	-8%
M3	3%	=	0%	0%
M4	5%	=	-2%	-2%
M5	=	1%	-1%	-1%
M6	-1%	=	2%	-3%
M7	6%	=	1%	0%
M8	=	=	5%	4%
M9	11%	=	2%	3%
M10	-5%	=	2%	-14%
M11	=	=	-2%	-7%
Total	1%	1%	1%	0%

For automatically-generated tests, the enhanced algorithms are also quite effective at reducing OD-test failures, though not as effective as the enhanced algorithms for human-written tests. For test selection, M5's unenhanced orders have OD-test failures, while the enhanced orders would completely remove all of the OD-test failures. Specifically, we see that the enhanced algorithms add in the extra positive dependee test to prevent the OD-test failures. Similarly, the enhanced test parallelization algorithms also add in some missing positive dependee tests, leading to a reduction in OD-test failures.

In summary, we find that the orders produced by all of the enhanced regression testing algorithms collectively reduce the number of OD-test failures by 81% and 71% for human-written and automatically-generated tests, respectively. When considering both types of tests together, the enhanced regression testing algorithms produce orders that reduce the number of OD-test failures by 80% compared to the orders produced by the unenhanced algorithms.

4.4 Efficiency

To accommodate test dependencies, our enhanced regression testing algorithms may add extra tests to the orders produced by the unenhanced test selection or parallelization algorithms (T_a on Line 4 in Figure 2). The added tests can make the orders produced by the enhanced algorithms run slower than those produced by the unenhanced algorithms. Table 9 shows the slowdown of running the

orders from the enhanced test selection and parallelization algorithms. We do not compare the time for orders where the enhanced and unenhanced algorithms produce the exact same orders (not only just running the same tests but also having the same ordering of the tests), since both orders should have the same running time modulo noise. We mark projects that have the same orders for enhanced and unenhanced with "=" in Table 9. For parallelization, we compare the runtime of the longest running subsuite from the enhanced algorithms to the runtime of the longest running subsuite from the unenhanced algorithms. For each of the projects in Table 9, we compute the percentage by summing up the runtimes for all of the enhanced orders in the project, subtracting the summed up runtimes for all of the unenhanced orders, and then dividing the summed up runtimes for all of the unenhanced orders. The overall percentage in the final row is computed the same way except we sum up the runtimes for the orders across all of the projects.

Overall, we see that the slowdown is rather small, and the small speedups (indicated by negative numbers) are mainly due to noise in the tests' runtime. For test parallelization of automatically-generated tests, we do observe a few cases that do not appear to be due to noise in the tests' runtime though. Specifically, we see that there are often speedups even when tests are added to satisfy test dependencies (e.g., M10). We find that in these cases the enhanced orders are faster because the OD-test failures encountered by the unenhanced orders actually slow down the test suite runtime more than the tests added to the enhanced orders that prevent the OD-test failures. This observation further demonstrates that avoiding OD-test failures is desirable, because doing so not only helps developers avoid having to debug non-existent faults in their changes, but can also potentially speed up test suite runtime.

The overall test suite runtime slowdown of the enhanced algorithms compared to the unenhanced ones is 1% across all of the orders produced and across all types of tests (human-written and automatically-generated) for all of the evaluation projects.

4.5 Findings

Our evaluation suggests the following two main findings.

Reducing Failures. The enhanced regression testing algorithms can reduce the number of OD-test failures by 81% for human-written test suites. The enhanced algorithms are less effective for automatically-generated test suites, reducing OD-test failures by 71%, but the unenhanced algorithms for these test suites generally cause fewer OD-test failures. Across all of the regression testing algorithms, the enhanced algorithms produce orders that cause 80% fewer OD-test failures than the orders produced by the unenhanced algorithms.

Efficiency. Our enhanced algorithms produce orders that run only marginally slower than those produced by the unenhanced algorithms. Specifically, for test selection and test parallelization, the orders produced by the enhanced algorithms run only 1% slower than the orders produced by the unenhanced algorithms.

5 DISCUSSION

5.1 General Approach vs. Customized Algorithms

In our work, we focus on a general approach for enhancing existing regression testing algorithms. Our approach works on any output of these existing regression testing algorithms to create an enhanced order that satisfies the provided test dependencies. While our approach works for many different algorithms that produce an order, it may not generate the most optimal order for the specific purpose of the regression testing algorithm being enhanced.

For example, we enhance the orders produced by test parallelization algorithms by adding in the missing tests for an OD test to pass on the machine where it is scheduled to run. A more customized test parallelization algorithm could consider the test dependencies as it decides which tests get scheduled to which machines. If the test dependencies are considered at this point during the test parallelization algorithms, then it could create faster, more optimized scheduling of tests across the machines. However, such an approach would be specific to test parallelization (and may even need to be specialized to each particular test parallelization algorithm) and may therefore not generalize to other regression testing algorithms. Nevertheless, it can be worthwhile for future work to explore how customized algorithms can enhance traditional regression testing algorithms.

5.2 Cost to Provide Test Dependencies

Developers may create OD tests purposefully to optimize test execution time by doing some expensive setup in one test and have that setup be shared with other, later-running tests. If developers are aware that they are creating such tests, then the human cost for providing test dependencies to our enhanced algorithms is low.

If developers are not purposefully creating OD tests, and they are unaware that they are creating OD tests, then it would be beneficial to rely on automated tools to discover such OD tests for them and use the outputs of these tools for our enhanced algorithms, as we demonstrate in our evaluation. The cost to automatically compute test dependencies (machine cost) is cheaper than the cost for developers to investigate test failures (human cost). Herzig et al. [33] quantified human and machine cost. They reported that the cost to inspect one test failure for whether it is a flaky-test failure is $9.60 on average, and the total cost of these inspections can be about $7 million per year for products such as Microsoft Dynamics. They also reported that machines cost $0.03 per minute. For our experiments, the longest time to compute test dependencies for an OD test is for M3, needing about half an hour, which equates to just about $0.90.

5.3 Removing OD Tests

OD-test failures that do not indicate faults in changes are detrimental to developers in the long run, to the point that, if these failures are not handled properly, one might wonder why a developer does not just remove these OD tests entirely. However, it is important to note that these tests function exactly as they are intended (i.e., finding faults in the code under test) when they are run in the original order. Therefore, simply removing them would mean compromising the quality of the test suite to reduce OD-test failures, being often an unacceptable tradeoff. Removing OD tests is especially undesirable for developers who are purposefully writing them for the sake of faster testing [6], evident by the over 197k Java files (on GitHub) that use some JUnit annotations or TestNG attributes to control the ordering of tests as of May 2020. As such, we hope to provide support for accommodating OD tests not just for regression testing algorithms but for a variety of different testing tasks. We believe that our work

on making regression testing algorithms dependent-test-aware is an important step in this direction.

5.4 Evaluation Metrics

In our evaluation, we focus on the reduction in the number of OD-test failures in enhanced orders over unenhanced orders as well as the potential increase in testing time due to the additional tests that we may need for test selection and test parallelization (Section 4.4). Concerning test prioritization algorithms, prior work commonly evaluates them using Average Percentage of Faults Detected (APFD) [55, 66]. Traditionally, researchers evaluate the quality of different test prioritization algorithms by seeding faults/mutants into the code under test, running the tests on the faulty code, and then mapping what tests detect which seeded fault/mutant. To compare the orders produced by the different test prioritization algorithms, researchers would measure APFD for each order, which represents how soon in an order comes a test that detects each fault/mutant.

In our work, we use real-world software that does not have failing tests due to faults in the code, ensured by choosing versions where the tests pass in the original order (Section 2.3). We do not seed faults/mutants as we want to capture the effects of real software evolution. As such, we do not and cannot measure APFD because there would be no test failures due to faults in the code under test; APFD would simply be undefined in such cases. Any test failures that we observe would be either OD-test failures or test failures due to other sources of flakiness.

6 THREATS TO VALIDITY

A threat to validity is that our evaluation considers only 11 modules from 8 Java projects. These modules may not be representative causing our results not to generalize. Our approach might behave differently on different programs, such as ones from different application domains or those not written in Java.

Another threat to validity is our choice of 12 traditional regression testing algorithms. Future work could evaluate other algorithms based on static code analysis, system models, history of known faults, and test execution results, and so forth. Future work could also enhance other techniques that run tests out of order, such as mutation testing [58, 68, 69], test factoring [23, 57, 65], and experimental debugging techniques [62, 67, 70].

Another threat is the presence of non-order-dependent flaky tests when we compute test dependencies. Non-order-dependent tests may also affect the metrics that we use for the regression testing algorithms (e.g., coverage and timing of tests can be flaky), thereby affecting the produced orders. We mitigate this threat by filtering out non-order-dependent flaky tests through the rerunning of tests in the original order. We also suggest that developers use tools [43, 59] to identify these tests and remove or fix them; a developer does not gain much from a test whose failures they would ignore. In future work, we plan to evaluate the impact of these issues and to improve our algorithms to directly handle these non-order-dependent tests.

7 RELATED WORK

7.1 Test Dependence Definitions and Studies

Treating test suites as sets of tests [34] and assuming test independence is common practice in the testing literature [20, 32, 35, 37, 38, 40, 41, 49–51, 55, 56, 61, 62, 69, 70]. Little prior research has considered test dependencies in designing or evaluating testing techniques.

Bergelson and Exman [18] described a form of test dependencies informally: given two tests that each pass, the composite execution of these tests may still fail. That is, if t_1 and t_2 executed by themselves pass, executing the sequence $\langle t_1, t_2 \rangle$ in the same context may fail. In the context of databases, Kapfhammer and Soffa [39] formally defined independent test suites and distinguished them from other suites. However, this definition focuses on program and database states that may not affect actual test outcomes [15, 16, 39]. We use a definition of OD test [71] based on test outcomes. Huo and Clause [36] studied assertions that depend on inputs not controlled by the tests themselves. These assertions are one source of OD-test failures; other sources include unexpected exceptions being thrown.

Some prior work [14, 27, 46] studied the characteristics of flaky tests — tests that have non-deterministic outcomes. Test dependencies do not imply non-determinism: a test may non-deterministically pass/fail without being affected by any other test. Non-determinism does not imply test dependencies: a program may have no sources of non-determinism, but two of its tests can be dependent. One line of work that mentions test determinism as an assumption defines it too narrowly, with respect to threads but ignoring code interactions with the external environment [32, 51]. Furthermore, a test may deterministically pass/fail even if it performs non-deterministic operations. Unlike our work in this paper, the preceding prior work neither evaluated the impact of OD-test failures on regression testing techniques, nor proposed an approach to accommodate them.

7.2 Techniques to Manage Test Dependence

Only a few techniques and tools have been developed to prevent or accommodate the impact of test dependence. Some testing frameworks provide mechanisms for developers to specify the order in which tests must run. For tests written in Java, JUnit since version 4.11 supports executing tests in lexicographic order by test method name [5], while TestNG [12] supports execution policies that respect programmer-written dependence annotations. Other frameworks such as DepUnit [10], Cucumber [9], and Spock [1] also provide similar mechanisms for developers to manually define test dependencies. These test dependencies specified by developers could be used directly by our general approach, or to improve the test dependencies computed using automatic tools (by adding missing or removing unnecessary test dependencies). Haidry and Miller [30] proposed test prioritization that prioritizes tests based on their number of OD tests, hypothesizing that running tests with more test dependencies is more likely to expose faults. In our work, we enhance existing, traditional test prioritization algorithms (along with other regression testing algorithms) to satisfy test dependencies.

Various techniques have been proposed to find test dependencies automatically. PolDet [29] finds tests that pollute the shared state — tests that modify some location on the heap shared across tests or on the file system. However, the tests that PolDet detects do not fail themselves, and while they pollute shared state, no other tests currently in the test suite may fail due to that polluted shared state. In contrast, we use DTDetector [71], which not only dynamically identifies dependee polluting tests but also identifies the tests that depend on and fail due to the polluted state (OD tests). DTDetector can also

identify test dependencies caused by other factors, such as database and network access. Biagiola et al. [19] studied test dependencies within web applications, noting the challenges in tracking the test dependencies in shared state between server- and client-side parts of web applications. They proposed a technique for detecting these test dependencies based on string analysis and natural language processing. Waterloo et al. [64] built a static analysis tool to detect inter-test and external dependencies. Electric Test [17] over-approximates test dependencies by analyzing data dependencies of tests. The tool is not publicly available, but the authors informed us that the work has evolved into PRADET [26]. Building upon Electric Test's analysis, Gambi et al. [26] created PRADET, a tool that outputs test dependencies. We have tried using PRADET but encountered issues with it on some of our projects. Lam et al. [43] released iDFlakies, a toolset similar to DTDetector. Both toolsets come with similar default strategies, namely to run tests in different orders as to detect OD tests. Lam et al. also released a dataset of flaky tests, with ~50% OD tests. We use their dataset for our evaluation.

VMVM [16] is a technique for accommomdating test dependencies through a modified runtime environment. VMVM's runtime resets the reachable shared state (namely, the parts of the in-memory heap reachable from static variables in Java) between test runs. The restoration is all done within one JVM execution of all tests, providing benefits of isolation per test without needing to start/stop separate JVMs per test. Similar to the work by Kapfhammer and Soffa [39], VMVM considers a test as an OD test if it accesses a memory location that has been written by another test, being neither necessary nor sufficient to affect the test outcome. Note that VMVM does not aim to detect OD tests, and resets shared state for *all* tests, regardless of pollution or not. In our work, we focus on coping with OD tests run in a single, standard JVM, and we propose enhanced regression testing algorithms to accommodate test dependencies as to reduce OD-test failures. Nemo [45] is a tool that considers test dependencies for test suite minimization. Our work aims at different regression testing techniques and can be used in conjunction with Nemo. iFixFlakies [60] is a tool for automatically fixing OD tests. iFixFlakies relies on finding code from "helpers", which are tests in the existing test suite that prevent OD-test failures when run before an OD test, to fix the OD tests. However, iFixFlakies cannot fix OD tests if these OD tests have no helpers. Developers would still need to use our enhanced algorithms that avoid OD-test failures even when there are no helpers or when they plan on using test dependencies to run their tests faster.

8 CONCLUSION

Test suites often contain OD tests, but traditional regression testing techniques ignore test dependencies. In this work, we have empirically investigated the impact of OD tests on regression testing algorithms. Our evaluation results show that 12 traditional, dependent-test-unaware regression testing algorithms produce orders that cause OD-test failures in 82% of the human-written test suites and 100% of the automatically-generated test suites that contain OD tests. We have proposed a general approach that we then use to enhance the 12 regression testing algorithms so that they are dependent-test-aware. We have made these 12 algorithms and our general approach publicly available [8]. Developers can use the enhanced algorithms with test dependencies manually provided or automatically computed using various tools, and the enhanced algorithms are highly effective in reducing the number of OD-test failures. Our proposed enhanced algorithms produce orders that result in 80% fewer OD-test failures, while being 1% slower to run than the unenhanced algorithms.

ACKNOWLEDGMENTS

We thank Jonathan Bell, Sandy Kaplan, Martin Kellogg, Darko Marinov, and the anonymous referees who provided helpful comments on a draft of the paper. This material is based on research sponsored by DARPA under agreement numbers FA8750-12-2-0107, AFRL FA8750-15-C-0010, and FA8750-16-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. This work was also partially supported by NSF grant numbers CNS-1646305, CNS-1740916, CCF-1763788, OAC-1839010, CNS-1564274, and CCF-1816615. Tao Xie is affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

REFERENCES

[1] 2011. Spock Stepwise. https://www.canoo.com/blog/2011/04/12/spock-stepwise.
[2] 2012. JUnit and Java 7. http://intellijava.blogspot.com/2012/05/junit-and-java-7.html.
[3] 2013. JUnit test method ordering. http://www.java-allandsundry.com/2013/01.
[4] 2013. Maintaining the order of JUnit3 tests with JDK 1.7. https://coderanch.com/t/600985/engineering/Maintaining-order-JUnit-tests-JDK.
[5] 2013. Test execution order in JUnit. https://github.com/junit-team/junit/blob/master/doc/ReleaseNotes4.11.md#test-execution-order.
[6] 2016. Running your tests in a specific order. https://www.ontestautomation.com/running-your-tests-in-a-specific-order
[7] 2019. Run tests in parallel using the Visual Studio Test task. https://docs.microsoft.com/en-us/azure/devops/pipelines/test/parallel-testing-vstest.
[8] 2020. Accommodating Test Dependence Project Web. https://sites.google.com/view/test-dependence-impact
[9] 2020. Cucumber Reference - Scenario hooks. https://cucumber.io/docs/cucumber/api/#hooks.
[10] 2020. DepUnit. https://www.openhub.net/p/depunit.
[11] 2020. SLOCCount. https://dwheeler.com/sloccount
[12] 2020. TestNG. http://testng.org.
[13] 2020. TestNG Dependencies. https://testng.org/doc/documentation-main.html#dependent-methods.
[14] Stephan Arlt, Tobias Morciniec, Andreas Podelski, and Silke Wagner. 2015. If A fails, can B still succeed? Inferring dependencies between test results in automotive system testing. In *ICST*. Graz, Austria, 1–10.
[15] Jonathan Bell. 2014. Detecting, isolating, and enforcing dependencies among and within test cases. In *FSE*. Hong Kong, 799–802.
[16] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *ICSE*. Hyderabad, India, 550–561.
[17] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *ESEC/FSE*. Bergamo, Italy, 770–781.
[18] Benny Bergelson and Iaakov Exman. 2006. Dynamic test composition in hierarchical software testing. In *Convention of Electrical and Electronics Engineers in Israel*. Eilat, Israel, 37–41.
[19] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. 2019. Web test dependency detection. In *ESEC/FSE*. Tallinn, Estonia, 154–164.
[20] Lionel C. Briand, Yvan Labiche, and S. He. 2009. Automating regression test selection based on UML designs. *Information and Software Technology* 51, 1 (January 2009), 16–30.
[21] Koen Claessen and John Hughes. 2000. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*. Montreal, Canada, 268–279.
[22] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An automatic robustness tester for Java. *Software: Practice and Experience* 34, 11 (September 2004), 1025–1050.
[23] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *FSE*. Portland, OR, USA, 253–264.
[24] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing test cases for regression testing. In *ISSTA*. Portland, OR, USA, 102–112.

[25] Gordon Fraser and Andreas Zeller. 2011. Generating parameterized unit tests. In *ISSTA*. Toronto, Canada, 364–374.

[26] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *ICST*. Vasteras, Sweden, 1–11.

[27] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. 2015. Making system user interactive tests repeatable: When and what should we control?. In *ICSE*. Florence, Italy, 55–65.

[28] Alex Groce, Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause reduction for quick testing. In *ICST*. Cleveland, OH, USA, 243–252.

[29] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*. Baltimore, MD, USA, 223–233.

[30] Shifa Zehra Haidry and Tim Miller. 2013. Using dependency structures for prioritization of functional test suites. *IEEE Transactions on Software Engineering* 39, 2 (2013), 258–275.

[31] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *SCAM*. 1–23.

[32] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression test selection for Java software. In *OOPSLA*. Tampa Bay, FL, USA, 312–326.

[33] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *ICSE*. Florence, Italy, 483–493.

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

[35] Hwa-You Hsu and Alessandro Orso. 2009. MINTS: A general framework and tool for supporting test-suite minimization. In *ICSE*. Vancouver, BC, Canada, 419–429.

[36] Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*. Hong Kong, 621–631.

[37] Bo Jiang, Zhenyu Zhang, W. K. Chan, and T. H. Tse. 2009. Adaptive random test case prioritization. In *ASE*. Auckland, NZ, 233–244.

[38] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*. Orlando, Florida, 467–477.

[39] Gregory M. Kapfhammer and Mary Lou Soffa. 2003. A family of test adequacy criteria for database-driven applications. In *ESEC/FSE*. Helsinki, Finland, 98–107.

[40] Jung-Min Kim and Adam Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*. Orlando, Florida, 119–129.

[41] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. 2013. Optimizing unit test execution in large software programs using dependency analysis. In *APSys*. Singapore, 19:1–19:6.

[42] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *ISSTA*. Beijing, China, 101–111.

[43] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*. Xi'an, China, 312–322.

[44] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: Continuous prioritization for continuous integration. In *ICSE*. Gothenburg, Sweden, 688–698.

[45] Jun-Wei Lin, Reyhaneh Jabbarvand, Joshua Garcia, and Sam Malek. 2018. Nemo: Multi-criteria test-suite minimization with integer nonlinear programming. In *ICSE*. Gothenburg, Sweden, 1039–1049.

[46] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*. Hong Kong, 643–653.

[47] John Micco. 2016. Flaky tests at Google and how we mitigate them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[48] John Micco. 2017. The state of continuous integration testing @ Google. https://ai.google/research/pubs/pub45880

[49] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. 2007. Parallel test generation and execution with Korat. In *ESEC/FSE*. Dubrovnik, Croatia, 135–144.

[50] Agastya Nanda, Senthil Mani, Saurabh Sinha, Mary Jean Harrold, and Alessandro Orso. 2011. Regression testing in the presence of non-code changes. In *ICST*. Berlin, Germany, 21–30.

[51] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *FSE*. Newport Beach, CA, USA, 241–251.

[52] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *ICSE*. Minneapolis, MN, USA, 75–84.

[53] Md Tajmilur Rahman and Peter C. Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *ESEC/FSE*. Lake Buena Vista, FL, USA, 857–862.

[54] Gregg Rothermel, Sebastian Elbaum, Alexey G. Malishevsky, Praveen Kallakuri, and Xuemei Qiu. 2004. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology* 13, 3 (July 2004), 277–331.

[55] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (October 2001), 929–948.

[56] Matthew J. Rummel, Gregory M. Kapfhammer, and Andrew Thall. 2005. Towards the prioritization of regression test suites with data flow information. In *SAC*. Santa Fe, NM, USA, 1499–1504.

[57] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. 2005. Automatic test factoring for Java. In *ASE*. Long Beach, CA, USA, 114–123.

[58] David Schuler, Valentin Dallmeier, and Andreas Zeller. 2009. Efficient mutation testing by checking invariant violations. In *ISSTA*. Chicago, IL, USA, 69–80.

[59] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*. Chicago, IL, USA, 80–90.

[60] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE*. Tallinn, Estonia, 545–555.

[61] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively prioritizing tests in development environment. In *ISSTA*. Rome, Italy, 97–106.

[62] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*. Lugano, Switzerland, 314–324.

[63] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An empirical study of flaky tests in Android apps. In *ICSME, NIER Track*. Madrid, Spain, 534–538.

[64] Matias Waterloo, Suzette Person, and Sebastian Elbaum. 2015. Test analysis: Searching for faults in tests. In *ASE*. Lincoln, NE, USA, 149–154.

[65] Ming Wu, Fan Long, Xi Wang, Zhilei Xu, Haoxiang Lin, Xuezheng Liu, Zhenyu Guo, Huayang Guo, Lidong Zhou, and Zheng Zhang. 2010. Language-based replay via data flow cut. In *FSE*. Santa Fe, NM, USA, 197–206.

[66] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (March 2012), 67–120.

[67] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 3 (February 2002), 183–200.

[68] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. 2013. Faster mutation testing inspired by test prioritization and reduction. In *ISSTA*. Lugano, Switzerland, 235–245.

[69] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2012. Regression mutation testing. In *ISSTA*. Minneapolis, MN, USA, 331–341.

[70] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*. Indianapolis, IN, USA, 765–784.

[71] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA*. San Jose, CA, USA, 385–396.

[72] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. 2011. Combined static and dynamic automated test generation. In *ISSTA*. Toronto, Canada, 353–363.