Test Dependence: Theory and Manifestation

Jochen Wuttke Kıvanç Muşlu Sai Zhang David Notkin
Computer Science & Engineering
University of Washington
Seattle, WA, USA
{wuttke|kivanc|szhang|notkin}@cs.washington.edu

ABSTRACT

Test dependence arises when executing a test in different environments causes it to return different results. In this paper, we show through a set of substantive real-world examples that test dependence arises in practice. We also show that test dependence can have potentially costly repercussions such as masking program faults, and can be hard to identify unless explicitly searched for: We found a dependence that only manifests when a sequence of three tests are run in a specified, non-default order.

We formally define test dependence in terms of test suites as ordered sequences of tests along with explicit environments in which these tests are executed. We use this formalization to formulate the concrete problem of detecting dependence in test suites, prove that a useful special case is NP-complete, and propose an initial algorithm that approximates solutions to this problem.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]:

Keywords

Software testing; test dependence; test selection; test prioritization

1. INTRODUCTION

Informally, *dependent tests* produce different test results when executed in different contexts. It is easy to construct an example of dependence between two tests `A` and `B`, where running `A` and then `B` leads to both tests passing, while running `B` and then `A` leads to either or both tests failing—the order of applying the tests, in this case, changes the execution context.

Definitions in the testing literature are generally clear that the conditions under which a test is executed may affect its result. The importance of context in testing has been explored in some depth in some domains including databases [11, 4, 17], with results about test generation, test adequacy criteria, etc., and mobile applications [32]. For the database domain, Kapfhammer and Soffa formally define and distinguish independent test suites from those that are *non-restricted* and thus "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" [17, p. 101].

At the same time, there is little focus on the core issues of test dependence itself. Is this because test dependence does not arise in practice (beyond domains such as databases)? Is it because, even if-and-when it does arise, there are few if any repercussions? Is it because it is difficult to notice if-and-when it arises?

1.1 Manifest Test Dependence

To explore these questions, we consider a narrow characterization of test dependence that:

- Adopts the results of the default, usually implicit, order of execution of a test suite as the *expected results*.
- Asserts *test dependence* when there is a possibly reordered subsequence of the original test suite that, when executed, has at least one test result that differs from the expected result for that test.

That is, we focus on a *manifest* perspective of test dependence, requiring a *concrete* variant of the test suite that *dynamically* produces different results from the expected. Our definition differs from that of Kapfhammer and Soffa by considering test results rather than program and database states. As we discuss later, considering only manifest test dependences allows us to more easily situate this research in the empirical domain.

1.2 Examples and Repercussions

We have identified a number of substantive examples of test suites from fielded programs that manifest dependences. We examined six projects and found in their human-written test suites a total of 75 dependent tests (1.4%). For the same set of programs, we also generated test suites automatically using Randoop [23] and found that on average 14% of the generated tests are dependent.

By analyzing these examples of test dependence, we identified three categories of problems that can arise due to the presence of dependent tests. First, test suites that unexpectedly contain dependent tests can *mask faults in a program.* We present examples where executing a test suite in the default order does not expose the fault, whereas executing the

same test suite in a different order does. Second, test suites that unexpectedly contain dependent tests can *conceal weaknesses in the test suite* itself. We present examples where exposing dependent tests can identify situations where some tests do not perform proper initialization. Third, a test suite containing undocumented test dependences can lead to *spurious bug reports*. We present an example where it took the developers more than a month to realize that the test dependences were intentional, allowing them to close the bug report without a change to the system.

1.3 Test Execution Environment

Our examples highlight varying execution environments as the unsurprising central cause of test dependence. Specifically, when a test is executed in different environments—global variables with different values, differences in the file system, differences in data obtained from web services, etc.—it has the potential to return a different result. Most of the dependences we see in our examples ultimately stem from incorrect or incomplete initialization of the program environment.

Why does this happen? Especially given frameworks such as JUnit that facilitate the process of clean setup by providing means to automatically execute methods (`setUp()` and `tearDown()` in JUnit 3.x, and methods annotated with `@Before` and `@After` in JUnit 4.x) that should handle all common setup and clean-up between test cases.

It appears that the answer is that developers are as likely to make mistakes when writing tests as when they are writing other code. And while frameworks make it easier to get environment setup right, they cannot ensure that it is done properly. Like with other code, this means that tests in some cases will have unintended and unexpected behaviors. And as programs increase in complexity, so may tests, which may increase the frequency of such problems in tests, which may in turn increase the frequency of test dependence.

Another situation in which the underlying test execution context can unexpectedly change is when a tool or technique that takes a test suite as input is used. Examples of such techniques include test selection techniques (that identify a subset of the input test suite that will guarantee some properties during regression testing) [12], test prioritization techniques (that reorder the input to increase the rate of fault detection) [9], test parallelization techniques (that schedule the input for execution across multiple CPUs), test factoring [27] and test carving [8] (which take system tests as input and create unit tests and differential unit tests, respectively), etc.

Of these techniques, we are most concerned about those that modify the organization of test suites, rather than the tests they contain. Many such downstream testing techniques implicitly assume that there are no test dependences in the input test suite. Our concern is that this assumption can cause *delayed problems* in the face of latent test dependence in the input. As an example, test selection may report a subsequence of tests that do not return the same results as they do when executed originally, as part of the full suite.

1.4 Contributions

At its heart, this paper addresses and questions conventional wisdom about the test independence assumption. It is intended to balance a precise characterization of test dependence with substantive empiric examples and concerns. The contributions of the paper include:

- A precise formalization of test dependence in terms of test suites as ordered sequences of tests and explicit execution environments for test suites, which enables reasoning about test dependence as well as a proof that finding *manifest* dependences is an NP-complete problem (Section 3).
- Examples from fielded software of test suites where manifest test dependences lead to identifiable concerns with the underlying programs or test suites (Section 4).
- Motivation for and presentation of our initial approximate algorithms and a supporting tool for identifying test dependences (Section 5).

Although we provide evidence that test dependence unexpectedly arises in practice, a broad study of how often test dependences arise, and the costs that these may lead to, is beyond the scope of this paper. We conclude the paper with a set of open questions addressing this and other possible concerns in Section 6.

2. RELATED WORK

Denoting a group of test cases as a "suite of test programs" began around the mid-1970's [3, p. 217]; similar terms include "testcase dataset" [21] and "scenario," which an IEEE Standard defines as "groups of test cases; synonyms are script, set, or suite" [14, p. 10]. Treating test suites explicitly as *mathematical sets* of tests dates at least to Howden [13, p. 554] and remains common in the literature. The execution order of tests in a suite is usually addressed implicitly or informally, suggesting that the potential of executing a given test in different contexts is immaterial to those results: that is, test independence is assumed.

2.1 Test Dependence

In addition to the work by Kapfhammer and Soffa [17], there are a handful of categorical references that acknowledge that tests can be dependent based on context, suggesting ways to document or find situations where the independence assumption fails to hold.

The IEEE Standard for Software and System Test Documentation (829-1998) §11.2.7, "Intercase Dependencies," says in its entirety: "List the identifiers of test cases that must be executed prior to this test case. Summarize the nature of the dependences" [14]. The succeeding version of this standard (829-2008) adds a single sentence: "If test cases are documented (in a tool or otherwise) in the order in which they need to be executed, the Intercase Dependencies for most or all of the cases may not be needed" [15].

McGregor and Korson discuss interaction tests that are intended to identify "two methods that may directly or indirectly cause each other to produce incorrect results" and suggest constructing such interaction tests by identifying the values shared via parameter passing between methods that two or more test cases share [20, p .69].

Bergelson and Exman characterize a form of test dependence explicitly: given two tests that each pass, the composite execution of these tests may still fail [1, p. 38]. That is, if $\langle t_1 \rangle$ executed by itself passes and $\langle t_2 \rangle$ executed by itself passes, executing the sequence $\langle t_1, t_2 \rangle$ in the same context may fail.

Some practitioners acknowledge test dependence as a possible, albeit low probability, event:

> Unit testing ... requires that we test the unit in isolation. That is, we want to be able to say, *to a very high degree of confidence* [emphasis added], that any actual results obtained from the execution of test cases are purely the result of the unit under test. The introduction of other units may color our results [31].

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

A few approaches allow developers to annotate dependent tests and provide supporting mechanisms to ensure that the test execution framework respects those annotations. DepUnit[1] 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[2] is a testing framework intended to improve upon JUnit, and allows dependence annotations and supports a variety of execution policies such as sequential execution in a single thread, execution of a single test class per thread, etc. that respect these dependences. What distinguishes our work from these approaches is that, while they allow dependences to be made explicit and respected during execution, they do not help developers *identify* dependences. A tool that finds dependences (Section 5.2) could co-exist with such frameworks by generating annotations for them.

2.2 Test Prioritization

Test prioritization seeks to reorder a test suite to detect software defects more quickly, and is the example of downstream testing tools that we focus on most closely both because it is characteristic of the other tools (in the dimensions we address) and also because of its focus on reordering (perhaps the most common way to change the execution environment of a test).

Early work in test prioritization [35, 25] laid the foundation for the most commonly used problem definition: consider the set of all permutations of a test suite and find the best award value for an objective function over that set [9]. The most common objective functions favor permutations where more faults in the underlying program are found with running fewer tests. A number of results carefully study various prioritization algorithms empirically, most by Rothermel and colleagues, spanning over a decade [25, 7, *et alia*]. These evaluations are based in part on the assumption that the set of faults in the underlying program is known beforehand; the possibility that test dependence may unmask additional faults in the program is not studied.

Kim and Porter proposed a technique that uses the history of test cases run in prior regression tests to prioritize those that have not yet run for creating new regression test suites [19]. Whether tests were executed is essential to the technique; the results of specific tests are not.

Echelon defines a heuristic that exploits both a mapping between tests and executed program paths and also a binary differencing between two program versions to select a subset of tests intended to quickly identify program faults [29]. Test dependence is not considered in the approach.

[1] `https://code.google.com/p/depunit/`
[2] `http://testng.org/`

Test independence is explicitly asserted as a requirement for prioritization by Rummel et al.:

> A test suite contains a tuple of tests $\langle T_1 \dots T_R \rangle$ that execute in a specified order. We require that each test is independent so that there are no test execution ordering dependencies. This requirement enables our prioritization algorithm to reorder the tests in any sequence that maximizes the suite's ability to isolate defects. The assumption of test dependence is acceptable because the JUnit test execution framework provides `setUp` and `tearDown` methods that execute before and after a test case and can be used to clear application state [26, p. 1500].

2.3 Syntactic and Semantic Test Dependencies

Dependences in testing are most often considered to be syntactic dependences between program units, for example methods calling other methods, and classes using other classes [1, 2]. *Syntactic* dependence here means that a unit `A` cannot be compiled and executed without unit `B` being present. If we test such a unit `A` without convincing ourselves first that `B` is correct, a test failure for `A` is harder to interpret, because it could just as well indicate a fault in `B`.

Zhang and Ryder extend this notion to *semantic* dependences, which is closer to our approach [37]. They use a notion of "test outcome" to determine whether or not syntactically dependent classes or methods can influence each others results, and consider only those that can to be semantically dependent. They give an informal definition of what it means for the execution of a test to influence the outcome of another test. We define this precisely, and we also define manifest test dependence in terms of execution environments and test execution order rather than in terms of code use.

Santelices et al. define a formal model of how changes might interact at the source code level and present a technique for detecting such interactions that arise at run-time [28]. In contrast to our approach, they identify changes that interact rather than tests that depend upon each other.

Another kind of dependence helps address the testing of configurable software, which can be combinatorial with respect to the set of configurable options [5, 6]. The dependences considered in this approach are not between tests, but rather within the configuration option space.

3. THEORY

A standard textbook states that "[a] test case includes not only input data but also any relevant *execution conditions* ..." [24, p. 152, emphasis added]. This characterization is consistent with the example that piqued our interest in test dependence: we serendipitously identified a bug in an open-source system when we found that running individual tests one-by-one—each in a newly initialized environment—produced different results from running the entire test suite normally (*i.e.*, with a single initialized environment followed by the sequential execution of each test in order) [22]. These tests shared global variables, and the test results varied depending on the values stored in these variables. That is, relevant execution conditions—specifically, pertinent parts of the implicit *environment* comprising global variables, the file system, operating system services, etc.—were neglected.

```
test1 { v1 = 4 }
test2 { assert v1==4 }
```

(a) Direct dependence

```
test1 { v1 = 1 }
test2 { v2 = v1 + 1 }
...
testn {
  assert vn-1 == n-1 }
```

(b) Chain dependence

Figure 1: Examples for basic causes of test dependences

To characterize the relevant execution conditions precisely, our formalism below explicitly represents the notions of (a) the order in which test cases are executed and (b) the environment in which a test suite is executed.

Consider two examples of how test dependences arise in terms of order and environment (Figure 1). In the leftmost example, `test2` checks the value of a variable that has been assigned elsewhere. If the tests are executed in the order $\langle$`test1`, `test2`$\rangle$, both tests will pass, while running `test2` first will make it fail. The rightmost example extends this principle to multiple tests. While none of the $n-1$ tests prior to `testn` will fail, they all must execute in this particular order for `testn` to pass.

The global variables involved are usually buried deep in the program code, and the assertions do not directly check them, but rather check values that have been computed from them. In any non-trivial real-world program, this deep nesting effectively hides potential dependencies from developers, and they may only become aware of them when a subtle bug leads them there. Therefore, we explicitly distinguish potential test dependences (Definition 6)—those that could cause a variation in test suite results under *some* environment and order—and manifest test dependences (Definition 7)—those that are guaranteed to cause a variation in test suite results under a *specific* environment and order.

3.1 Definitions

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

Definition 1 (Environment). *An* environment $\mathbf{E}$ *for the execution of a test consists of all values of global variables, files, operating system services, etc. that can be accessed by the test and program code exercised by the test case.*

Definition 2 (Test). *A test is a sequence of program statements, executed with fixed, well-defined inputs, and an oracle that decides whether a test passes or fails.*

Simplifying from Staats et al. [30], and without loss of generality, we consider an oracle to be a boolean predicate over tests and environments.

Definition 3 (Test Suite). *A test suite T is an n-tuple (i.e., ordered sequence) of tests $\langle t_1, t_2, \ldots, t_n \rangle$.*

Definition 4 (Test Execution). *Let T be a test suite and $\mathcal{E}$ the set of all possible environments. The function $\varepsilon : T \times \mathcal{E} \to \mathcal{E}$ is called test execution. ε maps the execution of a test $t \in T$ in an environment $\mathbf{E} \in \mathcal{E}$ to the new (potentially updated) environment $\mathbf{E}'$.*

For the execution of test suites $T = \langle t_1, t_2, \ldots, t_n \rangle$ we use the shorthand $\varepsilon(T, \mathbf{E})$ for $\varepsilon(t_n, \varepsilon(t_{n-1}, \ldots \varepsilon(t_1, \mathbf{E}) \ldots))$.

Definition 5 (Test Result). *The result of a test t executed in an environment $\mathbf{E}$, denoted $R(t|\mathbf{E})$ (and sometimes referred to as an oracle judgment), is defined by the test's oracle and is either PASS or FAIL.*

The result of a test suite $\langle t_1, \ldots, t_n \rangle$, executed in an environment $\mathbf{E}$, denoted $R(\langle t_1, \ldots, t_n \rangle|\mathbf{E})$ is a sequence of results $\langle o_1, \ldots, o_n \rangle$ with $o_i \in \{PASS, FAIL\}$.

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

Definition 6 (Potential Test Dependence). *Given a test suite T, a test $t_l \in T$ is* potentially dependent *on test $t_k \in T$, if and only if $\exists \mathbf{E} : R(T|\mathbf{E}) = \langle o_1, \ldots, o_n \rangle \wedge R(\langle t_k, t_l \rangle|\mathbf{E}) = \langle o_k, o_l \rangle \wedge R(t_l|\mathbf{E}) = \neg o_l$. We write $t_k \prec t_l$ when t_l is potentially dependent on t_k.*

This definition is *dynamic* because dependence arises only if there exists an environment in which actual test results would differ. It is *potential* as it only requires the existence of such an environment, but does not guarantee that the test suite will ever be executed in the context of such an environment.

We refine this definition of dependence to require a concrete environment guaranteed to *manifest* a dependence:

Definition 7 (Manifest Dependence). *Given a test suite T, two dependent tests $t_i, t_j \in T$, the dependence $t_i \prec t_j$* manifests *in a given environment $\mathbf{E}$ if $\exists S \subseteq T : t_i, t_j \in S \wedge R(T|\mathbf{E}) = \langle o_1, \ldots, o_n \rangle \wedge R(S|\mathbf{E}) = \langle \ldots, o_i, o_j \rangle \wedge R(t_j|\mathbf{E}) = \neg o_j$. We write $t_i \prec_{\mathbf{E}} t_j$ for manifest dependence.*[3]

Note that the dependent tests t_i and t_j do not have to be adjacent in the original test suite, but that they must be adjacent in the shortest test suite that manifests the dependence.

The intuition behind manifest dependences is that in practice we do not construct arbitrary environments to execute tests in. Rather, we use the natural environment $\mathbf{E}_0$ provided by frameworks such as JUnit, and the only modifications of this environment happen through the tests and the tested code. Hence, potential dependences manifest only if there is a sequence of tests S^* whose execution $\varepsilon(S^*, \mathbf{E}_0)$ produces the environment $\mathbf{E}'$ that will reveal the dependency. The algorithm we propose in Section 5 detects dependences by running tests and checking for different test results, hence it can only detect manifest dependences. To improve algorithms that are affected by test dependences, we are interested in the shortest test suite $S^* \subseteq T$ that manifests a dependence, because these define the partial order of test execution that such techniques must respect.

In later sections we often talk about executing tests in isolation, or executing all tests in a test suite in isolation. This is an important approximation to detecting test dependences.

Definition 8 (Test Isolation). *The result of executing a test t in isolation, given an environment $\mathbf{E}_0$ is the result $R(t|\mathbf{E}_0)$ of executing that test in the given environment.*

The result of executing all tests in a test suite $\langle t_1, \ldots, t_m \rangle$ in isolation is the sequence of results $\langle R(t_1|\mathbf{E}_0), \ldots, R(t_n|\mathbf{E}_0) \rangle$.

3.2 Detecting Dependent Tests

[3] $S \subseteq T$ means that S is a subsequence of T.

From a practical perspective, techniques that affect the ordering of test suites must respect dependences. Otherwise their results cannot be interpreted correctly in the presence of dependences. Detecting dependences in existing test suites is thus an interesting problem. In the following, we first give a precise definition of the problem of detecting dependent tests, and then prove that in general this problem is NP-complete. In Section 5 we outline an algorithm that approximates solutions efficiently.

DEFINITION 9 (DEPENDENT TEST DETECTION PROBLEM). *Given a set suite $T = \langle t_1, \ldots, t_n \rangle$ and an environment $\mathbf{E}_0$, for a given test $t_i \in T$, is there a test suite $S \subseteq T$ that manifests a test dependence involving t_i?*

We prove that this problem is NP-hard by reducing the NP-complete Exact Cover problem to the Dependent Test Detection problem [18]. Then we provide a linear time algorithm to verify any answer to the question. Together these two parts prove the the Dependent Test Detection Problem is NP-complete.

THEOREM 1. *The problem of finding a test suite that manifests a dependence is NP-hard.*

PROOF. In the Exact Cover problem, we are given a set $X = \{x_1, x_2, x_3, \ldots, x_m\}$ and a collection S of subsets of X. The goal is to identify a sub-collection S^* of S such that *each* element in X is contained in *exactly* one subset in S^*.

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

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

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

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

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

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

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

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

3.3 Discussion

This formalism has dual intents: to lay a foundation for reasoning about test dependence in a precise way; and to be consistent with and to allow for approximate and practical algorithms and tools (Section 5).

This second intent, of course, requires a balance of theory and practice. First, the dynamic nature of our our view on dependences allows us to avoid the complexity issues that come with a static approach. With a static approach, it would be essential to decide how to address undecidability. The most likely and common approach being to choose soundness with respect to all possible executions and accepting the consequent imprecision of the analysis. Second, our focus on manifest dependence, when realized in a tool will only identify true positives, although it may miss some dependences (false negatives). It is often easier to have tools with this kind of property accepted by practitioners than some other kinds. Third, the manifest test dependence problem is NP-complete; although that is daunting (but less so than undecidability), approximate algorithms can be defined for large classes of NP-complete problems.

The examples in the following section and the algorithm and tool following that give a better flavor for why we made these decisions. The degree to which these are the "right" (or at least effective) decisions is itself an empirical question beyond the scope of this paper.

4. MANIFESTATIONS

Dependent tests reach beyond theory and appear in real-world programs. In some cases, they are intentional, developers are aware of them and document them, but in other cases they are inadvertent. Test dependence can cause problems, not only when test suites are reordered, but even when they are executed in the intended order. This section presents concrete examples of test dependence found in well-known open source programs. Figure 2 summarizes the projects we studied and the results: The table summarizes the number of tests in the suites produced by the developers (MT), the number of tests we generated automatically with Randoop (AT), and the corresponding numbers of dependent tests in those test suites (MTD and ATD, respectively). The discussion of the examples in this section is distinguished by the problems caused by test dependence (*Kind*): when faults are masked because tests make incorrect assumptions about the global environment (Section 4.1); when tests do not respect required initialization protocols (Section 4.2); and when undocumented test dependence leads to spurious bug reports (Section 4.3). We also describe dependent tests in an automatically-generated test suite (Section 4.4). While this list—and associated set of examples—certainly is not exhaustive, it shows that there are several classes of dependence-related problems that have practical relevance.

4.1 Masking Faults

Masking is a particularly perplexing problem caused by dependence. The negative effect of masking is that it hides a fault in the program, *exactly* when the test suite is executed in its default order. Masking occurs when a test case t (a) *should* reveal a fault, (b) only does so when executed in a specific environment $\mathbf{E}_R$, but (c) tests executed before t in a test suite always generate environments different from $\mathbf{E}_R$. More precisely and without loss of gen-

Project	MT	MTD	AT	ATD	Kind	Revision
CLI	206	2	2821	20	Masking Faults	757051
JodaTime	3875	3	2663	711	Masking Faults	b609d7d66d
Crystal	75	18	2542	20	Test Structure	trunk version
XML Security	108	3	2947	925	Test Structure	version 1.0.4
Eclipse SWT	80*	49	–	–	Spurious Bug	version 3.0
Beanutils	1060	0	2692	299	Test Structure	version 1.8.3

Figure 2: Summary of all examples. Columns "MT" and "AT" are the numbers of human-written and automatically generated tests, respectively. Columns "MTD" and "ATD" are the numbers of test dependences in the corresponding test suites. Column "Kind" refers to the kind of problem associated with the dependency. * We only examined 80 tests in SWT manually, and found 49 dependencies among them.

```
1 public final class OptionBuilder {
2   private static String argName;
3
4   private static void reset() {
5     ...
6     argName = "arg";
7     ...
8   }
9
10  public static Option create(String opt){
11    Option option =
12      new Option(opt, description);
13    ...
14    option.setArgName(argName);
15    OptionBuilder.reset();
16    return option;
17  }
18 }
```

Figure 3: Fault-related code from `OptionBuilder.java`

erality, assume any environment with only a single variable. Then let $T = \langle t_1, \ldots, t_n \rangle$ be the test suite, and let $t_i, 1 < i \leq n$ be the test that should reveal the fault in environment $\mathbf{E}_R$. A dependency $t_k \prec t_i, k < i$ masks the fault if $\varepsilon(\langle t_1, \ldots, t_{i-1} \rangle, \mathbf{E}_0) \neq \mathbf{E}_R$.

The following two examples illustrate masking in practice.

CLI: A Long-Standing Bug.

A straightforward example of fault masking occurs in the Apache CLI library.[4] Two test cases fail when run in isolation: `test13666` and `testOptionWithoutShortFormat2` in test classes `BugsTest` and `HelpFormatterTest`, respectively.

A detailed study of the code under test revealed that both tests fail due to the same hidden dependence. The fault is located in `OptionBuilder.java` and is caused by not initializing a global variable early enough. Figure 3 shows code that illustrates the fault. By default, `argName` is initialized to `null` (line 2), and only set to its intended default value `"arg"` by the `create()` method via calling `reset()` (line 15). Consequently, if clients of CLI do not explicitly initialize the value of `argName`, the first option created will have `null` rather than `"arg"` as its argument name.

Both dependent tests can reveal this fault, since they create an option with the default argument as the first thing in their execution. However, in the default order of test execution, tests that create options with explicit arguments execute *before* these dependent tests. Thus, the tests that are executed before call `create()` at least once, which sets the default `argName` value, thus masking the fault.

This fault is reported in the bug database several times,[5] starting on March 13, 2004 (CLI-26). The report is marked as resolved *three years* later on March 15, 2007, but is then reopened as CLI-186 on July 31, 2009. On this report, one of the developers commented:

> I reproduced the issue, it requires a dedicated test case since it is tied to the initialization of a static field in OptionBuilder.

Despite the realization that a dedicated test is required, no such test was ever created. About one month later, the bug is duplicated as CLI-187, and the actual fix happens one year later on June 19, 2010, about six years after the bug was first reported (and four years total on the open-issue list).

JodaTime: Complex interactions that mask faults.

JodaTime[6] is an open source date and time library intended to improve upon the weaknesses of the date and time facilities provided by the standard JDK. It is a mature project that has been under active development for more than eight years.

JodaTime uses intricate caching mechanisms that are highly complex and coupled. All dependences we found are complex, in two cases even requiring a specific ordering of *three* tests to manifest.

In a simple dependence, JodaTime caches `PeriodType` objects, which contain an array of `DurationFieldTypes` (e.g., week, month). The order of `DurationFieldTypes` in the array is an important of the data representation, and two `PeriodTypes` with the same `DurationFieldTypes` in a different order are not equal internally in JodaTime, even though they are equal to JodaTime clients. To make this internal detail transparent to users of JodaTime, new `PeriodTypes` are normalized before they are cached. However, a fault in the code makes it possible to insert non-normalized `PeriodTypes` into the cache, leading to cache misses when searching for correctly normalized `PeriodTypes`.

A test that checks for correct normalization when caching

[4] `http://commons.apache.org/cli/`

[5] `https://issues.apache.org/jira/browse/CLI-26`
`https://issues.apache.org/jira/browse/CLI-186`
`https://issues.apache.org/jira/browse/CLI-187`

[6] `http://joda-time.sourceforge.net/`

objects fails in isolation but passes when the entire test suite executes in the default order; this happens because a prior test creates the expected `PeriodType`, and thus it is already in the cache for the later test. This behavior has been reported as a bug and has been fixed by the developers.

After inspecting the code, we reported the more complex dependence of three tests to the developers of JodaTime. They confirmed the phenomenon, but contended that it is due to interactions that are not intended in the design of the library [16]. In particular, one of the methods, `DateTimeZone.setProvider()`, is only supposed to be called a single time to initialize the library. In practice, multiple tests initialize the library, which leaves incorrect values in the cache and causes other tests to fail under some execution orders.

4.2 Poor Test Construction

Based on our interaction with the JodaTime developers, this last dependence does not mask a fault in the program. Instead, it represents a less severe consequence of test dependence that suggest that a test, or a test suite, has been constructed poorly in some dimension. While test dependences that mask faults correspond to a defect in the program source, these dependences correspond to defects in the test code.

The test dependences presented in this section arise due to incorrect initialization of program state by one or more tests. In the first case, tested program code relies on a global variable that is a part of the environment, but the test does not properly initialize it. In the second case, a test should but does not call an initialization function before later invocations to a complex library. This flaw in the test code is masked because the default test suite execution order includes other tests that initialize the library. The defect is inconsequential until and unless the flawed test is reordered, either manually or by a downstream tool, to execute before any other initializing test.

Crystal: Global Variables Considered Harmful.

Crystal[7] is a tool that pro-actively examines developers' code and precisely identifies and reports on textual, compilation, and behavioral conflicts.

The latest release of Crystal contains 81 human-written unit tests. Of those, 75 are fully automated, and 18 exhibit dependences. All these dependencies are caused by incomplete initialization of the environment when testing methods of three distinct classes (`DataSource, LocalStateResult, ConflictDaemon`). In all cases, one test initializes the environment correctly, and all other tests rely on that test executing first.

A short conversation with the developers confirmed that this was not intentional and most likely happened because the developers were not aware of the potential dependency caused by the use of global variables. Since we pointed out this problem, the developers treat the dependencies as undesirable and opened a bug report to have this issue resolved.[8]

XML Security: Global Initialization.

XML Security[9] is a component library implementing XML signature and encryption standards. Each released version of XML Security has a human-written JUnit test suite that achieves fairly high statement coverage.

Four stable released versions (1.0.2, 1.0.4, 1.0.5d2, and 1.0.71) of XML Security have been incorporated in the Software-artifact Infrastructure Repository (SIR).[10] We found that at least two out of the four versions contain dependent tests. Specifically, in versions 1.0.4 and 1.0.5d2, `test_Y1`, `test_Y2`, and `test_Y3` in class `ExclusiveC14NInterop` show dependent behavior. Since the dependences are the same in both versions, in the further discussion and in Figure 2, we consider only version 1.0.4.

For all three dependences, the cause of the dependence is the same: before any method in the library can be used, the global initialization function `Init.init()` has to be called. Internally, it initializes the static field that the code tested by the dependent tests rely on.

Given that the error when executing the dependent tests clearly explains the cause of the error, we speculate that developers either simply forgot to initialize the tests properly, or expected that these tests would always execute in the order defined in the test suite.

4.3 Spurious Bug Reports and Bug Fixes

Sometimes developers introduce dependent tests intentionally because it is easier, more efficient or more convenient to write unit tests for some modules in that way [17, 33]. Even though the developers are aware of these instances when they create them, this knowledge can get lost, and other people who are not aware of these dependences can get confused when they run a subset of the test suite that manifests the dependences.

As a result, they might report bugs backed by the failing tests, although this is exactly the expected behavior. If the dependence is not documented clearly and correctly, it can take a considerable amount of time to work out that these reported failures are spurious. Or worse, the developers may try to fix a bug that is not there.

Eclipse SWT: Causing Spurious Bug Reports.

The Eclipse Standard Widget Toolkit (SWT)[11] is a cross-platform GUI library developed within the Eclipse framework. Due to the difficulty of obtaining source, compiling and running test suites with the SWT project, we only examined some test cases manually, after a bug report indicated test dependence. The numbers reported in Figure 2 are the number of tests we manually examined, and the number of dependencies we found among those respectively.

As is common practice in GUI toolkits, SWT permits only one `Display` object per thread. Attempting to create multiple `Displays` in a single thread causes an `InvalidThreadAccessException`. To permit the reuse of `Displays`, SWT provides two methods: `Display.getDefault` and `new Shell`. These methods return the existing `Display` or create a new one if none exists.

In the test suite of SWT, all tests except those in the class `Test_org_eclipse_swt_widgets_Display` (`TestDisplay` for short) retrieve the current `Display` by using one of the lat-

[7] `http://crystalvc.googlecode.com`

[8] `https://code.google.com/p/crystalvc/issues/detail?id=57`

[9] `http://projects.apache.org/projects/xml_security_java.html`

[10] `http://sir.unl.edu`

[11] `http://eclipse.org/swt/`

ter methods. On the other hand, all tests in `TestDisplay` create their `Display` at the beginning of the test and dispose of it at the end.

In September 2003, a user reported a bug,[12] stating that tests throw an `InvalidThreadAccessException` if she runs any other test before `DisplayTest`. The cause of this is simple: any other test creates, but does not dispose of a `Display` object. Then the tests in `TestDisplay` attempt to create a new object, which fails, as one is already associated with the current thread. Since this is the expected and desired behavior, the bug report is spurious (except maybe it points to a problem in the test suite, rather than the code).

4.4 Dependence in Auto Generated Tests

Test dependence in automatically generated test suites is even more troublesome than in human-written suites. The reason for this is that all automated test generation tools we are aware of produce tests that are hard to read for humans, are undocumented, and their intent cannot easily be gleaned from naming conventions and other aids developers normally use. While there is some work to alleviate this problem, it still remains difficult to determine whether a failed test points to a bug in the program or a dependent test [10].

We already showed some evidence that test dependence is not uncommon in human-written tests. Given the increasing importance of automatically generated tests, we also wanted to at least get a glimpse of what is happening in that area. As a very preliminary, and by no means exhaustive or conclusive investigation, we applied Randoop to all the projects for which the source was readily available (this excludes SWT). The test suites were created by configuring Randoop to generate 5,000 tests for each program, and then drop what it considers to be redundant tests [23]. As the table in Figure 2 shows, in most projects, a large fraction of the remaining tests are dependent, while in some projects, there are almost no dependent tests. Why this strong division happens, and whether the differences between the programs can be used to derive guidelines for better testing is an interesting question left to future work.

In the following, we discuss one of the examples where tests generated with Randoop exhibit a large number of dependences, many of which could easily be fixed by humans.

Beanutils: Incomplete Automatic Generation.

Beanutils[13] is a library that provides services for collections of Java beans. Using Randoop, we generated a test suite consisting of 2692 unit tests for a recent release of Beanutils (version 1.8.3). The prototype tool we outline in Section 5 detected 299 dependent tests in this test suite.

After a close inspection of the automatically generated test code, we found the primary reason for the dependencies is missing initialization (cf. Sec. 4.2). Specifically, 248 tests attempt to retrieve values from a cache before anything has been added to the cache. This particular dependence could be fixed by adding a single line of setup code to each test. Most of the other dependencies could be fixed with similarly low effort, too. However, this particular fix requires understanding of at least part of the program semantics, which is a feat beyond the abilities of current test generation tools.

[12] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=43500`

[13] `http://commons.apache.org/beanutils/`

Given the high ratio of dependent tests in the automatically generated test suite, we speculate that the following two phenomena could be reasons for this.

First, developers usually know a lot about the intended purpose of a program when they write tests for it. This knowledge helps them to build well-structured and coherent test suites. Automated tools, on the other hand, have no such knowledge. One possible consequence of this is illustrated by the example: the automated tool does not understand the cache protocol and thus does not know that it must add values to the cache first.

Second, it is often hard for automated tools to understand that specific parts of the code depend on the environment, and thus may not explicitly generate code that sets up the environment correctly. If, at the same time, other tests are generated that as a side effect create the needed environment, test dependence ensues.

5. ALGORITHM AND IMPLEMENTATION

In this section we present an algorithm and a prototype tool to detect dependent tests. In the worst case, a naive, exhaustive search would execute all $n!$ permutations of the test suite to detect dependent tests. While this is not feasible for realistic n, our approximate algorithm uses our intuition that many dependences can be found by running only short subsequences of test suites, and introduces a bound k on the length of subsequences. That effectively bounds the execution time to $O(n^k)$, which for small k is tractable. At the same time, our prototype tool and the experiments we conducted with it, suggest that many dependences can be found for small k.

5.1 Algorithm

Since the general form of the dependent test detection problem is NP-complete, we do not expect to find an efficient algorithm for it. Instead, we developed an algorithm to approximate solutions by detecting a subset of dependent tests. For tractability, our algorithm in Figure 4 bounds the length of test execution sequences, and thus the number of permutations to execute. Instead of executing all permutations of the whole test suite, we execute all possible k-tuples for a bounding parameter k.

Given a test suite $T = \langle t_1, t_2, \ldots, t_n \rangle$, our algorithm executes $\varepsilon(T, \mathbf{E}_0)$ to obtain the *expected result* $R(T|\mathbf{E}_0)$ of each test (line 2). The environment $\mathbf{E}_0$ is the environment provided by the test execution framework. It then executes every k-tuple T_i^k of tests as $\varepsilon(T_i^k, \mathbf{E}_0)$, and checks whether any result $R(T_i^k|\mathbf{E}_0)$ differs from the expected result, i.e. that there is a dependence in T_i^k (lines 3–10). The algorithm returns the set of all tests $t_i \in T$ that have at least one dependence.

It is easy to extend this algorithm to return the shortest sequence of tests that manifest a dependency for a given test t_i, for example by reducing manifesting sequences with Delta Debugging [36].

5.2 Tool Implementation

We implemented our k-bounded dependent test detection algorithm in a prototype tool.[14] The tool is fully-automated and needs only a test suite and the bounding parameter k as inputs. Our current implementation supports JUnit 3.x

[14] Available at: `http://testisolation.googlecode.com`

```
Input: a test suite T, an execution length k
Output: a set of dependent tests depTests
 1: depTests ← ∅
 2: expectedResults ← R(T|E_0)
 3: for each T_i^k in getPossibleExecOrder(T, k) do
 4:    execResults ← R(T_i^k|E_0)
 5:    for each test t in T_i^k do
 6:       if execResults[t] ≠ expectedResults[t] then
 7:          depTests ← depTests ∪ t
 8:       end if
 9:    end for
10: end for
11: return depTests
```

Figure 4: k-bounded approximation algorithm to detect dependent tests. "getPossibleExecOrder" returns all permutations of tests from T of length k.

tests. We consider JUnit test results to be the same when the tests either both pass, or exactly the same exception or assertion violation leads to test failure. The tool creates a fresh JVM for each T_i^k, thus, ignoring external state such as files and OS services, the environment that the test suites are executed in is always the same $\mathbf{E}_0$. This ensures that there is no interaction between different T_i^k through shared memory.

We used the prototype to verify the dependent tests reported by users, developers, other researchers, and us, and to find new dependent tests in the example programs in Section 4 using isolated execution ($k = 1$) and pairwise execution ($k = 2$).

All the dependent tests reported in Figure 2, except for two dependent tests in JodaTime and the dependences in SWT, can already be found by isolated execution. Since we could not run the test suite of SWT, we could not check these dependences with our tool. During manual bug diagnosis in JodaTime, we identified two test dependences that require *three* tests to manifest. While these are easy to reproduce, we did not check that our tool finds them, because the time needed to run our naive algorithm on JodaTime with $k = 3$ is measured in months.

While we believe that most test dependences can be found with small k. This is in part because the set of dependent tests that can be found with a bound k is always a subset of the set of dependent tests that can be found with any bound $k' > k$. Additionally, our intuition and preliminary exploration seem to indicate that small k find many dependences, while larger k do not. However, in principle it is conceivable that any number of chain dependences with chains longer than any tried k exist in all the libraries we analyzed.

6. CONCLUSIONS

In the introduction we posed several questions why test dependence may have received little attention despite the ease of constructing concrete but contrived examples. Our contributions suggest answers, to differing degrees, to these questions:

Does test dependence arise in practice? YES.

Reproducible examples, both from human-written test suites and automatically generated test suites that we synthesized, show this in six fielded, substantative, open-source systems. This "existence proof" of test dependence is unlikely to surprise anyone, nor does it suggest that test dependence is common across *all* software. However, at least under our definition of manifest test dependence, this question is closed.

If and when test dependence arises, are there significant repercussions? SOMETIMES, ALTHOUGH THE EXTENT IS UNKNOWN.

Several of our examples identified situations in which test dependence masked faults in the underlying program. In another example, developers wasted time tracking down a non-existent fault because of a spurious report that was due to an undocumented test dependence. Even though these are real and reproducible examples, it is not possible to make any general claims about the frequency nor the significance of the repercussions of test dependence. At the same time, it seems unlikely that these are the *only* software systems where test dependence causes problems.

Is dependence easy to notice if-and-when it arises? AT PRESENT, USUALLY NOT.

This is a more subtle question, because the answer depends not only on the tools being used but also on the perceptions and insights of the developers. If tools always run tests in the same context, and if developers never consider the possibility of test dependence, then it is unlikely that dependence will be observed. Masking of faults in the underlying program is a good illustration of this. Our prototype tool shows the potential for revealing dependences, allowing developers to observe them and make conscious decisions about how, or even whether, to deal with the dependences.

We have also shown that, in principle, test dependence can compromise the application of downstream testing techniques such as selection, prioritization, and parallelization. Like contrived examples of test dependence itself, it is easy to produce simple examples where downstream techniques produce incorrect output when applied to dependent tests. We conjecture, based on intuition and very thin evidence, that if and when this happens in practice, it is hard to notice in part because tools do not surface the necessary information.

Our formalism provides a precise definition of manifest test dependence, allows reasoning about test dependence, and enables the proof that detecting manifest test dependence in test suites is NP-complete. Our prototype tool shows that even our approximate algorithm can reveal large numbers of important dependences. Faster and more precise approaches are plausible, especially as more understanding of test dependence "in the field" is acquired.

The question of the role of test dependence in the real world merits more aggressive empirical study. A better understanding of the frequency and scope of repercussions from test dependence should be developed. Of particular concern is the masking of program faults because, unlike weaknesses in test suites or spurious bug reports, masking faults could be costly to find by other methods or to leave in the program. Tools that surface test dependences may help researchers and practitioners study and deal with dependences more effectively.

As such deeper knowledge is acquired, we may better understand what contributes to creating test dependence. Is this phenomenon linked to particular testing levels (unit, integration, system, etc.), specified testing techniques and frameworks, the programming languages and/or the software development process employed, the relationship and

communication structures between developers and testers on a team, etc. In turn, any insights gained in these dimensions may lead to more systematic approaches to dealing with test dependence. There is already some work aiming at reducing the potential for dependences by refactoring programs to use less global state [34].

The question of how second-order testing techniques, like regression test selection and prioritization should handle dependences is also open. Most current techniques just assume independence and make no statement about what happens when this assumption is not true. One straightforward way to amend this situation might be to augment such techniques to respect a defined partial order among tests. And this partial order can be derived from knowledge about dependent tests.

Acknowledgments

Bilge Soran was a participant in the project that led to the initial result. Yuriy Brun and Colin Gordon provided advice about the formal notation. Reid Holmes and Laura Inozemtseva identified the initial JodaTime dependence. Mark Grechanik, Adam Porter, Michal Young, and Reid Holmes provided timely and insightful comments on a draft.

7. REFERENCES

[1] 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.

[2] Lionel C. Briand, Jie Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE '02*, pages 43–50, 2002.

[3] P. J. Brown. Programming and documenting software projects. *ACM Computing Surveys*, 6:213–220, December 1974.

[4] David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weber. A framework for testing database applications. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 147–157, 2000.

[5] David Cohen, Ieee Computer Society, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23:437–444, 1997.

[6] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 38–48, 2003.

[7] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering*, 36(5):593–617, 2010.

[8] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 253–264, 2006.

[9] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 102–112, 2000.

[10] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 364–374, 2011.

[11] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. *SIGMOD Rec.*, 23(2):243–252, 1994.

[12] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 312–326, 2001.

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

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

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

[16] Personal communication. `http://sourceforge.net/projects/joda-time/forums/forum/337835/topic/5111255`.

[17] Gregory M. Kapfhammer and Mary Lou Soffa. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 98–107, 2003.

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

[19] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 119–129, 2002.

[20] John D. McGregor and Timothy D. Korson. Integrated object-oriented testing and development processes. *Communincations of the ACM*, 37:59–77, September 1994.

[21] E. F. Miller, Jr. and R. A. Melton. Automated generation of testcase datasets. In *Proceedings of the International Conference on Reliable Software*, pages 51–58, 1975.

[22] Kıvanç Muşlu, Bilge Soran, and Jochen Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software*

Engineering, SIGSOFT/FSE '11, pages 496–499, 2011.

[23] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, 2007.

[24] Mauro Pezzè and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, 2007.

[25] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 179–188, 1999.

[26] Matthew J. Rummel, Gregory M. Kapfhammer, and Andrew 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.

[27] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for java. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 114–123, 2005.

[28] Raul Santelices, Mary Jean Harrold, and Alessandro Orso. Precisely detecting runtime change interactions for evolving software. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 429–438, 2010.

[29] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 97–106, 2002.

[30] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 391–400, 2011.

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

[32] Zhimin Wang, Sebastian Elbaum, and David S. Rosenblum. Automated generation of context-aware tests. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 406–415, 2007.

[33] James A. Whittaker, Jason Arbon, and Jeff Carollo. *How Google Tests Software*. Addison-Wesley Professional, 2012.

[34] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '09, pages 173–182, 2009.

[35] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ISSRE '97, pages 264–274, 1997.

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

[37] Weilei Zhang and Barbara G. Ryder. Discovering accurate interclass test dependences. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 55–62, 2007.