Are mutants a valid substitute for real faults in software testing?

René Just[1], Darioush Jalali[1], Laura Inozemtseva[2], Michael D. Ernst[1], Reid Holmes[2], and Gordon Fraser[3]
[1]University of Washington, Seattle, WA, USA
[2]University of Waterloo, Waterloo, ON, Canada
[3]University of Sheffield, Sheffield, UK

ABSTRACT

A good test suite is one that detects real faults. Because the set of faults in a program is unknowable, this definition is not useful to practitioners who are creating test suites nor to researchers who are creating and evaluating tools that generate test suites. In place of real faults, testing research often uses mutants, which are artificial faults — each one a simple syntactic variation — that are systematically seeded throughout the program under test. Mutation testing is appealing because large numbers of mutants can be automatically generated and used as a proxy for real faults.

Unfortunately, there is little experimental evidence to support the use of mutants as a proxy for real faults. This paper investigates whether mutants are indeed a valid substitute for real faults — that is, whether a test suite's ability to detect mutants is correlated with its ability to detect real faults that developers have fixed.

Our experiments used 357 real faults in 5 open-source applications totalling 321,000 lines of source code. Furthermore, our experiments used both developer-written and generated test suites. We found a statistically significant correlation between mutant detection and real fault detection, even when controlling for code coverage.

1. INTRODUCTION

Both industrial software developers and software engineering researchers are interested in measuring test suite quality: developers want to know if their suites have a good chance of detecting faults, while researchers want to be able to compare different testing or debugging techniques. Ideally, one would directly measure the number of faults a test suite can detect in a program. Unfortunately, the faults in a program are unknown, so a proxy measurement must be used instead.

A well-established proxy for test quality in testing research is the mutation score, which measures a test suite's ability to distinguish a program under test (*original version*) from many small syntactic variations, called *mutants*. The *mutation score* is the percentage of mutants that a test suite can distinguish from the original version. Mutants are created by systematically injecting small artificial faults into the program under test, based on *mutation operators*. Examples of such mutation operators are replacement of arithmetic operators (e.g., `x+y` $\mapsto$ `x-y`), modification of branch conditions, or deletion of statements. A test suite that can detect (or *kill*) more mutants — that is, it has a higher mutation score — is considered to be a better suite than one that detects fewer mutants.

This measurement is often used in software testing and debugging research. More concretely, mutation analysis is commonly used in the following use cases (e.g., [3, 13, 25, 26]):

1. *Test suite augmentation and generation*
   A test suite T is only augmented with a test t if this test increases the mutation score of T. Likewise, a mutation-based test generation approach generates and optimizes a test suite towards its mutation score based on the assumption that a higher mutation score indicates a better test suite.
2. *Test suite selection and evaluation*
   Suppose we have two unrelated test suites T_1 and T_2 that have the same mutation score and $|T_1| < |T_2|$. In the context of test suite selection, T_1 is a preferable test suite as it has fewer tests than T_2 but the same mutation score. Generally in the context of test suite evaluation, a test suite that has a higher mutation score is assumed to be more effective with respect to real faults.
3. *Test suite minimization*
   In the context of test suite minimization, a test suite T is reduced to $T \setminus \{t\}$ for every test $t \in T$ for which the reduction does not decrease the mutation score of T.
4. *Fault localization*
   A fault localization technique that precisely identifies a mutation location as the root cause of this artificial fault is assumed to be equally effective for real faults.

These uses of mutation analysis rely on the fundamental assumption that mutants are a valid substitute for real faults. Two test suites with the same mutation score are assumed to be equally effective — that is, they are assumed to have the same real fault detection capability. However, there is surprisingly little experimental evidence supporting this assumption, as discussed in greater detail in Section 4.

To the best of our knowledge, only two previous studies have explored the correlation between mutants and real faults [1,5]. These studies used small programs — the largest had only 5,905 LOC. In addition, one study investigated only 12 real faults [5] while the other examined 38 real faults [1]. Due to the lack of real faults, the latter study also used hand-seeded faults for 7 out of its 8 subject programs. However, it is not clear that these hand-seeded faults are equivalent to real inadvertently-introduced faults.

Besides, prior research also neglected the effect of structural code coverage when studying the correlation between mutant detection and real fault detection. A higher mutation score could simply be caused by a higher code coverage. Therefore, it is not clear how mutant detection is correlated with real fault detection independently of code coverage — that is, whether this correlation exists even if coverage is controlled for.

This paper extends previous work and explores the relationship between mutants and real faults using 5 large Java programs and

Table 1: Investigated subject programs. LOC and number of (JUnit) tests applies to the most recent version.

	Program	KLOC[1]	Test KLOC[1]	Tests	Coverage[2]	Source
Chart	JFreeChart	96	50	2,205	73%±16%	`http://sourceforge.net/projects/jfreechart`
Closure	Closure Compiler	90	83	7,927	90%±15%	`http://code.google.com/p/closure-compiler`
Math	Commons Math	85	19	3,602	90%±10%	`http://git.apache.org/commons-math.git`
Time	Joda-Time	28	53	4,130	91%± 7%	`http://github.com/JodaOrg/joda-time`
Lang	Commons Lang	22	6	2,245	88%±19%	`http://git.apache.org/commons-lang.git`

[1] Non-comment, non-blank lines of code, as reported by sloccount (`http://www.dwheeler.com/sloccount`)

[2] Statement coverage of developer-written test suite on files modified by the bug fix. Reported mean and standard deviation for the 357 analyzed faulty versions.

357 real faults. Specifically, this paper aims to confirm or refute the hypothesis that mutants are a valid substitute for real faults in software testing by answering the following research question:

RESEARCH QUESTION 1. *Is mutant detection correlated with real fault detection, independently of code coverage?*

A fundamental assumption in mutation testing is the existence of the *coupling effect* (cf. [20]). A complex fault is *coupled* to several simple faults if a test that detects all those simple faults also detects the complex fault. In other words, the coupling effect states that for each complex fault there exists no test that can detect all simple faults without detecting the complex one. Given the large number of mutants (i.e., simple faults) generated for a program, then (if the coupling effect holds) a test that detects a real fault (i.e., a complex fault) should always kill one or more mutants. Therefore, results derived from software testing experiments based on mutants only generalize to real faults if the following research question can be affirmed:

RESEARCH QUESTION 2. *Does a higher fault detection score imply a higher mutation score?*

Research question 2 studies the existence of the coupling effect between real faults and mutants without considering the number of coupled mutants. In addition, we also quantified the number of mutants coupled to a real fault by answering the following research question:

RESEARCH QUESTION 3. *How many additional mutants does a test suite detect if one stronger test is added that detects a single real fault?*

In a surprisingly high percentage (25%) of cases, a higher fault detection score did not lead to a higher mutation score. Based on this observation we addressed an additional research question:

RESEARCH QUESTION 4. *Which real faults are detected by a test that does not increase the mutation score of its test suite, i.e., for which real faults does the coupling effect not hold?*

The contributions of this paper are as follows:

- A new set of 357 developer-fixed and manually-verified bugs and test suites from 5 programs.
- The largest study to date of whether mutants are a valid substitute for real faults. The results show a statistically significant correlation, even when controlling for code coverage.
- Concrete suggestions for improving mutation analysis, and identification of inherent limitations (faults not coupled to mutants).

The paper is structured as follows: Section 2 describes how we collected data to answer the research questions above. Section 3 presents our analysis of that data. Section 4 reviews related work and Section 5 concludes.

2. DATA COLLECTION

Previous work assumes that mutant detection and real fault detection are well-correlated — if a given suite has better mutant detection, then it also has better real fault detection (and vice versa). We tested this assumption by conducting a study with real faults, using both developer-written test suites and automatically-generated test suites.

We used the following high-level methodology to answer our research questions:

1. Locate and isolate real faults that have been previously found and fixed, by analyzing several projects' version control and bug tracking systems; see Section 2.2.
2. Obtain developer-written test suites for both a buggy and a fixed program version for each fault. Obtain automatically-generated test suites for the fixed version of the program. See Section 2.3.
3. Generate mutants and perform mutation analysis for all fixed program revisions; see Section 2.4.
4. Conduct experiments using the faults and the test suites to answer our research questions; see Section 2.5.

2.1 Subject programs

Table 1 lists the 5 subject programs we used in our experiments. These programs satisfy the following desiderata:

1. Each program has a version control repository and bug tracker, enabling us to locate and isolate real defects.
2. Each program contains a comprehensive, developer-written test suite, enabling us to experiment with real test suites as well as generated ones.
3. Each program has been used in previous research, enabling an evaluation of whether prior research results derived from mutation analysis on those programs generalize to real faults.

2.2 Locating and isolating real faults

We obtained real faults from a project's version control history by identifying commits that corrected a failure in the program's source code. Ideally, we would like to have, for each real fault, two source code versions V_1 and V_2 which differ by only the bug fix. Unfortunately, developers do not always minimize their commits.

Table 2: Number of candidate revisions, compilable revisions, and reproducible and isolated faults for all subject programs.

	Candidate revisions	Compilable revisions	Reproducible faults	Isolated faults
Chart	80	62	28	26
Closure	316	227	179	133
Math	435	304	132	106
Time	75	57	29	27
Lang	273	186	69	65
Total	1179	836	437	357

Therefore, we had to locate and isolate the real fault in a bug-fixing commit.

We first examined the version control and bug tracking system of each program for indications of a bug fix (Section 2.2.1). We refer to a revision that indicates a bug fix as a *candidate revision*. For each candidate revision, we tried to reproduce the fault by exposing it with an existing test (Section 2.2.2). Finally, we isolated the fault by pruning all irrelevant code changes from the bug-fixing commit (Section 2.2.3). We discarded any fault that could not be reproduced and isolated. Table 2 summarizes the results of each step in which we discarded candidate revision pairs.

2.2.1 Candidate revisions for bug-fixing commits

We developed a script to determine revisions that a developer marked as a bug fix. The script mines the version control system for explicit mentions of a bug fix, such as a bug identifier of the project's bug tracking system.

Let rev_{fix} be a revision marked as a bug fix. We assume the previous commit in the version control history, rev_{bug}, to be faulty. (Later steps will validate this assumption.) Overall, we identified 1,179 candidate revision pairs $\langle rev_{bug}, rev_{fix} \rangle$.

2.2.2 Discarding non-reproducible faults

A candidate revision pair obtained in the previous step is not suitable for our experiments if we cannot reproduce and expose the real fault. Let V be the source code version of a revision *rev* and T be the corresponding test suite. The fault of a candidate revision pair $\langle rev_{bug}, rev_{fix} \rangle$ is reproducible if a test exists in T_{fix} that passes on V_{fix} but fails on V_{bug} due to the existence of the fault.

In some cases, test suite T_{fix} does not run on V_{bug}. If necessary, we fixed build system related issues and trivial errors such as imports of non-existent classes. However, we did not attempt to fix compilation errors if they were caused by semantic errors. This required discarding revisions with unresolvable compilation errors — 836 candidate revision pairs remained after this step.

After fixing trivial compilation errors, we discarded version pairs for which the fault was not reproducible. A fault might not be reproducible for three reasons. (1) The source code diff is empty — the difference between rev_{bug} and rev_{fix} was only to tests, configuration, or documentation. (2) No test in T_{fix} passes on V_{fix} but fails on V_{bug}. (3) No test in T_{fix} exposes the fault in V_{bug}. We manually inspected each test of T_{fix} that failed on V_{bug} while passing on V_{fix} to determine whether its failure is caused by the real fault. The overall number of candidate revision pairs for which we could reproduce the fault was 437.

2.2.3 Isolating the real fault

Since developers do not always minimize their commits, the source code of V_{bug} and V_{fix} might differ by both features and the bug fix. We isolated the bug fix for the purposes of our study. This is important to ensure that a test failure or success in a generated test is due to the bug or its fix, rather than other unrelated changes that could affect test results for other reasons. Other benefits include improved backward-compatibility of tests and the ability to focus our experiments on a smaller amount of modified code.

$$\begin{tikzcd} \boxed{V_{bug}} \arrow[r, leftrightarrow, "\text{features \& refactorings}"'] \arrow[rrr, bend left, leftrightarrow, "\text{source code diff}"] & V_1 \arrow[r, leftrightarrow, "\text{bug fix}"'] & V_2 \arrow[r, leftrightarrow, "="'] & \boxed{V_{fix}} \end{tikzcd}$$

Figure 1: Obtaining source code versions V_1 and V_2 that differ by only a bug fix. V_{bug} and V_{fix} represent the source code versions of two consecutive commits in the project's version control history.

For each of the 437 candidate revision pairs $\langle rev_{bug}, rev_{fix} \rangle$ for which we could reproduce the fault, we minimized the source code diff between V_{bug} and V_{fix}. For each revision pair, we manually reviewed the source code diff between V_{bug} and V_{fix} and divided it into a diff that represents the bug fix part vs. one that represents features and refactorings. At least two of this paper's authors performed this process for each pair, to ensure consistency.

The result of this process was two source code versions V_1 and V_2 such that V_1 and V_2 differ by exactly a bug fix — that is, no features were added and no refactoring was applied.

Figure 1 visualizes the relationship between the source code versions V_1 and V_2, and how they are obtained from the source code versions of a candidate revision pair. V_2 is equal to the version V_{fix} and the difference between V_1 and V_2 is the bug fix. V_1 is obtained by re-introducing the bug into V_2 — that is, applying the inverse bug-fixing diff. We discarded a version pair if we could not isolate the fault in the source code diff. Overall, we obtained 357 version pairs $\langle V_1, V_2 \rangle$ for which we could isolate the real fault.

2.3 Test suites

2.3.1 Developer-written test suites

This section describes how we obtained two related test suites T_{pass} and T_{fail} made up of developer-written tests, where T_{pass} passes on V_1 and T_{fail} fails on V_1 because of the real fault. These test suite pairs $\langle T_{pass}, T_{fail} \rangle$ reflect common and recommended practice. The developer's starting point is the source code version V_1 and a corresponding test suite T_{pass}, which passes on V_1. Upon discovering a previously-unknown fault in V_1, a developer augments test suite T_{pass} to expose this fault. The resulting test suite T_{fail} fails on V_1 but passes on the fixed source code version V_2. T_{pass} might be augmented by modifying an existing test (e.g., adding stronger assertions) or by adding a new test.

We cannot directly use the existing developer-written test suites T_{bug} and T_{fix} as T_{pass} and T_{fail}, because not all tests pass on each committed version and because the developer may have committed changes to the tests that are irrelevant to the fault. Therefore, we built the test suites T_{pass} and T_{fail} based on T_{bug} and T_{fix}, as subsequently described.

Section 2.2 describes how we obtained 357 suitable version pairs $\langle V_1, V_2 \rangle$. For each pair, at least one test exposes the fault in V_1 while passing on V_2. We refer to such a test as a *triggering test* $\hat{t}$. Let m be the number of triggering tests for a version pair; then $\hat{t}^i$ denotes the i-th triggering test ($1 \leq i \leq m$). Our goal was to obtain m pairs of test suites $\langle T^i_{pass}, T^i_{fail} \rangle$ with the following properties:

- $T^i_{fail} \subseteq T_{fix}$
- All tests of T^i_{fail} pass on V_2
- $\hat{t}^i \in T^i_{fail}$ is the only triggering test in T^i_{fail} — that is, only $\hat{t}^i$ exposes the fault in V_1
- All tests of T^i_{pass} pass on V_1 and V_2
- T^i_{pass} and T^i_{fail} differ by exactly one modified or added test

Figure 2 visualizes how the test suite pairs $\langle T^i_{pass}, T^i_{fail}\rangle$ are derived from the developer-written test suites T_{bug} and T_{fix}. In 80% of the cases, T_{fix} contained exactly one triggering test: developers usually augment a test suite by adding or strengthening one test to expose the fault.

In order to fairly compare the effectiveness of T^i_{pass} and T^i_{fail}, they must not contain irrelevant differences. In order to eliminate irrelevant differences between T^i_{pass} and T^i_{fail}, T^i_{pass} is derived from T_{fix}. If T^i_{pass} were derived from T_{bug} instead, two possible problems could arise. First, V_1 might include features (compared to V_{bug}, as described in Section 2.2) and T_{fix} might include corresponding feature tests. Second, tests unrelated to the real fault might have been added, changed, or removed in T_{fix}.

In summary, we applied the following steps to obtain all pairs $\langle T^i_{pass}, T^i_{fail}\rangle$ using the developer-written test suites T_{bug} and T_{fix}:

1. Manually fix all classpath- or configuration-related test failures in T_{bug} and T_{fix}, so all failures indicate genuine faults.
2. Exclude all tests from T_{bug} that fail on V_1, and exclude all tests from T_{fix} that fail on V_2.
3. Determine all triggering tests $\hat{t}^i_{fix}$ in T_{fix}.
4. Create one test suite pair $\langle T^i_{pass}, T^i_{fail}\rangle$ for each $\hat{t}^i_{fix} \in T_{fix}$ such that:
   - *T^i_{pass} passes on V_1 and V_2.*
     T^i_{pass} includes all tests of T_{base} plus the predecessor of the i-th triggering test $\hat{t}^i_{fix}$, if it exists. If $\hat{t}^i_{fix}$ was modified the previous version of the same test is added, otherwise $T^i_{pass} = T_{base}$.
   - *T^i_{fail} fails on V_1 but passes on V_2.*
     T^i_{fail} includes all tests of T_{base} plus the i-th triggering test $\hat{t}^i_{fix}$.

2.3.2 Generated suites

For our experiments, we used two test generation tools: EvoSuite and Randoop. We attempted to use a third tool, DSDCrasher, but found that it relies on the static analysis tool ESC/Java2. This tool does not work with Java 1.5 and higher, making it impossible to use DSDCrasher for this study. We plan to investigate an earlier version of the tool that does not use the static analysis component, JCrasher, in future work.

EvoSuite aims to satisfy one of several possible criteria. We selected branch coverage, weak mutation testing, and strong mutation testing. Randoop has no such bias. Therefore, we expected that EvoSuite test suites would have higher mutation scores, on average, than Randoop suites. If mutants are a good substitute for faults, this further implies that EvoSuite test suites should have higher

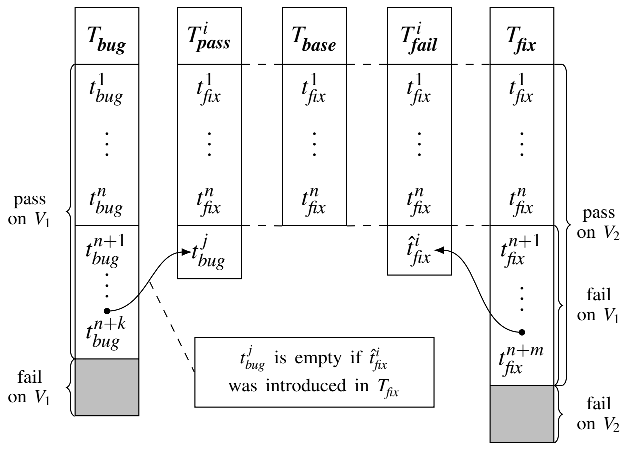

Figure 2: Relationship between the i-th obtained test suite pair $\langle T^i_{pass}, T^i_{fail}\rangle$ and the developer-written test suites T_{bug} and T_{fix}, which are derived from the project's version control system.

fault detection scores (on average) as well. For each fixed program version V_2, EvoSuite generated 30 test suites for each of the selected criteria and Randoop generated 12 test suites.

Each test generation tool created tests only for classes in C_m, the classes modified in the isolated bug fix — that is, the classes that differ between V_1 and V_2.

Each of the test generation tools might produce uncompilable test classes or tests that do not run without errors. Additionally, tests might sporadically fail due to the use of non-deterministic APIs such as time of day or random number generators. A test suite that (sporadically) fails is not suitable for our study — we generally refer to them as failing test suites. We repaired failing test suites in a semi-automatic fashion using the following workflow:

1. Automatically remove test methods that cause compilation errors.
2. Automatically fix test suites that compile but include failing test methods: remove all individual test methods that fail during execution on V_2.
3. Execute the generated test suites on V_2 to identify and eliminate non-deterministic tests: we assumed that a test suite does not include any further non-deterministic tests once it passed 5 times in a row.

The final output of this process is generated test suites that succeed on V_2. The workflow of repairing a test suite sometimes resulted in an empty set, when all tests failed and had to be removed. Therefore, for EvoSuite and Randoop the number of suitable test suites that succeed on V_2 is smaller than the total number of generated test suites. Table 3 summarizes the characteristics of all generated test suites that succeed on V_2.

We refer to a generated test suite as $\tilde{T}$. We ran each $\tilde{T}$ against V_1. If $\tilde{T}$ passed, it did not detect a fault. If $\tilde{T}$ failed, we manually verified that the failing tests are valid triggering tests — that is, we verified that the failures were due to the fault in V_1 rather than issues with the build system or configuration.

2.4 Mutation analysis

We employed the Major mutation framework [14, 16] and all its available mutation operators to create the mutant versions and to perform the mutation analysis. Major provides the following

Table 3: Characteristics of generated test suites. Test suites gives the total number and T_{fail} ratio for all test suites that succeeded on V_2. KLOC and Tests report mean and standard deviation of lines of code and number of (JUnit) tests for all test suites. Detected faults shows how many distinct real faults the test suites detected altogether and for how many versions at least one suitable test suite could be generated.

	Test suites		KLOC	Tests	Detected faults
	Total	T_{fail}			
EvoSuite	28,318	22.3%	98±491	68±133	182/354
-branch	10,133	21.1%	25±75	21±24	156/352
-weak	9,420	21.8%	29±80	24±27	158/352
-strong	8,765	24.1%	259±857	171±202	152/350
Randoop	3,387	18.0%	2,027±1,755	8,208±14,053	90/326
Overall	31,705	21.8%	335±995	1,066±5,599	198/357

mutation operators, which include the operators used in previous studies (cf. [18]):

- Replace constants
- Replace operators
- Modify branch conditions
- Delete statements

We only mutated the source code version V_2, consistent with the fundamental assumption in mutation analysis that the program under test is assumed to be defect-free.

For each of the developer-written and generated test suites, we computed mutation coverage and mutation score. A test is said to cover a mutant if it reaches and executes the mutated code. A test kills a mutant if the test outcome indicates a fault — that is, a test assertion fails or the test triggers an exception in the mutant. Mutation coverage is a necessary but not sufficient condition to kill a mutant.

Executing all tests on all mutants might be prohibitively expensive. Therefore, we exploited two common optimizations, which are supported by Major (cf. [15]): 1) a test is only executed on a mutant if it covers the mutant and 2) no further test is executed on a mutant once that mutant has been killed.

2.5 Experiments

This section describes our experiments to study the correlation between mutant detection and real fault detection. Our experiments use source code version V_1 (the faulty version).

The test suites T_{pass} and T_{fail} model how a developer usually augments a test suite. T_{fail} is a better suite — it detects a fault that T_{pass} does not. If mutants are a valid substitute for real faults, then T_{fail} should have a higher mutation score than T_{pass}.

We also investigated whether mutant detection is correlated with real fault detection for automatically-generated tests.

Controlling for coverage.

Structural code coverage is a widely-used measure of test suite quality. Differences in coverage often dominate other aspects of test suite generation, and a technique that creates larger test suites usually detects more faults for that reason alone. More specifically, if test suite T_x covers more code than T_y, then T_x is likely to have a higher overall mutation score and fault detection score, even if T_y does a better job testing a smaller portion of the program. Furthermore, no developer would use a complex, time-consuming test suite metric such as mutation analysis unless simpler ones such as structural coverage had exhausted their usefulness.

To account for these facts, we performed our experiments in two ways. First, we ignored coverage and simply determined the mutants killed by each test suite. Second, we controlled for coverage and only determined the mutants killed for test suites that cover the same code.

We include the first, questionable methodology for comparison with prior research that does not control for coverage. The second methodology controls for coverage. It better answers whether use of mutation analysis is profitable, under the assumption that a developer is already using the industry-standard coverage metric.

Our experiments use Cobertura [4] to compute statement and branch coverage over the classes modified by the bug fix.

2.5.1 *Does a higher fault detection score imply a higher mutation score, for developer-written tests?*

If mutants are a valid substitute for real faults, then any test suite T_{fail} that has a higher fault detection score than T_{pass} should have a higher mutation score as well.

The fault detection scores s_f for the test suites T_{pass} and T_{fail} are:

- $s_f(T_{pass}) = 0$
- $s_f(T_{fail}) = 1$

Mutation analysis determined the mutants (obtained from program version V_2) covered and killed by the test suites. We also computed structural coverage. Ultimately, this step determined for each test suite T_{pass} and T_{fail}:

- Statement and branch coverage
- Number of generated, covered, and killed mutants
- Real fault detection score

Using those values, we determined for every test suite pair $\langle T_{pass}, T_{fail} \rangle$, whether T_{fail} yields a higher mutation score than T_{pass}.

2.5.2 *How many additional mutants does a test suite detect if one stronger test is added that detects a single real fault?*

In this experiment we measured the sensitivity of the mutation score for a single fault. Specifically, we determined the increase in the number of detected mutants between T_{pass} and T_{fail}. Since T_{fail} is formed by adding the triggering test to T_{pass}, we considered the following 4 cases:

1. The triggering test is added and increases statement coverage.
2. The triggering test is added but does not increase statement coverage.
3. The triggering test is modified and increases statement coverage.
4. The triggering test is modified but does not increase statement coverage.

2.5.3 *Which real faults are not coupled to mutants?*

The key assumption of mutation analysis is that real faults are coupled to mutants. However, there exist real faults that are detected by a test that does not increase the mutation score of its test suite. The number of mutants detected by T_{pass} is equal to the number of mutants detected by T_{fail}, and there is no mutant such that detecting that mutant leads to detecting the fault.

To better understand these real faults that are not coupled to mutants, we manually investigated each such fault. This qualitative study sheds light on which types of real faults are not coupled to mutants generated by commonly used mutation operators. Moreover, this study reveals general limitations of mutation analysis and suggests new and stronger mutation operators taking the application domain into account.

2.5.4 *Is mutation score correlated with fault detection, for automatically-generated tests?*

In this experiment, we used the automatically-generated test suites $\tilde{T}$ and determined for each suite its mutation score and fault detection score. We only considered version pairs $\langle V_1,V_2 \rangle$ for which at least one generated test suite detects the real fault as we aimed at comparing the mutation adequacy of generated test suites that detect the real fault with test suites that do not.

Every generated test suite $\tilde{T}$ succeeds on V_2 by definition. If $\tilde{T}$ fails on V_1, we denote it as $\tilde{T}_{fail}$, otherwise as $\tilde{T}_{pass}$. The fault detection scores s_f for those a generated test suites are:

- $s_f(\tilde{T}_{pass}) = 0$
- $s_f(\tilde{T}_{fail}) = 1$

We computed the correlation between mutation score and fault detection score.

We first analyzed the entire pool of test suites derived from all generation tools to investigate whether mutation is generally a good metric to compare the effectiveness of arbitrary test suites.

Then, we applied the same analysis on a per-generation-tool basis to determine whether mutation is a good metric for the comparison of test suites derived from a specific source. Since the analysis needs at least one test suite that detects the fault, the numbers of version pairs differ for different test generation tools.

3. ANALYSIS

Section 2 described our data collection procedure. This section describes the results of our analyses. Recall that for each injected fault we have the following test suites and their mutation scores: a developer-written test suite that does not detect the injected fault (T_{pass}), a developer-written test suite that detects the injected fault (T_{fail}), and a large number of automatically-generated test suites ($\tilde{T}$).

Studying the relationship between real fault detection and mutant detection, our goal was twofold. 1) We wanted to determine whether a test suite augmentation that led to the detection of a real fault increases the mutation score. 2) We wanted to determine whether a higher mutation score indicates a stronger test suite. For the latter, we were interested whether this relationship existed for developer-written test suites and whether it would hold for automatically-generated test suites.

3.1 Developer-written test suites

Recall that the developer-written test suites contained more than one triggering test for 20% of the faults. Since we created a test suite pair $\langle T_{pass},T_{fail} \rangle$ for each triggering test, there are 480 pairs

Table 4: Wilcoxon signed-rank tests for developer-written test suites. All differences are significant at $p < .001$.

Test suites	N	M_1, SD_1	M_2, SD_2	Z	r
Coverage constant	258	67.5, 14.3	68.2, 14.0	-10.7	.67
Coverage increased	222	54.7, 23.0	60.3, 19.4	-12.5	.84
Test added	431	60.9, 20.3	64.1, 17.5	-15.7	.76
Test modified	49	67.4, 14.3	68.1, 14.0	-4.9	.70
All	480	61.6, 19.9	65.5, 17.2	-16.5	.75

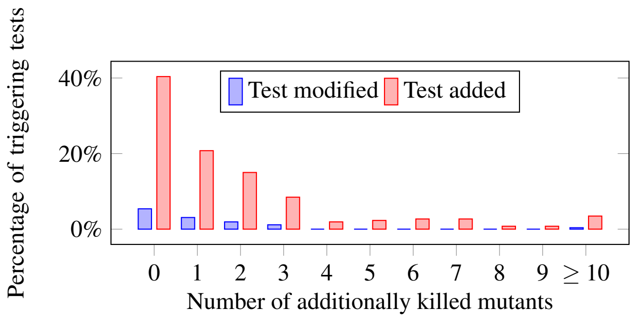

Figure 3: Number of mutants additionally killed by triggering tests that do not increase statement coverage. Total number of triggering tests (Test modified + Test added) is 258.

for the 357 faults. Of these 480 pairs, the mutation score of T_{fail} increased compared to T_{pass} for 362 of them (75%). Statement coverage increased only for 222 out of 480 (46%) pairs.

The mutation scores for the developer-written test suites were not normally distributed (evaluated by the Kolmogorov-Smirnov test), so a non-parametric statistical test is required. We ran the Wilcoxon signed-rank test over the 480 test suite pairs to determine whether the mutation score is significantly different between T_{pass} and T_{fail}. H_0 for this analysis is that there is no difference between the mutation scores between T_{base} and T_{pass} . Table 4 summarizes the results of the statistical analysis. The test suite augmentation ($T_{pass} \mapsto T_{fail}$) significantly increased the mean mutation score of T_{fail}. The mean difference in mutation score was statistically significant when coverage increased and when coverage did not increase. Coverage increased for 222 pairs and the mutation score increased for 209 out of those 222 pairs (94%). In contrast, the mutation score increased for 153 out of 258 pairs (59%) for which coverage did not increase.

3.1.1 *How many additional mutants does a test suite detect if one stronger test is added that detects a single real fault?*

In addition to determining whether the mutation score increased, we also measured the sensitivity of the mutation score with respect to a single real fault. This means that we measured the number of mutants that were additionally killed by the triggering test. Given the findings of the previous section, we again considered the influence of statement coverage. Figure 3 visualizes the number of additionally killed mutants if coverage did not increase and Figure 4 visualize the results when it did.

Overall, 118 out of 480 (25%) triggering tests did not detect any additional mutants while detecting a real fault. Considering that mutation analysis is used for test suite minimization, this result implies serious consequences. For 25% of all cases, test suite minimization

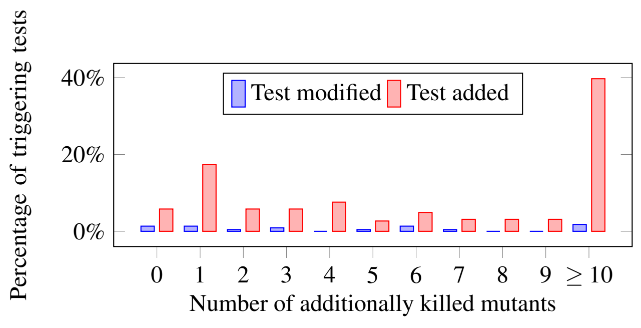

Figure 4: Number of mutants additionally killed by triggering tests that do increase statement coverage. Total number of triggering tests (Test modified + Test added) is 222.

(performed based on mutation results) would remove the triggering test as its removal does not affect the mutation score. Yet, removing the triggering test does decrease the real fault detection.

3.1.2 *Which real faults are not coupled to mutants?*

Given the findings of the previous section, we performed a qualitative study to investigate whether the results indicate a general limitation of mutation analysis for certain types of real faults.

We manually investigated all 95 real faults for which none of the corresponding 118 triggering tests increased the mutation score of its test suite. This means that for each of the 95 faults the mutation scores of T_{pass} and T_{fail} were equal for all triggering tests.

Table 5 summarizes the results of our manual analysis. We now discuss the three categories: cases where a mutation operator should be strengthened, cases where a new mutation operator should be introduced, and cases where no obvious mutation operator can generate mutants that are coupled to the real fault. In the latter case, results derived from mutation analysis do not generalize to those real faults.

Real faults requiring stronger mutation operators (25)

- *Statement deletion (12):* The statement deletion operator is usually not implemented for statements that manipulate the control flow. We surmise that this is due to technical challenges in the context of Java — removing `return` or `break`/`continue` statements changes the control flow and may lead to uninitialized variables or unreachable code errors. Figure 5a gives an example.

- *Argument swapping (6):* Arguments to a method call that have the same type can be swapped without causing type-checking errors. Argument swapping represents a special case of swapping identifiers, which is not a commonly-used mutation operator [21]. Figure 5b shows an example.

- *Argument omission (5):* Method overloading is error-prone when two methods differ in one extra argument — a developer might inadvertently call the method that requires fewer arguments. Figure 5c gives an example. Generating mutants for this type of fault requires a generalization of a suggested class-based mutation operator, which addresses method overloading to a certain extent [17].

- *Similar method called (2):* Existing mutation operators replace one method call by another only for calls to getter and setter methods. It would be both unfeasible and largely unproductive to replace every method call with every possible alternative that type-checks. Nonetheless, the method call replacement operator should be extended to substitute methods with related semantics. Figure 5d shows an example in which the fault is caused by using the wrong one of two similar methods (`indexOf` instead of `lastIndexOf`).

Table 5: Number of real faults not coupled to mutants generated by Major, categorized by the reason: a weak implementation of a mutation operator, a missing mutation operator, or no appropriate mutation operator exists.

	Weak operator	Missing operator	No such operator	Total
Chart	5 (19%)	1 (4%)	2 (8%)	8 (31%)
Closure	11 (8%)	2 (2%)	18 (14%)	31 (23%)
Math	4 (4%)	4 (4%)	30 (28%)	38 (36%)
Time	2 (7%)	0 (0%)	5 (19%)	7 (26%)
Lang	3 (5%)	0 (0%)	8 (12%)	11 (17%)
Total	25 (7%)	7 (2%)	63 (18%)	95 (27%)

```
    }
+   return false;
  }
  case 4: {
    char ch = str.charAt(0);
```
(a) LANG-365 fix

```
- Partial newPartial = new Partial(iChronology, newTypes,
      newValues);
+ Partial newPartial = new Partial(newTypes, newValues,
     iChronology);
```
(b) JodaTime-88 fix

```
- return solve(min, max);
+ return solve(f, min, max);
```
(c) MATH-369 fix

```
- int indexOfDot = namespace.indexOf('.');
+ int indexOfDot = namespace.lastIndexOf('.');
```
(d) Closure-747 fix

```
- return ... + toolTipText + ...;
+ return ... + ImageMapUtilities.htmlEscape(toolTipText)
    + ...;
```
(e) JFreeChart-591 fix

```
- return chromosomes.iterator();
+ return getChromosomes().iterator();
```
(f) MATH-779 fix

```
- FastMath.pow(2 * FastMath.PI, -dim / 2)
+ FastMath.pow(2 * FastMath.PI, -0.5 * dim)
```
(g) MATH-929 fix

Figure 5: Snippets of real faults that require stronger or new mutation operators.

Real faults requiring new mutation operators (7)

- *Omit chaining method call (4):* A developer might forget to call a method whose return type is equal to (or a subtype of) its argument type. Figure 5e gives an example in which a string needs to be escaped. A new mutation operator could replace such a method call with its argument, provided that the mutated code type-checks.

- *Direct access of field (2):* In case a class includes non-trivial getter or setter methods for a field (e.g., further side effects or post-processing), an object that accesses this field directly might cause an error. Figure 5f shows an example in which post-processing of the field `chromosomes` is required before the method `iterator()` should be invoked. A new mutation operator could replace calls to non-trivial getter and setter methods with a direct access to the field.

- *Type conversions (1):* Wrong assumptions about implicit type conversions and missing casts in arithmetic expressions can cause unexpected behavior. Figure 5g shows an example where the division should be performed on floating point numbers rather than integers (the replacement of the division by multiplication is unrelated to the real fault). A new mutation operator could replace a floating-point constant by an exact integer equivalent (e.g., replace `2.0` by `2`), remove explicit casts, or remove parentheses to manipulate operator precedence.

Real faults not coupled to mutants (63)

- *Algorithm modification or simplification (44):* Most of the real faults not coupled to mutants were due to incorrect algorithms. The bug fix was to re-implement or modify the algorithm. A bug fix that only removes special handling code also falls into this category.

- *Wrong method called (5):*

  Another common mistake is using a wrong method, which might either return wrong data or omit side-effects (e.g., method delegation with pre- or post-processing). Figure 6a shows an example for calling a wrong method. Note that this type of fault can be represented by mutants in theory. However, without deeper knowledge about the relation between methods in a program, replacing every identifier and method call with all alternatives would result in an unmanageable number of mutants.

- *Extraction of common code (4):* Suppose the access of a field that might be null is extracted to a utility method that includes a null check. A developer might forget to replace an instance of the field access with a call to this utility method. This rather subtle fault cannot be represented with mutants since it would require to inline the utility method (without the null check) for for every call. Figure 6b gives an example for this type of fault. The fault is that `this.startData` might be null — this condition is checked in `getCategoryCount()`. However, other tests directly or indirectly kill all mutants in `getCategoryCount()`, hence a test that exposes the fault does not kill any additional mutants.

- *Violation of pre/post conditions or invariants (3):* Some real faults were caused by the misuse of libraries. For example, the Java library makes assumptions about `hashCode` and `equals` methods of objects used as keys to a `HashMap`. Yet, a violation of this assumption cannot be generally simulated with mutants. Figure 6c gives an example for such a fault.

- *Numerical analysis errors (4):* Real faults caused by overflows, underflows, and improper handling of NaN values is poorly suited to be simulated by mutants, and hence also represents a general limitation. Figure 6d shows an example for a non-trivial case of this type of fault.

```
- return getPct((Comparable<?>) v);
+ return getCumPct((Comparable<?>) v);
```

(a) Math-337 fix

```
- if (categoryKeys.length != this.startData[0].length)
+ if (categoryKeys.length != getCategoryCount())
```

(b) JFreeChart-834 fix

```
- lookupMap = new HashMap<CharSequence, CharSequence>();
+ lookupMap = new HashMap<String, CharSequence>();
```

(c) LANG-882 fix[a]

[a]The result of comparing two `CharSequence` objects is undefined — the bug fix uses `String` to alleviate this issue.

```
- if (u * v == 0)
+ if ((u == 0) || (v == 0))
```

(d) MATH-238 fix

```
- {"\u00CB", "&Ecirc;"},
+ {"\u00CA", "&Ecirc;"},
+ {"\u00CB", "&Euml;"},
```

(e) LANG-658 fix

Figure 6: Snippets of real faults not coupled to mutants.

- *Specific literal replacements (3):* Literal replacement is a commonly used mutation operator that replaces a literal with a well-defined default (e.g., an integer with 0 or a string with the empty string). However, sometimes the real fault can only be exposed with a very specific replacement, which is not obvious. The literal replacement operator cannot simulate such a specific replacement. Figure 6e demonstrates an example that involves Unicode characters.

3.2 Automatically-generated test suites

The most common use of mutation testing is to compare automatically-generated test suites. We conducted an experiment to assess whether mutant detection of generated test suites is correlated with their real fault detection independent of coverage. In other words, is the mutation score generally a good proxy for test suite quality?

For the generated test suites we compared each $\tilde{T}_{pass}$ against every $\tilde{T}_{fail}$ for each program version. This means that if we have one $\tilde{T}_{pass}$ and four $\tilde{T}_{fail}$'s, we obtained four data points for our analysis. This within-subjects analysis focuses on differences between $\tilde{T}_{pass}$ and $\tilde{T}_{fail}$ while controlling for the subject program version.

In contrast to the first experiment where $T_{pass} \subset T_{fail}$, this property does not necessarily hold for two generated test suites $\tilde{T}_{pass}$ and $\tilde{T}_{fail}$. Therefore, we had to control for coverage in this experiment as this was not a property that could be naturally observed in the sample; we did this by only considering the intersection of mutants covered by both test suites.

The difference between mutation score in suites that detect real faults was significant for both EvoSuite and Randoop. The relationship held whether coverage was ignored (Table 6) or was held constant (Table 7). When controlling for coverage, the magnitude of the difference was smaller for EvoSuite, but not for Randoop.

3.3 Threats to validity

Our evaluation uses only 5 subject programs, all written in Java. Other programs might have different characteristics. Of specific note, all 5 subject programs are well-tested (see Table 1). This may limit the applicability of the results to programs that are not well-tested (e.g., programs under development).

Table 6: Paired t-test for generated test suites when coverage is ignored. All differences significant at $p < .001$.

Test suites	N	M_1, SE_1	M_2, SE_2	t	r
EvoSuite					
-branch	14,019	40.3, .19	44.5, .19	40.3	.19
-weak	10,278	40.7, .22	44.3, .22	29.9	.16
-strong	10,153	42.7, .25	48.4, .27	46.2	.22
Randoop	1,263	43.0, .67	46.8, .66	10.2	.16
All	175,163	37.4, .06	47.3, .06	200.9	.40

Table 7: Paired t-test for generated test suites when coverage is controlled for. All differences significant at $p < .001$.

Test suites	N	M_1, SE_1	M_2, SE_2	t	r
EvoSuite					
-branch	14,018	52.3, .14	55.9, .14	36.7	.22
-weak	10,276	54.6, .16	57.3, .16	25.4	.17
-strong	10,109	57.7, .19	62.4, .18	40.6	.25
Randoop	1,137	58.5, .58	63.0, .55	10.4	.24
All	168,604	53.3, .05	60.5, .04	167.3	.39

We evaluated only bug fix commits that span a single revision in the version control history. Our data collection methodology would miss bugs whose fixes and/or tests span multiple revisions, bug fixes that are not marked as such in the version control history, etc.

We could have considered every pair of consecutive commits or pairs across several commits to increase the number of reproducible faults. However, our methodology yielded a large number of faults nonetheless, and we have no reason to believe that our sample is biased.

When evaluating developer-written tests, our data contains only pairs $\langle T_{pass}, T_{fail}\rangle$ where $s_f(T_{pass}) = 0$ and $s_f(T_{fail}) = 1$. In other words, we did not consider the ways in which a developer might augment a test without increasing fault detection. When deciding whether to use mutation analysis, a developer wants to know, for an arbitrary addition to a test suite, whether an increased mutation score indicates increased quality (in terms of fault detection). We were not able to compute this because it is infeasible for us to determine, for an arbitrary change, whether there is any program fault that it detects. Our evaluation on automatically-generated test suites does consider ones that do and do not detect a single real fault.

4. RELATED WORK

This section discusses prior research related to our study on whether mutants are a valid substitute for real faults in software testing research. Moreover, it discusses research areas that rely on this assumption and the implications of our results.

4.1 Studies on the relationship between mutants and real faults

We are only aware of two previous studies that investigated the relationship of mutants and real faults — Table 8 summarizes these two studies.

Duran and Theévenod-Fosse [5] performed a study to determine whether mutations were similar to real faults. Test cases were randomly generated using information about the input domain, the program structure, and finite-state machine models of program and environment specifications. They found that errors (incorrect internal state) and failures (incorrect output) produced by programs containing mutants are similar to those produced by real faults. Of the 209 distinct errors and failures produced by mutants, 60% were among the 255 distinct errors and failures produced by real faults.

Table 8: Summary of the previous studies. All subjects investigated in both studies were written in C.

	Real programs	LOC	Real faults	Mutation operators	Mutants evaluated
[5]	1	1,000	12	Change constant Replace identifier Replace operator	1%
[1]	1	5,905	38	Replace constant Replace operator Negate branch condition Delete statement	10%

This study is limited in scope as it only involved one subject program with 1,000 lines of code, written in C. In addition, the study only employed three mutation operators and among the generated mutants, only 1% were arbitrarily chosen for evaluation.

Finally, the authors only compared the errors and incorrect internal states created by real faults and mutants; they did not compare the resulting failures. Testing research tends to focus on observed failures rather than internal program state. Therefore, while this study supports the idea that mutants are representative of real faults, the analysis cannot be directly used to draw conclusions about the correlation between real fault detection and mutant detection.

Andrews et al. [1] explored the relationship between mutants, seeded faults, and real faults. To test their subjects, the authors selected 5,000 test suites, each containing 100 tests, by randomly sampling a given test pool. The test pools were generated by creating random inputs and were augmented by control-flow graph coverage [10].

This study is also limited in scope. Among the eight subject programs, only one has any real faults. This program (Space) is written in C, contains roughly 5,900 lines of code and originates from a scientific domain (avionics). Furthermore, only 10% of the mutants produced were evaluated.

The authors found that for Space, there was no practically significant difference between the fault detection rate and the mutation score. However, for the seven programs with hand-seeded faults, the fault detection rate was lower than the mutation score — that is, mutants were easier to detect. After statistically ruling out other possible causes, the authors concluded that hand-seeded faults are not representative of real faults, but mutants are.

To the best of our knowledge, this paper is the first to undertake experimental evaluation of the relationship between mutants and real faults at such a scale in terms of number of real faults, number of mutants, subject size, and subject diversity. Moreover, previous work did not consider the impact of code coverage on the mutation score. While [1] considers program size, we believe that our study is the first to directly evaluate the effects of code coverage on the mutation score.

Another unique aspect of the study in this paper is dealing with Java, an object-oriented language. We are not aware of previous work evaluating the assumption that mutants are a valid substitute for real faults in the context of object-oriented programs. While previous work [17,19] suggested class level mutation operators, they are neither implemented in modern mutation tools that have been shown to work on large code bases (Major, Javalanche, PIT) nor are they commonly used in experiments. Even though we did not include class level operators when performing mutation analysis, our study addresses whether and how mutation analysis could benefit from adding specialized versions of those operators.

4.2 Software testing research using mutation analysis

The fundamental assumption that mutant detection and real fault detection are well correlated is widely accepted in software testing research. This can be witnessed in hundreds of research papers that used mutants or hand-seeded faults in their evaluations. This section discusses the subjects and type of faults that were commonly used in previous studies. It also discusses the research areas in which those subjects and faults were used.

4.2.1 Commonly used subjects

Google scholar lists approximately 1,400 papers that used the subjects of the Siemens benchmark suite [12] for their evaluations. This suite consists of 7 C programs, whose sizes vary between 141 and 512 LOC. Faulty versions of the subjects were obtained by manually seeding faults. The authors described these seeded faults in the subjects as follows: “The faults are mostly changes to single lines of code, but a few involve multiple changes. Many of the faults take the form of simple mutations or missing code.” [12] Since the seeded faults mostly represent ordinary mutants, our study directly confirms or refutes whether the results derived from using this suite generalize to real faults.

The publication summary of the software-artifact infrastructure repository (SIR) [24] lists more than 500 papers that reference it. SIR provides 81 subjects written in Java, C, C++, and C# and 36 of the subjects come with real faults. The median size of those 36 subjects is 120 LOC, ranging between 24 and 8,570. 35 out of the 36 subjects with real faults are written in Java. Unfortunately, the number of faults was not available to us. Compared to the subjects available in SIR, we developed a fault database for 5 real-world programs considering several years of development. Moreover, all 5 subjects feature comprehensive test suites.

4.2.2 Test generation

The core idea of mutation-based test input generation goes back to DeMillo et al. [6], who used constraints to capture under which conditions the execution state differs between test execution on a mutant and the original program. Solving these constraints together with symbolic path conditions leads to mutant killing test inputs. More recently this approach has been enhanced using dynamic symbolic execution (e.g., [22,27]). Bottaci proposed a variation of the constraint-based approach where the constraints are re-interpreted to guide search-based test generation. This was implemented by Ayari et al. [2] using the ant colony optimization meta-heuristic, and by Fraser et al. [8,9] using genetic algorithms. Fraser and Zeller [9] use the mutants not only to guide test generation, but also to guide generation of test assertions and to minimize test suites. Harman et al. [11] use a hybrid approach where dynamic symbolic execution is used to derive test inputs that reach mutants and infect the execution state, whereas search-based techniques similar to [9] are used to propagate the state changes to observable outputs.

However, none of the studies evaluated the effectiveness of the generated test suites with respect to real faults. Hence, all previous studies left open the question whether test suites derived from mutants are effective in practice.

4.2.3 Test prioritization and minimization

Prior research on test suite prioritization and minimization commonly used the hand-seeded faults of the Siemens suite to evaluate the effectiveness of prioritized or minimized test suites (e.g., [7,23]). Given the nature of faults in the Siemens test suite, which are essentially mutants, our study sheds light on whether the results of prior studies generalize to real faults.

4.2.4 Fault localization

Seeded faults and mutants are indispensable when evaluating a fault localization technique (e.g., [13]) since real faults are rare. Yet, given the findings of our qualitative study, which showed that a substantial number of real faults are not coupled to commonly used mutants, it is not clear whether a fault localization technique performs equally well on real faults and simple faults such as mutants or hand-seeded ones.

5. CONCLUSION

Mutants are intended to be used as practical replacements for real faults in software testing research and in practice by developers. This is valid only if mutation score is correlated with fault detection. Our study confirms this relationship empirically by examining 357 real faults on five large, mature, actively-maintained, open-source projects.

By examining mutants and real faults using developer-written tests, a statistically significant correlation was observed. This confirms that techniques such as test selection and prioritization can use mutants to evaluate developer-written tests. The data indicate that this is not only due to the increase of coverage, but also suggest a deeper, statistical coupling between mutants and real faults.

Our work also reveals cases in which a developer-written test could detect a fault without increasing mutation score. We identified ways to improve mutation testing by strengthening mutation operators or introducing new ones. We also discovered that almost 20% of faults cannot be simulated by mutations, which is a significant limitation of mutation testing. These results have practical implications for test selection or minimization techniques — a test suite that is minimized based on mutants might have a reduced real fault detection, even if the mutation score does not decrease.

Examining mutants and real faults on automatically generated tests by state of the art tools, EvoSuite and Randoop, also confirmed a statistically significant correlation. This implies that mutation score can be used as a proxy for test quality when comparing test generation techniques. This significance holds even if coverage is controlled for.

Acknowledgments

Some of computations in this work were performed using the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: `www.sharcnet.ca`).

6. REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2005.

[2] K. Ayari, S. Bouktif, and G. Antoniol. Automatic mutation test input data generation via ant colony. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1074–1081, New York, USA, 2007. ACM.

[3] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2006.

[4] Cobertura. The official web site of the Cobertura project, Accessed Jan 28, 2014. `http://cobertura.sourceforge.net`.

[5] M. Daran and P. Thevenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 1996.

[6] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering (TSE)*, 17(9):900–910, 1991.

[7] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE TSE*, 28(2):159–182, Feb. 2002.

[8] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering (ESEM)*, 2014. To appear.

[9] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 28(2):278–292, 2012.

[10] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE*, pages 60–71, May 2003.

[11] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 212–222. ACM, 2011.

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 191–200, 1994.

[13] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 273–282, 2005.

[14] R. Just, G. M. Kapfhammer, and F. Schweiggert. Do redundant mutants affect the effectiveness and efficiency of mutation analysis? In *Proceedings of the International Workshop on Mutation Analysis (Mutation)*, pages 720–725, 2012.

[15] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, pages 11–20, 2012.

[16] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 612–615, 2011.

[17] S. Kim, J. A. Clark, and J. A. McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proceedings of the Net. Object Days Conference on Object-Oriented Software Systems*, pages 9–12, 2000.

[18] A. Namin, J. Andrews, and D. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 351–360, 2008.

[19] J. Offut, Y.-S. Ma, and Y.-R. Kwon. The class-level mutants of mujava. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 78–84, 2006.

[20] A. J. Offutt. Investigations of the software testing coupling effect. *ACM TOSEM*, 1(1):5–20, Jan. 1992.

[21] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[22] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, pages 121–130. IEEE, 2010.

[23] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 34–43, 1998.

[24] SIR: Software-artifact Infrastructure Repository. SIR usage information, Accessed Mar 4, 2014. `http://sir.unl.edu/portal/usage.php`.

[25] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2008.

[26] M. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013.

[27] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.