How Closely are Common Mutation Operators Coupled to Real Faults?

Gregory Gay
Chalmers | University of Gothenburg
Gothenburg, Sweden
greg@greggay.com

Alireza Salahirad
University of South Carolina
Columbia, SC, USA
alireza@email.sc.edu

***Abstract*—In mutation testing, faulty versions of a program are generated through automated modifications of source code. These mutants are used to assess and improve test suite quality, under the assumption that detection of mutants is indicative of a test suite's ability to detect real faults—i.e., that mutants and faults have a *semantic* relationship. Improving the effectiveness—in both cost and quality—of mutation testing may lie in better understanding this relationship, in particular with regard to how individual mutation operators (types) couple to real faults.**

In this study, we examine coupling between 32,002 mutants produced by 31 mutation operators and 144 real faults, using a scale based on number of failing tests and reasons for failure. Ultimately, we observed that 9.92% of the mutants are strongly coupled to real faults, and 51.03% of the faults have at least one strongly coupled mutant. We identify and examine mutation operators with the highest median coupling, as well as the operators that tend to produce non-compiling mutants, undetected mutants, and mutants that cause tests other than those that detect the actual fault to fail. We also examine how coupling could be used to filter the set of operators employed, leading to potentially significant cost savings during mutation testing. Our findings could lead to improvements in how mutation testing is applied, improved implementation of specific mutation operators, and inspiration for new mutation operators.

***Index Terms*—Software Testing, Mutation Testing, Mutation Analysis, Fault Analysis, Mutation Operators**

I. Introduction

Software testing—the process of applying stimuli to software and judging the resulting reaction—is the most common means of ensuring that software operates correctly. When designing test cases, past experience can be used to estimate the potential effectiveness of the test suite. If we have known software **faults**—mistakes in the source code [1]—we can use detection of these faults to predict whether test cases will be effective against unknown future faults. Essentially, this is an estimation of the sensitivity of the test suite to changes in the source code. In practice, we typically lack a sufficiently large collection of faults to draw reasonable conclusions. Instead, we make use of synthetic faults, known as **mutants** [2].

Mutation testing [3] is a technique in which a user generates many mutants through automated modifications of the original code [2], [4]. **Mutation operators** define types of transformations over code structures [5]. For example, an operator may change one arithmetic operation into another—turning $A + B$ into $A * B$—permute the order of two statements, or remove a `static` modifier. There are many mutation operators [5], [6]. These operators vary in complexity and effect, but are intended to reflect mistakes that developers make.

Mutation testing is common technique in both research and practice to compare testing techniques [4] and suggest areas of improvement in test design [7] under the hypothesis that test suites that detect mutants will also detect real faults, as they are sensitive to these small changes in the code [8]. This hypothesis hinges on the idea that mutants can stand in for real faults. Mutants clearly bear little *syntactic* resemblance to real faults [9]—they tend to be simple, one-line changes to the code [10], [11]. Real faults often affect multiple lines of code and require complex changes to fix [9].

Instead, the idea that mutants can substitute for real faults is based on the assumption of a *semantic* relationship, built on two hypotheses. The first, the "competent programmer hypothesis", suggests that many programs are close to correct, with only minor changes required to fix them. The second, the "coupling effect", suggests that detection of many simple mutants will enable detection of a complex fault affecting the same code [1], [11]. However, the exact nature of the relationship between mutants and real faults is not clear. Past studies have found that many factors affect the potential correlation between mutant and fault detection [2], [4], [12]–[14]. Further, even if mutation testing can improve test quality, the immense cost of applying mutation to a large codebase [11] suggests the need for improvement in the implementation and application of the practice.

We hypothesize that improving the effectiveness—in terms of both cost and quality—of mutation testing lies in better understanding the semantic relationship between mutants and real faults, also known as their *coupling*. In particular, and in contrast to past studies, we focus on examining different mutation operators. That is, what degree of coupling do specific mutation operators have with real faults?

We investigate the degree of coupling by executing developer-written test suites against both mutated and faulty versions of classes from multiple open-source Java projects, based on 144 case examples from the Defects4J fault database [15]. In particular, we focus on the **trigger tests**—the tests that detect

the real fault. A mutant that is most strongly coupled to a real fault will be detected only by the trigger tests, and those tests will fail for the same reasons—i.e., the same exception or error. Mutants that are more weakly coupled may cause additional—or fewer—tests to fail or cause tests to fail for different reasons. We have defined a scale rating the strength of the coupling between a mutant and a corresponding fault, based on number of failing tests and reasons for failure. This scale, in turn, allows us to contrast 31 mutation operators. Ultimately, we observed:

- 61.08% of mutants are detected. 9.92% of the mutants are strongly coupled to faults, and a further 9.03% are strongly coupled with additional tests failing. 51.03% of the faults have at least one strongly coupled mutant.
- The level of coupling of individual mutants is relatively low—a median score of 2.00 (of 10.00). 16 mutation operators (45.71%) have a median score < 2.00.
- EMM, $ASRS$, ISD, COI, $PRVOU_{SMART}$, and EOC yield mutants with the highest median coupling. The average EMM or $ASRS$ mutant strongly substitutes for corresponding faults.
- ISI, JTI, AMC, OAN, and LVR have the lowest median scores. They largely produce mutants resulting in compilation errors.
- JTD, SOR, $AODU$, $PRVOR_{SMART}$, and $AORU$ have the largest percentage of mutants not detected. $PRVOR_{SMART}$ yields subtle mutants with, often, strong coupling. The other operators could be selectively useful, but may yield equivalent mutants or cause non-trigger tests to fail.
- SOR, $PRVOR_{REFINED}$, $AORB$, AOD, and EOA have the largest percentage of mutants that are *only* detected by non-trigger tests. These mutants lack a significant relationship with their corresponding faults.
- Using past coupling to filter operators could offer cost reductions while retaining the power of mutation testing to assess test suite sensitivity. In our experiment, a ≥ 4.0 threshold yields an 81.48% reduction in the number of mutants while retaining a diverse subset of operators and mutants with strong coupling.

Understanding this semantic relationship could enable improvements in how mutation testing is applied. For example, identifying strongly-coupled operators allows prioritization during testing. Exclusion of weakly-coupled operators could lead to cost savings and filtering of "noise" from test suite adequacy estimation. In addition, understanding semantic coupling enables potential improvements in the implementation of existing mutation operators—e.g., ensuring that mutants will compile—and may suggest new mutation operators. To inspire future research, we also make our data available[1]

[1]Available from https://doi.org/10.5281/zenodo.7261554.

II. Background

Mutation Testing: Mutation testing [3] is a technique where a user generates many faulty versions of program through modifications of the original code, generally through automated code transformation [2], [4]. Usually a single modification is made to form each **mutant**, such as changing an expression (e.g., substituting addition for subtraction). The mutations introduced match one or more models of mistakes that developers make (**mutation operators**). Each mutation operator reflects a repeatable program change that can be automatically imposed on statements that fit the correct pattern.

While mutations are *individually* simpler than real faults, two hypotheses suggest that, *together*, mutants are helpful for determining the thoroughness of a test suite. The "Competent Programmer Hypothesis" [8], [16] states that programmers tend to develop programs that are close to correct. Although there may be faults, such faults can be corrected with a few simple changes. Mutations are intended to represent simple changes that are made in practice [16]. The "Coupling Effect" [8] states that tests that distinguish a large number of mutants from the original program are so sensitive that they also will implicitly distinguish more complex errors as well. This hypothesis postulates that mutation testing is an effective sensitivity analysis, and that test suites that detect more mutants are sensitive to even subtle real-world faults.

Generally, mutants are introduced with the intent that they not be trivially detected—they are both syntactically valid and semantically useful [1]. That is, effective mutants will compile, and will not trivially cause test cases to fail [1]. Mutations can be used to assess the effectiveness of a test suite by examining how many mutants are **killed** (that is, detected) by the tests within the test suite. The **mutation score** divides the number of killed mutants by the total number of mutants. Mutants are considered **equivalent** if no test can corrupt program state for that mutant. Deciding equivalence is generally undecidable [17], but techniques exist that can determine equivalence for a subset of mutations [11].

Fault Analysis: It is important to first establish concepts and terminology related to faults and fault analysis. We broadly adopt the same conventions followed by our experimental subject, Defects4J [15]. In this study, each fault meets the following three properties:

1) Each case example consists of faulty and fixed source code versions. Changes imposed by the fix must be to source code, not to other artifacts(e.g., build files).
2) Each fault must be reproducible—all tests pass on the fixed version and at least one test fails on the faulty version, exposing the fault.
3) Each fault is isolated—faulty and fixed versions differ only by a minimal set of changes related to addressing the fault, free of unrelated changes (e.g., refactoring).

When discussing faults, we use the following terminology:

- **Trigger tests** are developer-written test cases that fail only on the faulty version.
- **Modified classes** are classes altered to fix the fault.
- **Relevant tests** are all test cases that load at least one of the modified classes. These are the full set of tests that could detect mutations of modified classes. Relevant tests include all trigger tests, as well as additional tests that pass on both the fixed and faulty versions of modified classes (but could still detect mutants).
- **Loaded classes** are classes loaded by the Java Virtual Machine during execution of relevant tests.

III. Methodology

We hypothesize that improving the quality and cost effectiveness of mutation testing requires examining the *semantic* relationship between mutation operators and real faults. In this study, we investigate this semantic relationship by assessing the degree of coupling between mutants and real faults using a spectrum of outcomes, based on the *number of failing trigger tests* and the *reasons for failure*. A mutant that is strongly coupled will be detected by only the trigger tests, and those tests will fail for the same reasons—i.e., will trigger the same exception or error. Mutants that are more weakly coupled may cause additional—or fewer—tests to fail or cause tests to fail for different reasons. Specifically, we address the following research questions:

- **RQ1:** What is the degree of coupling between mutants and real faults?
  - **RQ1.1:** Which mutation operators yield mutants that most *strongly* couple to faults?
  - **RQ1.2:** Which mutation operators yield mutants that tend to result in compilation errors or lack of detection?
  - **RQ1.3:** Which mutation operators yield mutants that tend to be detected only by non-trigger tests?
- **RQ2:** What potential cost savings could be achieved by selectively omitting weakly coupled mutation operators?

To answer these questions, we performed the following:

1) **Collected Case Examples:** We have used 144 case examples, from five Java projects, as case examples (Section III-A).
2) **Generated Mutants:** For each fixed modified class for each case example, we generated mutants for the specific lines of code that differ between the faulty and fixed versions of the classes. We perform this generation using 31 mutation operators offered by the muJava++ framework (Section III-B).
3) **Executed Test Suites:** For each mutant, we execute all relevant tests from the developer-written test suite (Section III-C).
4) **Recorded Failure Information:** For each mutant, we measure the number of failing trigger tests, number of failing non-trigger tests, and the reasons for failure (i.e., exceptions or error messages) (Section III-C).

Table I: ID numbers of studied faults from Defects4J. Faults in bold lacked strongly-substituting mutants.

Project	Bugs
Chart	1, **2**, 3, 4, **5, 6, 7**, 9, 10, 11, 13, **14**, 15, **16, 17, 18, 19**, 21, **22**, 24, **25**, 26
Closure	9, 12, **23, 34, 52**, 56, **65, 77, 85, 99, 100, 102, 123**, 124, **128**, 131, **147, 161, 162, 164**, 169, **173**
Lang	**2, 4**, 5, **7**, 11, **12**, 16, **19, 20**, 22, 23, 24, 27, 28, **29, 30**, 31, **34, 37**, 39, **40**, 42, **43**, 44, 45, **46, 47, 48**, 49, **51**, 52, 54, 55, **57**, 58, **59**, 60, 61, 62, 63, 65
Math	3, 5, **8**, 9, 11, **15, 17**, 19, **22**, 23, 24, **27, 29, 30, 37**, 40, 41, **43, 46, 47**, 48, 49, 51, 53, **54**, 56, 60, **64, 66**, 67, **68, 69**, 70, 72, 73, 82, 84, 85, **89**, 95, **96**, 97, **98**, 102, 103, 105, 106
Time	2, **3, 4, 5, 6, 7, 12**, 14, 15, 16, 18, **27**

5) **Assessed Coupling:** For each mutant, we use the information gathered above to assess the degree of coupling of that mutant to the real fault using a scale that reflects the outcomes of relevant test execution (Section III-D).

A. Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [15][2] used extensively in test generation [18], automated program repair [19], and fault localization [20] experiments. We generate mutants for modified classes for 144 faults from five projects: Chart (22 faults), Closure (22), Lang (41), Math (47), and Time (12). The specific faults are listed in Table I. Some faults from the version used, 1.5.0, were excluded because (a) the lines of code that differ between the faulty and fixed versions of classes yielded no mutants for the employed mutation operators, (b) the lines of code that we attempted to mutate used Java features not supported by muJava++, or (c), muJava++ failed to produce usable mutants due to errors.

B. Mutant Generation

We used the muJava++ mutation framework, an extended version of muJava[3]. We employ muJava++ because (a) it offers a large number and variety of mutation operators, (b) it can be applied to user-specified lines of code, and (c), it can export mutants as Java files for execution and analysis.

Mutation Operators: muJava++ supports 62 mutation operators, although some operators have multiple variations. We were able to produce compiling mutations for 31 of the operators. These operators are explained in Table II.

Mutant Generation: We generate mutants by applying all operators to the *specific lines of code* that differ between the fixed and faulty versions of all modified classes. We apply this restriction because we are interested in the mutants that could semantically replicate an actual fault. It is unlikely that a mutant in another class or an unrelated portion of a modified class could replicate the real fault. Therefore, mutants outside of these lines potentially mislead the analysis.

We generate all possible mutants for all mutation operators for these lines of code. We apply operators to the fixed code, as

[2] Available from http://defects4j.org
[3] Available from https://github.com/saiema/MuJava.

Table II: Mutation operators used from muJava++.

Operator Name	Description
AMC	Changes the access modifier of methods and class fields.
AOD	Replaces an arithmetic operation with one of its members. For example, (a = b + c) becomes (a = b) or (a = c).
$AODU$	Deletes basic unary arithmetic operators. (+, -)
$AOIS$	Inserts short-cut arithmetic operators (++, --).
$AOIU$	Inserts unary arithmetic operators (+, -).
$AORB$	Replaces arithmetic operators (*, /, %, +, -) with other operators.
$AORS$	Replaces short-cut arithmetic operators (++, --) with other operators.
$AORU$	Replaces unary arithmetic operators (+, -)
$ASRS$	Replace short-cut assignment operators (+=, -=, *=, /=, %=, &=, |=, ˆ=, $<<=$, $>>=$, $>>>=$) with other operators.
COD	Deletes conditional operators (&&, ||, &, |, ˆ).
COI	Inserts conditional operators (&&, ||, &, |, ˆ).
COR	Replaces conditional operators (&&, ||, &, |, !) with other operators.
$CRCR$	A constant C in the code is mutated to be one of (1, 0, -1, -C, C+1, C-1).
EAM	Changes an accessor method name for other compatible accessor method (where methods have the same signature).
EMM	Changes a setter method name for other compatible setter method (where methods have the same signature).
EOA	Replaces an assignment of an object reference with a copy of the object, using the `clone()` method. Only performed if the object has a declared `clone()` method.
EOC	Changes an object reference check to object content comparison through Java's `equals()` method.
ISD	Deletes occurrences of the `super` keyword so that a reference to a variable or method is no longer to the parent class' variable or method.
ISI	Inserts the `super` keyword so a reference to a variable or method in a child class uses the parent variable or method.
JTD	Deletes uses of the keyword `this`.
JTI	Inserts uses of the keyword `this`.
LOI	Inserts logical operators (&, |, ˆ).
LVR	Replaces a literal with a default value. Numeric literals become (0, 1, -1), Booleans become (`true`, `false`), Strings are replaced with an empty string.
OAN	Changes the number of arguments in method invocations, but only if there is an overloading method that can accept the new argument list.
$PRVOL_{SMART}$	The $PRVO$ operator changes object references in assignment statements to instead refer to other objects of a compatible type. $PRVOL$ mutates references on the left-hand side of the assignment, and mutations must be compatible with the right side. $PRVOL_{SMART}$ only uses references to reachable variables.
$PRVOR_{SMART}$	Same as $PRVOL_{SMART}$, except applied to the right-hand side of the assignment, and mutations must be compatible with the left side.
$PRVOR_{REFINED}$	Same as $PRVOR_{SMART}$, except it also uses literals found inside the method.
$PRVOU_{SMART}$	Same as $PRVOR_{SMART}$, except applied to `return` expressions.
$PRVOU_{REFINED}$	Same as $PRVOR_{REFINED}$, except applied to `return` expressions.
ROR	Replace relational operators with other relational operators (==, !=, $<$, $\leq$, $>$, $\geq$), or replace an entire predicate with true and false.
SOR	Replaces shift operators ($>>$, $<<$, $>>>>$) with other operators.

the mutants represent alternative "buggy" versions. Table III lists the number of mutants produced for each operator.

muJava++ exports each mutant as a Java file that can be substituted for the real file during test execution. This allows us to execute mutants using Defects4J's test execution capabilities. This also enables further qualitative analysis through inspection of mutated code.

Table III: Number of mutants produced for each operator across each project from Defects4J (and overall).

Operator	Chart	Closure	Lang	Math	Time	Overall
$PRVOU_{REFINED}$	5287	4880	3125	3663	1724	18679
$PRVOU_{SMART}$	540	299	53	2760	54	3706
$PRVOR_{REFINED}$	121	31	297	940	226	1615
$AOIS$	185	120	492	295	68	1160
ROR	139	152	489	223	113	1116
ISI	142	121	312	338	66	979
$PRVOR_{SMART}$	92	159	24	368	77	720
LOI	68	44	278	262	51	703
JTI	84	63	211	269	42	669
$CRCR$	51	27	223	236	59	596
$AORB$	4	4	160	204	24	396
$PRVOL_{SMART}$	95	0	57	216	4	372
COI	44	47	149	67	22	329
$AOIU$	27	5	73	147	22	274
COR	9	90	129	18	6	252
AOD	6	2	57	112	12	189
EAM	11	1	40	0	44	96
COD	5	8	13	2	0	28
$ASRS$	0	0	0	18	0	18
EOC	7	2	6	1	1	17
EOA	8	0	4	0	4	16
LVR	0	4	10	2	0	16
AMC	0	0	12	3	0	15
$AORS$	5	1	3	6	0	15
$AORU$	4	0	1	10	0	15
$AODU$	4	0	1	8	0	13
OAN	0	1	5	5	0	11
SOR	0	0	0	4	0	4
ISD	1	0	2	0	0	3
JTD	1	0	0	1	0	2
EMM	0	0	0	2	0	2
Overall	**6932**	**6061**	**6223**	**10169**	**2617**	**32002**

C. Data Collection

To examine the relationship between mutants and faults, we execute the relevant tests against all mutated versions of the modified classes for each case example. To perform test execution, we use the `defects4j test` utility, which executes the relevant tests in a controlled execution environment that ensures that expected behavior is preserved and that all constraints and assumptions of the dataset hold. Specifically, we perform the following for each mutant:

- We checkout the fixed version of the case example and exchange the modified class for the mutant.
- We execute `defects4j compile`. If the mutant does not compile, we abort execution and record the result.
- We execute `defects4j test` and record any tests that fail or result in an error, as well as the stack trace and any error messages.
- We compare the test failures and reasons for failure to the trigger tests for that case example. The metadata for Defects4J includes the stack traces and messages for each trigger test, preserving the reasons that trigger tests fail for the real fault.

We check that the same assertions fail or that the same exceptions are thrown. However, we do not check that the same output is issued. For example, an assertion might check

Table IV: Number and percentage of detected mutants for each project (and overall).

Project	Detected	Total	Percentage
Chart	4175	6932	60.23
Closure	3072	6061	50.68
Lang	4160	6223	66.85
Math	6625	10169	65.15
Time	1515	2617	57.89
Overall	**19548**	**32002**	**61.08**

that the return value is 10. If a mutant returns 7 and the fault returns 12, we consider the "reason" to be the same—the return value was not 10—even though the mutant and fault do not return the same incorrect value.

Based on the test executions, we create a dataset with one line per mutant execution. For each mutant, we record:

- **Basic metadata:** the project, fault ID, modified class, mutant ID, mutation operator, and number of trigger tests.
- **Test failure data:** the number of tests that failed or resulted in an error, the number of trigger tests that failed or resulted in an error, the number of trigger tests that failed for the same reason, and the number of failing non-trigger tests.

D. Coupling Categorization

To assess the coupling of mutants to faults, we employ the following scale. This scale accounts for all possibilities when executing relevant tests against each mutant:

- **Strong Substitution:** All trigger tests fail for the same reasons that they failed for the real fault. No additional tests fail. This represents a semantic replacement of the real fault, given the developer-written test suite.
- **Test Substitution:** All trigger tests fail, but one or more fail for differing reasons. No additional tests fail.
- **Partial Substitution:** Some, but not all, trigger tests fail. All failing trigger tests fail for the same reasons. No additional tests fail.
- **Partial Test Substitution:** Some, but not all trigger tests fail. Not all failing trigger tests fail for the same reasons. No additional tests fail.
- **(Strong/Test/Partial/Partial Test) + Additional Tests Fail:** The same definitions as above apply, but additional non-trigger tests fail.
- **No Substitution:** Only non-trigger tests fail.
- **Mutant Not Detected**
- **Compilation Error**

For each mutant, we assign a categorization from this spectrum of possibilities. In some analyses, we also assign a numeric value: (0) Compilation Error, (1) Not Detected, (2) No Substitution, (3) Partial Test + Additional, (4) Partial Test, (5) Partial + Additional, (6) Partial, (7) Test + Additional, (8) Test, (9) Strong + Additional, and (10), Strong Substitution.

Table V: Number of faults with at least one corresponding "strongly substituting" mutant.

Project	With Strongly Substituting Mutants	Total	Percentage
Chart	12	22	54.55%
Closure	6	22	27.27%
Lang	23	41	56.10%
Math	28	47	59.57%
Time	5	12	41.67%
Overall	**74**	**144**	**51.03%**

IV. Results and Discussion

A. Coupling Between Mutants and Real Faults (RQ1)

Table IV presents an overview of the number and percentage of mutants detected by the relevant test cases for each project from Defects4J. As a baseline, we observe:

> Overall, 61.08% of mutants are detected.

This percentage is relatively consistent across projects, with the lowest percentage being 50.68% in Closure. The remaining mutants either are not detected (18.31%) or resulted in compilation errors (20.58%).

Table VI categorizes each result according to the scale previously defined in Section III-D. We also assign a numeric score to each category, with higher scores indicating closer coupling. Table V further indicates the number of faults with at least one mutant categorized as "strong substitution".

> 9.92% of mutants are strongly coupled to real faults, and a further 9.03% are strongly coupled with additional tests failing. 51.03% of faults have at least one strongly coupled mutant.

The strongly-substituting mutants can serve as stand-ins for the real faults, yielding the same failing test cases and the same test outcomes. Approximately half of the studied faults have at least one strongly-substituting mutant, with relatively consistent results across all projects—other than Closure, where only 27.27% of faults have strongly substituting mutants.

The overall percentage of strongly-substituting mutants falls behind compilation errors (20.58%), not detected (18.31%), no substitution (11.39%)—where only non-trigger tests fail–and partial substitution (12.73%)—where only a subset of trigger tests fail, but those that fail offer the same reasons for failure. Test substitution cases—where trigger tests fail, but for alternative reasons—are rarer than strong substitutions, but still present. Mutations to these lines still cause test cases to fail, but they do not replicate the semantic effect of the fault. Commonly, these are cases where a mutation causes an exception to be thrown when one was not expected.

The distribution of mutants belonging to each category varies somewhat between projects. The Closure project particularly stands out, as only 0.94% of mutants strongly couple to the faults. In this project—among the detected mutants—the largest category is test substitution (with additional failing

Table VI: Categorization of coupling for each mutant, for each project (and overall).

Category	Score	Chart	Closure	Lang	Math	Time	Overall
Compile Error	0	1405 (20.23%)	2724 (44.94%)	785 (12.60%)	1440 (14.16%)	237 (9.05%)	6591 (20.58%)
Not Detected	1	1352 (19.47%)	265 (4.37%)	1278 (20.52%)	2104 (20.68%)	866 (33.07%)	5865 (18.31%)
No Substitution	2	68 (0.98%)	486 (8.02%)	647 (10.39%)	1778 (17.48%)	669 (25.54%)	3648 (11.39%)
Partial Test + Additional	3	74 (1.07%)	251 (4.14%)	436 (7.00%)	40 (0.39%)	18 (0.69%)	819 (2.56%)
Partial Test Substitution	4	453 (6.52%)	0 (0.00%)	256 (4.11%)	110 (1.08%)	0 (0.00%)	819 (2.56%)
Partial + Additional	5	21 (0.30%)	401 (6.62%)	305 (4.90%)	143 (1.41%)	279 (10.65%)	1149 (3.59%)
Partial Substitution	6	2071 (29.82%)	107 (1.77%)	541 (8.69%)	1345 (13.22%)	13 (0.50%)	4077 (12.73%)
Test + Additional	7	131 (1.89%)	1276 (21.05%)	148 (2.38%)	467 (4.59%)	212 (8.10%)	2234 (6.98%)
Test Substitution	8	25 (0.36%)	0 (0.00%)	362 (5.81%)	333 (3.27%)	11 (0.42%)	731 (2.28%)
Strong + Additional	9	374 (5.39%)	494 (8.15%)	568 (9.12%)	1296 (12.74%)	159 (6.07%)	2891 (9.03%)
Strong Substitution	10	958 (13.80%)	57 (0.94%)	897 (14.40%)	1113 (10.94%)	153 (5.84%)	3178 (9.92%)

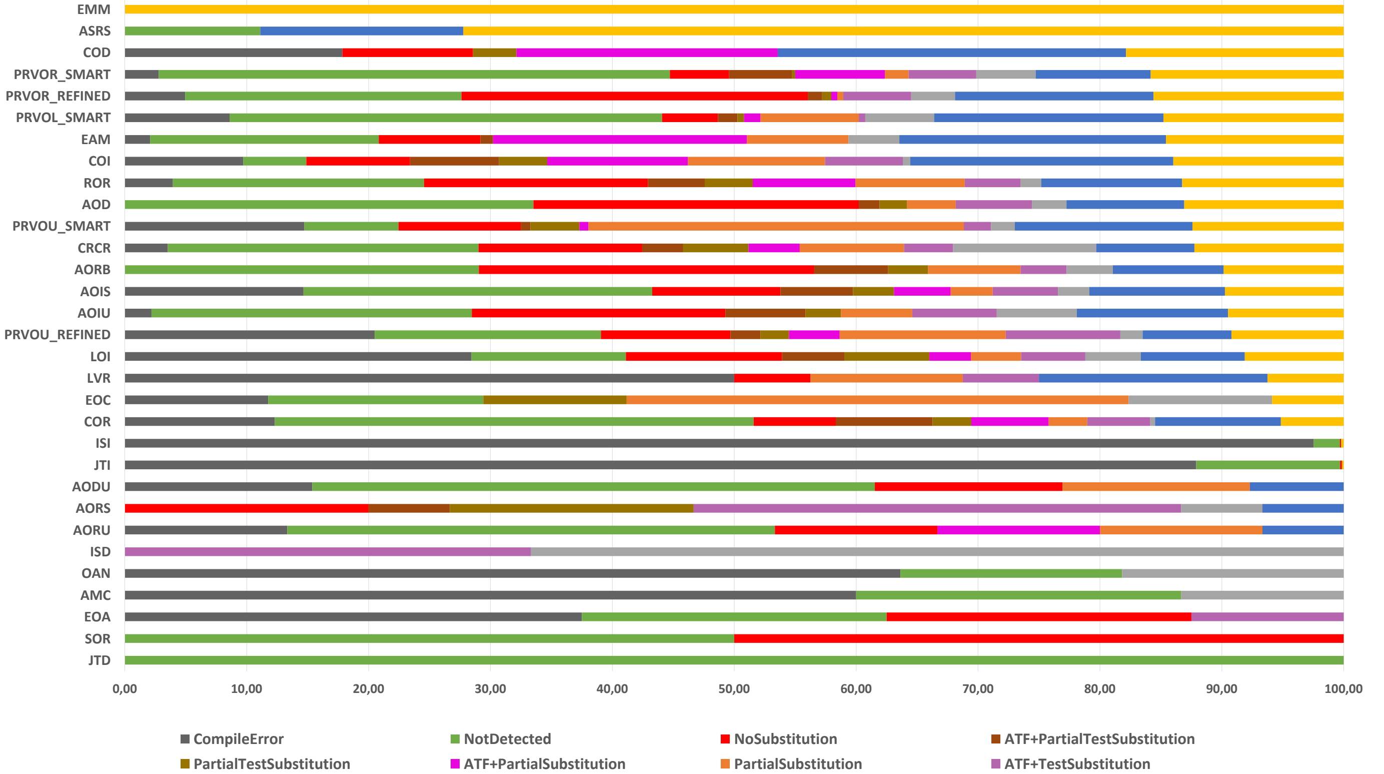

Figure 1: Percentage of mutants for each operator matching each category, sorted by the percentage strongly substituting.

tests). The Closure compiler has complex validity checking code[4] for the abstract syntax tree produced during compilation. Many mutations are caught by this code. This produces a large number of test failures, often for alternative reasons than expected by the test designers.

The percentage of mutants belonging to each category is depicted for each mutation operator in Figure 1. Using the assigned values, we also note the median and average "scores", as well as the standard deviation and number of mutants, for each operator in Table VII. Overall, we observe:

> The level of coupling of individual mutants is low—a median of 2.00 of 10.00. 16 mutation operators (45.71%) have a median score < 2.00.

[4]e.g., https://github.com/google/closure-compiler/blob/ca23c597fc17c95ece2aae07f4e503a34c88c61f/src/com/google/javascript/jscomp/ValidityCheck.java.

We will discuss specific operators in more detail in the coming subsections. However, we can make initial observations about variance. We observe the highest variance from the LVR, $PRVOL_{SMART}$, $PRVOR_{REFINED}$, COD, and $PRVOR_{SMART}$ operators. EMM, JTD, ISD, ISI, and SOR have the lowest variance. The level of variance does not suggest a particular relationship with the level of coupling. Operators with low variance can tend towards strong coupling (e.g., EMM) or not being detected at all (e.g., JTD or SOR). Operators with high variance tend to span the range of outcomes—e.g., COD is split between compilation errors/lack of substitution and strong substitution/strong with additional tests. However, operators with high variance should be considered further in future research. Implementation details of such operators could be considered. Certain types of statements could potentially be avoided or prioritized with the goal of increasing the potential coupling.

Table VII: For each mutation operator, the number of mutants, median and average coupling score, and standard deviation in coupling score.

Operator	Mutants	Median	Average	Std. Dev.
EMM	2	10.00	10.00	0.00
$ASRS$	18	10.00	8.83	2.79
ISD	3	7.00	7.67	0.47
COI	329	6.00	5.69	3.36
$PRVOU_{SMART}$	3706	6.00	5.21	3.44
EOC	17	6.00	4.65	2.95
COD	28	5.00	5.79	3.73
EAM	96	5.00	5.69	3.38
$AORS$	15	4.00	5.33	2.33
$CRCR$	596	4.00	4.73	3.46
ROR	1116	4.00	4.65	3.41
$PRVOL_{SMART}$	372	3.00	4.73	3.94
$AOIU$	274	3.00	4.42	3.44
$PRVOR_{SMART}$	720	3.00	4.38	3.72
$PRVOU_{REFINED}$	18679	3.00	3.97	3.50
$PRVOR_{REFINED}$	1615	2.00	4.62	3.79
AOD	176	2.00	4.09	3.52
$AORB$	396	2.00	3.98	3.29
$AOIS$	1160	2.00	3.81	3.57
LOI	703	2.00	3.55	3.54
COR	252	1.00	3.24	3.23
$AORU$	15	1.00	2.73	2.65
$AODU$	13	1.00	2.39	2.68
EOA	8	1.00	1.63	2.18
SOR	4	1.00	1.50	0.50
JTD	2	1.00	1.00	0.00
LVR	16	0.00	3.63	4.01
OAN	11	0.00	1.64	3.02
AMC	15	0.00	1.33	2.65
JTI	669	0.00	0.14	0.51
ISI	976	0.00	0.04	0.48
Overall	**32002**	**2.00**	**4.01**	**3.58**

B. Strongly-Coupled Mutation Operators (RQ1.1)

> EMM, $ASRS$, ISD, COI, $PRVOU_{SMART}$, and EOC yield mutants with the highest median coupling. The average EMM or $ASRS$ mutant strongly substitutes for its fault.

The EMM operator replaces a setter method reference for another compatible setter. EMM mutants are very rare—both mutations were for fault Math-106[5], where `setIndex(...)` was changed to `setErrorIndex(...)` in different occurrence. ISD mutants—which delete the `super` keyword—are also quite rare. It is difficult to generalize from few examples, so both should be further examined in future work.

EOC changes `==` to `.equals()`. For example, in Lang 39[6], `replacementList[i] == null` is changed to `replacementList[i].equals(null)`. EOC mutants are also relatively rare—with only 17 examples, mostly in Chart and Lang. While some of these strongly substitute, the majority result in partial substitution.

The $ASRS$ operator replaces short-cut assignment operators, e.g., changing `+=` to `/=`. All 18 mutants for this operator appear in the Math project. Because the Math project focuses on mathematical functions, such mutants may also be more likely in this project to match the actual mistakes made by developers. For example, in Math 102[7], multiple ASRS mutations led to the same result as the fault.

COI and $PRVOU_{SMART}$ both yield a much larger number of faults. Naturally, the median coupling for both is lower than the three rarer operators, but is still high. COI inserts conditional operators. For example, in Math 84[8], COI changes an instance of `true` to `!true`. This causes a `return` in the same (incorrect) location as the real fault.

$PRVOU_{SMART}$ replaces object references with other compatible references—variables or methods—in `return` expressions. For example, in Math 5[9], it replaces a reference to INF with calls to float-returning methods (e.g., `this.atan()`) from the class-under-test. Because both COI and $PRVOU_{SMART}$ are widely applicable—but still tend towards a strong relationship to real faults—they are potentially useful for use in assessing and expanding test suites.

C. Compilation Errors and Non-Detection (RQ1.2)

We observe two primary reasons for very low median scores—either a large percentage of mutants result in compilation errors or are not detected. We discuss both situations below.

> ISI, JTI, AMC, OAN, and LVR have the lowest median scores. They largely produce mutants resulting in compilation errors.

All five operators have a median score of 0.00, indicating that the majority of mutants result in compilation errors. This is confirmed in Figure 1. These operators make changes that can easily "break" code without proper precautions. For example, ISI inserts the `super` keyword, JTI inserts the `this` keyword, and AMC changes access modifiers for methods and fields. All three can yield useful mutants, but they also assume conditions of the code that may not be true—e.g., in the case of ISI, not all classes have parents.

Mutations resulting in compilation errors—for these and other operators—are not useful for evaluating the strength of a test suite. Although some time is saved by not needing to execute the test suite against these mutants, time and effort are still wasted on attempting compilation and analyzing the resulting failure. Cost savings could be achieved from either avoiding the use of such mutation operators altogether or improving their implementation such that compilation errors are avoided.

The developers of mutation frameworks should explore measures that would prevent generation of non-compiling mutants.

[5] https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/106.src.patch

[6] https://github.com/rjust/defects4j/blob/master/framework/projects/Lang/patches/39.src.patch

[7] https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/102.src.patch

[8] https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/84.src.patch

[9] https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/5.src.patch

For example, certain types of code structures could be avoided. At the very least, mutation testing frameworks should check whether assumptions are met. In the above examples, the ISI operator implementation should check that a class has a parent before inserting the `super` keyword or the implementation of the JTI operator should check whether a call is to a static method before inserting `this`.

> JTD, SOR, $AODU$, $PRVOR_{SMART}$, and $AORU$ have the the largest percentage of undetected mutants. $PRVOR_{SMART}$ yields subtle mutants with, often, strong coupling to real faults. The other operators could be selectively useful, but may yield many equivalent mutants or cause non-trigger tests to fail.

Mutants that do not compile detract from the effectiveness of mutation testing. Those that are not detected can either detract—if they are equivalent to the original code—or can be *very* useful—if they are subtle and require sensitive test cases to detect.

The JTD operator deletes uses of the keyword `this`. There were only two mutants of this type in our set, and both were equivalent. There are many situations where this operator could yield equivalent mutants. Therefore, we would suggest only employing this operator in cases where behavior might be affected significantly when the keyword is removed.

The $AODU$ operator—which deletes unary arithmetic operators, e.g., changing -1 to 1—and $AORU$ operator—which replaces such operators—may also be selectively useful when unary operators are employed. However, both also lack strong coupling with faults. When such mutants are detected, tests outside of the trigger tests tend to fail.

SOR replaces shift operators (e.g., $>>$) with other operators. This operator produced four mutants for Math 40[10], where two were not detected and two caused non-trigger tests to fail. Shift expressions are typically only used in specialized code, but it seems this operator could be useful for assessing test suite adequacy when such operators are used.

$PRVOR_{SMART}$ is similar to the previously-discussed $PRVOU_{SMART}$. It replaces object references with other compatible references on the right-hand side of assignment expressions. When mutants are detected, they often strongly substitute for real faults. Some of the not-detected mutants are equivalent. Many, however, could be detected with the addition of further test cases. Therefore, this operator could be useful in improving test suite quality.

D. Detection By Non-Trigger Tests (RQ1.3)

One further situation to examine is when mutation operators tend to yield mutants only detected by non-trigger tests. This is the "no substitution" category in our scale.

[10]https://github.com/rjust/defects4j/blob/master/framework/projects/Math/patches/40.src.patch

> SOR, $PRVOR_{REFINED}$, $AORB$, AOD, and EOA have the the largest percentage of mutants that are only detected by non-trigger tests. These mutants lack a significant relationship with their corresponding faults.

$PRVOR_{REFINED}$ is an extended version of $PRVOR_{SMART}$, discussed previously. The difference between these operators is that $PRVOR_{REFINED}$ is able to also reference literals found in the method. The use of literals seems to lead to a large number of cases where tests outside of the trigger tests fail, including both "no substitution" and "additional tests fail" outcomes.

$AORB$—which replaces arithmetic operators—and AOD—which replaces an entire arithmetic expression with one of its member variables—yield mutants that span the entire spectrum of possibilities. These operators lead to many "no substitution" outcomes, as well as many cases where mutants are not detected or strongly substitute. Based on these observations, as well as the earlier observations of similar operators often producing mutants that are not detected, it seems that mutation operators related to arithmetic expressions lack a predictable semantic relationship with real faults. Arithmetic expressions are common when programming. Developers *do* make mistakes involving such expressions. However, such expressions also are not predictive of the existence of a fault.

The EOA operator replaces an assignment of a object reference with a clone of that object, in situations where the `clone()` operation is defined. Such mutants are relatively rare, but largely fall into the "not detected" and "no substitution" categories. This operator may be of selective use in cases where `clone()` is implemented.

E. Potential Cost Savings Through Filtering Operators (RQ2)

Mutation testing is a notoriously expensive practice, as test cases must be executed against each mutant [7]. For mutation testing to be a viable technique in industrial development, that cost must be reduced. Past research has examined methods of filtering the set of mutants or mutation operators employed (e.g., [21], [22]).

Similarly, the degree of coupling of an operator to a set of known faults could be used to select a subset of mutation operators—focusing on operators that have previously been observed to have a close relationship to real faults. If a threshold is carefully selected, this could lead to a small subset of mutants that are—we hypothesize—useful for assessing the strength of existing test cases and for targeting in the design of additional test cases because weakly-coupled operators have been filtered.

There are many possible methods of using the assessed coupling for selecting a subset of operators for assessing the test suites of new projects. We explore one such method—using the median "score" of an operator to determine its inclusion in the subset employed. In this scenario, we would select a

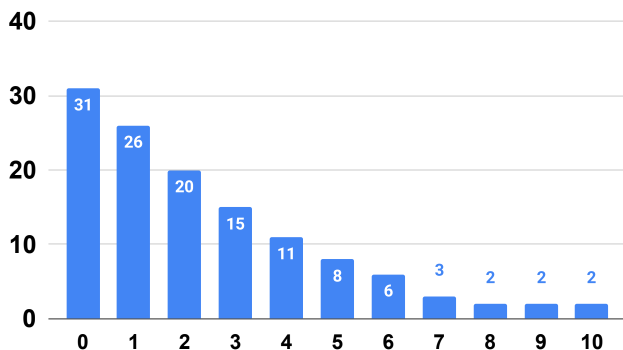

Figure 2: Number of operators remaining if the median level of coupling is used as a threshold for determining the subset of operators employed.

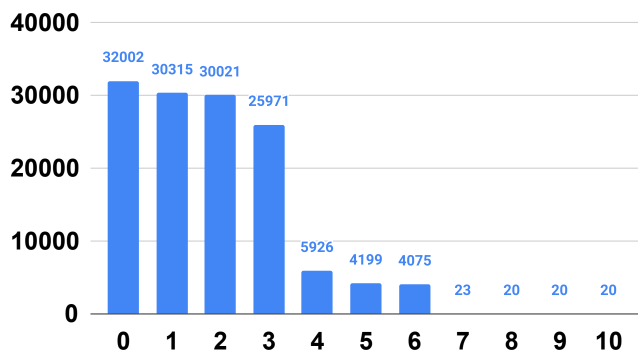

Figure 3: Number of mutants remaining if the median level of coupling is used as a threshold for determining the subset of operators employed.

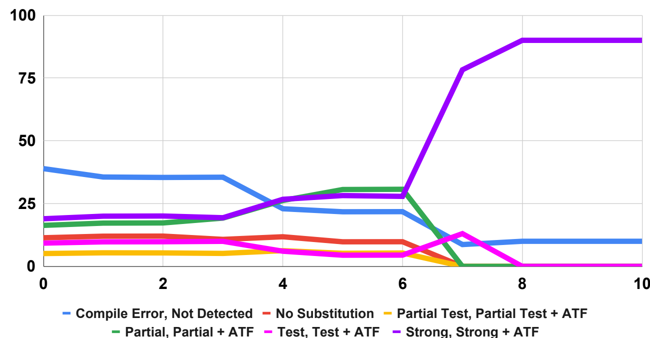

Figure 4: Percentage of mutants in the remaining subset (if median level of coupling is used as a threshold) belonging to each coupling category.

score threshold and compare the median score for an operator to this threshold. If it falls above this threshold, we would include it in the subset employed. If not, the operator would be removed.

Figure 2 indicates the number of mutation operators that would fall in this subset for all thresholds (0–10). Figure 3, then, indicates the number of mutants in the set considered in our experiment that would be included in that subset. Finally, Figure 4 indicates the distribution of mutants in this subset matching the different levels of coupling.

Figure 2 shows a steady drop in the number of operators. However, there are massive reductions in the number of mutants in the subset (Figure 3) when the threshold moves from 3–4, due to the loss of the $PRVOU_{REFINED}$ operator, and from 6–7, due to the $PRVOU_{SMART}$ operator.

There are multiple thresholds that could make sense. Ultimately, a developer should utilize a subset with a reasonable *variety* of operators to ensure that tests are sensitive to different types of faults. In addition, that subset should still contain a reasonably high number of mutants—while still remaining cost-effective—to ensure that tests are robust across a wide span of the codebase. If a threshold is too low, the set of mutants will remain unreasonably expensive to assess. If it is too high, the set of mutants will be too small and lack diversity—reducing the power of mutation testing to assess the sensitivity of a test suite across the codebase.

Based on Figures 2–3, a threshold of 4–6 seems most reasonable. In the scale defined previously, we would expect the "average" mutant in this subset to range from "partial test substitution" to "partial substitution". These threshold yield a reasonably-sized set of mutants compared to lower thresholds, while still retaining a variety of operators. As shown in Figure 4, the distributions of coupling categories for the mutants in the subsets considered in our experiment are relatively stable in this range, with approximately 25% being strongly coupled and approximately 75% being detected. Higher thresholds yield an increasing percentage of strongly coupled mutants, but the total number of mutants in the subset becomes so small that the subset would lose its utility to assess the sensitivity of test cases to changes to the code.

> Using median level of coupling to filter operators could offer cost reductions while retaining the power of mutation testing to assess test suite sensitivity. In our set of mutants, a $\geq$ 4.0 threshold yields an 81.48% reduction in the number of mutants while retaining a diverse subset of strongly coupled operators.

V. Threats to Validity

External Validity: Our study has focused on five open-source Java projects—a relatively small number—from a single benchmark. Therefore, we cannot claim that our results will generalize to all fault types, projects, or languages. Nevertheless, we believe that the projects studied are representative of, at minimum, other small to medium-sized Java systems. The projects are popular case examples, used in past research. We also believe that Defects4J offers enough fault

examples that our results are generalizable to similar projects. As Defects4J has been applied in many studies and we have made data available, our results can be compared with related research, replicated, and extended.

Internal Validity: We have used 31 mutation operators from a single framework. Other frameworks may offer different operators. muJava++ was the only framework that met our experimental constraints. It offers a large variety of operators—more than Major, and substantially overlapping with PIT. muJava++ also offers operators not available in PIT. We believe that the operators employed are sufficiently varied.

We use only developer-written test suites to examine coupling. These tests do not form a complete specification of behavior, meaning that strong coupling with respect to the existing tests may not hold with additional test cases. However, the developer-written tests reflect the intent and priorities of the developers. For the studied case examples, the test suites are often extensive. Therefore, these tests are appropriate for use for determining the coupling between mutants and faults.

VI. Related Work

Many researchers have examined whether tests that detect mutants also detect real faults (e.g., [2], [4], [13], [14], [23]–[25]). Andrews et al. suggest that, when using appropriate mutation operators and removing equivalent mutants, mutant detection predicts for fault detection [4], [24]. Just et al. found a significant correlation between mutant and fault detection [2]. They examined the relationship between mutants and faults using a simple definition of coupling—if any test fails, for any reason, there is coupling. They find that 73% of real faults are coupled to at least one mutant, but the number of mutants coupled to each fault is small. Moreover, conditional operator replacement, relational operator replacement, and statement deletion mutants are more often coupled than other operators. In contrast, Papadakis et al. found that—when test suite size is controlled—the correlation between mutant and fault detection is weak [13]. They also examine coupling, using a similarity measurement based on test failures and code coverage. Mutants that affect the same statements as a real fault and that are detected by tests that detect the real fault are considered more similar. They find that less than 1% of mutants represent the behavior of real faults. In contrast to our study, they do not consider the reasons that tests fail, consider tests independent, and do not examine mutation operators.

Gopinath et al. [9] found that the syntactic differences between fixed and buggy program are usually *not* equivalent to traditional mutation operators. They dispute the competent programmer hypothesis—observing that syntactic difference between faulty and fixed programs is often significant. A common assumption is that mutants that are syntactically similar to real faults will also be semantically similarity. Using the same notion of semantic similarity as Papadakis et al., Ojdanic et al. found that syntactic similarity has no predictive relationship to semantic similarity [26].

Broadly, we differ from prior work in our methodology for examining coupling. Our focus is on semantic similarity, rather than syntactic. In addition, we use a complex scale, rather than a simple binary assessment or a score that does not account for failure reasons. Past work did not tend to differentiate operators. We also employ a larger number of mutation operators. We additionally perform a large-scale experiment, making use of many real faults for multiple, complex Java projects, each containing many different classes. Collectively, these factors enable a thorough analysis of how specific mutation operators relate to real faults. We do not examine whether there is a correlation between mutant and fault detection. However, our research is complementary in that the degree of coupling between an operator and faults could impact the correlation between mutant and fault detection. Future work should consider this factor.

VII. Conclusion

We hypothesize that improving the effectiveness—in terms of both cost and quality—of mutation testing lies in better understanding the semantic relationship between mutants and real faults. Ultimately, we observed that 9.92% of the mutants are strongly coupled to real faults, and 51.03% of the faults have at least one strongly coupled mutant. EMM, $ASRS$, ISD, COI, $PRVOU_{SMART}$, and EOC yield mutants with the highest median level of coupling. ISI, JTI, AMC, OAN, and LVR have the lowest median scores. They largely produce mutants resulting in compilation errors. JTD, SOR, $AODU$, $PRVOR_{SMART}$, and $AORU$ have the largest percentage of mutants that are not detected. SOR, $PRVOR_{REFINED}$, $AORB$, AOD, and EOA have the largest percentage of mutants that are only detected by non-trigger tests. These mutants *are* detected, but lack a significant relationship with the corresponding real faults. We also found that using the median coupling to filter operators could offer cost reductions while retaining the power of mutation testing to assess test suite sensitivity. In our set of mutants, a ≥ 4.0 threshold yields an 81.48% reduction in the number of mutants while retaining a diverse subset of operators and mutants with strong coupling.

Understanding this semantic relationship could enable improvements in how mutation testing is applied, improved implementation of specific mutation operators, and inspiration for new mutation operators. We plan to explore all three further in future work by (1) further analyzing the operators identified above in experiments on additional fault examples, (2) expanding the range of mutation operators considered and contrasting implementations of operators from different frameworks, (3) empirically evaluating cost savings and impact on mutation score from different filtering methods, and (4), exploring how coupling could be used to design and generate (e.g., via machine learning [27]) new mutation operators.

VIII. Acknowledgements

This research was supported by an ASPIRE-1 grant from the University of South Carolina.

References

[1] M. Pezze and M. Young, *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.

[2] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?” in *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, Hong Kong, November 18–20, 2014, pp. 654–665.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2nd ed. New York, NY, USA: Cambridge University Press, 2016.

[4] J. Andrews, L. Briand, Y. Labiche, and A. Namin, “Using mutation analysis for assessing and comparing testing coverage criteria,” *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608 –624, aug. 2006.

[5] 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 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 612–615. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100138

[6] J. Offutt, Y.-S. Ma, and Y.-R. Kwon, “The class-level mutants of mujava,” in *Proceedings of the 2006 International Workshop on Automation of Software Test*, 2006, pp. 78–84.

[7] G. Petrović and M. Ivanković, “State of mutation testing at google,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 163–171. [Online]. Available: https://doi.org/10.1145/3183519.3183521

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[9] R. Gopinath, C. Jensen, and A. Groce, “Mutations: How close are they to real faults?” in *25th International Symposium on Software Reliability Engineering*, Nov 2014, pp. 189–200.

[10] Y. Jia and M. Harman, “Higher order mutation testing,” *Information and Software Technology*, vol. 51, no. 10, pp. 1379 – 1393, 2009, source Code Analysis and Manipulation, SCAM 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584909000688

[11] ——, “An analysis and survey of the development of mutation testing,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[12] J. Andrews, L. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” *Proc of the 27th Int’l Conf on Software Engineering (ICSE)*, pp. 402–411, 2005.

[13] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, “Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 537–548. [Online]. Available: https://doi.org/10.1145/3180155.3180183

[14] M. Kim, N. Kim, and H. P. In, “Investigating the relationship between mutants and real faults with respect to mutated code,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 30, no. 08, pp. 1119–1137, 2020.

[15] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[16] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Mutation analysis.” GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, Tech. Rep., 1979.

[17] A. J. Offutt and J. Pan, “Automatically detecting equivalent mutants and infeasible paths,” *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1997. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291099-1689%28199709%297%3A3%3C165%3A%3AAID-STVR143%3E3.0.CO%3B2-U

[18] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, “Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges,” in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE 2015. New York, NY, USA: ACM, 2015.

[19] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, “Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset,” *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, Aug 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9470-4

[20] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 609–620. [Online]. Available: https://doi.org/10.1109/ICSE.2017.62

[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*, vol. 5, no. 2, p. 99–118, 1996.

[22] W. Ma, T. Titcheu Chekam, M. Papadakis, and M. Harman, "Mudelta: Delta-oriented mutation testing at commit time," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021.

[23] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 158–171, 1996.

[24] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings. 27th International Conference on Software Engineering*, 2005, pp. 402–411.

[25] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, "How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2426–2463, 2018.

[26] M. Ojdanic, A. Garg, A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Syntactic vs. semantic similarity of artificial and real faults in mutation testing studies," *arXiv preprint arXiv:2112.14508*, 2021.

[27] T. Titcheu Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, no. 1, pp. 434–487, 2020.