AN ABSTRACT OF THE DISSERTATION OF

Rahul Gopinath for the degree of Doctor of Philosophy in Computer Science presented on June 9, 2017.

Title: On the Limits of Mutation Analysis

Abstract approved: ______________________________

Carlos Jensen

Mutation analysis is the gold standard for evaluating test-suite adequacy. It involves exhaustive seeding of all small faults in a program and evaluating the effectiveness of test suites in detecting these faults. Mutation analysis subsumes numerous structural coverage criteria, approximates fault detection capability of test suites, and the faults produced by mutation have been shown to be similar to the real faults.

This dissertation looks at the effectiveness of mutation analysis in terms of its ability to evaluate the quality of test suites, and how well the mutants generated emulate real faults. The effectiveness of mutation analysis hinges on its two fundamental hypotheses: The *competent programmer hypothesis*, and the *coupling effect*. The *competent programmer hypothesis* provides the model for the kinds of faults that mutation operators emulate, and the *coupling effect* provides guarantees on the ratio of faults prevented by a test suite that detects all simple faults to the complete set of possible faults. These foundational hypotheses determine the limits of mutation analysis in terms of the faults that can be prevented by a mutation adequate test suite. Hence, it is important to understand what factors affect these assumptions, what kinds of faults escape mutation analysis, and what impact interference between faults (coupling and masking) have.

A secondary concern is the computational footprint of mutation analysis. Mutation analysis requires the evaluation of numerous mutants, each of which potentially requires complete test runs to evaluate. Numerous heuristic methods exist to reduce the number of mutants that need to be evaluated. However, we do not know the effect of these heuristics on the quality of mutants thus selected. Similarly, whether the possible improvement in representation using these heuristics are subject to any limits have also not been studied in detail.

Our research investigates these fundamental questions to mutation analysis both empirically and theoretically. We show that while a majority of faults are indeed small, and hence within a finite neighborhood of the correct version, their size is larger than typical mutation operators. We show that strong interactions between simple faults can produce complex faults that are semantically unrelated to the component faults, and hence escape first order mutation analysis. We further validate the *coupling effect* for a large number of real-world faults, provide theoretical support for fault coupling, and evaluate its theoretical and empirical limits. Finally, we investigate the limits of heuristic mutation reduction strategies in comparison with random sampling in representativeness and find that they provide at most limited improvement.

These investigations underscore the importance of research into new mutation operators and show that the potential benefit far outweighs the perceived drawbacks in terms of computational cost.

©Copyright by Rahul Gopinath
June 9, 2017
All Rights Reserved

On the Limits of Mutation Analysis

by

Rahul Gopinath

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 9, 2017
Commencement June 2017

Doctor of Philosophy dissertation of Rahul Gopinath presented on June 9, 2017.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Rahul Gopinath, Author

ACKNOWLEDGEMENTS

I came to United States from India with hopes of doing a great Ph.D. Unfortunately, circumstances mostly beyond my control resulted in a number of setbacks both financial and personal. I had hit the nadir in terms of my hopes and stamina when I met my advisor Dr. Carlos Jensen. He accepted me to his group and believed in me. He made sure that he was approachable, provided the guidance where I required it, and gave me freedom to pursue what I thought was interesting for my research. He helped me overcome my problems due to the cultural shift, and helped me regain my confidence. He helped me understand what to look for in choosing a good research path, in writing good papers, in choosing a career and even in giving a good presentation. He taught me to value ethics, to be inclusive, to care about diversity, and to go the extra mile in order to make what I do accessible and acceptable to everyone. Hence, I would like to express my sincere gratitude to him for being there, and for being my advisor. Carlos, I couldn't have completed my Ph.D. without your help. I would also like to thank my co-advisor Dr. Alex Groce for always being there to discuss my research. He always helped me to clarify my thoughts, pointed out the inconsistencies or implicit assumptions where they existed, and helped me shape the direction of my research. As a second language speaker of English, my language has often been less than stellar, and I would like to thank him for spending countless hours making my research actually readable. A share of credit for my work goes to my co-authors Mohammad Amin Alipour, Iftekhar Ahmed, and also to Amirhosein Emerson Azarbakht for his careful reviews.

I would also like to express my heartfelt gratitude towards Praveen Chandrasekharan for motivating and guiding me towards my Ph.D. Praveen mentored me, believed in me, and helped me when I really needed it. Praveen, your help and guidance are sincerely appreciated!.

A major share of credit goes to my beloved wife Dhara Sheth for standing by me and supporting me even at the cost of her own career. She made my path toward Ph.D. smooth, took the role of an accountant, an assistant, and a manager so that I could focus on my Ph.D. Without her, this Ph.D. would not have been possible. Dhara, more than me, this Ph.D. belongs to you.

CONTRIBUTION OF AUTHORS

Task	Conception	Data Collection	Empirical Analysis	Final Draft
Chapter 2	RG, AG, CJ	RG	RG	RG, AG, CJ
Chapter 3	RG	RG	RG	RG, AG, CJ
Chapter 4	RG, AG, CJ	IA, AA, RG	RG	RG, AG, CJ

Table 1: Author Contributions

The following authors contributed to the manuscript: Iftekhar Ahmed (**IA**), Amin Alipour (**AA**), Rahul Gopinath (**RG**), Alex Groce (**AG**), Carlos Jensen (**CJ**). Their individual contributions to various papers are summarized in Table 1.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF ALGORITHMS

Chapter 1: Introduction

The ideal measure of the efficacy of a test suite is its fault detection adequacy based on a set of faults discovered during the lifetime of the program under consideration. However, it is often hard to measure even in a research setting [42] due to the unavailability of an exhaustive set of real faults for a given program. The most popular alternative is the coverage criteria [7], which describes the degree to which various program structures were exercised by the test suite. However, relying on the coverage criteria misses out on the actual verification of execution results – that is, the strength of its assertions. If one considers both parts – coverage, and assertion strength, mutation score is considered to be the best proxy [7,31] for the efficacy of a test suite. The mutation score is obtained from mutation analysis[1], which involves the introduction of simple syntactic changes to the program, followed by evaluation of the ability of the test suite under consideration to detect these changes. Indeed, it is easy to show that mutation analysis subsumes numerous coverage criteria [7,69,81] in that a corresponding set of mutants can only be detected if a given criterion is satisfied. A mutation adequate test suite detects more bugs than the coverage adequate (edge-pair, all-uses, prime-paths) test suites [61], and produces faults that are more similar to real-world bugs when compared to hand-seeded faults [8,9]. We also found that mutation score tracks the fault detection capability of test suites [3]. These factors make mutation analysis an attractive tool for researchers investigating test suite quality.

Mutation analysis assumes two fundamental hypotheses [31]: The *competent programmer hypothesis*[2] and the *coupling effect*. The *competent programmer hypothesis* asserts that a non-pathological [17] program written by a competent programmer is syntactically very close to the correct version. The *coupling effect* is concerned with the semiotics[3] of fault interaction. According to the *coupling effect*, complex faults are coupled to simple faults such that a test suite capable of detecting all simple faults in isolation will, with high probability, detect most of the complex faults in a program [55,76,77].

[1]Also called program mutation and mutation testing.
[2]Also called the finite neighborhood hypothesis
[3]The relation between syntax and semantics of faults

The effectiveness of mutation analysis as a test suite adequacy measure is reliant on these foundational hypotheses. However, whether the real world faults actually obey these axioms, and if they do, to what extent, has not been adequately investigated. Indeed, it is possible that not all the faults are covered by the *competent programmer hypothesis* and the *coupling effect*. Secondly, mutation analysis suffers from a heavy computational footprint [17, 55], which makes traditional mutation analysis infeasible for even moderately large code bases. Hence, numerous heuristics have been proposed to limit the number of mutants evaluated. However, it is an open question whether these heuristics are uniformly good for all programs, and whether there are any limits to the improvement that they can provide.

We expand on the limits of mutation analysis in the coming sections.

1.1 Limits of the Foundational Hypotheses of Mutation Analysis

In this section, we consider the limits of mutation analysis due to the foundational hypotheses.

1.1.1 The *competent programmer hypothesis*

The *competent programmer hypothesis* asserts that programmers tend to make small mistakes. That is, given a program P, and the set of mutants described by $\Phi_\delta(P)$, which are at most δ changes away from P, the *competent programmer hypothesis* asserts that the correct program lies within $\Phi_\delta(P)$. The traditional mutation operators rely on a stricter assertion in that they typically tend to produce mutants that differ from the correct version by a single token. That is, $\delta = 1$. Indeed, if we consider large programs with numerous bugs, this traditional interpretation without qualifications is obviously incorrect. However, the idea is that if we consider specific independent bugs, these tend to be fairly small. While it is certainly plausible, are a majority of fault fixes similar to the mutation operators that are traditionally single token modifications? What effect does the language have, on the syntactic size of faults? Are some languages more likely to lead to larger or smaller fixes? The mutation operators implicitly assume that the possibility of a fault at any point in the source code is proportional to the number of valid modifications that can be made at that point. Is this assumption correct? Does the ratio of possible mutations correspond to the actual faults? The correspondence of the *competent programmer hypothesis* to the real world determines the faults that mutation analysis can emulate and hence prevent. This thesis investigates the limits to the correspondence of competent programmer hypothesis and the real world, especially under different languages.

1.1.2 The *coupling effect*

The *coupling effect* asserts that test suites that can detect all simple faults can detect a high percentage of complex faults. Mutation analysis relies on the *coupling effect* to limit the mutation operators to only first order modifications. Unfortunately, the *coupling effect* is rather ambiguous as to what is a simple fault and vague about the ratio of higher order faults that can be expected to be found by the test suite that detects all simple faults. An exhaustive empirical validation of the *coupling effect* assumption is infeasible as there is an exponential number of higher order mutants for any set of first order mutants. Hence, empirical evaluation has been limited to just until mutants of third order [76]. Previous theoretical analysis [92] has been limited in that it considers only programs with same domain and range. It is also incomplete in that it does not account for recursion and iteration, which are common constructs in programming. This thesis provides a stronger axiomatization of the *coupling effect*– The *composite fault hypothesis: Tests detecting a fault in isolation will (with probability $K \approx 99\%$) continue to detect the fault even when it occurs in combination with other faults.* Our theory of composite faults clarifies what an *atomic* fault is, and provides theoretical support and strong empirical evidence for the *composite fault hypothesis*. The *composite fault ratio* is the lower limit of the *general coupling ratio*. That is, we can expect more test cases to fail for complex faults than for the component faults due to the strong interaction between faults.

1.2 Limits of Mutation Reduction Strategies

The biggest problem with mutation analysis is its heavy computational footprint due to the enormous number of mutants that need to be evaluated for even a moderately large program. Numerous strategies have been proposed to reduce the number of mutants to be evaluated, and these strategies often report a high ratio of reduction. However, what impact these strategies have, on the effectiveness of the reduced mutant set, and whether the improvement in effectiveness has any limits has not been investigated. This thesis comprehensively investigates the limits of mutation reduction strategies in terms of improvement possible with respect to a random sample of mutants of the same size. We show that there is a theoretical limit to the amount of improvement possible using any mutation reduction strategies, and provide empirical evidence for the same.

This thesis investigates these fundamental limits in mutation analysis empirically as well as theoretically.

1.3 Research Goals

This dissertation has the following high-level goals

1. Investigate the limits of the *competent programmer hypothesis*: Identify the limitations in using first order mutation to emulate simple faults, especially with regard to the typical size of real faults, and the factors that affect the distribution of faults. Investigate whether all atomic faults are first order faults.

2. Investigate the limits of the *coupling effect*: Investigate the interference between faults in terms of both constructive (coupling effect) and destructive (fault masking) interference. Can we rely on coupling effect even on extremely large code bases?

3. Evaluate the empirical and theoretical limits of redundancy reduction using selective mutation strategies.

1.4 Structure

This thesis is based on the following papers:

- Chapter 2 presents our paper "*Mutations: How close are they to real faults?*" which was published at *ISSRE 2014*. This paper primarily investigates the first high-level goal. It investigates real-world faults in $5,000$ real-world programs especially with regard to the validity of the *competent programmer hypothesis*. We investigate the typical program neighborhood, and the real-world distribution of faults corresponding to different mutation operators. We also evaluate the impact of programming language on the distribution and size of changes.

- Chapter 3 presents our paper "*The Theory of Composite Faults*" which was published at *ICST 2017*. In this paper, we investigate the first as well as the second high-level goal, and we make two major contributions: As the first contribution, we investigate the precise definitions of what an atomic fault is, and also whether all atomic faults are first order. We further provide a stronger theory of faults: *The composite fault hypothesis*. Our second contribution is an empirical evaluation of the general coupling effect, resulting in precise empirical values of the general coupling ratio as well as the composite fault ratio.

- Chapter 4 presents our paper "*On The Limits Of Mutation Reduction Strategies*" which was Published at *ICSE 2016*. This paper investigates the third high-level goal. We make two main contributions. The first is a theoretical framework to model the limits of mutation reduction strategies against random samples given the distribution of mutants. We further evaluate the theoretical limits of selection strategies in a simple system with uniform mutant redundancy. The second contribution is the identification of empirical limits of heuristic mutation reduction strategies in selecting non-redundant mutants.

- Chapter 5 concludes this dissertation and presents our ideas on how to take our research forward.

1.5 Contributions

The contributions of this dissertation are:

1. Identification of the size of typical program neighborhoods, and the impact of programming language on the distribution of faults. (Chapter 2)

2. A formal and precise definition of the *atomic fault* (Chapter 3): *An atomic fault is a fault that can not be semantically decomposed.*

3. An improved theory of fault masking called the *theory of composite faults* (Chapter 3) *Composite fault hypothesis: Tests detecting a fault in isolation will (with probability $K \approx 99\%$) continue to detect the fault even when it occurs in combination with other faults.*

4. Empirical evaluation of general coupling ratio and the composite fault ratio (Chapter 3)

5. A theoretical framework for evaluating the limit of improvement possible using selective reduction strategies given the distribution of mutants (Chapter 4)

6. A precise empirical limit for improvement possible using selective reduction strategies in real world (Chapter 4)

Mutations: How close are they to real faults?

Rahul Gopinath, Carlos Jensen, Alex Groce

2014 IEEE International Symposium on Software Reliability Engineering
Naples, Italy
pp. 189-200

Chapter 2: Mutations: How close are they to real faults?

2.1 Introduction

Mutation analysis is a fault injection technique originally proposed by Lipton [63] and is often used in software testing. It is used as a means of comparison between different testing techniques [9], as a means of estimating whether a test suite has reached adequacy [103], and as a means of emulating software faults for the purposes of estimating software reliability [15]. In fact, the validity of mutation analysis is an assumption underlying considerable work in other suite evaluation techniques, such as code coverage criteria [42].

Mutation analysis involves systematic transformation of a program through introduction of first order syntactical changes, and determines whether tests can distinguish the mutated code from the original (presumed correct) source code. A mutation score, which measures how many mutants were distinguished from the original code by at least one test, is used as a measure of the effectiveness of the test suite [7] because it is believed to correlate well with the effectiveness of the test suite in detecting real faults [8].

Mutation analysis relies on two assumptions: (1) the *competent programmer hypothesis* and (2) the *coupling effect* [18]. The *competent programmer hypothesis* suggests that the version of program produced by a competent programmer is close to the final correct version of a program, while the *coupling effect* claims that a test suite capable of catching all the first order mutations will also detect most of the higher order mutations. In practice, a strong *competent programmer hypothesis* (that for programs of any size the initial version is syntactically close to correct) is fairly obviously incorrect for large programs. However, mutation analysis only rests on a weaker version: that the *competent programmer hypothesis* holds with respect to each individual fault.

For mutation analysis to be successful, the mutants it produces should ideally be similar in character to the faults found in real software. This property has been used in practice by many researchers [9, 40, 47, 83], for generating plausible faults[1]. While this

[1]These articles do not explicitly call upon the *competent programmer hypothesis*, but we believe

has been investigated by a few researchers [8, 27, 75, 86], the evidence is largely based on the real faults from a single program. Further, except for the study by DeMillo et al. [86], the similarities investigated were constrained to the error trace produced [27], and the ease of detection [8, 75].

The existing body of work, especially by DeMillo et al. underscores the necessity of further studies, with a much larger sample of programs, especially in light of the proliferation of programming languages and availability of open source software. We expand the work by DeMillo et al. — which investigated 296 bug-fixes from a single program (TeX) — to faults from 5,000 programs in four different programming languages (C, Java, Python, and Haskell) — a total of 240,000 bug-fixes.

We also extended our investigation to localized patches that contain just a single modification in them, which should contain a simple fault, and hence should be similar to those produced by mutation operators. The incidence of bug fixes and localized changes in the overall population is summarized in Table 2.1. The details of our data collection are summarized in Section 3.4.

Our analysis, summarized in Section 3.5, suggests there is a huge variation in the incidence of different classes of mutations, which are dependent on the kind of programming language chosen. Further, there are a significant number of change patterns which are different from the single token change captured by standard mutation operators. Hence, using all mutations equally would not be representative of the real faults in software, and most real faults do not match any mutation operator. Further, the choice of mutation operators also needs to be guided by the programming language used. We provide a basis for future investigations in this regard.

The data for this study is available at Dataverse [43].

2.2 Related Work

Our work is an extension of the work done by DeMillo et al. [86], Daran et al. [27] Andrews et al. [8] and Namin et al. [75] which attempts to relate the characteristics of mutation operators to that of real faults. In the remainder of this paper, we use the term mutation operator to indicate, in context, either actual mutation operators applied

that their use of mutation analysis-generated faults instead of a fault seeding approach based on fault distributions is essentially based on the Competent Programmer assumption.

during mutation analysis, or the actual small changes made to code in bug fixes.

DeMillo et al. [86] were the first researchers to investigate the representativeness of mutations to real faults empirically. The investigated the 296 errors in TeX, and found that 21% were simple faults (single token changes), while the rest were complex errors.

Daran et al. [27] investigated the representativeness of mutation operators to real faults empirically. They studied the 12 real faults found in the program developed by a student, and 24 first order mutants. They found that 85% of the mutants were similar to the real faults. While this paper highlights the importance of relating the actual mutations to real faults, they were constrained by their small sample size, a single program. More importantly, the conclusions were based on only 12 real faults.

Another important study by Andrews et al. [8] investigated the ease of detecting a fault for both real faults and hand seeded faults, and compared it to the ease of detecting faults induced by mutation operators. The ease is calculated as the percentage of test cases that killed each mutant. Their conclusion was that the ease of detection of mutants was similar to that of real faults. However, they based this conclusion on the result from a single program, which makes it unconvincing. Further, their entire test set was eight C programs, which makes the statistical inference drawn liable to type I errors. We also observe that the programs and seeded faults were originally from Hutchins et al. [51] where the programs were chosen such that they were subject to certain specifications of understandability, and the seeded faults were selected such that they were neither too easy nor too difficult to detect. In fact they eliminated 168 faults for being either too easy or too hard to detect, ending up with just 130 faults. This is clearly not an unbiased selection. More seriously, this selection can not tell us anything about the ease of detection of hand seeded faults (because the criteria of selection itself is confounding).

These acute problems were highlighted in the work of Namin et al. [75] who used the same set of C programs, but combined them with analysis of four more Java classes from JDK. They used a different set of mutation operators on the Java programs, and used fault seeding using student programmers on them. Their analysis concluded that we have to be careful when using mutation analysis as a stand-in for real faults. They found that programming language, the kind of mutation operators used, and even test suite size has an impact on the relation between mutations introduced by mutation analysis and real faults. In fact, using a different mutation operator set, they found that there is only a weak correlation between real faults and mutations. However, their study was constrained by the paucity of real faults available for just a single C program (same as Andrews et al. [8]). Thus they were unable to judge the ease of detection of real faults in these Java programs. Moreover, the students who seeded the faults had knowledge of mutation analysis which may have biased the seeded faults (thus resulting in high correlation between seeded faults and mutants). Finally, the manually seeded faults in C programs, originally from Hutchins et al. [51], were confounded by their selection criteria which eliminated the majority of faults as being either too easy or too hard to detect.

These previous efforts prompted us to look at evaluating mutation analysis from a different direction. We wondered if the ease of detection was the only relevant criteria when comparing mutation operators and real faults. Why not compare them directly, by comparing the syntactical patterns of both? Even if it is argued that there may be interdependent changes that make it difficult to compare, we can still get a reasonable result by restricting our analysis to small localized changes that are limited to a single change in a single file.

There has been other research in related fields that takes a similar approach. Christmansson et al. [20, 21] analyzed field data to come up with an error model that mimics real faults, and used these to inject errors to simulate faults. Their study classified the defects based on their *semantics* using Orthogonal Defect Classification [19]. While this research is useful in its domain, it is inapplicable to mutation analysis, which is primarily a syntactical technique. We want to easily generate bugs that look like and feel like real bugs with relatively little context. We certainly don't want to understand the semantic content, e.g. whether a mutation introduces a functional error, an algorithmic error, or a serialization error (classifications of ODC).

Duraes et al. [37, 38] analyzed the change patterns in 9 open source C projects, and collected a total of 668 faults. They adapted Orthogonal Defect Classification to provide a finer classification of errors into missing, wrong, and extraneous language constructs. They find 64% of the faults were due to missing constructs, 33% due to changes, and only 2.7% were due to extraneous constructs. While this study is the closest to our approach, they are also limited by concentration on a single language (C), and a comparatively small number of faults (532 faults) to ours. Further, while the classification they provide is finer grained than ODC, it is still at a higher level than the typical mutation operator implementations. In comparison, our analysis of a larger set of data indicates that addition and deletion were relatively similar in prevalence, while changes dominated in all the languages we analyzed.

A larger study of similar nature by Pan et al. [82] extracted 27 bug fix patterns from the revision history of 7 projects, which cover up to 63.3% of the total changes, and computed the most frequent patterns. Their study, like the previous one by Duraes analyzed the patterns from a higher level than typical mutation operators, and hence is not directly applicable to mutation analysis. Further, their analysis is restricted to Java programs which, along with the limited number of projects reduces their applicability.

Our research uses machine learning techniques to automatically classify patches as bug-fix or non bug-fix based on an initial set of changes that we manually classified. Mokus et al. [70] first used a classifying approach that relied on the presence of keywords. They classified changes into categories of fixes, refactoring and features.

Another related study is by Purushotam et al. [84] who analyzed the change history of a large software project, specifically focusing on small (one line) changes. They were interested in finding the patterns of changes that can induce an error with high probability in software. The study is interesting for the distribution they found for small changes, which we also consider. They found that 10% of the total changes involved a single line of code, and 50% were below 10 lines, dropping to 5% for those above 50 lines. They also suggest that most changes involved inserting new lines of code. Our study found that small (localized) changes can range from 26.2% in C to 62.7% in Haskell (see Table 2.1).[2]

	C	Java	Python	Haskell
Localized changes	26.241	27.278	43.770	62.685
Bug-fixes	44.314	29.612	34.395	31.009
Localized bug-fixes	10.464	9.053	16.541	16.486

Table 2.1: Localized changes and bug-fixes prevalance in %

2.3 Methodology

We were primarily interested in finding answers to the following questions.

Q Can we find empirical evidence for or against the *competent programmer hypoth-*

[2]The percentages given in Table 2.1 are overlapping. The set of changes is divided into bug-fixes and feature updates, and orthogonal to that, as localized (single line), and non-localized (mult-line) changes

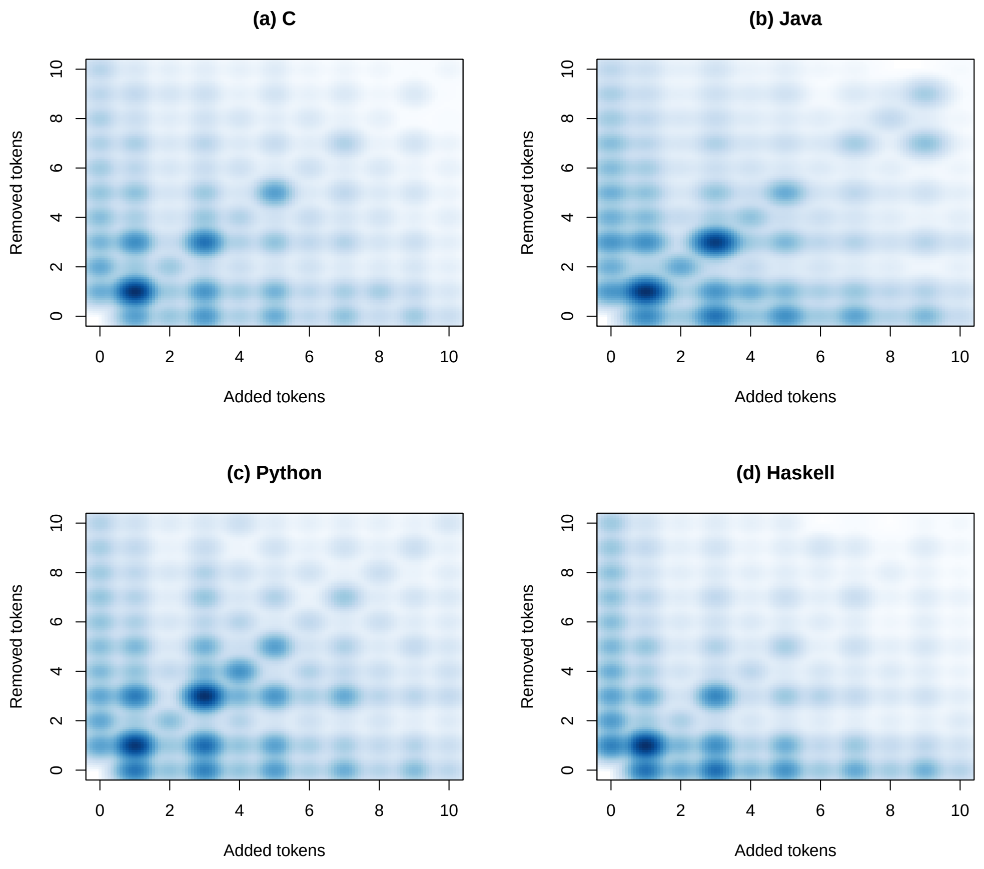


Figure 2.2: Density plot of added vs removed number of tokens in replacement changes for full distribution

esis? Can we find any support for the assumption that real faults look like those produced by typical mutation operators? Can we do this by analysis of patches (whether it be the complete set of changes or a subset that is identified as bug-fixes or localized small bug-fixes that should be fixes for simple faults)?

Q How much of an effect does programming language have on the distribution of change patterns? Can we extend the results from the distribution of syntactical changes or fault patterns in one language to another? We especially want to make sure that we compare apples to apples here and look at a common set of mutation

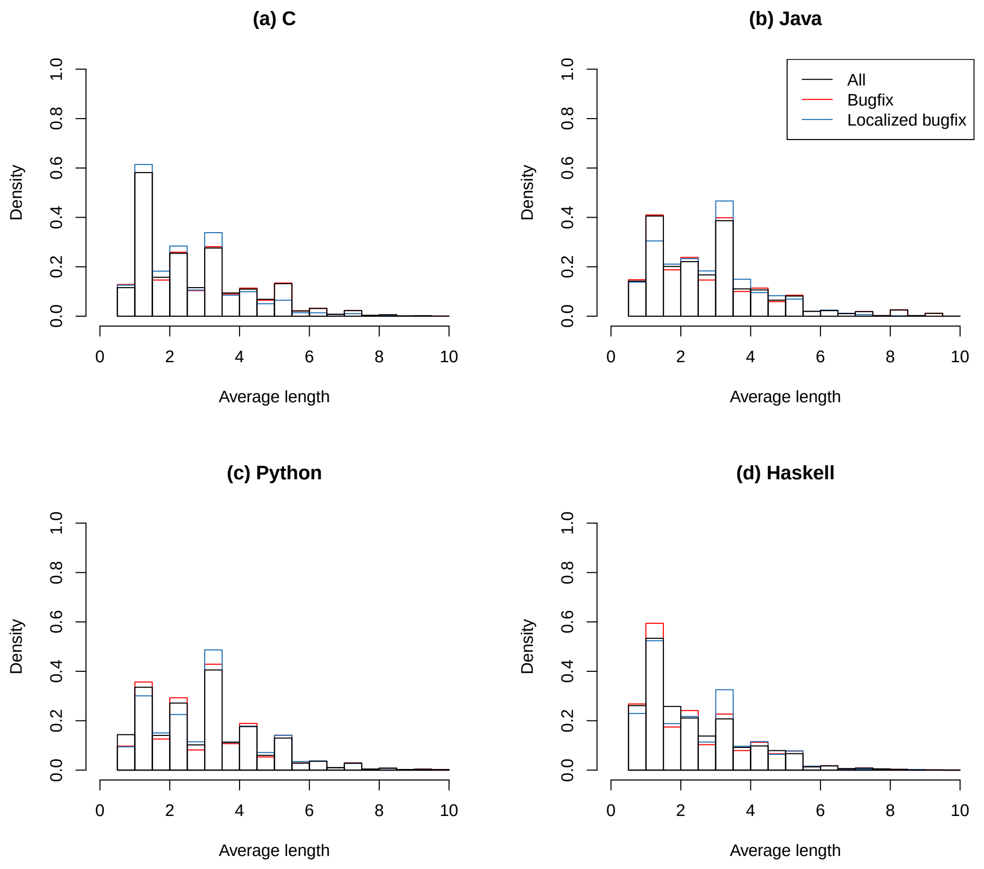


Figure 2.3: Average length of added vs removed tokens

operators across different languages.

Q What are the most common mutation operators? Are they different from the traditional mutation operators that are commonly used? Can we provide any guidance to future implementors of mutation tools so that mutation operators produced look similar to real faults?

We wanted our results to be applicable to a wide variety of languages and ensure that our analysis did not suffer from bias for a particular language group. We chose four languages, each representative of an important kind of development. We chose C as the dominant systems programming language, widely used in the most critical systems

```

class MyClass {
  int loop(int counter) {
    int i = 0;
    while(i < counter) {
*      <count = count +1 | i++ >;
    }
*    return <count | i>;
  }
}
```

Fig 1: An example patch

for testing. Java was chosen as a popular programming language used in enterprise applications. The choice of Python was driven by its status as one of most popular languages in the dynamically typed community, and its use in many domains including statistics, mathematics, and web development. Finally Haskell, while less popular than the other three, is a popular strongly typed functional language preferred in academic research.

To ensure that we had a relatively unbiased population from each language, we searched for projects in Github [41] with criteria `stars :>= 0` and filtered by the language side bar. We used this criteria since this is a nil-filter—the stars start from '0'—and hence no project was excluded. This search resulted in 1850 projects for C, 1128 for Java, 1000 for Python, and 1393 for Haskell.

2.3.1 Classifying patches

Each project from Github came with its entire revision history, which is accessible as a set of patches. To answer our research questions we had to differentiate between bug-fixes—where some pre-existing fault was fixed—and patches that were not bug-fixes. Since we lacked resources to manually classify our entire dataset, we made use of machine learning techniques. We manually classified 1200 patches as bugs or non-bugs for each of the languages. Out of these 4800 classified patches, we used 4000 to train our classifiers, and used 800 (200 from each) to cross validate our trained classifiers. We achieved an accuracy of 78.87% using CRM114 classifier [26] which gave us the highest accuracy out of Bayesian, Bishop, LSI, and SVM classifiers. The acceptance accuracy for bugs was 73.19%, while the rejection accuracy was 81.24%. We got overall better results by combining training examples from all the languages than by training on each in isolation. For example, using individual training, accuracy obtained for Java was 76.5% (acceptance: 64.4%, rejection 81.5%), 77% (acceptance: 76.8%, rejection: 77.1%) for Python, 71% (acceptance: 69.7%, rejection: 71.8%) for C, and 76% (acceptance: 70.9%, rejection: 76.9%) for Haskell. This rate is close to the rate obtained by leading research [11] in classification of bugs and non-bugs, which obtained an accuracy between 77% to 82% using change tracking logs.

After classification, we found that 44.31% of commits in C were bug fixes, 29.6% for Java, 34.39% for Python and 31.01% for Haskell. The distribution is given in Table 2.1.

2.3.2 Generating normalized patches

Next, we wanted to collect the patches in each project, after discounting for the differences due to whitespace and formating changes. To accomplish this, for each project, the following procedure was applied to collect normalized patches for each projects.

First, the individual revisions of files were extracted, and they were cleaned up by stripping comments, joining multi-line statements, and hashing string literals. These were then re-formated by passing through a pretty-printer. This removed the differences due to addition or removal of comments or due to formating changes. Next, successive revisions were diffed against each other using a token-based diffing algorithm, and the patches thus produced were collected.

2.3.3 Sampling

We were interested in finding the distribution of token changes, unbiased by effects of project size, developer or project maturity, or other unforeseen factors. For statistical inference to be valid to a given population, the observations from which the inference is drawn should be randomly sampled from the targeted population. For this purpose, we decided to generate random samples of patches from the projects we had.

We generated 10 random samples, with each containing 1,000 patches for combinations of the following sets—whether they are bugs or not (bug, nonbug, all), whether the bug fix was localized or otherwise (small, all), and each of the languages (C, Java, Python, Haskell). This generated $3 \times 2 \times 4 \times 10 = 240$ samples (240,000 patches, but some may be repeated in multiple samples).

2.3.4 Collecting chunks

Each patch is composed of multiple segments in the file where some text was removed, or added, or some text was replaced (remove + add). An example patch is given in Figure 1. This patch contains two chunks. The first chunk is in line 6 and involves removal of 5 tokens, and addition of two tokens. The second one is in line 8, and involves removal of a single token and addition of another. These chunks were extracted and processed further by eliminating syntactic sugar elements such as parenthesis[3], commas, etc. and collapsing strings to their checksums for easier processing. The tokens thus identified were then passed through a lexical identifier which replaced each lexical element by its class. We use chunk and change interchangeably in this paper.

2.3.5 Identifying mutation operators

For ease of comparison between different languages, we chose to use a single set of mutation operators applicable across different programming languages. We started with the original 77 operators proposed for the C programming language by Agrawal et al. [2]. We then removed operators that could not be matched from the context of changes. The mutation operator variants that were mirror images were collected under a single name. Further, a few mutation operators were discarded because they were inapplicable in other languages. The mutation operators were further grouped into classes for analysis. Further, the added, removed and changed patterns that could not be classified under any existing mutation operators were grouped in their own categories, resulting in ten mutation operator categories. A complete listing is provided in Table 2.6. The distribution of average token count is provided in Table 2.13 where max ϵ is the largest percentage detected in the remaining token bins.

[3]We may therefore miss some changes that involved semantically meaningful parenthesis additions, but this is also not a standard mutation operator.

	C	Java	Py	Hs
Add:oth	16.483	17.757	17.082	30.114
Change:oth	32.219	25.115	29.443	32.363
Rem:oth	13.372	14.526	12.215	23.263
Twiddle	0.219	0.057	0.047	0.070
Const	5.425	2.515	6.205	2.270
Var.Const	4.981	2.045	3.372	1.045
Var	26.641	37.744	31.487	10.721
BinaryOp	0.119	0.031	0.033	0.026
Negation	0.428	0.186	0.098	0.102

Table 2.2: Summary of mutation operators for all changes

	C	Java	Py	Hs
Add:oth	28.781	29.805	22.699	31.607
Change:oth	23.352	21.078	26.439	29.294
Rem:oth	12.877	13.139	11.614	19.742
Twiddle	0.614	0.286	0.094	0.160
Const	12.126	13.086	21.442	8.545
Var.Const	5.533	4.095	4.384	2.240
Var	14.979	17.279	12.881	7.979
BinaryOp	0.453	0.307	0.158	0.054
Negation	0.932	0.728	0.189	0.332

Table 2.3: Summary of mutation operators for localized changes

2.4 Analysis

According to a naive interpretation of the *competent programmer hypothesis*, a majority of changes we see should be simple and localized and look like traditional mutants. The traditional mutation operators all operate on changing a single token. In order to investigate whether this is the case, we plotted the number of tokens added versus the number of tokens deleted in each change. The result of this analysis is shown in Figure 2.2 for each language. This figure shows that while there are a significant number of changes that are one token (ϵ changes), there is a large number of changes that include more than one token in both added and deleted counts. We note that these are not captured by the traditional mutants.

A second concern we had was about the difference between the distributions of bug-

	C	Java	Py	Hs
Add:oth	15.705	19.519	18.714	29.788
Change:oth	32.823	27.470	29.971	33.395
Rem:oth	13.284	15.418	13.529	23.352
Twiddle	0.126	0.049	0.010	0.020
Const	4.242	2.741	7.614	2.101
Var.Const	3.648	2.240	4.808	1.511
Var	29.776	32.314	25.094	9.726
BinaryOp	0.058	0.013	0.019	0.011
Negation	0.296	0.222	0.229	0.086

Table 2.4: Summary of mutation operators for bug-fixes

	C	Java	Py	Hs
Add:oth	29.861	33.103	26.629	33.162
Change:oth	21.686	19.032	28.097	28.149
Rem:oth	12.168	13.082	11.308	18.696
Twiddle	0.852	0.392	0.161	0.259
Const	11.899	10.779	14.226	8.060
Var.Const	5.359	3.634	4.461	2.156
Var	15.856	18.366	14.678	8.773
BinaryOp	0.646	0.326	0.163	0.130
Negation	1.198	1.093	0.165	0.490

Table 2.5: Summary of mutation operators for localized bug-fixes

fixes and other changes, and the impact of different languages. We plotted the histograms of average change lengths (computed as the average of added and removed tokens per change) for each of the languages. This is shown in Figure 2.3. The plot indicates that bug-fixes do not significantly differ from the main change patterns. However, the figure indicates a difference in distribution between different languages.

To confirm our finding, we use statistical methods. Students two-sample t-test is a statistical test that checks whether two sets of data differ significantly. We use it to determine whether essential characteristics of changes differ between bug-fixes and other commits, and between different languages. We also provide a comparison with difference in mean between the two distributions obtained by running Students t-test. These were significant for $p < 0.05$ except where indicated. The difference between the bug-fix changes and others are tabulated in the Table 2.7. We note that the difference between

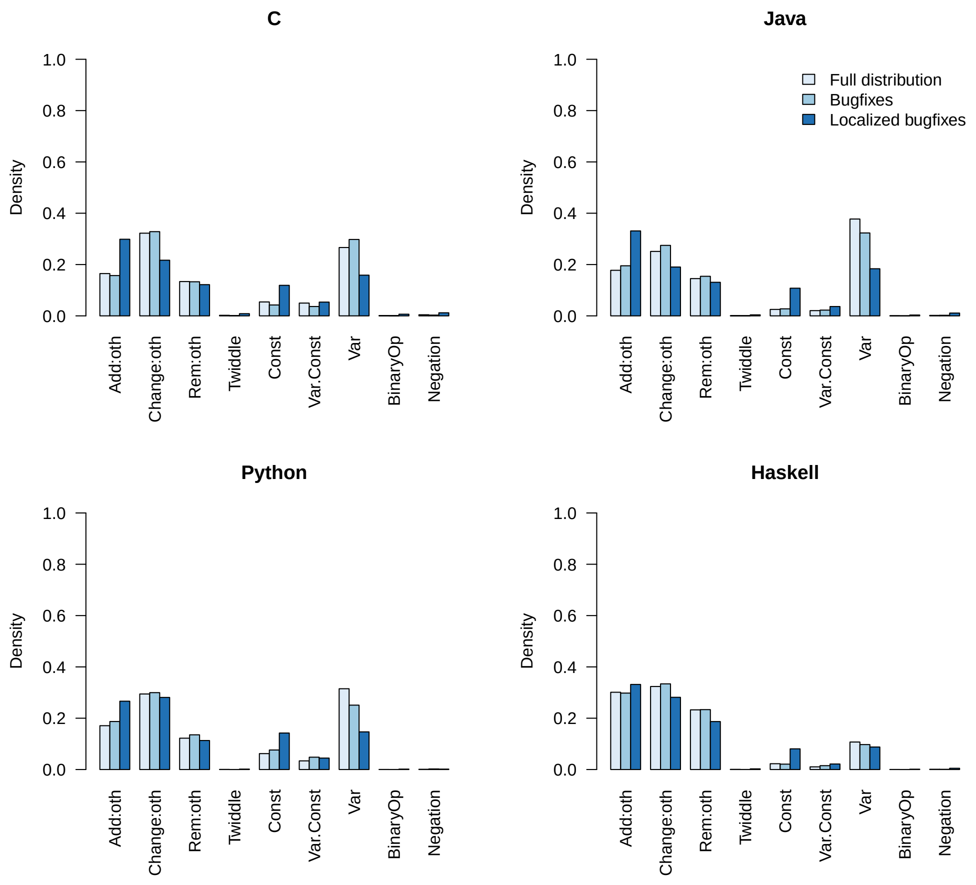


Figure 2.4: Relative occurrence of mutation operators

bug-fix and others changes is universally very low for all four programming languages, confirming our initial finding from Figure 2.3.

Next we compare the distributions of tokens between different languages. The mean difference from Students t-test is given in Table 2.8. These were not significant for the pair C and Java, but was significant with $p < 0.05$ for all other language pairs.

We observe here that while the difference between languages seems small, there is a large similarity between C and Java patterns, and Haskell is closer to Python than others. This seems somewhat intuitive if we consider that C and Java are descendants of the Algol family, while Python to a large part supports functional programming paradigms,

Class	Explanations
Add:oth	Added tokens not including twiddle, negation, unary and statement mutation operators
Change:oth	Replaced tokens not classified under any of other changes
Rem:oth	Removed tokens not including twiddle, negation, unary and statement mutation operators
Twiddle	Addition or removal of +/- 1 or the use of unary increment or decrement operators
Const	Change in constant value
Var.Const	Changing a constant to a variable or reverse
Var	Changing a variable to another variable
BinaryOp	Changing a binary operator to another
Negation	Negation of a value (includes arithmetic, bitwise, and logical)

Table 2.6: Explanations of mutation operator classes

	Bug-fix	Nonbug	*SBug	LowCI	HighCI	MeanD	Pval
C	4.19	4.17	3.08	-0.06	0.09	0.02	0.65
Java	4.22	4.18	3.18	-0.07	0.14	0.03	0.53
Python	4.39	4.22	3.91	0.07	0.27	0.17	0.00
Haskell	4.48	4.46	3.93	-0.08	0.13	0.02	0.69

Table 2.7: Average tokens changed between bug-fixes and other changes

of which Haskell is an exemplar.

2.4.1 Mutation operator distribution

A major part of our analysis is the comparison of mutation operator distributions across different languages and kinds of patches. We analyze the difference between the complete distribution, that of just bug-fixes alone, and localized bug-fixes. This is visualized in Figure 2.4. The summary of mutation operators are also provided as in Table 2.2, and a summary of mutation operators for localized changes are given in Table 2.3. Finally, Table 2.4 tabulates the distribution of mutation operators for bug-fixes, and Table 2.5 the distribution of localized bug fixes. The mutation operator class explanations are given in Table 2.6.

2.4.2 Regression Analysis

Regression analysis is a statistical process that helps us to understand the relative contributions of different variables. Here, we make use of regression analysis to assess the contribution of class of mutation operator, programming language, and the kind of

	C			Java			Python			Haskell		
	MD	LCI	HCI	MD	LCI	HCI	MD	LCI	HCI	MD	LCI	HCI
C	0	0	0	-0.02	-0.09	0.04	-0.1	-0.2	-0.06	-0.3	-0.4	-0.2
J	-0.02	-0.09	0.04	0	0	0	-0.1	-0.2	-0.03	-0.3	-0.3	-0.2
P	-0.1	-0.2	-0.06	-0.1	-0.2	-0.03	0	0	0	-0.2	-0.2	-0.09
H	-0.3	-0.4	-0.2	-0.3	-0.3	-0.2	-0.2	-0.2	-0.09	0	0	0

Table 2.8: Mean difference for average tokens changed between different languages ($p < 0.05$ except C x Java)

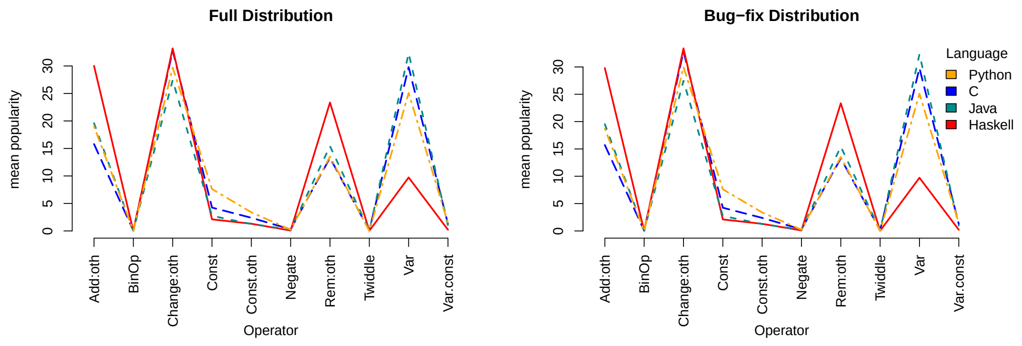


Figure 2.5: $Op \times Language$ interaction

change (bug-fix or otherwise) to the prevalence of the mutation operator.

First we run our analysis for the complete distribution, analyzing which model fits best. Next, we run our analysis on only the patches classified as bug-fixes, and finally on those localized bug-fixes. We use the keys given in Table 2.9 to refer to the variables in the model.

Variable	Name
P	Prevalence of mutation operator
O	Operator (Mutagen operator)
L	Language
B	Bug-fix or otherwise

Table 2.9: Explanations of model variables

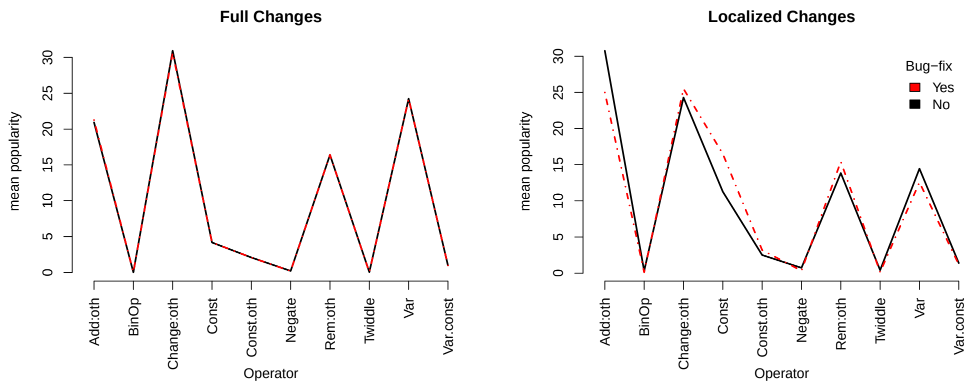


Figure 2.6: $Op \times Bug$ interaction

2.4.2.1 Complete Distribution

We started with the full model containing the full interactions between all given variables.

$$\mu\{P|O, L, B\} = O + L + B + O \times L + O \times B + L \times B + O \times L \times B$$

However, not all the variables were significant contributors towards the prevelance of the mutation operator. We sequentially eliminated non-significant variables resulting in

$$\mu\{P|O, L, B\} = O + L + O \times L$$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Op	9	101703.67	11300.41	1724.49	0.0000
Language	3	0.00	0.00	0.00	1.0000
Op:Language	27	10669.20	395.16	60.30	0.0000
Residuals	760	4980.20	6.55		

Table 2.10: Results of the model fit for complete distribution

This provides us the best fit given in Table 2.10, and has correlation coefficient R^2 =0.955. This suggests that a patch has similar change patterns irrespective of whether it is a bug-fix or otherwise. This is also suggested by the interaction plot between mutation operator bug-fixes given in Figure 2.6. Further, we also see the evidence of non additive interaction between mutation operators and language in Figure 2.5 and in the ANOVA results in Table 2.10.

2.4.2.2 Localized Change Distribution

Next, we analyze the localized changes. These are changes that modify only a single file in a single part such that the change is restricted to a single chunk. We investigate localized changes because they are closest to the changes produced by mutation operators.

$$\mu\{P|O,L,B\} = O + L + B + O \times L + O \times B + L \times B + O \times L \times B$$

Interestingly, for localized distribution, the interaction between mutation operators, language, and bug-fix is significant, which makes the full model the one with the best fit. The model has R^2 =0.955, and the model ANOVA is given in Table 2.11.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Op	9	79759.07	8862.12	4088.18	0.0000
Language	3	0.00	0.00	0.00	1.0000
Bug	1	0.00	0.00	0.00	1.0000
Op:Language	27	5319.62	197.02	90.89	0.0000
Op:Bug	9	1378.17	153.13	70.64	0.0000
Language:Bug	3	0.00	0.00	0.00	1.0000
Op:Language:Bug	27	860.12	31.86	14.70	0.0000
Residuals	720	1560.77	2.17		

Table 2.11: Results of the model fit for localized distribution

2.4.2.3 Localized Bug-fix Distribution

The previous result induced us to also look at the distribution of localized bug-fixes. These are localized changes that were also identified as bug-fixes. This results in a very close fit model with a coefficient of correlation R^2 =0.991. The result of ANOVA is given in Table 2.12.

$$\mu\{P|O,L\} = O + L + O \times L$$

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Op	9	42910.04	4767.78	4880.83	0.0000
Language	3	0.00	0.00	0.00	1.0000
Op:Language	27	2002.98	74.18	75.94	0.0000
Residuals	360	351.66	0.98		

Table 2.12: Results of the model fit for complete distribution

2.5 Results

Our first question was whether we could quantify the *competent programmer hypothesis*, and verify whether real faults look like mutation operators. Our analysis shows that a significant number of changes are larger than the common mutation operators. A typical change modifies about three to four tokens in all the programming languages surveyed. This increases to addition or removal of about six to eight tokens if we consider addition or removal changes rather than replacement. This increases to five tokens (ten tokens for addition or removal) if we wish to include at least 80% of the real faults, and remains relatively the same even when we consider localized bug-fixes which we had expected to have a distribution similar to that produced by mutation analysis, provided the *competent programmer hypothesis* is applicable to the mutants produced. This suggests that our understanding of the *competent programmer hypothesis*, at least as suggested by typical mutation operators, may be incorrect.

This also suggests that in at least one dimension—that of patterns of change—mutations are different from real faults.

Our next effort was to identify whether programming language had any effect on the distribution of mutants, first without considering the different mutation operators, and later, including the differences between mutation operators. Our initial analysis in Table 2.7 and Table 2.8 indicated that while there are interesting affinities between different languages with regard to the syntactical distance, the effect itself was weak when different mutation operators were not considered. However, once we consider the different classes of mutation operators, as shown in the interaction plot in Figure 2.5, there is a significant difference in mutation distribution between different programming languages. Finally we conclusively showed using regression analysis that language is an important contributor to the mutation operator distribution in Table 2.10. The result

	C all	J all	P all	H all	C bug	J bug	P bug	H bug
0.5	4.9	6.1	6.2	10.5	5.5	6.3	4.2	10.8
1	29.7	23.6	20.5	31.9	30.2	23.9	19.4	34.7
1.5	36.4	32.2	26.5	42.3	36.4	32.0	24.8	41.7
2	47.2	41.7	38.2	50.7	47.5	42.2	37.3	51.4
2.5	52.1	48.9	42.5	56.3	51.9	48.5	40.8	55.5
3	63.9	65.5	59.9	64.6	63.9	65.7	59.2	64.7
3.5	67.9	70.3	64.8	68.3	67.7	69.9	63.7	67.8
4	72.6	74.9	72.3	72.2	72.5	74.8	71.8	72.4
4.5	75.5	77.6	74.9	75.4	75.3	77.4	74.1	75.0
5	81.1	81.2	80.4	78.1	81.0	81.0	80.2	78.1
5.5	82.8	83.1	82.6	80.2	82.7	82.9	82.1	79.9
6	85.1	84.6	84.8	82.0	85.0	84.5	84.6	82.1
6.5	86.2	86.0	86.2	83.7	86.1	85.8	85.8	83.5
7	88.1	87.5	88.3	85.1	88.0	87.5	88.1	85.0
7.5	89.0	88.5	89.3	86.6	88.9	88.4	89.0	86.3
8	90.0	90.2	90.4	87.5	89.9	90.1	90.3	87.5
8.5	90.6	90.9	91.3	88.4	90.5	90.8	91.0	88.2
9	91.4	91.9	92.2	89.2	91.4	91.8	92.1	89.1
9.5	92.0	92.3	92.9	90.0	91.9	92.2	92.6	89.8
10	92.7	92.9	93.5	90.6	92.6	92.8	93.3	90.6
$\max_\epsilon$	0.5	0.6	0.6	0.7	0.5	0.6	0.7	0.6

Table 2.13: Cumulative density(%) of average token changes

holds true even for localized bug-fixes as shown in Table 2.12.

This quite strongly suggests that while the average change involves touching about four tokens in all languages examined, different languages encourage different mutation patterns. This suggests that we have to be careful while adapting the results from a different language.

As our final step, we investigated the most common mutation operators. Our results shown in Figure 2.4, and tabulated in Table 2.2, Table 2.3, and Table 2.5 show that different languages have different mutation patterns. Addition, deletion, and the replacement of tokens, especially those that did not come under traditional mutation operators, dominated the mutation operator distribution. This suggests a need for more effective ways to simulate real faults.

An interesting result is also that the distribution we identified between changes of addition and removal (which are somewhat similar in magnitude in each language surveyed) is somewhat at odds with previous research [37] which finds addition of statements to be the highest category (64%), while deletion was small at 2.7%.

Another interesting finding is also the difference between Haskell and other languages in the prevalence of localized changes. We found that for Haskell, more than 60% of the changes were localized changes. Further, we also found that Haskell showed a higher affinity with Python than other languages with regard to change length distribution (Table 2.8).

2.6 Discussion

Mutation analysis is a very useful technique that is commonly used by researchers as a stand-in for test suite quality. Its theoretical foundations rely on two important concepts: that of the *competent programmer hypothesis*, and the *coupling effect*. While the *coupling effect* has been investigated to some extent both theoretically [93,95] and empirically [77], relatively little research has investigated the *competent programmer hypothesis*.

In this paper, we investigated the *competent programmer hypothesis*. According to Budd et al. [17] and DeMillo [28], a competent programmer constructs programs that are at most one simple fault [77] away from correctness, and the program, together with the mutants generated—the finite neighborhood $\Phi(P)$—would include the correct program. The implicit claim is that real world programmers are in fact competent, at least most of the time and with regard to a particular program unit and fault. Mutation analysis looks for tests that are adequate relative to Φ.

For the ease of discourse, we define different versions of the *competent programmer hypothesis*, differentiated by their syntactical finite neighborhood $\bar{\Phi}_\delta(P)$, that is, $\bar{\Phi}_1(P)$ are all the mutants that are at most one token away.

The current generation of mutation operators are overwhelmingly members of $\bar{\Phi}_1$ (excepting a few OO operators for Java [66] and the statement deletion operator [78]). However, our finding is that real faults appear to have a mean token distance of three to four, for all languages examined.

This also brings us to the question of effectiveness of the coupling effect on these larger changes. Coupling has been demonstrated to work only using the entire domain of higher order faults. We note that the actual empirical data indicates that real faults occur in such a way as to ensure that the real higher order faults are drawn not from the entire domain, but a much restricted domain of (what we suspect is) a semantic neighborhood of the correct program. It could be that detecting mutants from the $\bar{\Phi}_1$ family does detect 90% or more of mutants from the full $\bar{\Phi}_{\delta>1}$ family, but that real faults fall heavily into the 10% of mutants hard to detect, for example, since the distributions do not resemble the syntactic space of higher order operators. Hence, we suggest further research needs to be done to empirically show that the Coupling Effect holds on real faults, especially on those belonging to $\bar{\Phi}_{\delta>1}$.

We also note that the effectiveness of mutation analysis need not be tied to its theoretical basis. That is, if suites that effectively kill mutants based on $\bar{\Phi}_1$ also have a very high likelyhood, in a purely empirical sense, of also detecting faults very well, that the mutants do not resemble the faults does not matter. However, this itself is in fact the real Coupling Effect that needs to be demonstrated, and as we noted in Section 4.2 the current evidence is not strong enough to place mutation analysis on a sound footing.

2.7 Threats to Validity

While we have taken utmost care to avoid errors, our results are subject to various threats. First, our samples have been from a single source—open source projects in Github. This may be a source of bias, and our inferences may be limited to open source programs. However we have not seen any evidence of open source programs differing from closed source programs in terms of fault patterns.

Github selection mechanisms favoring projects based on some confounding criteria may be another source of error. However, we believe that the large number of projects sampled more than adequately addresses this concern.

Another source of error is in the bug classification of patches. However, we have followed current research recommendations, and obtained a result in classification that is close to that obtained from current best research in the field.

2.8 Conclusion

One of the main assumptions in mutation analysis is the *competent programmer hypothesis*, which claims that real programs are very close to correct. If this assumption holds true, then mutation analysis will produce faults that are similar to real faults. However, except for an initial small scale research by DeMillo et al., there has been a lack of research quantifying the syntactic changes involved in real faults, especially with an adequate number of subjects.

Our research attempts to quantify the syntactic differences found in real faults, and finds that faults produced by typical mutation operators are not representative of real faults. Therefore the competent programmer hypothesis, at least from a syntactical perspective, may not be applicable. This suggests that mutation analysis requires further research to place the use of mutants to evaluate suites on a firm empirical footing. Moreover, the differences between results for different programming languages suggest that mutation operators may need to vary even more than has been suspected in order to work in new languages.

The theory of composite faults

Rahul Gopinath, Carlos Jensen, Alex Groce

2017 IEEE International Conference on Software Testing, Verification and Validation
Tokyo, Japan
pp. 47-57

Chapter 3: The theory of composite faults

3.1 Introduction

Fault masking occurs when interactions between component faults in a complex fault result in expected (non-faulty) values being produced for particular test inputs. This can result in faults being missed by test cases, and undeserved overconfidence in the reliability of a software system.

The *coupling effect* [31] hypothesis concerns the semiotics[1] of fault masking. It asserts that *"complex faults are coupled to simple faults in such a way that a test data set that detects* ***all simple faults in a program*** *will detect a high percentage of the complex faults."* [55, 76, 77].

This is relied upon by software testers to assert that fault masking is indeed rare. However, our understanding of the *coupling effect* is woefully inadequate. We do not know when (and how often) fault coupling can happen, whether multiple faults will always result in fault coupling, or the effect of increase in number of faults on the number of faults masked. Further, the formal statement of the coupling effect itself is ambiguous and inadequate as it covers only the case where all simple faults are detected. Even worse, it has no unambiguous definition of what a simple (or atomic) fault is. We propose a stronger version of the *coupling effect* (called the *composite fault hypothesis* to avoid confusion):

Composite fault hypothesis: *Tests detecting a fault in isolation will (with high probability κ) continue to detect the fault even when it occurs in combination with other faults.*

We investigate our hypothesis theoretically and empirically. The terms used in this paper are given in Note 1.

[1]The relation between syntax and semantics of faults.

3.1.1 Theory

Wah et al. [92, 93, 96] investigated the theory of the *coupling effect*, which assumes that any software is built by composition of q independent functions, with a few restrictions:

- Functions have the same *domain* and *range* (order n), and the functions are *bijective*. The non-*bijective* functions are modeled as *degenerate* functions.
- *Separability of faults*: A program with two faults can be split into two independent faulty programs [2].
- *Democratic assumption*: Any applicable function may be chosen as the faulty representation with equal probability.
- The number of functions considered, q, is much smaller than the size of the domain. That is, $q \ll n$. Wah suggests that as q nears n, the *coupling effect* weakens.

For q functions, the survival ratio of I and II order test sets are $\frac{1}{n}$ and $\frac{1}{n^2}$. Wah also makes an observation, used as a heuristic, that the survival ratio of a multi-fault alternate is $\frac{p+1}{n}$ if there are p fault free functions left over after the last faulty function. That is, there are $2^{p-1} - 1$ multi fault alternates with last faulty function at p, and the expected number of survivors for q-function composition is:

$$\frac{1}{n^r}\sum_{p=1}^{q}(2^{p-1}-1)(q-p+1)^r$$

for test sets of order r. Wah's analysis lacks wider applicability due to these constraints. Real programs vary widely in their *domain* and *co-domain*. Second, the number of mathematical functions with same *domain* and *co-domain* is not identical to that of programs with same type. Third, the democratic assumption ignores the impact of syntactical neighborhood. That is, it is possible that a *quick sort* implementation can have a small bug, resulting in an incorrect sort. However, it is quite improbable that it is replaced by an algorithm for — say — *random shuffle*, which has the same *domain* and *co-domain* as that of a sorting function. While syntactical nearness does not completely capture semantic nearness, it is closer than assuming any function is a plausible fault for

[2] Wah assumed this to be true for all general functions, but Section 3.3 shows that it is not.

(Semantic) Separability of faults: Two faults present in a function are said to be separable *if and only if* the smallest possible chunk containing both faults can be decomposed into two functions g and h such that each fault is isolated within a single function (providing g_a and h_b as faulty functions), the behavior of composition $h \circ g$ equals the behavior of the original chunk in terms of input and output , and composition $h_b \circ g_a$ equals the behavior of the chunk with both faults.
Simple fault (first order fault): A fault that cannot be *lexically* separated into other independent smaller faults.
Complex fault: (or *higher order* or *combined* fault) A fault that can be *lexically* separated into smaller independent faults.
Constituent fault: A fault that is *lexically* contained in another.
Atomic fault: A fault that cannot be *semantically* separated.
Composite fault: A fault that can be *semantically* separated.
Traditional coupling ratio (C): The ratio between the percentage of complex faults detected and the percentage of simple faults that were detected by a test suite.
Composite coupling ratio (κ): The ratio between the percentage of complex faults detected by the same set of test cases that detected the constituent simple faults, and the percentage of constituent simple faults detected.
Domain *of a function*: The set of all values a function can take as inputs (this is practically the input type of a function).
Co-Domain *of a function*: The set of all values that a function can produce when it is provided with a valid input from its *domain* (this is practically the output type of a function).
Range of a function: The set of all values in *co-domain* that directly maps to a value in the *domain*.
Syntactic neighborhood: The set of functions that can be reached from a given function by modifying its *syntactical representation in a given language* a given number of times.

Note 1: Terms used in this paper

any other function. Next, the separability of complex faults, as we show in Section 3.3, is valid only in certain cases, and does not account for recursion and iteration. Finally, Wah's analysis suggested that the survival ratio of mutants is dependent on the *domain* of the function. We show that the survival ratio of a mutant is actually dependent on the *co-domain* of the function examined, but bounded by *domain*.

We propose a simpler theory of fault coupling that uses a similar model to Wah's, but with relaxed constraints, and incorporates differing *domain* and *co-domain*. We clarify the semantic separability of complex faults, and show how it affects the coupling effect. We also show that certain common classes of complex faults may not be semantically separable. This provides us with a definition of an *atomic fault*: a fault that cannot be semantically separated into simpler faults. This is important because two faults that may be lexically separate but inseparable can be expected to produce a different behavior than either fault considered independently. Further, we consider the impact of syntactic neighborhood. Using both case analysis and statistical argument, we show that our analysis remains valid even when the syntactic neighborhood is considered.

3.1.2 Empirical Validation

Lipton et al. [31, 64], and Offutt [76, 77], observed that the tests for first order mutants were sufficient to kill up to 99% of all 2^{nd} order mutants, and 99% of 3^{rd} order mutants sampled. Further research [8, 9, 30, 36, 56, 61] confirms that mutants are coupled to real faults.

Offutt suggests [76, 77] that there are two distinct definitions of coupling involved. The *general coupling effect*: simple faults are coupled to more complex faults such that test data adequate for simple faults will be able to kill a majority of more complex faults. The *mutation coupling effect*: test data adequate for simple first order mutants will be able to detect a majority of more complex mutants. Previous research validates *mutation coupling effect* but not *general coupling effect*.

Our empirical analysis aims to accomplish the following: First, we empirically evaluate the composite coupling ratio κ for numerous real-world projects. This gives us confidence in the assumptions made in the theoretical analysis, and serves to validate the *composite fault hypothesis*. Second, we empirically evaluate the general coupling effect for faults, and compute the traditional coupling ratio C. Lastly, as the size of the faults increase, it is possible that strong interactions also increase, which can produce semantically different faults. Hence, it is important to empirically validate both composite coupling and the general coupling effect for syntactically large fault clusters.

What is the relation between the composite coupling ratio κ and the traditional coupling ratio C? We can regard the composite coupling ratio as a lower limit of the traditional coupling ratio. As we explain further, the general coupling ratio does not discount the effect of strong fault interactions, which can produce complex faults semantically independent from the constituent faults. Hence, C is not bounded by any number, and will often be larger than κ, with $\kappa < 1$.

Contributions:

- We propose the *composite fault hypothesis* that resolves vagueness and ambiguity in the formal statement of the *coupling effect* for non-adequate mutation scores.
- Our theoretical analysis results in the *composite fault hypothesis* for general functions. We find the composite coupling ratio to be $1 - \frac{1}{n}$, where n is the *co-domain*.
- We show that our analysis remains valid even when considering recursion and loops.
- Using 25 projects, we compute the composite coupling ratio κ to be greater than 0.99, with 95% confidence. This helps substantiate the impact of composite coupling.

Our full data set is available for replication[3].

3.2 Related Work

Fault masking in digital circuits was studied before it was studied in software. Dias [35] studies the problem of fault masking, and derives an algebraic expression that details the number of faults to be considered for detection of all multiple faults. Morell [72] provided a theoretical treatment of fault based testing, and also [71] gave a formal treatment of the *coupling effect*. and shows impossibility of a general algorithm to identify fault coupling. Wah et al. [92–94, 96] using a simple model of finite functions (the *q-function* model, where q represents the number of functions thus composed) showed that the survival

[3] http://eecs.osuosl.org/rahul/icst2017/

ratio of first and second order test sets are respectively $\frac{1}{n}$ and $\frac{1}{(n^2-n)}$ where n is the order of the *domain* [55]. A major finding of Wah is that *the coupling effect weakens as the system size (in terms of number of functions in an execution path) increases (i.e. q increases), and it becomes unreliable when the system size nears the domain of functions.* Another important finding was that *minimization of test sets has a detrimental effect.* That is, for n faults, one should use n test cases, with each test case able to detect $n-1$ faults (rather than a single fault) to ensure that the test suite minimizes the risk of missing higher order faults due to fault masking. Kapoor [57] proved the existence of the *coupling effect* on logical faults. Voas et al. [91] and later Woodward et al. [99] suggested that functions with a high *DRR* (*domain* to *range* ratio) tend to mask faults. Al-Khanjari et al. [4], found that in some programs there is a strong relationship between *DRD* (Dynamic Range to Domain) ratio and testability.

Androutsopoulos et al. [10] found that one in ten tests suffered from failed error propagation. Clark et al. [23] found that likelihood of collisions was strongly correlated with an information theoretic measure called *squeeziness*, related to the amount of information destroyed on function application.

Our research is an extension of the theoretical work of Wah [93] and Offutt [76, 77]. The major theoretical difference from Wah [93] is that, given a pair of faulty functions that compose, we try to find the probability that, for given test data, the second function masks the error produced by the first one. On the other hand, Wah [93] tries to show that the *coupling effect* exists considering the *entire* program composed of q functions, each having a single fault (given by q in the q-function model). Next, Wah [93] assumes semantic separability of all complex faults. However, as we show, there exist a class of complex faults that are not semantically separable. We make this restriction clear. Further, our analysis shows that the probability of coupling is related to the *co-domain*, not the *domain*, as Wah [93] suggests. In fact, Wah [93] considers only functions which have exactly same *domain* and *range*, and hence are more restricted than our analysis. Finally, we show that even if syntax is considered, our analysis remains valid.

While Offutt [77] evaluates the traditional coupling effect, and shows the empirical relation with respect to *all* simple faults and their combinations, we aim to demonstrate the *composite fault hypothesis* and evaluate the relation between any pair of faults, and the combined fault including both.

3.3 Theory of Fault Coupling

We start with a function compositional view of programs (similar to Wah [96]). While Wah considered composition of q functions, with as many as q faults, we consider only pairs of faulty functions, since any faulty program with a number of separable faults can be modeled as composition of two functions with (possibly complex) faults.

We have the following assumptions, and simplifications (also made by Wah [93]): our biggest simplification is modeling programs by mathematical functions. While, theoretically, there can be an infinite number of alternatives to any given program, practically, the *domain* and *co-domain* often determines the plausible syntactical alternatives. Next, we assume a finite *domain* and *co-domain*, and consider only total functions. We also assume that faulty versions have same *domain* and *co-domain* (that is, the same type) as that of the non-faulty version. Since any function can be regarded as a single parameter function by considering the input as composed of a tuple of all the original parameters, we restrict our analysis to single parameter functions. While Wah considers how a known number of test inputs (1, 2, 3, or more than 3), some of which can detect some of the component faulty functions, can together detect the composite faulty function, we consider the probability of any single test input that can detect a fault being masked by a new fault. This allow us to significantly simplify our analysis.

Note that the theory *does not* rely on the constituent faults being considered to be simple.

A major idea in our analysis is the semantic separability of faults. Two faults present in a function are said to be separable *if and only if* the smallest possible chunk containing both faults can be decomposed into two functions g and h such that each fault is isolated within a single function (providing g_a and h_b as faulty functions), the behavior of composition $h \circ g$ equals the behavior of the original chunk in terms of input and output , and composition $h_b \circ g_a$ equals the behavior of the function with both faults. A *chunk* here is any small section of the program that can be replaced by an independent function preserving the behavior.

That is, given a function:

```
def functionX(x, y, n)
  for i in (1..n):
    y = faultyA(x)           (1)
    if odd(i): x = faultyB(y) (2)
    x += 1
```

The lines (1) and (2) together form a chunk. The interaction between the faults and their separability is discussed next.

3.3.1 Interaction Between Faults

There are two kinds of interaction between faults: *weak*, and *strong*. *Weak* interactions occur when faults can be semantically separated. That is, given two faults $\hat{a}$ and $\hat{b}$ in a function f, which can be split into $f_{ab} = h_b \circ g_a$, where g_a and h_b are faulty functions, the only interaction between $\hat{a}$ and $\hat{b}$ is because the fault $\hat{a}$ modifies the input of h (or h_b) from $g(i_0)$ to $g_a(i_0)$ (where i_0 is an input for f). That is, the interaction can be represented by a modified input value.

Strong interactions happen when the interpretation of the second fault is affected by the first, and hence faults can't be semantically separated. For example, consider:

```
def swap(x,y): x,y=y,x
```

Say this was mutated into

```
def swap(x,y): x,y=x,y
```

Clearly, there were two independent lexical changes: $x \to y$ and $y \to x$. However, consider the disassembly:

```
>>> dis.dis(swap)
  1    0 LOAD_FAST      1 (y)
       3 LOAD_FAST      0 (x)
       6 ROT_TWO
       7 STORE_FAST     0 (x)
      10 STORE_FAST     1 (y)
      13 LOAD_CONST     0 (None)
      16 RETURN_VALUE
```

The changes in source resulted in intertwined bytecode changes, and hence cannot be separated. Since the faults cannot be separated, *strong* interactions produce faults with different characteristic from the component simple faults, and hence should be considered independent atomic faults[4]. Why should we consider the semantically inseparable faults as independent faults? An intuitive argument is to consider two functions that implement id (these are not strongly interacting). That is, given any value x, we have $g(x) = h(x) = x$. If two faults $\hat{a}$, and $\hat{b}$ occur as we suggest above in g and h, causing inputs i to g_a and inputs j to h_b to fail, then the faulty inputs for $h_b \circ g_a$ are bounded by $i \cup j$, where i represents inputs to f that result in faulty outputs due to faulty g and j, inputs to f resulting in faulty outputs due to faulty h.

What about fault masking? Any input i that failed for g_a could possibly result in an input value that would cause a failure for h_b. For any element outside of i, there is no possibility of two faults acting on it, and hence no possibility of fault masking. However, if the faults are not semantically separable, one cannot make these guarantees, as the faulty inputs may be larger than $i \cup j$ or even completely different. In the general case, when the interaction is weak, we expect the faulty output for up to $i \cup j$.

For formal proof, consider a function f that has *domain* x, represented as $h \circ g$ using two functions. Replacing g with g_a causes $i \in x$ inputs to result in faults. Similarly, replacing h with h_b causes $j \in x$ inputs to f to result in faults. Joining together to form f_{ab}, we know that any of $i \in x$ has a potential to produce a faulty output unless it was masked by h_b. Similarly, any of $j \in x$ also has the possibility of producing a faulty output. Now, consider any element k not in either i or j. It will not result in a faulty output while passing through g_a because it is not in i, further, the value $g_a(k) = g(k) = k_1$. We already know that k_1 would not result in a faulty output from h_b because $k \notin j$. Hence, any element $k \notin i \cup j$ will not be affected by faults $\hat{a}$ and $\hat{b}$.

We can make this assertion only because we can replace g and h separately. If $\hat{a}$ and $\hat{b}$ interacted strongly, any function could potentially replace f. Hence, any element in x may potentially result in a fault when f_{ab} is applied. Harman et al. [48] calls these de-coupled higher order mutants.

Depending on the language used, other features causing strong faulty interaction may exist.

[4]Wah [93] ignores strong interaction of faults.

3.3.2 Analysis

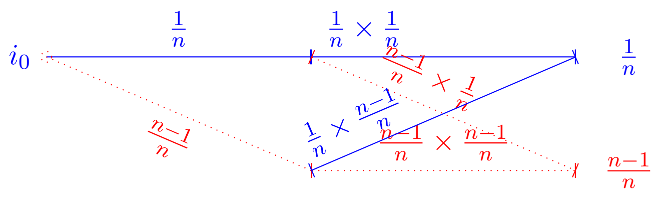
Consider a program f with two simple faults $\hat{a}$, and $\hat{b}$, which can be applied to f to produce two functions f_a and f_b containing one fault each, and f_{ab} containing both faults (Figure 3.2). Say such a program can be partitioned into two functions g and h ($f = h \circ g$) with restriction that $\hat{a}$ lies in g, producing alternative g_a, and $\hat{b}$ lies in h producing h_b, such that the new faulty version of f containing both is given by $f_{ab} = h_b \circ g_a$. We note that the particular kind of fault depends on the syntax and semantics of the programming language used, and there can be fault pairs that cannot be separated cleanly. As stated previously, we ignore these kinds of fault pairs as they are syntax dependent and strongly interacting. Hence, no general solution is possible for these faults.

Given that we can distinguish a fault in isolation using a given input, what is the probability that another fault would not result in the masking of that fault for the same input? That is, given a test input i_0 for f, able to distinguish $(f,\ f_a)$, what is the probability that $(f,\ f_{ab})$ can be distinguished by the same input?

Since we know that f_a is distinguished from f, we know that $g_a(i_0) \neq g(i_0)$. Hence, the function h_b will have a different input than h. Thus, the question simplifies to: given an alternate input for function h (or anything that can be substituted in its place), what is the probability that a faulty h, with the new input $g_a(i_0)$ will result in same output as the old h, with the old input $g(i_0)$?

Let us assume for simplicity that functions g and h have fixed *domain* and a *co-domain* given by $g \in G : \mathbb{L} \rightarrow \mathbb{M}$ and $h \in H : \mathbb{M} \rightarrow \mathbb{N}$. That is, h belongs to a set of functions H, which has a *domain* M, and a *co-domain* N such that $m = |M|$ and $n = |N|$. Considering all possible functions in H, with the given *domain* and *co-domain*, there will be n^m unique functions in H (separated by at least one different $\{input, output\}$ pair).

The only constraint on h_b we have is that $h_b(g_a(i_0))$ should result in the same output as $h(g(i_0))$. We are looking for functions that can vary in every other $\{input, output\}$ pair except for the pair given by $\{g_a(i_0), h(g(i_0))\}$. There are n^{m-1} functions that can do that out of $|H| = n^m$ functions. That is, the composite coupling ratio is given by $\kappa = 1 - \frac{n^{m-1}}{n^m}$, which is simplified to $1 - \frac{1}{n}$ of the total number of eligible functions where m is the size of *domain*, and n is the size of *co-domain* of the function. That is, given



i_0
$\frac{1}{n}$
$\frac{1}{n} \times \frac{1}{n}$
$\frac{1}{n}$
$\frac{n-1}{n} \times \frac{1}{n}$
$\frac{1}{n} \times \frac{n-1}{n}$
$\frac{n-1}{n}$
$\frac{n-1}{n} \times \frac{n-1}{n}$
$\frac{n-1}{n}$

Figure 3.1: Recursive interaction. The *blue solid* lines represent the masking values where the values are same as what would be expected before the fault was introduced, and the *red dotted* lines represent values that are different from the non-faulty version so that faults could be detected.

any test input, the probability of the composite coupling effect where the fault in one constituent is not masked by the fault in another is $1 - \frac{1}{n}$, and $\frac{1}{n}$ tends to be very small when the *co-domain* of the function (n) is large.

A symmetric argument can be made when the function fixed is h, and g varies. There are m^l functions in G, of which m^{l-1} can be used as a replacement without affecting $\{input, output\}$, in which case, the probability of composite coupling effect is $1 - \frac{1}{m}$ where m is the *co-domain*[5].

3.3.3 Recursion and Iteration

Recursion and *iteration* can present challenges to our analysis. For example, consider:

```
while y > 0: y = h(g(y))
```

The two functions g and h are otherwise independent. However, the input of h influences g, and vice versa. Here, we do not know when the loop will end, and any faults will be detected. The faults may be detected after a larger or smaller number of iterations than the non faulty version. Hence, we consider the chances of propagation of the faulty value after each iteration. That is, if a faulty value is present after executing the function g_a once, what are the chances that it will be caught at the end of each iteration?

Let f denote the program segment composed of g and h. After the first iteration of

[5]We note that the logic of probability is very similar to Wah [93], and this is the same value derived by Wah for single test input, where n is the *domain* of the function as Wah does not consider functions that have a different *domain* and *co-domain*.

f, we will have $\frac{1}{n}$ possibility for fault masking as we discussed before, and $\frac{n-1}{n}$ possibility for detectable faulty values. Now, consider the next iteration. In this case, of the original $\frac{1}{n}$ masked outputs, $\frac{1}{n}$ will again be masked, for a total of $\frac{1}{n^2}$, and the remaining $\frac{(n-1)}{n^2}$ will have a value that is faulty. Consider the original $\frac{n-1}{n}$ that had faulty values in the first iteration. Out of that, $\frac{1}{n}$ will be masked in the second iteration (i.e. $\frac{n-1}{n^2}$). Similarly, $\frac{(n-1)^2}{n^2}$ of the original faulty outputs will remain faulty. That is, after second iteration, we will have $\frac{1}{n^2} + \frac{n-1}{n^2} = \frac{1}{n}$ masked output values. Similarly, we will have $\frac{n-1}{n^2} + \frac{(n-1)^2}{n^2} = \frac{n-1}{n}$ possibility of faulty output values. That is, after each iteration, we will have $\frac{1}{n}$ possibility of fault masking (See Figure 3.1). Hence, composite fault hypothesis will hold even for recursion and iteration.

3.3.3.1 Premature loop exits

What if a fraction of inputs – say x – diverge so much (crashes or gets detected by asserts) that they never make it through all iterations? We can model this as the case where the remaining fraction ($y = 1 - x$) of inputs belong to a function with reduced *domain* and hence *co-domain*. This is more involved because functions with a smaller *co-domain* are more prone to fault masking. We need to show that the total fraction of masked values is lesser than the original $\frac{1}{n}$, or show the other side

$$x + \frac{ny-1}{ny} \geq \frac{n-1}{n} \tag{3.1}$$

We assume that $nx \geq 1$ (at least one input causes a crash) and $ny \geq 1$ (at least one input reaches the end – otherwise, there is no fault masking involved).

We simplify Equation 3.1 by first making the denominator the same (ny) and then simplifying, which results in the equation $nxy + ny - 1 \geq ny - y$. On expanding y to $1 - x$, and simplifying, we get $ny \geq 1$. Note that this was our original assumption. Hence, premature loop exits result in a stronger coupling between faults.

What happens if instead of a fixed fraction, we have say $r\%$ input values detected at the end of each iteration? Of course, any finite number of loops could be modeled as we did above. If instead, we rely on the crashes alone to distinguish faulty values, we are still in luck. Each iteration detects $r\%$ of the input values, and the remaining $q = 1 - r\%$ of the values restart the iteration. This results in $r + rq + rq^2 + \ldots rq^{n-1}$ values getting detected at the end of n^{th} iteration. This infinite sum converges to 1. That is, no faults will be masked.

3.3.3.2 Different execution paths

Another wrinkle is the pattern where iteration proceeds in different paths during different executions. For example:

```
for i in 1..10:
  if odd(i): x = g(y)
  else: y = h(x)
```

In programs such as this, one may unroll the loop, i.e.

```
for i in 1..10:2:
    x = g(y)
    y = h(x)
```

which can make it amenable to the above treatment. Recursion can be resolved similarly. We do not claim that this is exhaustive. There could exist other patterns of recursion or iteration that do not fit this template. However, most common patterns of recursion and iteration could be captured in this pattern.

Can we extend the bounds we found ($i \cup j$ for faulty outputs) to recursion? Unfortunately, it is possible for a faulty function to interact with its own output during recursion, and hence mask a failure. Hence, we can not bound the failure causing inputs in a doubly faulty function that incorporates recursion.

3.3.4 Accounting for Multiple Faults

What happens when there are multiple faults? Say, we have a system modeled by $p \circ q \circ r \circ s \circ t \circ u$, where any of the functions may be faulty or not faulty, for example $p_a \circ q \circ r_b \circ s_c \circ t_d \circ u$. We can not directly apply the technique in recursion because there are non-faulty functions interspersed. The thing to remember here is that a non faulty function immediately adjacent to a faulty function can together be considered a faulty function. Hence, the above reduces to $(p_a \circ q) \circ r_b \circ s_c \circ (t_d \circ u)$, or equivalently

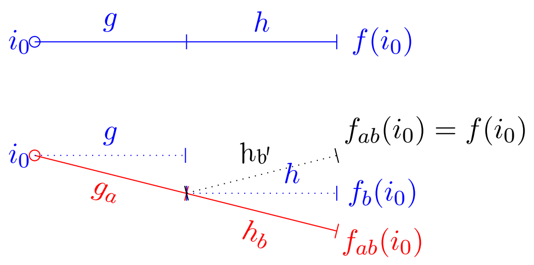


Figure 3.2: Fault interaction ($g_a(i_0)$ is masked by $h_{b'}$)

$pq_a \circ r_b \circ s_c \circ tu_d$. This is now amenable to the treatment in Figure 3.1 because each function now can produce $\frac{1}{n}$ non-faulty and $\frac{n-1}{n}$ faulty outputs. An additional complication is that a general expression is not possible unless we simplify further, and assumes that *domain* and *co-domain* of all functions are same. With this simplification, even when we consider a number of faulty functions, the mean ratio of fault masking remains the same at $\frac{1}{n}$. Indeed, this is one of the significant differences from Wah. Wah does not attempt to collapse the non-faulty functions to their neighbours. Why do we do this? Because we know that each faulty function on its own was detected by the test suite. That is, we know that $p \circ q_a \circ r \circ s \circ t \circ u$ would have been detected. Hence, we can certainly consider $pq_a \circ r \circ s \circ t \circ u$ as the set of functions where the function pq_a is the function with an atomic fault.

3.3.5 Dynamically Checked Languages

In the case of *dynamically checked* or *unityped* languages, every single function has the same type (*domain*, *co-domain*), and alternatives are large (but finite), because one may not identify a faulty input type until execution. Hence, we can expect large composite coupling ratio.

3.3.6 Impact of Syntax

In order to model composite coupling, we assumed that all faults are equally probable, which is often not the case, with faults that are closer syntactically being more probable than faults which are not in the syntactic neighborhood of correctness. In fact, we have some reasonable estimate of the distribution of size of faults that programmers make [46].

Implementation of functions as code need not necessarily follow the same distribution as that of their mathematical counterparts. For example, for mathematical functions, there exist only 4 functions that map from a boolean to a boolean. However, there can be an infinite number of program implementations of that function. The way it can be made tractable is again to consider the human element. The *competent programmer hypothesis* suggests that faulty programs are close (syntactically) to the correct versions. So one need only consider a limited number of alternatives (the number of which is a function of the size of the correct version, if one assumes that each token may be legally replaced by another).

As soon as we speak about syntactic neighborhood, the syntax of a language can have a large influence on which faults can be considered to be in a neighborhood. However, we note that most languages seem to follow a similar distribution of faults with a size below 10 tokens for 90% of faults [46].

Let us call the original input to h, $g(i_0) = j_0$, and the changed value $g_a(i_0) = j_a$. Similarly, let $f(i_0) = k_0$, $f_a(i_0) = k_a$, $f_b(i_0) = k_b$, and $f_{ab}(i_0) = k_{ab}$. Given two inputs i_0, and i_1 for a function f, we call i_0, and i_1 semantically close if their execution paths in f follow equivalent profiles, e.g taking the same branches and conditionals. We call i_0 and i_1 semantically far in terms of f if their execution profiles are different.

Consider the possibility of masking the output of g_a by h_b ($h_{b'}$ in Figure 3.2)). We already know that $h(j_a) = k_a$ was detected. That is, we know that j_a was sufficiently different from j_0, that it propagated through h to be caught by a test case. Say j_a was semantically far from j_0, and the difference (i.e the skipped part) contained the fault $\hat{b}$. In that case, the fault $\hat{b}$ would not have been executed, and since $k_{ab} = k_a$, it will always be detected.

On the other hand, say j_a was semantically close to j_0 in terms of g and the fault $\hat{b}$ was executed. There are again three possibilities. The first is that $\hat{b}$ had no impact, in which case the analysis is the same as before. The second is that $\hat{b}$ caused a change in the output. It is possible that the execution of $\hat{b}$ could be problematic enough to always cause an error, in which case we have $k_{ab} = k_b$ (error), and detection. Thus masking requires k_{ab} to be equal to k_0.

Even if we assume that the function h_b is close syntactically to h, and that this implies semantic closeness of functions h and h_b, we expect the value k_{ab} to be near k_a, and not k_0. This suggests that masking, even when considered in the light of syntactical neighborhood, is still unlikely, but this belief requires empirical verification since we are unable to assign probabilities to the cases above. Our empirical data (provided in the next section of this paper) should shed light on the actual incidence of masking when syntactic/semantic neighborhoods are taken into account, since real faults are likely in the syntactic and semantic neighborhood of the correct code.

A statistical observation can further buttress our argument. We know that if all functions were equally probable, fault masking has low probability. Now, consider the functions that are syntactically close to a given function. For most input values, we can assume that the syntactically close functions will have same output as that of the given function, more so than functions that are far away lexically. If h did not mask a value originally, (which we know since we were able to detect fault $h(g_a(i_0)))$, then the syntactically close functions to h will with a higher probability than a uniform sample, produce the same value as $h(g_a(i_0))$, which will be detected as faulty.

3.3.7 Can Strong Interaction be Avoided?

The *coupling effect* argues that if a test suite can find all atomic faults, then by composite fault hypothesis, a large percentage (κ) of complex faults will also be found. However, when can one assert that all atomic faults have been found? Any strong fault interaction has the potential to produce an atomic fault.

Given that the strong interaction is dependent on the execution, can runtime environment or compiler order computation so that strong interaction is no longer present?

Consider the function *swap (a,b) = (b,a)* that we examined earlier. We see how one may mistakenly use *id (a,b) = (a,b)* instead, and cause a strong interaction. Now, the question is, does there exist a way to split the two functions, so that the condition of separability can be satisfied? Given that there are only four possible functions that can operate on a tuple, (*swap (a,b) = (b,a)*, *id (a,b) = (a,b)*, *dupleft (a,b) = (a,a)*, *dupright (a,b) = (b,b)*) we could check it exhaustively. The condition is that the functions representing single faults should individually cause a detectable deviation on their own, and on composition, result in same behavior as *id*. Now, it can be seen that, neither of the single fault functions can behave like *swap* since that represents *no* fault, so they can not behave like *id*, since that suggests that the other faulty function behaves like *swap*. Hence, no compiler or runtime environment can remove the strong interaction in *swap*.

Where can we expect strong interaction to appear? While we can not provide an exhaustive overview of possible language features, we can demonstrate that even very simple languages such as the λ-calculus are vulnerable. Consider the λ-calculus expression $\lambda x\, y\, .y\, x$, and its faulty version $\lambda x\, y\, .x\, y$. There are two lexical points where the faults have been injected $\{x \to y, y \to x\}$. However, they cannot be separated out. That is, even such simple features can cause strong interaction.

3.4 Methodology for Assessment

Our methodology was guided by two principles [88]: We sought to minimize the number of variables, and tried to be as general as possible. Hence, we selected Apache commons for analysis.

For our set of projects, we iterated through their commit logs, and generated reverse patches for each commit. For each patch thus created, we applied the patch on the latest repository, and removed any changes to the test directory, thus ensuring that the test suite we tested with was always the latest. Any patch that resulted in a compilation error was removed. This resulted in a set of patches for each project that could be independently applied. The complete test suite for the project was executed on each of the patches left, and any patch that did not result in a test failure was removed. The failed test cases that corresponded to each patch were thus collected. At this point, we had a set of patches that introduce specific test case failures. The set of Apache projects, along with the set of reverse patches thus found, are given in Table 3.1.

We conducted our remaining analysis in two parts. For the first part, we generated patch pairs by joining together two random patches for any given project. For the projects where the total number of unique pairs was larger than 100, we randomly sampled 100 of the pairs produced. After removing patch combinations that resulted in compilation errors, we had 1,126 patch combinations. We evaluated the test suite of each project against the pair-patches thus generated, and collected the test cases which failed against these. Adopting the terminology of Jia et al. [54], out of 1,126, we had 1,126 coupled higher order mutants, and 56 subsuming mutants.[6] Out of these, there

[6]Of course, our patches are derived from actual faulty code, not mutants in the traditional sense of

Table 3.1: Apache Commons Libraries

	Projects	SLOC	TLOC	CPatches	Fails
1	commons-bcel	30,175	3,155	148	6
2	commons-beanutils	11,640	21,665	63	5
3	commons-cli	2,665	3,768	71	5
4	commons-codec	6,599	11,026	179	4
5	commons-collections	27,820	32,913	333	16
6	commons-compress	18,746	13,496	430	65
7	commons-configuration	26,793	37,806	322	78
8	commons-csv	1,421	3,168	150	8
9	commons-dbcp	11,259	8,487	98	18
10	commons-dbutils	3,064	3,699	43	1
11	commons-discovery	2,320	268	171	1
12	commons-exec	1,757	1,601	90	5
13	commons-fileupload	2,389	1,946	129	8
14	commons-imaging	31,152	6,525	174	4
15	commons-io	9,813	17,968	177	18
16	commons-jexl	10,921	9,509	54	10
17	commons-jxpath	18,773	6,137	10	2
18	commons-lang	25,468	43,981	571	49
19	commons-mail	2,720	3,869	48	5
20	commons-math	84,809	89,336	954	142
21	commons-net	19,749	7,465	454	21
22	commons-ognl	13,139	6,873	190	3
23	commons-pool	5,242	8,042	149	12
24	commons-scxml	9,524	5,119	74	7
25	commons-validator	6,681	7,926	126	17

SLOC is the program size in LOC, TLOC is the test suite size in LOC, CPatches is the number of compiled patches, and Fails is the number of test failures.

were only 2 strongly subsuming mutants.

We tried to reduce the number of external variables further for the second part, and chose a single large project — *Apache commons-math*. We generated a set of combined patches by joining 2, 4, 8, 16, 32, and 64 patches at random, and evaluated the test suite for commons-math against each of these k^{th} order patches. We removed all patches that resulted in any compilation errors, producing 342 patch combinations.

For both parts of our analysis, we generated two sets. The first set containing the unique failures from the constituent faults in isolation, and the second containing the combined patches.

3.5 Analysis

There are two questions that we tackle here. The first investigates the fraction of test cases that detect any of the constituent mutants that also detect the combined mutant. That is, evaluates the following prediction from the model: "Given two faults, and the test cases killing each, (assuming a sufficiently large *domain* and *co-domain*, and ignoring the effects of strong interaction), there is a high probability for the same test cases to kill the combined fault."

The second investigates the general coupling effect. Since the general coupling ratio does not distinguish between strong and weak interaction, this also serves as an evaluation of the strong interaction between faults where inputs other than the original i and j – that is, outside $i \cup j$ – becomes faulty (where i represents faulty inputs to f due to faults in h, and j represents faulty inputs to f due to faults in g).

Indeed, we believe that strong interaction between different faults is rarer than weak interaction. While there is no easy way to verify it, one may look at the newer faults (new test failures) that are introduced by a combination of patches when compared to the original patches as instances of strong fault interaction, which may be considered a reasonable proxy. Our empirical evaluation does not require individual patches to be simple faults. Our theory suggests that irrespective of whether the faults are complex or not, we can expect the same fault masking probability.

generated modification.

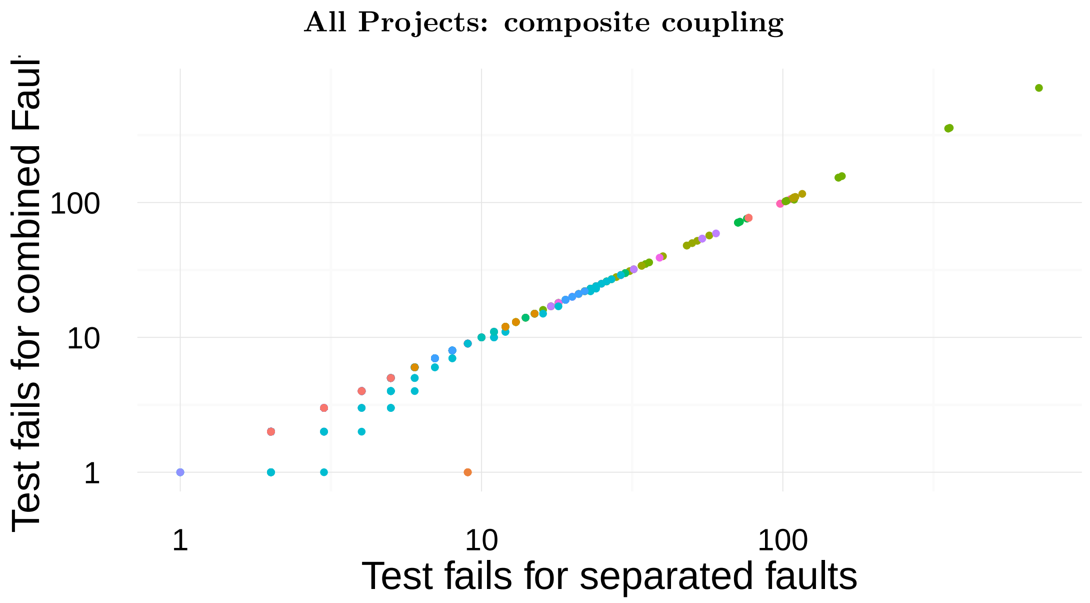


Figure 3.3: The size of set of the test cases able to detect the faults when they were separate is in the *x-axis*, and the *subset of the same test cases* able to detect the combined fault is in the *y-axis*. Colors correspond to projects.

3.5.1 All Projects

This section investigates fault pairs from all projects.

3.5.1.1 The Composite Fault Model

Here, we try to answer the question: *what percentage of test cases detecting constituent faults can detect the complex faults?*

Figure 3.3 plots the set of test cases able to detect the faults when they were separate with the set of test cases able to detect the combined fault. To analyze the fraction of test cases expected to detect the combined mutant, we evaluate the regression model given by:

$$\mu\{AfterT|BeforeT\} = \beta_0 + \beta_1 \times BeforeT \qquad (3.2)$$

where $BeforeT$ is the size of the test suite that includes all test cases that can detect both faults separately, and $AfterT$ is the size of the test suite which is a subset of

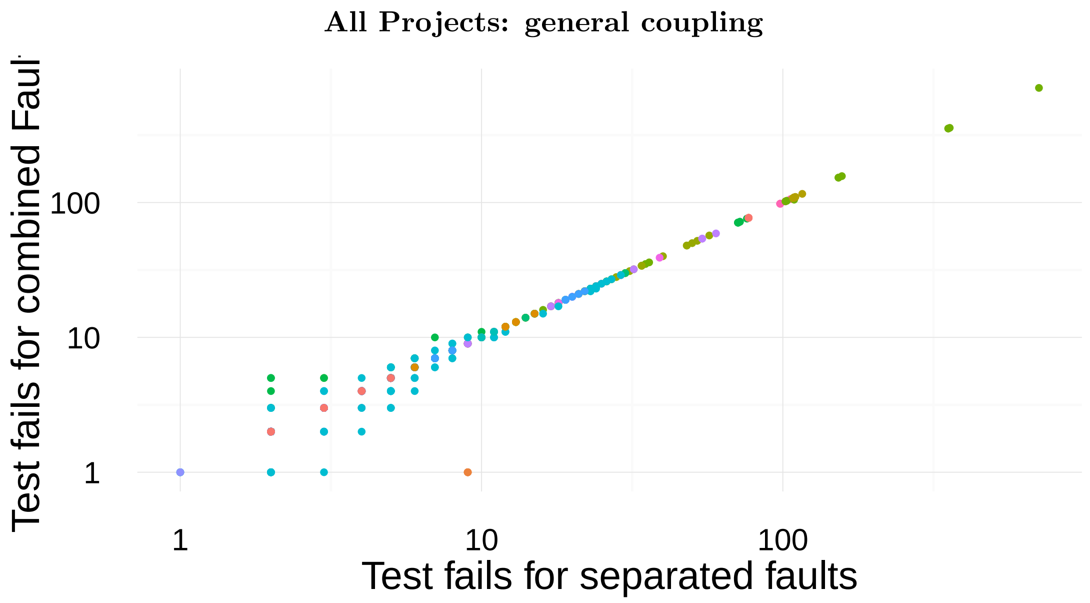


Figure 3.4: The size of the set of test cases able to detect the faults when they were separate is the *x-axis*, and the *set of all test cases* able to detect the combined fault is in the *y-axis*. Colors correspond to projects.

$BeforeT$ that can detect the fault pair when combined. We force β_0 to zero to account for the fact that if no test cases detected the original mutant, then the question of their fraction does not arise. This linear regression model lets us predict the number of test fails for combined faults from the test fails for separated faults.

We note that we are interested in β_1 for another purpose. β_1 is also the composite coupling ratio κ. Thus this regression provides us with a model for prediction, its goodness of fit (R^2), and also the composite coupling ratio.

3.5.1.2 The General Coupling Model

Figure 3.4 plots the general coupling of faults. We evaluate the following regression model.

$$\mu\{NewT|BeforeT\} = \beta_0 + \beta_1 \times BeforeT \tag{3.3}$$

where $BeforeT$ is the size of the test suite that includes all test cases that can detect both faults separately, and $NewT$ is the size of the test suite that can detect the fault pair when combined. Note that we do not set $\beta_0 = 0$ here as the combined fault pair may be detected by a new test case even if its constituents were not detected. In fact, β_0 represents the complex faults that became detectable due to interaction even though the constituent faults are not detectable.

However, if one wishes to investigate the general coupling ratio, we have to investigate a simpler regression model, because the general coupling ratio does not permit an intercept.

$$\mu\{NewT|BeforeT\} = \beta_1 \times BeforeT \tag{3.4}$$

Here, similar to the previous section, β_1 corresponds to the general coupling ratio C.

3.5.1.3 Strong fault interaction

The incidence of strong fault interaction may be ascertained by the average number of new test cases that failed for the combined patch. Note that this number is not exhaustive, as some of the original test cases may fail for new faulty behavior too, even if the behavior is not same as that of the component faults.

3.5.1.4 Effect of size of codebase

The effect of size of codebase may be ascertained by inspecting the correlation of SLOC with the coupling ratios.

3.5.2 Apache Commons-math

3.5.2.1 The Composite Fault Model

We try to answer the question *what percentage of test cases detecting constituent faults can detect the complex faults?* for Commons-math. We rely on the regression given by Equation 3.2. Figure 3.5 plots test cases able to detect the faults when they were separate with the test cases able to detect the combined fault.

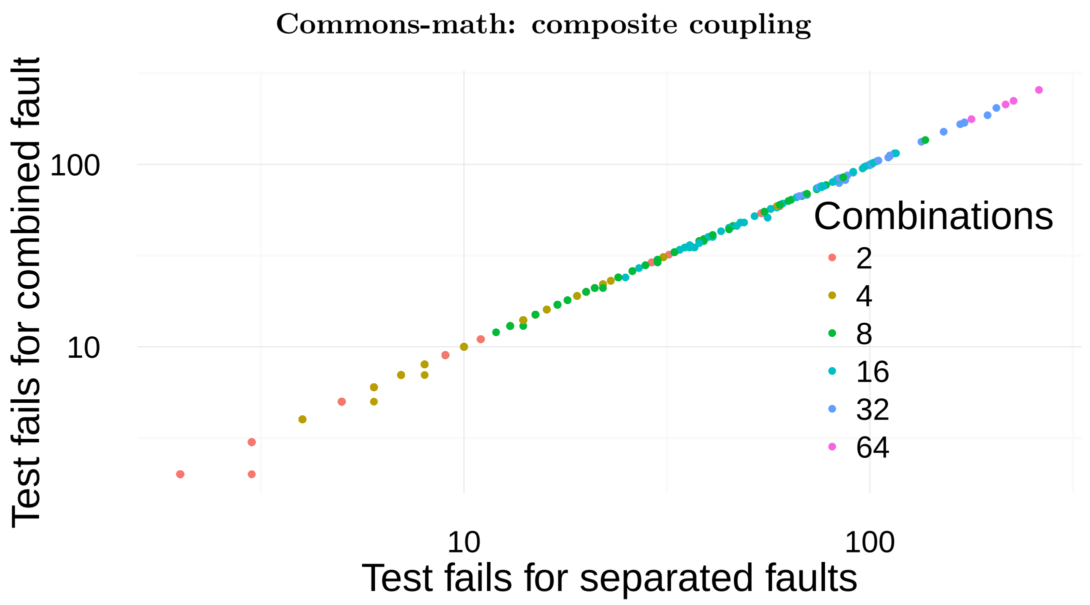


Figure 3.5: The set of test cases able to detect the faults when they were separate is in the *x-axis*, and the *subset of the same test cases* able to detect the combined fault is in the *y-axis*. Colors correspond to number of patch combinations.

3.5.2.2 The General Coupling Model

We rely on the regressions given by Equation 3.3 and Equation 3.4. Figure 3.6 plots the general coupling of faults for *Apache commons math*.

3.5.2.3 Strong fault interaction

The incidence of strong fault interaction may be ascertained by the average number of new test cases that failed for the combined patch. The difference of note here is that the number of patches are larger, and hence the chances of strong interaction are correspondingly larger.

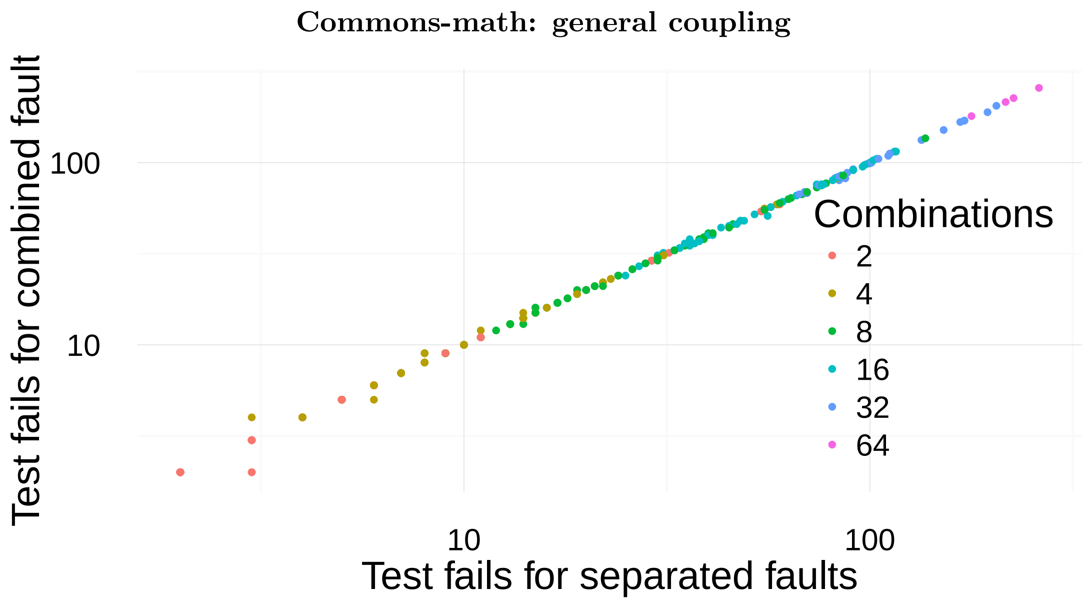


Figure 3.6: The set of test cases able to detect the faults when they were separate is in the *x-axis*, and the *set of all test cases* able to detect the combined fault is in the *y-axis*. Colors correspond to number of patch combinations.

Table 3.2: All projects for composite coupling ratio. R^2 =0.99975

	Estimate	Std. Error	t value	Pr(>|t|)
SeparateFaults	0.9992	0.0005	2,116.13	0.0000

3.6 Results

3.6.1 All Projects

The results for regression for Equation 3.2 for all projects is given in Table 3.2. The correlation between the dependent and independent variable is 0.99975. The composite coupling ratio was found to be 0.99916. The results for regression for Equation 3.3 for all projects is given in Table 3.3. The correlation between the dependent and independent variable is 0.99967. The results for regression for Equation 3.4 for all projects is given in Table 3.4. The general coupling ratio was found to be 0.99931. Further, the mean number of faulty test cases that were not present in the component faults were found to

Table 3.3: All projects R^2 =0.99967

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-0.0399	0.0189	-2.12	0.0343
SeparateFaults	0.9997	0.0005	1,847.83	0.0000

Table 3.4: All projects for general coupling ratio. R^2 =0.9997

	Estimate	Std. Error	t value	Pr(>|t|)
SeparateFaults	0.9993	0.0005	1,939.82	0.0000

be 0.0417. See Table 3.5 for the summary.

3.6.1.1 Effect of size of codebase

The effect of *SLOC* on coupling is given in Table 3.6[7].

3.6.2 Apache commons-math

The results for regression for Equation 3.2 for all projects is given in Table 3.7. The correlation between the dependent and independent variable is 0.99983. The composite coupling ratio was found to be 0.98956. The results for regression for Equation 3.3 for commons-math is given in Table 3.8. The correlation between the dependent and independent variable is 0.99971. The results for regression for Equation 3.4 for commons-math is given in Table 3.9. The general coupling ratio was found to be 0.9944. Further, the mean number of faulty test cases that were not present in the component faults were found to be 0.137. See Table 3.10 for the summary.

3.7 Discussion

Fault masking is one of the key concerns in software testing. The *coupling effect* hypothesis asserts that fault masking is rare. Unfortunately, little is known about the theory

[7] This table is not present in the published version, but generated from the data published along with the paper. Including it here for the context.

Table 3.5: Summary for all projects.

	SeparateFaults	JoinedFaults	RemovedFaults	AddedFaults
bcel	27.73	27.73	0.00	0.00
beanutils	4.80	1.60	3.20	0.00
cli	7.60	7.60	0.00	0.00
codec	2.50	2.50	0.00	0.00
collections	16.49	16.49	0.00	0.00
compress	11.60	11.60	0.00	0.00
configuration	37.19	37.16	0.04	0.02
csv	2.00	2.00	0.00	0.00
dbcp	10.60	10.91	0.01	0.32
exec	16.50	16.50	0.00	0.00
fileupload	4.64	4.64	0.00	0.00
imaging	6.50	6.50	0.00	0.00
io	7.93	7.55	0.54	0.17
jexl	3.58	3.56	0.02	0.00
jxpath	3.00	3.00	0.00	0.00
lang	4.46	4.46	0.00	0.00
mail	2.30	2.30	0.00	0.00
math	7.67	7.64	0.03	0.00
net	4.13	4.13	0.00	0.00
ognl	27.00	27.00	0.00	0.00
pool	4.48	4.45	0.05	0.02
scxml	36.62	36.62	0.00	0.00
validator	3.07	3.06	0.01	0.00

Table 3.6: All projects – correlation between size of codebase (SLOC) and coupling ratios. (Significance given in lower triangle)

	SLOC	Coupling Ratio	Composite Ratio
SLOC		0.009	0.082
Coupling Ratio	.		0.666
Composite Ratio	**	*	

Table 3.7: C-math for composite coupling ratio. R^2 =0.99983

	Estimate	Std. Error	t value	Pr(>|t|)
SeparateFaults	0.9896	0.0007	1,418.94	0.0000

Table 3.8: C-math R^2 =0.99971

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.0924	0.0482	1.92	0.0563
SeparateFaults	0.9933	0.0009	1,090.92	0.0000

Table 3.9: C-math for general coupling ratio. R^2 =0.99983

	Estimate	Std. Error	t value	Pr(>|t|)
SeparateFaults	0.9944	0.0007	1,401.55	0.0000

Table 3.10: Summary for all Commons-math.

	SeparateFaults	JoinedFaults	RemovedFaults	AddedFaults
2	7.67	7.64	0.03	0.00
4	14.67	14.70	0.05	0.07
8	30.53	30.42	0.17	0.06
16	59.25	59.09	0.42	0.25
32	109.85	108.96	1.37	0.48
64	220.25	219.50	3.00	2.25

behind fault coupling. We study the *coupling effect* and fault masking using theoretical and empirical methods.

Our theoretical evaluation of the *composite fault hypothesis*. shows that for any pair of separable faults, composite coupling effect exists. We find that composite coupling ratio $\kappa = 1 - \frac{1}{n}$, where n is the *co-domain* of the function being considered, and that syntactical neighborhood does not have an adverse impact on our result. Further, while Wah suggests that, as system size increases the *coupling effect* weakens exponentially, our results suggest that the mean coupling ratio remains the same at $\frac{1}{n}$ for an idealized system, and strengthens when premature exits from recursion or iteration is encountered. Indeed, our results from Table 3.6 suggests that there is a very small, but statistically significant positive correlation between composite coupling ratio and the size of the code base[8].

Why is our prediction on fault masking so important? Basic testing relies on fault masking. Say you are unit testing a function with multiple faults, and some of the faults are left undetected due to fault masking. Wah's analysis suggests that when we integrate these units into a larger system, the faults in the larger system have a much higher (indeed exponential) tendency to self correct, and avoid failure due to masking. Our analysis suggests that even on larger systems composed of smaller systems, the rate of fault masking remains the same.

We proposed the existence of strongly interacting faults, which cannot be accounted for within the formal coupling theory. Our empirical analysis (see Table 3.5 and Table 3.10) indicates that strong interaction is possibly rare, occurring at a similar frequency as fault masking. Figure 3.3 suggests that while there is some reduction in the combined faults for the faults with smaller semantic footprint (as given by the number of test cases that failed for that fault) with respect to constituent faults, the difference vanishes when the size of the fault increases. This same effect is also seen in Figure 3.5.

The results for regression (Equation 3.2) also suggest a similar observation — that test cases that are able to detect a fault in isolation will with very high probability detect the same fault in combination with other faults.

Overall, our statistical analysis suggests that there is a very high probability (between {0.998 & 1.000} for all projects, and {0.988 & 0.991} for commons-math — 95% confi-

[8]This conclusion is drawn from the data published along with the paper but was not present in the published version. Including it here for context.

dence interval with statistical significance $p < 0.0001$) that when two faults are paired to produce a combined fault, any test cases that detected either of the faults continue to detect the combined fault.

Our results for Table 3.4 suggests that between $\{0.998 \ \& \ 1.000\}$ of complex faults are caught (95% confidence interval, $p < 0.0001$). This is again confirmed by the deeper analysis of *Apache commons-math*, using larger size faults in Table 3.9 which suggests that between $\{0.993 \ \& \ 0.996\}$ fraction of complex faults are caught (95% confidence interval, $p < 0.0001$). We note that this is the first confirmation of the *general coupling effect* (unlike the *mutation coupling effect* which has been validated multiple times). Why is validating the general coupling effect important? We already know that faults emulated by traditional mutants are only a subset of the possible kinds of faults (Just et al. [56] found that up to 27% of faults were inadequately represented by mutants). Hence, it is important to verify the *general coupling effect* using real faults so that our results are applicable for faults in general, and especially for possible future mutation operators. Indeed, the *mutation coupling effect* has been validated multiple times, and we do not attempt it again here.

3.8 Conclusion

The *coupling effect* hypothesis is a general theory of fault interaction, and is used to quantify *fault masking*. It also finds use in mutation analysis. While there is compelling empirical evidence for the *coupling effect*, our theoretical understanding is lacking. The extant theory by Wah is too restrictive to be useful for real world systems. We address this limitation, and provide a stronger, modified version of the theory called the *composite fault hypothesis*.

Our theoretical analysis suggests that the composite fault hypothesis has a high probability of occurring ($1 - \frac{1}{n}$, where n is the *co-domain* of the function under consideration) under the assumptions of total functions, finite *domain*, and separability of faults, *irrespective of the size of the system.*

Our empirical study provides validation, and an empirical approximation of the composite coupling ratio κ (0.99), with 99% of the test cases that detected a fault in isolation continuing to detect it when it is combined with other faults.

On the limits of mutation reduction strategies

Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, Alex Groce

2016 IEEE/ACM International Conference on Software Engineering
Texas, United States
pp. 511-522.

Chapter 4: On the limits of mutation reduction strategies

4.1 Introduction

The quality of software is a pressing concern for the software industry, and is usually determined by comprehensive testing. However, tests are themselves programs, (usually) written by human beings, and their quality needs to be monitored to ensure that they in fact are useful in ensuring software quality (e.g., it is important to determine if the tests are also a quality software system).

Mutation analysis [18,63] is currently the recommended method [9] for evaluating the efficacy of a test suite. It involves systematic transformation of a program through the introduction of small syntactical changes, each of which is evaluated against the given test suite. A mutant that can be distinguished from the original program by the test suite is deemed to have been *killed* by the test suite, and the ratio of all such mutants to the set of mutants identified by a test suite is its mutation (kill) score, taken as an effectiveness measure of the test suite.

Mutation analysis has been validated many times in the past. Andrews et al. [8,9], and more recently Just et al. [56], found that faults generated through mutation analysis resemble real bugs, their ease of detection is similar to that of real faults, and most importantly for us, a test suite's effectiveness against mutants is similar to its effectiveness against real faults.

However, mutation analysis has failed to gain widespread adoption in software engineering practice due to its substantial computational requirements — the number of mutants generated needs to be many times the number of program tokens in order to achieve exhaustive coverage of even first order mutants (involving one syntactic change at a time), and each mutant needs to be evaluated by a potentially full test suite run. A number of strategies have been proposed to deal with the computational cost of mutation analysis. These have been classified [80] orthogonally into *do faster*, *do smarter*, and *do fewer* approaches, corresponding to whether they improve the speed of execution of a single mutant, parallelize the evaluation of mutants, or reduce the number of mutants evaluated.

A large number of *do fewer* strategies — mutation reduction methods that seek to intelligently choose a smaller, representative, set of mutants to evaluate — have been investigated in the past. They are broadly divided into operator selection strategies, which seek to identify the smallest subset of mutation operators that generate the most useful mutants [79, 85], and strata sampling [1, 16] techniques, which seek to identify groups of mutants that have high similarity between them to reduce the number of mutants while maintaining representativeness and diversity [101, 102]. Even more complex methods using clustering [33, 65], static analysis [52, 59] and other intelligent techniques [89] are under active research [34].

These efforts raise an important question: What is the actual effectiveness of a perfect mutation reduction strategy over the baseline – random sampling – given any arbitrary program?

We define the **efficiency** of a selection technique as the amount of reduction achieved, and the **effectiveness** as the selection technique's ability to choose a representative reduced set of mutants, that require as many test cases to kill as the original set of mutants. The ratio of effectiveness of a technique to that of random sampling is taken as the **utility** of the technique.

We approach these questions from two directions. First, we consider a simple theoretical framework in which to evaluate the improvement in effectiveness for the best mutation reduction possible, using a few simplifying assumptions, and given oracular knowledge of mutation kills. This helps set the base-line. Second, we empirically evaluate the best mutation reduction possible for a large number of projects, given post hoc (that is, oracular) detection knowledge. This gives us practical (and optimistic) limits given common project characteristics.

Our contributions are as follows:

- We find a theoretical upper limit for the effectiveness of mutation reduction strategies of 58.2% for a uniform distribution of mutants — the distribution most favorable for random sampling. We later show that for real world programs, the impact of distribution is very small (4.467%) suggesting that uniform distribution is a reasonable approximation.

- We find an empirical upper limit for effectiveness through the evaluation of a large number of open source projects, which suggests a maximum practical utility of 13.078% on average, and for 95% of projects, a maximum utility between 12.218% and 14.26% (*one sample u-test* $p < 0.001$)[1].

- We show that even if we consider a set of mutants that are distinguished by at least by one test (thus discounting the impact of skew in redundant mutants) we can expect a maximum utility of 17.545% on average, and for 95% of projects, a maximum utility between 16.912% and 18.876% (*one sample u-test* $p < 0.001$).

What do our results mean for the future of mutation reduction strategies? Any advantage we gain over random sampling is indeed an advantage, however small. However, our understanding of mutant semiotics[2] is as yet imperfect, and insufficient to infer whether the kind of selection employed is advantageous. In fact, our research shows that current operator selection strategies seldom provide any advantage over random sampling, and even strata sampling based on program elements never achieves more than a 10% advantage over pure random sampling. Our results suggest that the effort spent towards improving mutant selection mechanisms should be carefully weighed against the potential maximum utility, and the risks associated with actually making things worse through biased sampling. We note that even selection based on subsumption is not [62] a foregone conclusion.

Our research is also an endorsement of the need for further research into new mutators. Consider what would happen if one uses of new mutators to produce new mutants and then randomly sample the same number of mutants as that of the original set. In the theoretical framework we developed, with the simplifications such as uniform redundancy of mutants, and non-overlapping test cases, the worst case is when the added mutants duplicate the already available mutants in the population. However, sampling from that would still result in the same number of unique mutants on average. That is, there is no disadvantage to using new mutators, thus increasing the number of mutants, and using sampling for reduction. On the other hand, assuming that the sample size is larger than the number of unique mutants present, there is no upper bound on the

[1]We use the non-parametric Mann-Whitney *u-test* as it is more robust to normality assumption, and to outliers. We note that a *t-test* also gives similar results.

[2]Here semiotics is the relation between a syntactic change and its semantic impact.

advantage for adding new mutators (or it is dependent on the sample size).

What about the real world?, it is possible that the logic for adding new mutators was flawed, and resulted in a set of redundant mutants in similar distribution as that of the original population. In such a case, a random sample from the new population will have as many unique mutants as the original population, which means no advantage or disadvantage. (Indeed, this is not the worst case. It is possible for the new set of redundant mutants to skew the population of mutants completely, but given that the new mutants are to be added after adequate investigation, we believe this to be of low probability). The best case is when all the added mutants are representatives of unique faults. In such a case, the advantage gained by a random sample from the new population from a random sample from the original sample can be bounded only by the sample size. If the original sample size was much larger than the number of unique mutants, the advantage thus gained may be huge (and potentially unbounded if one considers larger and larger samples).

The asymmetry between improvement obtained by operator removal and operator addition is caused by the difference in population from which the random comparison sample is drawn. For operator selection, the perfect set remaining after removal of operators is a subset of the original population. Since the random sample is drawn from the original population, it can potentially contain a mutant from each strata in the perfect set. For operator addition, the new perfect set is a superset of the original population, with as many new strata as there are new mutants (no bounds on the number of new strata). Since the random sample is constructed from the original population, it does not contain the newly added strata.

Our results suggest a higher payoff in **finding newer categories** of mutations, than in trying to reduce the mutation operators already available.

4.2 Related Work

According to Mathur [68], the idea of mutation analysis was first proposed by Richard Lipton, and formalized by DeMillo et al. [31] A practical implementation of mutation analysis was done by Budd et al. [17] in 1980.

Mutation analysis subsumes different coverage measures [16, 69, 81]; the faults produced are similar to real faults in terms of the errors produced [27] and ease of detection [8, 9]. Just et al. [56] investigated the relation between mutation score and test case effectiveness using 357 real bugs, and found that the mutation score increased with effectiveness for 75% of cases, which was better than the 46% reported for structural coverage.

Performing a mutation analysis is usually costly due to the large number of test runs required for a full analysis [55]. There are several approaches to reducing the cost of mutation analysis, categorized by Offutt and Untch [80] as: do *fewer*, do *smarter*, and do *faster*. The *do fewer* approaches include selective mutation and mutant sampling, while weak mutation, parallelization of mutation analysis, and space/time trade-offs are grouped under the umbrella of *do smarter*. Finally, the *do faster* approaches include mutant schema generation, code patching, and other methods.

The idea of using only a subset of mutants was conceived along with mutation analysis itself. Budd [16] and Acree [1] showed that even 10% sampling approximates the full mutation score with 99% accuracy. This idea was further explored by Mathur [67], Wong et al. [97, 98], and Offutt et al. [79] using Mothra [29] for Fortran.

A number of studies have looked at the relative merits of operator selection and random sampling criteria. Wong et al. [98] compared x% selection of each mutant type with operator selection using just two mutation operators and found that both achieved similar accuracy and reduction (80%). Mresa et al. [73] used the cost of detection as a means of operator selection. They found that if a very high mutation score (close to 100%) is required, x% selective mutation is better than operator selection, and, conversely, for lower scores, operator selection would be better if the cost of detecting mutants is considered.

Zhang et al. [102] compared operator-based mutant selection techniques to random sampling. They found that none of the selection techniques were superior to random sampling. They also found that uniform sampling is more effective for larger programs compared to strata sampling on operators[3], and the reverse is true for smaller programs. Recently, Zhang et al. [101] confirmed that sampling as few as 5% of mutants is sufficient for a very high correlation (99%) with the full mutation score, with even fewer mutants having a good potential for retaining high accuracy. They investigated eight sampling

[3]The authors choose a random operator, and then a mutant of that operator. This is in effect strata sampling on operators given equal operator priority.

strategies on top of operator-based mutant selection and found that sampling strategies based on program components (methods in particular) performed best.

Some studies have tried to find a set of *sufficient mutation operators* that reduce the cost of mutation but maintain correlation with the full mutation score. Offutt et al. [79] suggested an n-selective approach with step-by-step removal of operators that produce the most numerous mutations. Barbosa et al. [14] provided a set of guidelines for selecting such mutation operators. Namin et al. [74, 87] formulated the problem as a variable reduction problem, and found that just 28 out of 108 operators in Proteum were sufficient for accurate results.

Using only the statement deletion operator was first suggested by Untch [90], who found that it had the highest correlation ($R^2 = 0.97$) with the full mutation score compared to other operator selection methods, while generating the smallest number of mutants. This was further reinforced by Deng et al. [32] who defined deletion for different language elements, and found that an accuracy of 92% is achieved while reducing the number of mutants by 80%.

A similar mutation reduction strategy is to cluster similar mutations together [34,50], which has been attempted based on domain analysis [52] and machine learning techniques based on graphs [89].

In operator and mutant subsumption, operators or mutants that do not significantly differ from others are eliminated. Kurtz et al. [60] found that a reduction of up to 24 times can be achieved using subsumption alone, even though the result is based on an investigation of a single program, `cal`. Research into subsumption of mutants also includes Higher Order Mutants (HOM), whereby multiple mutations are introduced into the same set of mutants, reducing the number of individual mutants by subsuming component mutants. HOMs were investigated by Jia et al. [53, 54], who found that they can reduce the number of mutants by 50%.

Ammann et al. [6] observe that the set of minimal mutants corresponding to a minimal test suite has the same cardinality as the test suite, and provides a simple algorithm for finding both a minimal test suite and a corresponding minimal mutant set. Their work also suggests this minimal mutant set as a way to evaluate the quality of a mutation reduction strategy. Finally, Ammann et al. also found that the particular strategies examined are rather poor when it comes to selecting representative mutants. Our work is an extension of Ammann et al. [6] in that we provide a theoretical and empirical bound to the amount of improvement that can be expected by *any* mutation reduction strategy.

In comparison with previous work [101, 102] our analysis is backed by theory and compares random sampling to the *limit of selection*. That is, the results from our study are applicable to techniques such as clustering using static analysis, and even improved strata sampling techniques. Further, we are the first to evaluate the effectiveness of non-adequate test suites (Zhang et al. [101] evaluates only the predictive power of non-adequate test suites, not effectiveness). Finally, previous research [101, 102] does not compare the effectiveness of the *same number* of mutants for sampling and operator selection, but rather different operator-selections with samples of increasing size such as 5%, 10% etc. We believe that practitioners will be more interested in comparing the effectiveness achieved by the same numbers of mutants.

4.3 Theoretical Analysis

The ideal outcome for a mutation reduction strategy is to find the minimum set of mutants that can represent the complete set of mutants. A mutation reduction strategy accomplishes this by identifying redundant mutants and grouping them together so that a single mutant is sufficient to represent the entire group. The advantage of such a strategy over random sampling depends on two characteristics of the mutant population. First, it depends on the number of redundant mutants in each group of such mutants. Random sampling works best when these groups have equal numbers of mutants in them (uniform distribution), while any other distribution of mutants (skew) results in lower effectiveness of random sampling. However, this distribution is dependent on the program being evaluated. Since our goal is to find the mean advantage for a perfect strategy for an arbitrary program, we use the conservative distribution (uniform) of mutants for our theoretical analysis (we show later that the actual impact of this skew is less than 5% for real world mutants).

The next consideration regards the minimum number of mutants required to represent the entire population of mutants. If a mutant can be distinguished from another in terms of tests that detect it, then we consider both to be *distinguishable* from each other in terms of faults they represent, and we pick a representative from each set of indistinguishable mutants. Note that, in the real world, the population of distinguishable mutants is often larger than the minimum number of mutants required to select a *minimum test suite*[4] able to kill the entire mutant population. This is because while some mutants are distinguishable from others in terms of tests that detect them, there may not be any test that uniquely kills them[5]. Since this is external to the mutant population, and also because such a minimum set of mutants does not represent the original population fully (we can get away with a lower number only because the test suite is inadequate), we assume that distinguishable mutants are uniquely identified by test cases. We note however, that having inadequate test suites favors random sampling, and hence lowers the advantage for a perfect mutation reduction strategy, because random sampling can now miss the mutant without penalty. We derive the limits of mutation reduction for this system using the best strategy possible, given oracular knowledge of mutant kills.

Impact of deviations of parameters:

Skew: The presence of skew reduces the effectiveness of random sampling, and hence increases the utility of the perfect strategy.

Distinguishability: Any distinguishable mutant that is not chosen by the strategy (due to not having a unique detecting test case) decreases the effectiveness of the selection strategy, decreasing its utility.

Before establishing a theoretical framework for utility of mutation reduction strategies, we must establish some terminology for the original and reduced mutant sets and their related test suites.

Terminology: Let M and $M_{strategy}$ denote the original set of mutants and the reduced set of mutants, respectively. The mutants from M killed by a test suite T are given by $kill(T, M)$ (We use M_{killed} as an alias for $kill(T, M)$). Similarly the tests in T that kill mutants in M are given by $cover(T, M)$.

$$\text{kill} : \mathbb{T} \times \mathbb{M} \rightarrow \mathbb{M}$$

$$\text{cover} : \mathbb{T} \times \mathbb{M} \rightarrow \mathbb{T}$$

[4]A *minimum* test suite with respect to a set of mutants is the smallest test suite that can kill all mutants in the set, and a *minimal* test suite is a test suite from which no further tests can be removed without decreaseing mutation score. Our approach tries to approximate the actual *minimum* test suite using the *greedy* algorithm that has an approximation bound of $k \cdot ln(n)$ where k is the true minimum, and n is the number of elements. Since we have a strong bound on the approximation, and since the algorithm is robust in practice, we use the *minimal* computed by the *greedy* algorithm as a proxy for the *minimum test suite*.

[5] Consider the mutant×test matrix (1 implies the test kills the mutant) $\{\{1, 1, 0\}, \{1, 0, 1\}, \{0, 1, 1\}\}$. While all the mutants are distinguishable, just two test cases are sufficient to kill them all.

The test suite $T_{strategy}$ can kill all mutants in $M_{strategy}$. That is, $kill(T_{strategy}, M_{strategy}) = M_{strategy}$. If it is minimized with respect to the mutants of the strategy, we denote it by $T^{min}_{strategy}$.

Two mutants m and m' are distinguished if the tests that kill them are different: $cover(T, \{m\}) \neq cover(T, \{m'\})$.

We use M^{uniq}_{killed} to denote the set of distinguished mutants from the original set such that $\forall_{m,m' \in M} cover(T, \{m\}) \neq cover(T, \{m'\})$.

The *utility* ($U_{strategy}$) of a strategy is improvement in effectiveness due to using that strategy compared to the baseline (the baseline is random sampling of the same number[6] of mutants). That is,

$$U_{strategy} = \left| \frac{kill(T^{min}_{strategy}, M)}{kill(T^{min}_{random}, M)} \right| - 1$$

Note that $T^{min}_{strategy}$ is minimized over the mutants **selected** by the strategy, and it is then applied against the **full set** of mutants (M) in $kill(T^{min}_{strategy}, M)$.

This follows the traditional evaluation of effectiveness, which goes as follows: start with the original set of mutants, and choose a subset of mutants according to the strategy. Then select a minimized set of test cases that can kill all the selected mutants. This minimized test suite is evaluated against the full set of mutants. If the mutation score obtained is greater than 99%, then the reduction is deemed to be effective. Note that we compare this score against the score of a random set of mutants of the same size, in order to handle the case where the full suite itself is not mutation adequate (or even close to adequate). Our utility answers the question: does this set of mutants better represent the test adequacy criteria represented by the full set of mutants than a random sample of the same size, and if so, by how much?

The strategy that can select the perfect set of representative mutants (the smallest set of mutants such that they have the same minimum test suite as the full set) is called the *perfect strategy*, with its utility denoted by $U_{perfect}$[7].

We now show how to derive an expression for the maximum $U_{perfect}$ for the idealized

[6]For the rest of the paper, we require that efficiency of random sampling is the same as that of the strategy it is compared to, i.e. $|M_{strategy}| = |M_{random}|$.

[7]Where unambiguous, we shorten the subscript such as p for *perfect*, and r for *random*.

system with the following restrictions.

1. We assume that we have an equal number of redundant mutants for each distinguished mutant.

From here on, we refer to a set of non-distinguished mutants as a *stratum*, and the entire population is referred to as the *strata*. Given any population of detected mutants, the mutation reduction strategy should produce a set of mutants such that if a test suite can kill all of the reduced set, the same test suite can kill all of the original mutant set (remember that $T_{strategy}$ kills all mutants in $M_{strategy}$). Hence,

$$kill(T_{perfect}, M) = kill(T, M)$$

The quality of the test suite thus selected is dependent on the number of unique mutants that we are able to sample. Since we have supposed a uniform distribution, say we have x elements per stratum, and total n mutants. Our sample size s would be $p \times k$ where k is the number of strata, p is the number of samples from each stratum, and is a natural number; i.e. the sample would contain elements from each stratum, and those would have equal representation. Note that there will be at least one sample, and one strata: i.e., $s \geq 1$. Since our strata are perfectly homogeneous by construction, in practice $p = 1$ is sufficient for perfect representation, and as we shall see below, ensures maximal advantage over random sampling.

Next, we evaluate the number of different (unique) strata expected in a random sample of the same size s.
Let X_i be a random variable defined by:

$$X_i = \begin{cases} 1 & \text{if strata } i \text{ appears in the sample} \\ 0 & \text{otherwise.} \end{cases}$$

Let X be the number of unique strata in the sample, which is given by: $X = \sum_{i=1}^{k} X_i$, and the expected value of X (considering that all mutants have equal chance to be sampled) is given by:

$$E(X) = E(\sum_{i=1}^{k} X_i) = \sum_{i=1}^{k} E(X_i) = k \times E(X_1)$$

Next, consider the probability that the mutant 1 has been selected, where the sample size was $s = p \times k$:

$$P[X_i = 1] = 1 - \left(\frac{k-1}{k}\right)^{pk}$$

The expectation of X_i:

$$E(X_1) = 1 \times P(X_i = 1)$$

Hence, the expected number of unique strata appearing in a random sample is:

$$k \times E(X_1) = k - k \times \left(\frac{k-1}{k}\right)^{pk}$$

We already know that the number of *unique* strata appearing in each strata-based sample is k (because it is perfect, so each strata is unique). Hence, we compute the utility as the difference divided by the baseline.

$$U_{max} = \frac{k - \left(k - k \times \left(\frac{k-1}{k}\right)^{pk}\right)}{k - k \times \left(\frac{k-1}{k}\right)^{pk}} = \frac{1}{(\frac{k}{k-1})^{pk} - 1} \tag{4.1}$$

This converges to[8]

$$\lim_{k\to\infty} \frac{1}{(\frac{k}{k-1})^{pk} - 1} = \frac{1}{e^p - 1} \tag{4.2}$$

and has a maximum value when $p = 1$.

$$U_{max} = \frac{1}{e-1} \approx 58.2\% \tag{4.3}$$

Note that this is the *mean* improvement expected over random sampling for *uniform distribution* of redundant mutants in strata (and with oracular knowledge). That is, individual samples could still be arbitrarily advantageous (after all, the perfect strata sample itself is one potential random sample), but on average this is the expected gain over random samples.

How do we interpret this result? If you have a robust set of test cases that is able to uniquely identify distinguishable mutants, then given an arbitrary program, you can

[8]While we can expect k to be finite for mutation testing, we are looking at the maximum possible value for this expression.

Table 4.1: PIT Mutation Operators. The (*) operators were added or extended by us.

IN	Remove negative sign from numbers
RV	Mutate return values
M	Mutate arithmetic operators
VMC	Remove void method calls
NC	Negate conditional statements
CB	Modify boundaries in logical conditions
I	Modify increment and decrement statements
NMC	Remove non-void method calls, returning default value
CC	Replace constructor calls, returning null
IC	Replace inline constants with default value
RI*	Remove increment and decrement statements
EMV	Replace member variable assignments with default value
ES	Modify switch statements
RS*	Replace switch labels with default (thus removing them)
RC*	Replace boolean conditions with true
DC*	Replace boolean conditions with false

expect a perfect strategy to have at least a mean 58.2% advantage over random sample of the same efficiency in terms of effectiveness. However, if the program produces redundant mutants that are skewed, then the advantage of perfect strategy with oracular knowledge will increase (depending on the amount of skew). Similarly, if the tests are not sufficient to identify distinguishable mutants uniquely, we can expect the advantage of the perfect strategy to decrease. Finally, strategies can rarely be expected to come close to perfection in terms of classifying mutants in terms of their behavior without post hoc knowledge of the kills. Hence the advantage held by such a strategy would be much much lower (or it may not even have an advantage).

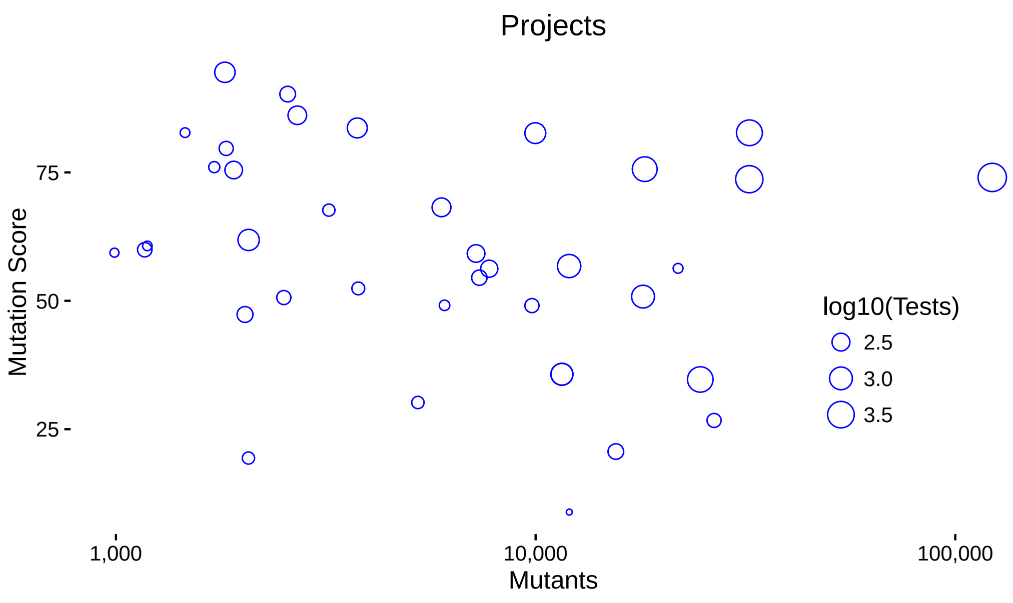


Figure 4.1: Distribution of number of mutants and test suites. It shows that we have a reasonable non-biased sample with both large programs with high mutation scores, and also small low scoring projects.

4.4 Empirical Analysis

The above analysis provides a theoretical framework for evaluating the advantage a sampling method can have over random sampling, with a set of mutants and test suite constructed with simplifying assumptions. It also gives us an expected limit for how good these techniques could get for a uniform distribution of mutants. However, in practice, it is unlikely that real test suites and mutant sets meet our assumptions. What advantage can we expect to gain with real software systems, even if we allow our hypothetical method to make use of prior knowledge of the results of mutation analysis? To find out, we examine a large set of real-world programs and their test suites.

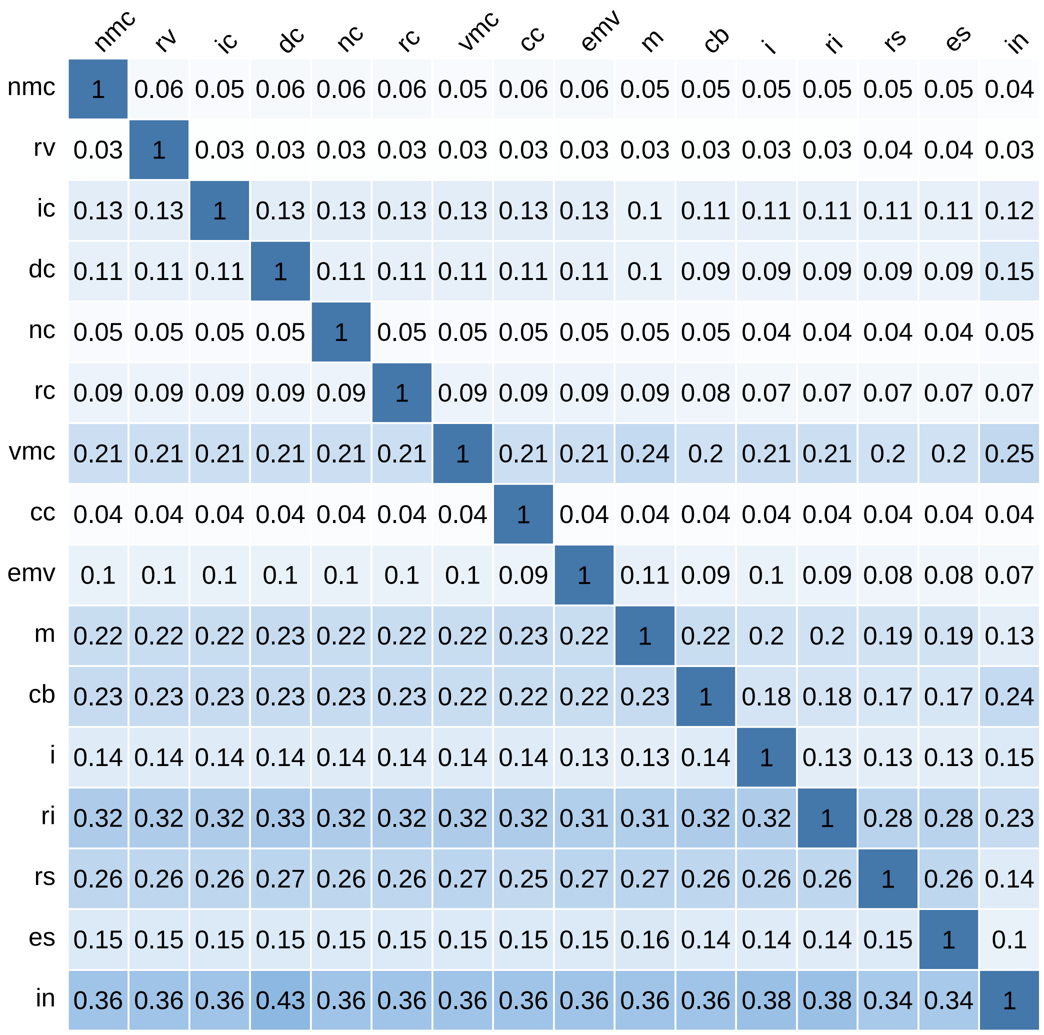
Our selection of sample programs for this empirical study of the limits of mutation reduction was driven by a few overriding concerns. Our primary requirement was that our results should be as representative as possible of real-world programs. Second, we strove for a statistically significant result, therefore reducing the number of variables present in the experiments for reduction of variability due to their presence.

We chose a large random sample of Java projects from Github [41][9] and the Apache Software Foundation [12] that use the popular Maven [13] build system. From an initial 1,800 projects, we eliminated aggregate projects, and projects without test suites, which left us with 796 projects. Out of these, 326 projects compiled (common reasons for failure included unavailable dependencies, compilation errors due to syntax, and bad configurations). Next, projects that did not pass their own test suites were eliminated since the analysis requires a passing test suite. Tests that timed out for particular mutants were assumed to have not detected the mutant. The tests that completely failed to detect any of the mutants were eliminated as well, as these were redundant to our analysis. We also removed all projects with trivial test suites, leaving only those that had at least 100 test cases. This left us with 39 projects. The projects are given in Table 4.2.

We used PIT [24] for our analysis. PIT was extended to provide operators that it was lacking [5] (accepted into mainline). We also ensured that the final operators (Table 4.1) were not redundant. The redundancy matrix for the full operator set is given in Figure 4.2. A mutant *m1* is deemed to subsume another, say *m2* if any tests that kills *m1* is guaranteed to kill *m2*. This is extended to mutation operators whereby the fraction of mutants in *o1* killed by test cases that kills all mutants in *o2* is taken as the degree of subsumption of *o1* by *o2*. The matrix shows that the maximum subsumption was just 43% — that is, none of the operators were redundant. For a detailed description of each mutation operator, please refer to the PIT documentation [25]. To remove the effects of random noise, results for each criteria were averaged over ten runs. The mutation scores along with the sizes of test suites are given in Figure 4.1.

It is of course possible that our results may be biased by the mutants that PIT produces, and it may be argued that the tool we use produces too many redundant mutants, and hence the results may not be applicable to a better tool that reduces the redundancy of mutants. To account for this argument, we run our experiment in two parts, with similar procedures but with different mutants. For the first part, we use the detected mutants from PIT as is, which provides us with an upper bound that a practicing tester can expect to experience, now, using an industry-accepted tool. For the second part, we choose only distinguishable mutants [6] from the original set of detected

[9]Github allows us access only a subset of projects using their search API. We believe this should not confound our results.



	nmc	rv	ic	dc	nc	rc	vmc	cc	emv	m	cb	i	ri	rs	es	in
nmc	1	0.06	0.05	0.06	0.06	0.06	0.05	0.06	0.06	0.05	0.05	0.05	0.05	0.05	0.05	0.04
rv	0.03	1	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.04	0.04	0.03
ic	0.13	0.13	1	0.13	0.13	0.13	0.13	0.13	0.13	0.1	0.11	0.11	0.11	0.11	0.11	0.12
dc	0.11	0.11	0.11	1	0.11	0.11	0.11	0.11	0.11	0.1	0.09	0.09	0.09	0.09	0.09	0.15
nc	0.05	0.05	0.05	0.05	1	0.05	0.05	0.05	0.05	0.05	0.05	0.04	0.04	0.04	0.04	0.05
rc	0.09	0.09	0.09	0.09	0.09	1	0.09	0.09	0.09	0.09	0.08	0.07	0.07	0.07	0.07	0.07
vmc	0.21	0.21	0.21	0.21	0.21	0.21	1	0.21	0.21	0.24	0.2	0.21	0.21	0.2	0.2	0.25
cc	0.04	0.04	0.04	0.04	0.04	0.04	0.04	1	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04
emv	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.09	1	0.11	0.09	0.1	0.09	0.08	0.08	0.07
m	0.22	0.22	0.22	0.23	0.22	0.22	0.22	0.23	0.22	1	0.22	0.2	0.2	0.19	0.19	0.13
cb	0.23	0.23	0.23	0.23	0.23	0.23	0.22	0.22	0.22	0.23	1	0.18	0.18	0.17	0.17	0.24
i	0.14	0.14	0.14	0.14	0.14	0.14	0.14	0.14	0.13	0.13	0.14	1	0.13	0.13	0.13	0.15
ri	0.32	0.32	0.32	0.33	0.32	0.32	0.32	0.32	0.31	0.31	0.32	0.32	1	0.28	0.28	0.23
rs	0.26	0.26	0.26	0.27	0.26	0.26	0.27	0.25	0.27	0.27	0.26	0.26	0.26	1	0.26	0.14
es	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.16	0.14	0.14	0.14	0.15	1	0.1
in	0.36	0.36	0.36	0.43	0.36	0.36	0.36	0.36	0.36	0.36	0.36	0.38	0.38	0.34	0.34	1

Figure 4.2: Subsumption rate between operators. Note that subsumption is not a symmetrical relation. No operators come close to full subsumption. This suggests that **none of the operators studied are redundant**.

Table 4.2: The projects mutants and test suites

Project	$\|M\|$	M_{killed}	M_{killed}^{uniq}	$\|T\|$	$\|T_{min}\|$
events	1171	702	59	180	33.87
annotation-cli	992	589	110	109	38.97
mercurial-plugin	2069	401	102	138	61.77
fongo	1461	1209	175	113	70.73
config-magic	1188	721	204	112	74.55
clazz	5242	1583	151	140	64.00
ognl	21852	12308	2990	114	85.43
java-api-wrapper	1715	1304	308	125	107.04
webbit	3780	1981	325	147	116.93
mgwt	12030	1065	168	101	90.65
csv	1831	1459	411	173	117.97
joda-money	2512	1272	236	173	128.48
mirror	1908	1440	532	301	201.21
jdbi	7754	4362	903	277	175.57
dbutils	2030	961	207	224	141.53
cli	2705	2330	788	365	186.24
commons-math-l10n	6067	2980	219	119	109.02
mp3agic	7344	4003	730	206	146.79
asterisk-java	15530	3206	451	214	196.32
pipes	3216	2176	338	138	120.00
hank	26622	7109	546	171	162.88
java-classmate	2566	2316	551	215	196.57
betwixt	7213	4271	1198	305	206.35
cli2	3759	3145	1066	494	303.86
jopt-simple	1818	1718	589	538	158.37
faunus	9801	4809	553	173	146.11
beanutils2	2071	1281	465	670	181.00
primitives	11553	4125	1365	803	486.71
sandbox-primitives	11553	4125	1365	803	488.56
validator	5967	4070	759	383	264.35
xstream	18030	9163	1960	1010	488.25
commons-codec	9983	8252	1393	605	444.69
beanutils	12017	6823	1570	1143	556.67
configuration	18198	13766	4522	1772	1058.36
collections	24681	8561	2091	2241	938.32
jfreechart	99657	32456	4686	2167	1696.86
commons-lang3	32323	26741	4479	2456	1998.11
commons-math	122484	90681	17424	5881	4009.98
jodatime	32293	23796	6920	3973	2333.49

mutants. What this does is to reduce the number of samples from each stratum to 1, and hence eliminate the skew in mutant population. Note that this requires post-hoc knowledge of mutant kills (not just that the mutants produce different failures, but also that *available* tests in the suite can distinguish between both), and is the best one can do for the given projects to enhance the utility of any strategy against random sampling. We provide results for both the practical and more theoretically interesting distinguishable sets of mutants. Additionally, in case adequacy has an impact, we chose the projects that had plausible mutation-adequate test suites, and computed the possible advantage separately.

4.4.1 Experiment

Our task is to find the $U_{perfect}$ for each project. The requirements for a perfect strategy are simple:

1. The mutants should be representative of the full set. That is,

$$kill(T_p, M) = kill(T, M)$$

2. The mutants thus selected should be non-redundant. That is,

$$\forall_{m \in M_p} kill(T_p, M_p \setminus \{m\}) \subset kill(T_p, M_p)$$

The minimal mutant set suggested by Ammann et al. [6] satisfies our requirements for a perfect strategy, since it is representative — a test suite that can kill the minimal mutants can kill the entire set of mutants — and it is non-redundant with respect to the corresponding minimal test suite.

Ammann et al. [6] observed that the cardinality of a minimal mutant set is the same as the cardinality of the corresponding minimal test suite. That is,

$$|M^{min}_{perfect}| = |MinTest(T, M)| = |T^{min}_{all}|$$

Finding the true minimal test suite for a set of mutants is NP-complete[10]. The best

[10]This is the *Set Covering Problem* [6] which is NP-Complete [58].

possible approximation algorithm is Chvatal's [22], using a greedy algorithm where each iteration tries to choose a set that covers the largest number of mutants. This is given in Algorithm 1. In the worst case, if the number of mutants is n, and the smallest test suite that can cover it is k, this algorithm will achieve a $k \cdot ln(n)$ approximation. We note that this algorithm is robust in practice, and usually gets results close to the actual minimum k (see Figure 4.3). Further, Feige [39] showed that this is the closest approximation ratio that an algorithm can reach for set cover so long as $NP \neq P$[11].

Since it is an approximation, we average the greedily estimated minimal test suite size over 100 runs. The variability is given in Figure 4.3, ordered by the size of minimal test suite. Note that there is very little variability, and the variability decreases as the size of test suite increases. All we need now is to find the effectiveness of random sampling for the same number of mutants as produced by the *perfect* strategy.

Algorithm 1 Finding the minimal test suite

```
function MinTest($Tests, Mutants$)
    $T \leftarrow Tests$
    $M \leftarrow kill(T, Mutants)$
    $T_{min} \leftarrow \emptyset$
    while $T \neq \emptyset \vee M \neq \emptyset$ do
        $t \leftarrow random(\max_t |kill(\{t\}, M)|)$
        $T \leftarrow T \setminus \{t\}$
        $M \leftarrow kill(T, Mutants)$
        $T_{min} \leftarrow T_{min} \cup \{t\}$
    end while
    return $T_{min}$
end function
```

Next, we randomly sample $|M^{min}_{perfect}|$ mutants from the original set M_{random}, obtain the minimal test suite of this sample T^{min}_{random}, and find the mutants from the original set that are killed by this test suite $kill(T^{min}_{random}, M)$, which is used to compute the utility of

[11] We avoided the *reverse greedy algorithm* given by Ammann et al. [6] for two reasons. First, while the approximation ratio of the *greedy* algorithm is at most $k \cdot ln(n)$ where k is the actual minimum, that of *reverse greedy* is much larger [49] (if any). Secondly, the number of steps involved in *reverse greedy* is much larger than in *greedy* when the size of minimal set is very small compared to the full set. We also verified that the *minimum frequency* of kills of the set of mutants by the minimal test suite was 1. A larger *minimum frequency* indicates that at least that many tests are redundant, which is a rare but well-known problem with the *greedy* algorithm.

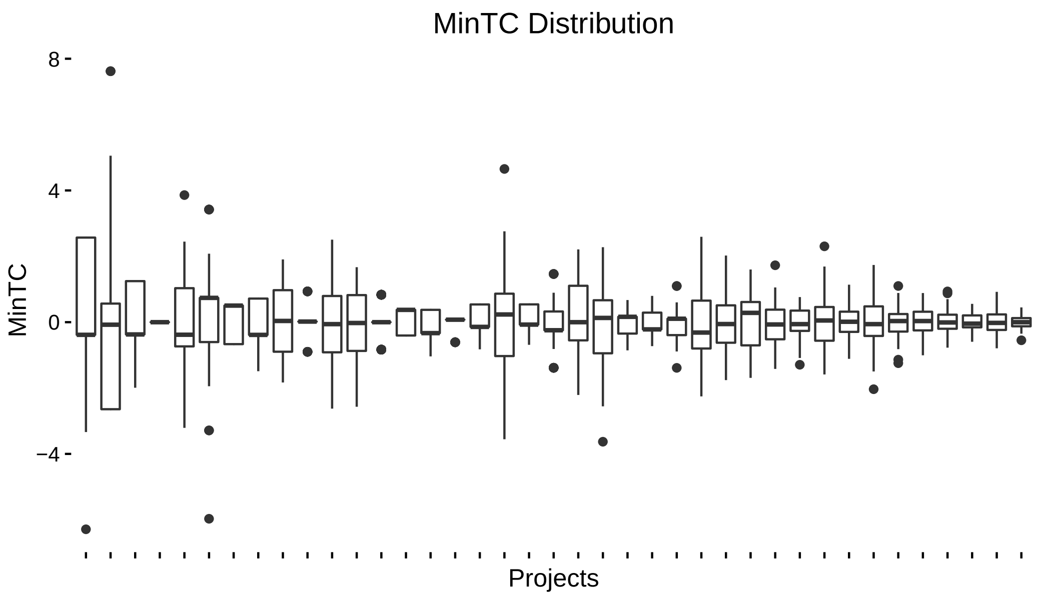


Figure 4.3: Variation of minimal test cases for each sample as a percentage difference from the mean ordered by mean minimal test suite size. There is very little variation, and the variation decreases with test suite size.

perfect strategy with respect to that particular random sample. The experiments were repeated 100 times for each project, and averaged to compute $U_{perfect}$ for the project under consideration.

4.5 Results

4.5.1 All Mutants

Our results are given in Table 4.3. We found that the largest utility achieved by the perfect strategy was 17.997%, for project *faunus*, while the lowest utility was 1.153%, for project *joda-money*. The mean utility of the perfect strategy was 13.078%. A one sample *u-test* suggests that 95% of projects have maximum utility between 12.218% and 14.26% ($p < 0.001$). The distribution of utility for each project is captured in Figure 4.6. Projects are sorted by average minimal test suite size.

One may wonder if the situation improves with either test suite size or project size.

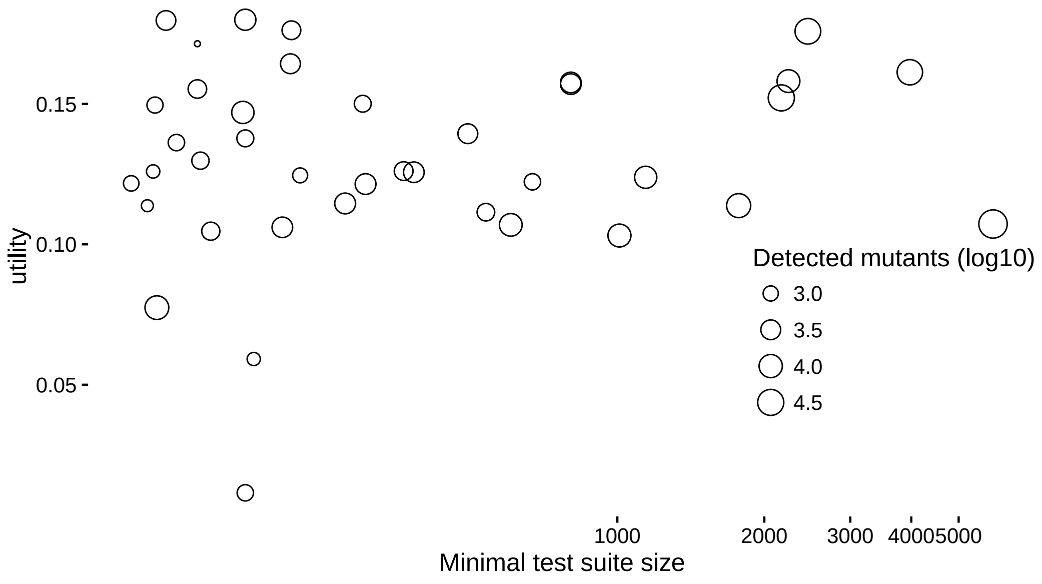


Figure 4.4: The figure plots utility (y-axis) against the average minimal test suite size (log10). Bubble size represents the magnitude of detected mutants (log10). The figure suggests that there is no correlation between utility and average minimal test suite size.

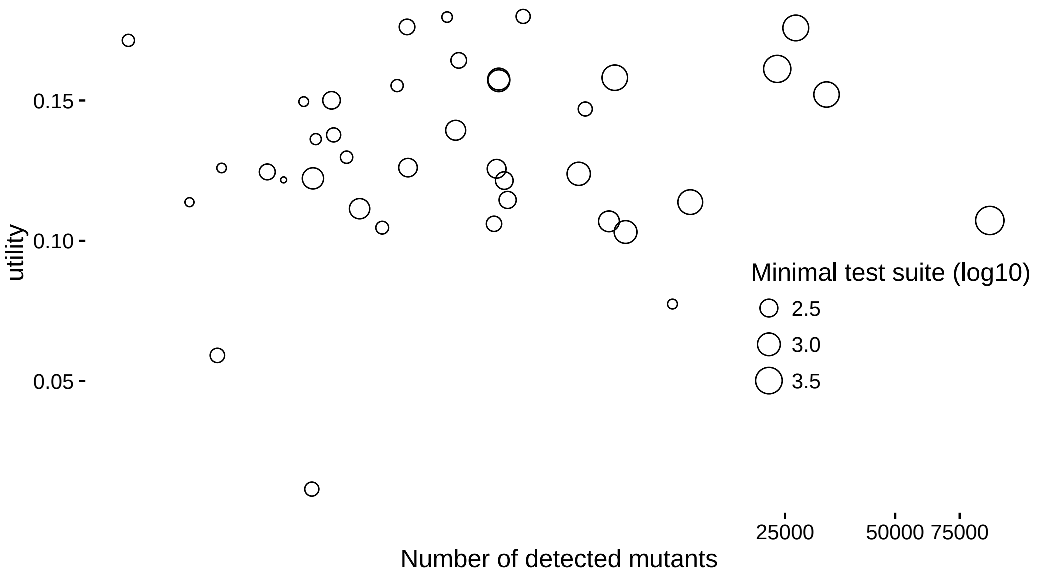


Figure 4.5: The figure plots utility (y-axis) against the number of detected mutants. Bubble size represents the magnitude of average minimal test suite size (log10). The figure suggests that there is no correlation between utility and number of detected mutants.

Table 4.3: The maximum utility achievable by a perfect strategy for each project

Project	$\|kill(T,M)\|$	$\|kill(T_r,M)\|$	U_{perf}
events	702	662.97	0.06
annotation-cli	589	529.51	0.11
mercurial-plugin	401	342.91	0.17
fongo	1209	1052.99	0.15
config-magic	721	640.91	0.13
clazz	1583	1402.39	0.13
ognl	12308	11426.09	0.08
java-api-wrapper	1304	1148.52	0.14
webbit	1981	1793.96	0.10
mgwt	1065	949.96	0.12
csv	1459	1282.93	0.14
joda-money	1272	1257.55	0.01
mirror	1440	1252.50	0.15
jdbi	4362	3914.73	0.11
dbutils	961	854.83	0.12
cli	2330	2069.84	0.13
commons-math-l10n	2980	2527.66	0.18
mp3agic	4003	3620.41	0.11
asterisk-java	3206	2754.69	0.16
pipes	2176	1884.73	0.16
hank	7109	6200.08	0.15
java-classmate	2316	1969.76	0.18
betwixt	4271	3809.19	0.12
cli2	3145	2760.66	0.14
jopt-simple	1718	1546.21	0.11
faunus	4809	4078.22	0.18
beanutils2	1281	1141.73	0.12
primitives	4125	3565.83	0.16
sandbox-primitives	4125	3563.85	0.16
validator	4070	3616.71	0.13
xstream	9163	8307.12	0.10
commons-codec	8252	7455.50	0.11
beanutils	6823	6071.53	0.12
configuration	13766	12359.89	0.11
collections	8561	7392.63	0.16
jfreechart	32456	28171.19	0.15
commons-lang3	26741	22742.46	0.18
commons-math	90681	81898.25	0.11
jodatime	23796	20491.96	0.16

Table 4.4: The maximum utility achievable by a perfect strategy for each project using distinguishable mutants

Project	$\|kill(T,M)\|$	$\|kill(T_r,M)\|$	U_{perf}
events	59	49.15	0.20
annotation-cli	110	93.68	0.18
mercurial-plugin	102	80.95	0.26
fongo	175	145.13	0.21
config-magic	204	171.60	0.19
clazz	151	129.24	0.17
ognl	2990	2835.77	0.05
java-api-wrapper	308	259.87	0.19
webbit	325	280.89	0.16
mgwt	168	140.60	0.20
csv	411	349.30	0.18
joda-money	236	230.76	0.02
mirror	532	444.17	0.20
jdbi	903	783.99	0.15
dbutils	207	170.60	0.21
cli	788	688.05	0.15
commons-math-l10n	219	177.86	0.23
mp3agic	730	639.01	0.14
asterisk-java	451	372.25	0.21
pipes	338	288.41	0.17
hank	546	465.52	0.17
java-classmate	551	450.46	0.22
betwixt	1198	1055.30	0.14
cli2	1066	903.30	0.18
jopt-simple	589	514.36	0.15
faunus	553	467.03	0.18
beanutils2	465	392.30	0.19
primitives	1365	1155.09	0.18
sandbox-primitives	1365	1155.01	0.18
validator	759	647.36	0.17
xstream	1960	1691.84	0.16
commons-codec	1393	1192.29	0.17
beanutils	1570	1341.04	0.17
configuration	4522	3934.21	0.15
collections	2091	1750.05	0.19
jfreechart	4686	3910.15	0.20
commons-lang3	4479	3663.98	0.22
commons-math	17424	15139.90	0.15
jodatime	6920	5801.10	0.19

We note that the utility U_p has low correlation with total mutants, detected mutants (shown in Figure 4.5), mutation score, and minimal test suite size (shown in Figure 4.4). The correlation factors are given in Table 4.5.

An analysis of variance (ANOVA) to determine significant variables affecting $U_{perfect}$

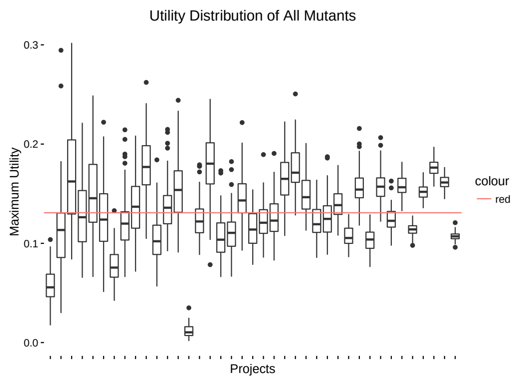


Figure 4.6: Using all mutants.

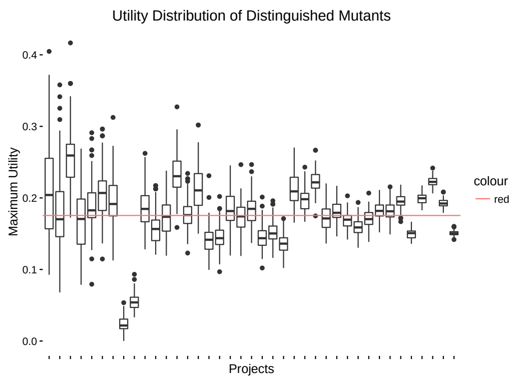


Figure 4.7: Using distinguished mutants.

Distribution of mean utility using distinguished mutants across projects. The projects are ordered by the cardinality of mean minimal test suite. The red line indicates the mean of all observations.

Table 4.5: The correlation of utility for all mutants, killed mutants, mutation score, and minimal test suite size, based on both full set of mutants, and also considering only distinguished mutants

	R^2_{all}	$R^2_{distinguished}$
M	-0.02	-0.03
M_{kill}	-0.03	-0.01
M_{kill}/M	-0.02	-0.00
T^{min}	-0.01	-0.02

suggests that the variability due to project is a significant factor ($p < 0.001$) and interacts with $kill(T_{random}, M)$ strongly.

$$\mu\{U_p\} = project + kill(T_r, M) + project \times kill(T_r, M)$$

The variable *project* has a correlation of 0.682 with the $U_{perfect}$, and the combined terms have a correlation of 0.9995 with $U_{perfect}$.

4.5.2 Distinguishable Mutants

Our results are given in Table 4.4. We found that the largest utility achieved by the perfect strategy was 26.159%, for project *mercurial-plugin*, while the lowest utility was 2.283%, for project *joda-money*.

The mean utility of the perfect strategy was 17.545%. A one sample *u-test* showed that 95% of projects have a maximum utility between 16.912% and 18.876% ($p < 0.001$).

The utility distribution for each project is captured in Figure 4.7. The projects are sorted by the average minimal test suite size.

This situation does not change with either test suite or project size.

The utility U_p has low correlation with total mutants, detected mutants, mutation score, and minimal test suite size. The correlation factors are given in Table 4.5.

An analysis of variance (ANOVA) to determine significant variables affecting $U_{perfect}$ found that the variability due to project is a significant factor ($p < 0.001$) and strongly interacts with $kill(T_{random}, M)$.

$$\mu\{U_p\} = project + kill(T_r, M) + project \times kill(T_r, M)$$

The variable *project* has a correlation of 0.734 with the $U_{perfect}$, and the combined terms have a correlation of 0.9994 with $U_{perfect}$.

4.5.3 Adequate Mutants

Finally, one may ask if adequacy has an impact on the effectiveness of selection strategies. Following the industry practice of deeming well-tested projects adequate after discounting equivalent mutants [87, 100–102], we chose large well tested projects that had at least $10,000$ mutants and a mutation score of at least 70% (in the range of similar studies above) which were deemed adequate. We evaluated the utility for *configuration, commons-lang3, commons-math, jodatime* and found that they have a mean maximum utility of 13.955%. These same projects have a distinguished mean maximum utility of 17.893%. This suggests that adequacy does not have a noticeable impact on the effectiveness of selection strategies.

4.6 Discussion

Mutation analysis is an invaluable tool that is often difficult to use in practice due to hefty computational requirements. There is ongoing and active research to remedy this situation using different mutation reduction strategies. Hence, it is important to understand the amount by which one can hope to improve upon the simplest baseline strategy for reduction — pure random sampling.

Our theoretical analysis of a simple idealized system finds a mean improvement of 58.2% over random sampling for a mutation reduction strategy with oracular knowledge of mutation kills given a uniform distribution of mutants. This serves as an upper bound of what any known mutation reduction strategy could be expected to achieve (under the assumption that the mutant distribution is reasonably close to uniform).

Our empirical analysis using a large number of open source projects reveals that the practical limit is much lower, however, on average only 13.078% for mutants produced by PIT. Even if we discount the effects of skew, by using only distinguished mutants, the potential improvement is restricted to 17.545% on average.

It is important to distinguish the different questions that the theory and empirical analysis tackle. The theoretical limit shows the best that can be done by a perfect mutation strategy given the worst distribution of mutants one may encounter. On the other hand, the empirical analysis finds the average utility of a perfect strategy without regard to the distribution of mutants in different programs. However, given that the effects of skew were found to be rather weak (only 4.467%) the theoretical bound is reasonable for the empirical question too.

The empirical upper bounds on gain in utility are surprisingly low, and call into question the effort invested into improving mutation reduction strategies. Of course, one can still point out that random sampling is subject to the vagaries of chance, as one can get arbitrarily good or bad samples. However, our results suggest that the variance of individual samples is rather low, and the situation improves quite a bit with larger projects — e.g. the variance of *commons-math* is just 0.397%. Hence the chances for really bad samples are very low in the case of projects large enough to really need mutant reduction, and drop quickly as the number of test cases increases. One may wonder if the adequacy of test suites has an impact, but our analysis of projects with adequate test suites suggests that there is very little difference due to adequacy ($U_{perfect}$ =13.955%). In general, using accepted standard practices for statistical sampling to produce reasonably-sized random mutant samples should be practically effective for avoiding unusually bad results due to random chance. The added advantage is that random sampling is easy to implement and incurs negligible overhead.

We note that our framework is applicable not only to selective mutation, but also to mutation implementors looking to add new mutators. Say a mutation implementor has a perfect set of mutation operators such that their current set of mutants does not have any redundant mutants (practically infeasible given our shallow understanding of mutant semiotics). Even if we consider the addition of a new set of random mutants that *do not* improve the mutation set at all, in that they are redundant with respect to the original set (rare in practice, given that we are introducing new mutants), the maximum disadvantage thus caused is bounded by our limit (18.876% upper limit for 95% of projects). However, at least a few of the new mutants can be expected to improve the representativeness of a mutation set compared to the possible faults. Since we can't bound the number of distinguishable mutants that may be introduced, there is no upper bound for the maximum advantage gained by adding new mutation operators. Adding new operators is especially attractive in light of recent results showing classes of real faults that are not coupled to any of the operators currently in common use [56].

Our previous research [45] suggests that a constant number of mutants (a theoretical maximum of $9,604$, and 1,000 in practice for 1% accuracy) is sufficient for computing mutation score with high accuracy irrespective of the total number of mutants. This suggests that sampling will lead to neither loss of effectiveness nor loss of accuracy, and hence addition of new mutation operators (and sampling the required number of mutants) is potentially a very fruitful endeavour.

4.7 Threats to Validity

While we have taken care to ensure that our results are unbiased, and have tried to eliminate the effects of random noise. Random noise can result from non-representative choise of project, tool, or language, and can lead to skewed strata and bias in empirical result. Our results are subject to the following threats.

Threats due to approximation: We use the greedy algorithm due to Chvatal [22] for approximating the minimum test suite size. While this is guaranteed to be $H(|M|)$

approximate, there is still some scope for error. We guard against this error by taking the average of 100 runs for each observation. Secondly, we used random samples to evaluate the effectiveness of random sampling. While we have used 100 trials each for each observation, the possibility of bias does exist.

Threats due to sampling bias: To ensure representativeness of our samples, we opted to use search results from the Github repository of Java projects that use the `Maven` build system. We picked *all* projects that we could retrieve given the Github API, and selected from these only based on constraints of building and testing. However, our sample of programs could be biased by skew in the projects returned by Github.

Bias due to tool used: For our study, we relied on PIT. We have done our best to extend PIT to provide a reasonably sufficient set of mutation operators, ensuring also that the mutation operators are non-redundant. Further, we have tried to minimize the impact of redundancy by considering the effect of distinguished mutants. There is still a possibility that the kind of mutants produced may be skewed, which may impact our analysis. Hence, this study needs to be repeated with mutants from diverse tools and projects in future.

4.8 Conclusion

Our research suggests that blind random sampling of mutants is highly effective compared to the best achievable bound for mutation reduction strategies, using perfect knowledge of mutation analysis results, and there is surprisingly little room for improvement. Previous researchers showed that there is very little advantage to *current* operator selection strategies compared to random sampling [101, 102]. However, the experiment lacked direct comparison with random sampling of the *same* number of mutants. Secondly it was also shown that *current* strategies fare poorly [6] when compared to the actual minimum mutant set, but lacked comparison to random sampling. Our contribution is to show that there is a theoretical limit to the improvement that any reduction strategy can have *irrespective* of the intelligence of the strategy, and also a *direct* empirical comparison of effectiveness of the best strategy possible with random sampling.

Our theoretical investigation suggests a mean advantage of 58.2% for a perfect mutation reduction strategy with oracular knowledge of kills over random sampling given an arbitrary program, under the assumption of no skew in redundant mutants. Empirically, we find a much lower advantage 13.078% for a perfect reduction strategy with oracular knowledge. Even if we eliminate the effects of skew in redundant mutant population by considering only distinguished mutants, we find that the advantage of a perfect mutation reduction strategy is only 17.545% in comparison to random sampling. The low impact of skew (4.467%) suggests that our simplifying assumptions for theoretical analysis were not very far off the mark. The disparity between the theoretical prediction and empirical results is due to the inadequacies of real world test suites, resulting in a much smaller minimum mutant set than the distinguishable mutant set. We note that mutation reduction strategies routinely claim high reduction factors, and one might expect a similar magnitude of utility over random sampling, which fails to materialize either in theory or practice.

The second takeaway from our research is that a researcher or an implementor of mutation testing tools should consider the value of implementing a mutation reduction strategy carefully given the limited utility we observe. In fact, our research [44] suggests that popular operator selection strategies we examined have reduced utility compared to random sampling, and even strata sampling techniques based on program elements seldom exceed a 10% improvement. Given that the variability due to projects is significant, a testing practitioner would also do well to consider whether the mutation reduction strategy being used is suited for the particular system under test (perhaps based on historical data for that project, or projects that are in some established sense similar). Random sampling of mutants is not extremely far from an empirical upper bound on an ideal mutation reduction strategy, and has the advantage of having little room for an unanticipated bias due to a "clever" selection method that might not work well for a given project. The limit reported here is based on using full knowledge of the mutation kill matrix, which is, to say the least, difficult to attain in practice.

Perhaps the most important takeaway from our research is that it is possible to improve the effectiveness of mutation analysis, not by removing mutation operators, but rather by further research into newer mutation operators (or new categories of mutation operators such as domain specific operators for concurrency or resource allocation). Our research suggests that there is little reduction in utility due to addition of newer operators, while there is no limit to the achievable improvement.

Chapter 5: Conclusion

The central thesis of this dissertation is in identifying limits to the effectiveness of mutation analysis. The effectiveness of mutation analysis depends on its ability to emulate the first order faults – for which it depends on the *competent programmer hypothesis*, and subsume the higher order faults – for which it depends on the *coupling effect*. Hence, a focus of this dissertation is to identify limits of mutation analysis with respect to the faults that it tries to emulate and subsume. In Chapter 2 and Chapter 3, we investigated the limits of mutation analysis with regard to its foundational hypothesis — the *competent programmer hypothesis* and the *coupling effect*. The effectiveness of mutation analysis also depends on generating mutants that are as non-redundant as possible. Hence, another major focus of the dissertation is to identify the limits of removal of redundancy possible by mutation selection. In Chapter 4, we investigated the limits of improvement one can achieve by removing redundant mutants. We summarize the three major findings in the coming sections.

5.1 Real faults are different from typical mutation operators

The *competent programmer hypothesis* or the finite neighborhood hypothesis suggests that any program is a finite distance away from the correct version. The traditional mutation analysis theory and practice assume that the distance from the correct version is at most one single fault [17,28,77]. We investigated the size of real world faults in Chapter 2 using faults from $5,000$ different programs in four different programming languages and found that while there exist a large amount of single token changes, the majority of them involves changes of more than a single token. Since the traditional mutation operators are overwhelmingly first order, our results suggest that the traditional mutation operators do not adequately represent the possible real-world faults. Our investigations in Chapter 2 further suggest that the incidence of faults in real world systems are dependent on the language being used. In fact, there were interesting affinities attributable to language paradigms in the distribution of faults, and further, mutation operators that are adequate for emulating faults in one language may not be adequate for another language. We found that the implicit assumption that each possible mutation point or each mutation possible have an equal probability of occurrence does not reflect the real world. In fact, we found that the largest category of faults does not match any of the traditional mutation operators.

Our results from Chapter 3 also suggest that language features such as naming and variables can produce complex faults that are not decomposable to first order faults. Hence, our results indicate that traditional mutation analysis is limited in terms of the representativeness of faults it generates. Indeed, for any language that supports at least naming, one can always find faults that are larger than a fixed limit. This can be generalized to the *theory of incompleteness of program neighborhoods.* That is, given a language with facilities such as naming or recursion, no fixed size program neighborhood will contain the complete set of atomic faults for all programs. Further, it also suggests that one needs to consider other factors such as the language used, in order to get a realistic sense of the fault-proneness of the program and the actual strength of the test suite.

5.2 A theory of fault interaction

The *coupling effect* asserts that a test suite capable of finding all simple faults can detect a high percentage of complex faults. Unfortunately, the *coupling effect* does not clarify what is a simple fault, and what may be counted as a complex fault. It provides no help in estimating the number of complex faults detected by a mutation adequate test suite that detects all first order mutants. Our investigations in Chapter 3 identified the limitations of the *coupling effect*. We showed that strong fault interactions can lead to complex but non-decomposable faults and should be considered as atomic. We also provide a stronger and precise formulation for the *coupling effect*, called the *composite fault hypothesis.* Our investigations suggest that theoretically, the probability of coupling is as high as $\frac{n-1}{n}$ where n is the *co-domain* of the function being examined. Our theory suggests that, when the effects of premature exits from loops or recursion are taken into account, the coupling ratio can increase. This is borne out by empirical evidence. Empirically, it was found that in the programs we examined, approximately 99% of the higher order faults are indeed coupled to simpler faults.

5.3 Mutation reduction strategies considered harmful

The largest impediment to wider use of mutation analysis, and indeed in interpreting its results is the problem of redundancy in mutants generated. Numerous researchers have tried to tackle this problem by suggesting various heuristics for identifying representative mutants — commonly called the selective mutation strategies. While these strategies achieve significant amounts of reduction, there are few studies that evaluate these strategies with regard to the accuracy of representation of unique faults. We investigated the limits of these heuristic reduction strategies with regard to the accuracy of representation in Chapter 4. We found that there is a theoretical as well as an empirical limit to the improvement possible by any given strategy when compared to the baseline – random sampling. Considering a simple system with an equal number of redundant mutants per unique fault, and no overlap between mutants in terms of tests, the maximum improvement is limited to 58.2%. Empirically, we find a maximum limit of 15% when comparing the representativeness of the minimal set of mutants with that of a random sample of the same size. Our research suggests that researchers can gain better results by focusing on increasing variation within mutation operators and using sampling to reduce the number of mutants rather than using selective mutation strategies.

5.4 Limitations and Future Research

We have seen how first order mutation analysis is inadequate to emulate and subsume all possible faults. One of the ways forward is to consider larger neighborhoods, with δ nearer to real world magnitudes (10 tokens for up to 95% coverage). However, this is likely to lead to an exponential increase in the number of mutants. One avenue to control this explosion is to provide limited context sensitivity to mutation operators or provide domain specific higher order mutation operators that better represent common errors that programmers make. Indeed, as our research in Chapter 4 suggests, it is better to increase the number of mutation operators and sample to reduce than to reduce the number of mutation operators altogether. On the theoretical side, while we have shown by case analysis that some of the general patterns of recursion and iteration does not lead to an increase in fault masking, the evaluation is not exhaustive. A complete theory of fault interaction is hence, an important avenue for future research.

The second area of future research is in finding better limits of mutation reduction. We have provided in Chapter 4 the framework for evaluating the limits of mutation reduction strategies given a mutant distribution, and have shown how a simple system may be evaluated. We would like to improve this framework by using a mutant distribution that is similar to the real world, and reducing assumptions such as no overlap between mutants. Similarly, we note that our theoretical framework in Chapter 3 for evaluating fault interaction also depends on a few simplifying assumptions such as an assumed distribution of functions, and incomplete case analysis. This is another area that needs to be further investigated.

An area of research that we have not touched upon, but still extremely important for the effectiveness of mutation analysis is the question of equivalent mutants. It is well-known that a general solution is impossible [17], and a number of heuristic methods such as coverage and state changes have been proposed. However, a comparative study on the merits of these techniques in large codebases have yet to happen and is one area of future research. We would like to examine if a pragmatic solution to the equivalent mutant problem could be the statistical sampling of the input domain, especially making use of techniques such as combinatorial testing of multiple mutants, incremental execution, and split stream execution.

In summary, we find that there are certain well-defined limits to the kinds of simple faults that mutants can emulate, and even when faults can be emulated, the ratio of faults that can be subsumed. We also find empirical as well as theoretical limits on the accuracy of representation for any given heuristic reduction strategy. Finally, we propose a few avenues of further research.

Bibliography

[1] Allen Troy Acree. *On Mutation.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1980.

[2] Hiralal Agrawal, Richard A DeMillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P Mathur, and Eugene Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN,, March 1989.

[3] Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. Can testedness be effectively measured. In *ACM SIGSOFT Symposium on The Foundations of Software Engineering.* ACM, 2016.

[4] Zuhoor A. Al-Khanjari and Martin R. Woodward. Investigating the partial relationships between testability and the dynamic range-to-domain ratio. *Australasian Journal of Information Systems*, 11, 2003.

[5] Paul Ammann. Transforming mutation testing from the technology of the future into the technology of the present. In *International Conference on Software Testing, Verification and Validation Workshops.* IEEE, 2015.

[6] Paul Ammann, Marcio Eduardo Delamaro, and A Jefferson Offutt. Establishing theoretical minimal sets of mutants. In *International Conference on Software Testing, Verification and Validation*, pages 21–30, Washington, DC, USA, 2014. IEEE Computer Society.

[7] Paul Ammann and A Jefferson Offutt. *Introduction to software testing.* Cambridge University Press, 2008.

[8] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411. IEEE, 2005.

[9] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[10] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M. Hierons, and Mark Harman. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *International Conference on Software Engineering*, pages 573–583, New York, NY, USA, 2014. ACM.

[11] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 23:304–23:318, New York, NY, USA, 2008. ACM.

[12] Apache Software Foundation. Apache commons. http://commons.apache.org/.

[13] Apache Software Foundation. Apache maven project. http://maven.apache.org.

[14] Ellen Francine Barbosa, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the determination of sufficient mutant operators for c. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001.

[15] Alfredo Benso and Paolo Prinetto. *Fault injection techniques and tools for embedded systems reliability evaluation*, volume 23. Springer, 2003.

[16] Timothy Alan Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, USA, 1980.

[17] Timothy Alan Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233. ACM, 1980.

[18] Timothy Alan Budd, Richard J Lipton, Richard A DeMillo, and Frederick G Sayward. *Mutation analysis*. Yale University, Department of Computer Science, 1979.

[19] Ram Chillarege. Orthogonal defect classification. *Handbook of Software Reliability Engineering*, pages 359–399, 1996.

[20] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *Annual Symposium on Fault Tolerant Computing, 1996.*, pages 304–313, 1996.

[21] J. Christmansson and P. Santhanam. Error injection aimed at fault removal in fault tolerance mechanisms-criteria for error selection using field data on software faults. In *International Symposium on Software Reliability Engineering*, pages 175–184, 1996.

[22] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[23] David Clark and Robert M Hierons. Squeeziness: An information theoretic measure for avoiding fault masking. *Information Processing Letters*, 112(8):335–340, 2012.

[24] Henry Coles. Pit mutation testing. http://pitest.org/.

[25] Henry Coles. Pit mutation testing: Mutators. http://pitest.org/quickstart/mutators.

[26] CRM114. Crm114 classifier. http://crm114.sourceforge.net.

[27] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 158–171. ACM, 1996.

[28] Richard A DeMillo. Completely validated software: Test adequacy and program mutation (panel session). In *International Conference on Software Engineering*, pages 355–356, New York, NY, USA, 1989. ACM.

[29] Richard A DeMillo, Dana S Guindi, WM McCracken, A Jefferson Offutt, and KN King. An extended overview of the mothra software testing environment. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 142–151. IEEE, 1988.

[30] Richard A. DeMillo and Aditya P Mathur. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. Technical Report SERC-TR92-P, Software Engineering Research Center, Purdue University, West Lafayette, IN,, 1991.

[31] Richard A DeMillo and Richard J LiptonFrederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[32] Lin Deng, A Jefferson Offutt, and Nan Li. Empirical evaluation of the statement deletion mutation operator. In *IEEE 6th ICST*, Luxembourg, 2013.

[33] Anna Derezińska. A quality estimation of mutation clustering in c# programs. In *New Results in Dependability and Computer Systems*, pages 119–129. Springer, 2013.

[34] Anna Derezińska. Toward generalization of mutant clustering results in mutation testing. In *Soft Computing in Computer and Information Science*, pages 395–407. Springer, 2015.

[35] F. J. O. Dias. Fault masking in combinational logic circuits. *IEEE Trans. Comput.*, 24(5):476–482, May 1975.

[36] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32(9):733–752, 2006.

[37] J. Duraes and H. Madeira. Definition of software fault emulation operators: a field data study. In *International Conference on Dependable Systems and Networks*, pages 105–114, 2003.

[38] Joao A Duraes and Henrique S Madeira. Emulation of software faults: a field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

[39] Uriel Feige. A threshold of ln n for approximating set cover. *J. ACM*, pages 634–652, 1998.

[40] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[41] GitHub Inc. Software repository. http://www.github.com.

[42] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2013.

[43] Rahul Gopinath. Mutagen census. http://dx.doi.org/10.7910/DVN/24329, 2014.

[44] Rahul Gopinath, Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability*, 67(0), 2017. accepted for publication.

[45] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. How hard does mutation analysis have to be, anyway? In *International Symposium on Software Reliability Engineering*. IEEE, 2015.

[46] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *International Symposium on Software Reliability Engineering*, pages 189–200, Nov 2014.

[47] Ralph Guderlei, René Just, Christoph Schneckenburger, and Franz Schweiggert. Benchmarking testing strategies with tools from mutation analysis. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 360–364. IEEE, 2008.

[48] Mark Harman, Yue Jia, and William B Langdon. A manifesto for higher order mutation testing. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 80–89. IEEE, 2010.

[49] Anonymous (http://cstheory.stackexchange.com/users/37275/anonymous). What is the reverse of greedy algorithm for setcover? Theoretical Computer Science Stack Exchange. url:http://cstheory.stackexchange.com/q/33685 (version: 2016-01-29).

[50] Shamaila Hussain. Mutation clustering. Master's thesis, Kings College London, Strand, London, 2008.

[51] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *International Conference on Software Engineering*, pages 191–200. IEEE Computer Society Press, 1994.

[52] Changbin Ji, Zhenyu Chen, Baowen Xu, and Zhihong Zhao. A novel method of mutation clustering based on domain analysis. In *SEKE*, pages 422–425, 2009.

[53] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258. IEEE, 2008.

[54] Yue Jia and Mark Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51(10):1379–1393, October 2009.

[55] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[56] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 654–665, Hong Kong, China, 2014. ACM.

[57] Kalpesh Kapoor. Formal analysis of coupling hypothesis for logical faults. *Innovations in Systems and Software Engineering*, 2(2):80–87, 2006.

[58] Richard M Karp. *Reducibility among combinatorial problems.* Springer, 1972.

[59] Bob Kurtz, Paul Ammann, and A Jefferson Offutt. Static analysis of mutant subsumption. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 1–10, April 2015.

[60] Robert Kurtz, Paul Ammann, Marcio Eduardo Delamaro, A Jefferson Offutt, and Lin Deng. Mutant subsumption graphs. In *Workshop on Mutation Analysis*, 2014.

[61] Nan Li, Upsorn Praphamontripong, and A Jefferson Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 220–229. IEEE, 2009.

[62] Birgitta Lindstrm and Andrs Mrki. On strong mutation and subsuming mutants. ieee workshop on mutation analysis (mutation 2016). In *Workshop on Mutation Analysis*, 2016.

[63] Richard J Lipton. Fault diagnosis of computer programs. Technical report, Carnegie Mellon University, 1971.

[64] Richard J Lipton and Fred G Sayward. The status of research on program mutation. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 355–373, December 1978.

[65] Yu-Seung Ma and Sang-Woon Kim. Mutation testing cost reduction by clustering overlapped mutants. *Journal of Systems and Software*, 115:18–30, 2016.

[66] Yu-Seung Ma, Yong-Rae Kwon, and A Jefferson Offutt. Inter-class mutation operators for java. In *International Symposium on Software Reliability Engineering*, pages 352–363. IEEE, 2002.

[67] Aditya P Mathur. Performance, effectiveness, and reliability issues in software testing. In *Annual International Computer Software and Applications Conference, COMPSAC*, pages 604–605, 1991.

[68] Aditya P Mathur. *Foundations of Software Testing.* Addison-Wesley, 2012.

[69] Aditya P Mathur and Weichen Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[70] A. Mockus and L.G. Votta. Identifying reasons for software changes using historic databases. In *IEEE International Conference on Software Maintenance*, pages 120–130, 2000.

[71] Larry J Morell. A model for code-based testing schemes. In *Fifth Annual Pacific Northwest Software Quality Conf*, page 309, 1987.

[72] Larry Joe Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

[73] Elfurjani S Mresa and Leonardo Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability*, 9(4):205–232, 1999.

[74] Akbar Siami Namin and James H Andrews. Finding sufficient mutation operators via variable reduction. In *Workshop on Mutation Analysis*, page 5, 2006.

[75] Akbar Siami Namin and Sahitya Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 342–352, New York, NY, USA, 2011. ACM.

[76] A Jefferson Offutt. The Coupling Effect : Fact or Fiction? *ACM SIGSOFT Software Engineering Notes*, 14(8):131–140, November 1989.

[77] A Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, 1992.

[78] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[79] A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. An experimental evaluation of selective mutation. In *International Conference on Software Engineering*, pages 100–107. IEEE Computer Society Press, 1993.

[80] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[81] A Jefferson Offutt and Jeffrey M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, Technical Report ISSE-TR-96-01, Information and Software Systems Engineering, George Mason University, 1996.

[82] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.

[83] Matthew Patrick, Rob Alexander, Manuel Oriol, and John A Clark. Probability-based semantic interpretation of mutants. In *Workshop on Mutation Analysis*, 2014.

[84] Ranjith Purushothaman and Dewayne E Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[85] David Schuler and Andreas Zeller. Javalanche: Efficient mutation testing for java. In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 297–298, August 2009.

[86] Purdue University. Software Engineering Research Center (SERC)., RA DeMillo, and AP Mathur. *On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software*. Software Engineering Research Center, Purdue University, 1991.

[87] Akbar Siami Namin, James H Andrews, and Duncan J Murdoch. Sufficient mutation operators for measuring test effectiveness. In *International Conference on Software Engineering*, pages 351–360. ACM, 2008.

[88] Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In *International Conference on Software Engineering*, pages 9–19, Piscataway, NJ, USA, 2015. IEEE Press.

[89] Joanna Strug and Barbara Strug. Machine learning approach in mutation testing. In *Testing Software and Systems*, pages 200–214. Springer, 2012.

[90] Roland H Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *Annual Southeast Regional Conference*, ACM-SE 47, pages 71:1–71:4, New York, NY, USA, 2009. ACM.

[91] Jeffrey M. Voas and Keith W. Miller. Semantic metrics for software testability. *The Journal of Systems and Software*, 20(3):207 – 216, 1993.

[92] K S How Tai Wah. Fault coupling in finite bijective functions. *Software Testing, Verification and Reliability*, 5(1):3–47, 1995.

[93] K S How Tai Wah. A theoretical study of fault coupling. *Software Testing, Verification and Reliability*, 10(1):3–45, 2000.

[94] K. S. How Tai Wah. Theoretical insights into the coupling effect. In Weichen Eric Wong, editor, *Mutation testing for the new century*, pages 62–70. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[95] K S How Tai Wah. An analysis of the coupling effect i: single test data. *Science of Computer Programming*, 48(2):119–161, 2003.

[96] K S How Tai Wah. An analysis of the coupling effect i: single test data. *Science of Computer Programming*, 48(2):119–161, 2003.

[97] Weichen Eric Wong. *On Mutation and Data Flow.* PhD thesis, Purdue University, West Lafayette, IN, USA, 1993. UMI Order No. GAX94-20921.

[98] Weichen Eric Wong and Aditya P Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185 – 196, 1995.

[99] Martin R Woodward and Zuhoor A Al-Khanjari. Testability, fault size and the domain-to-range ratio: An eternal triangle. *ACM SIGSOFT Software Engineering Notes*, 25(5):168–172, 2000.

[100] Jie Zhang, Muyao Zhu, Dan Hao, and Lu Zhang. An empirical study on the scalability of selective mutation testing. In *International Symposium on Software Reliability Engineering.* ACM, 2014.

[101] Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. Operator-based and random mutant selection: Better together. In *IEEE/ACM Automated Software Engineering.* ACM, 2013.

[102] Lu Zhang, Shan-Shan Hou, Jun-Jue Hu, Tao Xie, and Hong Mei. Is operator-based mutant selection superior to random mutant selection? In *International Conference on Software Engineering*, pages 435–444, New York, NY, USA, 2010. ACM.

[103] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computer Survey*, 29(4):366–427, December 1997.