Finding Fault(s) with Mutation Analysis

Rahul Gopinath,
gopinath@eecs.oregonstate.edu
EECS Department, Oregon State Univeristy

***Abstract*—Mutation analysis is the gold standard for test-suite adequacy. It subsumes other structural coverage criteria, approximates fault detection capability of test-suites, and the faults produced by mutation have been shown to behave similar to real faults. It is also used in other domains that require simulation of bug patterns such as Reliability Engineering and Security Testing.**

However, mutation analysis is resource intensive, requiring multiple test runs to complete, and its foundational assumption — *The Competent Programmer Hypothesis* — is taken on faith. The relationship between mutation analysis and the test-suite adequacy also needs more clarity.

Our research proposes to investigate these fundamental issues in mutation analysis by running large scale studies of real-world bug fixes. We propose to identify the characteristics of bug patterns in real-world, and in multiple programming languages, to validate the competent programmer hypothesis.

We will also investigate the relationship between mutation score and test-suite adequacy, and also quantify the contributions of different mutation operators.

Finally, we aim to find cheap alternatives to mutation analysis that can be used in practice without significant loss of accuracy.

The results from our research will improve the efficacy of mutation analysis while reducing its computational footprint, paving way for its large-scale adoption by the testing world.

***Keywords*—software testing, mutation analysis, test-suite adequacy**

I. Introduction

The ideal measure of the efficacy of a test-suite is its fault detection adequacy. However, it is often hard to measure even in a research setting [1]. The next best is often considered to be mutation analysis [2], which introduces simple syntactic changes to the program and measures the ability of test-suite to distinguish the semantic difference caused by these changes. Mutation analysis subsumes condition coverage criteria [3], [4], detects more bugs [5], produces mutations that have some characteristics of real-world bugs [6], [7], and tracks the fault detection capability of test-suites. These factors make mutation analysis an attractive tool for researchers investigating test-suite quality.

Mutation analysis relies on two assumptions for its effectiveness. *The competent programmer hypothesis* [2] asserts that a *non-pathological* [8] program written by a *competent programmer* is syntactically very close to the correct version. According to *the coupling effect*, when separation between correct version and the produced program is composed of multiple simple faults (a fault tuple), a test case capable of detecting any one of the component faults will be capable of detecting the tuple. That is, the semantic difference of a fault tuple will be greater than the semantic difference of its component faults.

While mutation analysis is attractive as an accurate test-suite adequacy measure, it has a heavy computational footprint [8], requiring as many test-suite executions as there are mutants. This makes mutation analysis unsuitable for large code bases, or where the developers require a fast turnaround, such as in prototyping and initial development where a large amount of unit tests get written.

Another concern is its reliance on the assumption of *competent programmer*, which is yet to be verified[1]. *The competent programmer hypothesis* is relied upon by researchers to assert that the mutants produced are plausible real faults. If they aren't similar enough to real faults in all respects, their applicability to other domains may be limited. Production of mutants dissimilar to real faults is also inefficient if our intention is to measure the fault detection ability, and these spurious mutants can adversely affect the quality of prediction from mutation analysis.

Finally, large-scale non biased studies quantifying the relationship between mutation score and the fault detection capability of test-suites are lacking. This is a serious concern if mutation analysis is to be relied upon as a means of evaluating the quality of a test-suite.

Our overarching aim is to provide developers with an efficient, fast, and accurate criteria to measure the effectiveness of their test-suites. To this end, we propose to investigate how mutation analysis and other criteria relate to real faults. Since mutation analysis is resource-intensive even under a curtailed number of mutants, we intend to investigate alternatives to mutation analysis that can be used with minimum resources for fast turnaround. Our aim is to rectify the identified shortcomings of mutation analysis, to improve mutation analysis — both by improving its accuracy of prediction, and by reduction of its computational requirements — thus increasing its applicability.

We propose to validate and quantify *the competent programmer hypothesis*, especially considering effect of conditions such as different languages, and paradigms such as functional programming, thereby strengthening its theoretical underpinnings, making it amenable to further improvements driven by theory.

Our research tackles the following open questions.

- Q1 *Validation* of mutation analysis assumptions: Exploring latent factors in limits of competent programmer hypothesis. Does its validity depend on the kind of language used to program?
- Q2 *Verification* of mutation analysis claims: Investigate the accuracy of mutation analysis and relative contribution

[1]We are talking about patterns of change here. Other relationships between faults and mutants have been subject to other studies [9]

of mutagens with respect to detecting real faults in real programs, using large-scale studies, and identify ways to improve the accuracy of prediction and reduction of computational requirements.

Q3 Improving the *variety* of techniques: Investigate alternatives to mutation analysis that can predict fault detection capability without significant loss in accuracy.

We also aim to foster a robust research environment for testers and researchers to replicate and evaluate proposed modifications and alternatives to mutation analysis and other test-suite adequacy measures. To this end, we share our data [10] and environment [11], [12] with other researchers under an open license.

A. Terminology

The rest of this document uses these definitions. A *mutation* is made when a rule is applied to the original program to produce a variant. The rule being applied is called a *mutagen* and the variant produced is called a *mutant*. The term *mutagen* is synonymous with both *mutation-operator* and *mutation-transformer* in literature. When only a single mutation separates the mutant from the original program, it is called a *first order mutant* or *FOM*. A *higher order mutant HOM* is separated from the original by multiple mutations.

A *fault* is a concrete structural defect in the program source [13] (a syntactic difference in the program text). An *error* is a deviation from the required behavior [13], [14] (a semantic difference from the correct behavior) [2]. The program along with its variants is collectively known as the *program neighborhood*. A *simple fault* is assumed to be a fault that can be fixed by making a syntactically valid *atomic change*[3] to the source of the program [16].

II. Related Work

This research synthesizes prior work in three broad areas. The theoretical validity of mutation analysis, research on empirical verification and improvement of mutation analysis with regard to its predictive power, and finding cheaper alternatives to mutation analysis.

Research in mutation analysis has been surveyed multiple times in the past. We rely on the comprehensive review by Jia et al. [17] which summarizes the research until 2010, and that of Offutt et al. [18] for categorization of mutagen reduction efforts.

A. Evolution of Mutation Analysis

The idea of mutation analysis was first espoused by Richard Lipton in a term paper called “Fault Diagnosis of Computer Programs” in 1971 [19]. Its principles were flushed out by DeMillo et al. [2] and the theoretical edifice was further consolidated by Budd et al. [8]. The principles of *competent programmer hypothesis* and *coupling effect* were first laid out in these papers.

Mutation analysis was found to subsume condition coverage measures [20], [3], [4], [21]. However, this result was not without its share of controversy. Frankl et al. [22] showed that mutation coverage — using mutagens limited to branch coverage specific mutagens — did not subsume decision coverage, and that according to them, most mutagens do not do a good job of branch testing. This was contested by DeMillo et al. [23], who responded with empirical results showing mutation analysis does work well on branch testing, along with theoretical support for this result. The takeaway seems to be that the results of mutation testing are heavily dependent on the language and mutagens used.

An early influential survey from DeMillo [24] collected the initial mutation studies. He also provided a comprehensive list of mutagens. The justification for the mutagens was based on an early study by Youngs [25] which synthesized error data from 42 programmers (30 novices and 12 professional programmers), on coding 69 elementary programming exercises. We note that the syntactical errors (caught at compile time) were a main part of the study. While the study was intended to be language independent, the authors stress that further data is needed before the findings could be evaluated. This study (which is cited as evidence for *competent programmer hypothesis* [24]) does not seem to provide evidence that *competent programmers* (the 12 professionals here) commit simple mistakes[4].

The concept of *competent programmer hypothesis* can be understood from practical and theoretical perspectives. From a purely practical aspect, it is taken to mean that the kind of mistakes programmers make that are non-pathological [8] tend to be simple. In fact, the initial mutagens were justified using the error data collected from projects [2], [8], [20] [5]. The theoretical understanding is based on the concept of program neighborhoods. A program neighborhood is a set of programs composed of the original program and other programs separated from it by simple faults [26], [27]. That is, while the theoretical interpretation is based on semantics [28], the practical interpretation is based on syntax.

B. Similarity of mutants to real faults

A few studies have attempted to verify if the mutations generated by mutation analysis are similar to real faults found in software. Daran et al. [14] compared 24 first order mutants with 12 real faults from a student developed program. They found that the errors caused by 85% of mutants were similar to the errors caused by real faults. Unfortunately, this study is hobbled by its very small sample size of real faults, and a single program from which mutants were created, which makes it vulnerable to statistical noise.

Studies by Andrews et al. [6], [7] compared the ease of detection for both real and hand seeded faults, which was then

[2]We note that the definition of error does not seem to have been consistent, with some opting to define it as a mistake in programmer's thinking leading to a fault [15], and others opting to define it as a mistake in program internal state

[3]A change that can not be decomposed further

[4]If they were claiming that they were simple semantically, then we could accept their claim. However, mutation analysis requires proof that the errors introduced are syntactically small because mutation analysis produces syntactic mutants

[5]Budd [20] cites studies other than Youngs [25] which we were unable to obtain

compared against the faults induced by mutagens. The metric *ease of detection* is calculated as the percentage of test cases that killed each mutant. Their study concluded that the *ease of detection* of mutants was similar to real faults, and was somewhat different from seeded faults. However, their results rely on real faults from a single program, which limits the scope of inference that can be drawn. The entire study used only eight C programs, making it susceptible to biases due to non representative data.

A further problem is that the hand seeded faults were originally from Hutchins et al. [29]. These seeded faults were chosen such that they were neither too easy nor too difficult to detect. In fact, they eliminated 168 faults for violating these constraints, ending up with just 130 faults. This makes these faults unsuitable for a study of the ease of detection of faults in a general population (because they are non representative).

Namin et al. [30] 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 mutagens on the Java programs, and used fault seeding by student programmers. 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 mutagens used, and even test-suite size have an impact on the relation between mutations introduced by mutation analysis and real faults. In fact, using a different mutagen 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. (Just for *space.c* — same as that used by Andrews et al. [6]). 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. [29], were confounded by their selection criteria which eliminated the majority of faults as being either too easy or too hard to detect.

A recent study by Just et al. [9] investigated the relation between mutation score and test case effectiveness using 357 bugs from 5 opensource applications. They found that mutation score increased along with test-suite effectiveness for 75% of the cases while for coverage, the increase happened only for 46% of the cases. They also note that when coverage increased, mutation score did increase along with it for 94%. However, an increase in mutation score did not result in an increase in coverage in 59% of the cases. That is, coverage increase results in mutation coverage increase, but the reverse does not seem to happen[6]. (This effect was also observed by Wei et al. with regard to branch coverage [32]). While this study is an improvement over older studies, it still suffers from the comparatively small number of test subjects (5 programs), and could suffer from biases due to developmental methods, capabilities of programmers, project-specific culture etc.

[6]It seems to us that it implies a causal relationship between coverage and mutation score, and justifies our explanation for the high coverage observed between statement coverage and mutation score in [31]. There is also the possibility that coverage tracks mutation score when mutation score is low, but not after a certain high coverage level

C. Validation of assumptions

There has been various attempts to validate the assumptions in mutation analysis. In particular, different researchers have tried to validate *the coupling effect*. Theoretical studies by Wah [33], [34] using a simplified finite function model suggests that the survival rate of *first order mutants* is about $\frac{1}{n}$, and that of *second order mutants* is about $\frac{1}{n^2}$ where n is the order of the input domain. Empirical studies by Offutt [35], [16] also suggests that software fault coupling occurs infrequently, with 99% of the second order mutants killed using tests that detect first order mutants. Recent research by Debroy et al. [36] and later by DiGiuseppe et al. [37] on a much larger scale, investigating n^{th} order fault interaction finds that fault obfuscation – where the effect of one fault is hidden by the others – is much more prevalent (about 70% of the interactions result in obfuscation) than expected. We note that Offutt studied only second order faults while DiGiuseppe studies the full spectrum of fault interaction given the available number of faults – about 17 per program studied.

However, *the competent programmer hypothesis* does not seem to have been extensively investigated.

D. Cost of Mutation Analysis

The problem of computational cost in mutational analysis has also been extensively researched. Offutt et al. [18] categorized these efforts into *do fewer*, *do faster*, and *do smarter* approaches. The *do fewer* approach seeks to reduce the number of mutants tested without a significant loss in accuracy of the final result, while the *do faster* approach tries to make the process of running each mutants as fast as possible, and *do smarter* approach tries to distribute the load over several machines, or save and reuse the invariant results of a previous run to make the current run faster.

1) Do Fewer: The *do fewer* approaches include selective mutation, constrained mutation, and statistical sampling of mutants. Selective mutation and constraint mutation attempts to identify mutagens that provide the maximum value, and ignore those mutagens that contribute least. Investigating *constrained mutation*, Mathur, Wong et al. [38], [39] suggested using only two mutagens: *abs* and *ror*, with the reasoning that it forced the tester to examine inputs and boundary conditions. It resulted in 80% reduction of mutants with less than 5% loss in accuracy.

Offutt et al. [40] investigated *selective mutation* whereby the mutagens producing the largest amount of mutants are dropped (*n-selective* approach). With this approach, at *6-selective* stage, 60% savings in mutagens is achieved, with just 1% reduction in accuracy. In a later research [41], it was found that using only 5 expression related mutagens out of 22 mutagens in the mutation platform Mothra resulted in a 77.6% reduction of mutants while losing only about 1% in accuracy.

Barbosa et al. [42] provided a set of guidelines for systematically selecting the mutagens that are most suitable for *selective mutation*. Barbosa's study found that this procedure resulted in an accuracy of more than 99% with a reduction of 65% of mutants (However, they find that the best benefit is obtained with 10% mutants with *x% random selection*, which provided 97% accuracy). This was used by Maldonado et al. [43] to obtain 80% reduction in mutants while keeping the accuracy close to 99%.

Namin et al. [44], [45] looked at the problem of finding sufficient selective mutagen set as a statistical variable reduction problem. They report 92.6% reduction in the number of mutants (just 28 out of 108 mutagens were sufficient) with very high correlation to the full mutation score.

Untch [46] suggested using only a single mutagen — the statement deletion mutagen. Using only this mutagen achieved a higher reduction than other mutagen selection approaches, with a higher correlation to the full mutation score. This was further investigated by Deng et al. [47] who found that savings of 80% fewer mutants could be achieved, with an accuracy of 92%, and a reduction of equivalent mutants over 40%. This was extended further by Delamaro et al. [48] by designing deletion mutagen for all kinds of expressions, and found that operator deletion mutagen achieved higher savings than statement deletion while keeping the accuracy high at 95%.

The statistical sampling also had a significant share of success. Wong et al. [39] found that even sampling of only 10% mutants is effective in achieving an accuracy close to 80%.

Mresa et al. [49] tried to find the most *efficient mutagens* to use, factoring in the average cost to generate and detect. They compared the cost of test run to random selection and found that when an accuracy very close to 100% is needed, random selection is likely to be more efficient than selective mutation using this strategy, but even with a little loss in accuracy (as low as 2%), using *efficient mutagens* resulted in higher cost reduction (nearly twice as that of random selection — about 20% of the full mutation).

Lu Zhang et al. [50] compared mutagen selection techniques to random sampling and found that random sampling was as good as the mutagen selection techniques in terms of accuracy, with same number of mutants. They also found that a single-step sampling of mutants is more effective for larger programs, while a two-step random sampling of mutagens and then mutations was better for smaller programs. Lingming Zhang et al. [51] showed that a sampling rate of 5% was sufficient for a high correlation (99%) with full mutation score, and that it could be reduced further without an appreciable loss in accuracy. They also found that sampling strategies based on program elements performed better than other sampling strategies.

Patrick et al. [52], [53] use static-analysis to identify mutations that are closest to the program. These are harder to kill, and hence contribute more to the mutation analysis.

Another promising area is that of subsumption of mutants. Kurtz et al. [54] found that a minimal set of mutants are sufficiently representative of the entire set of mutants – a test set that kills the *sufficient set* kills the remaining mutants too. They suggest that it can produce a savings of up to 2400% (although, their conclusion is based on a single program *cal*).

Research to reduce the number of mutants executed also includes using a selection of higher order mutants [55], [56]. They attempt to find *subsuming higher mutants* that can provide as much information as component mutants with first order mutations. Multiple strategies are used to increase the chances of coupling, and hence utility of selected mutants. These include *random selection*, *different mutagens*, *based on order of production* [57] or based on dominator analysis [58]. The best hybrid strategy achieves about 50% reduction in effort, with 1.75% average loss in accuracy (They also found that covering just 10% of mutants is sufficient to cover the remaining too, which mirrors the results from our own study).

2) Do Smarter: The *do smarter* approaches include weak mutation, parallelization of mutant execution and incremental mutation approaches.

Weak mutation is a relaxation of the constraints on mutation analysis whereby only a difference in internal state after execution of the mutated subunit is necessary (rather than full test failure) [59]. However, weak mutation testing is really a coverage measure [60], and orthogonal to our research. *Firm mutation* is a related technique that postpones the checking of infected state to some time later than the immediate execution of mutated subunit [61].

Other *do smarter* approaches involve parallelization of mutation analysis by taking advantage of various parallel and distributed architectures [18]. Offutt [18] also describes algorithmic approaches that enable space-time trade offs for mutation analysis.

3) Do Faster: The *do faster* approaches include schema based mutation analysis, compiler integrated mutation, and incremental mutation.

Untch [62] pioneered the schema based mutation analysis. All the mutations are encoded within the same source code, and the code is compiled once, resulting in savings by avoiding multiple compilation of non mutating parts. The bytecode based approaches such as in MuJava [63] also help in reducing the cost of computation and fall under the *do faster* approach.

Another approach called conditional mutation analysis by Just et al. [64] inserts triggering conditions into the source code, such that each mutant can be triggered separately. This eliminates the separate compilation overhead.

Study by Cachia et al. [65] suggests finding invariants from previous runs of mutation analysis, and thus reducing the number of tests to run. They accomplish this by assuming that the program and test sets can be split into two sections — one invariant, and other changed — and shows that we only need to rerun the test cases for the changed section. They were able to obtain speed improvements of up to 91%. A similar approach is also taken by Lingming Zhang et al. [66]. They use reachability analysis to find test cases that need not be executed again. This results in savings from 50% to 90% mutants.

E. Correlation with other criteria

Andrews et al. [7] identified that statement coverage (along with other coverage criteria) had a high correlation with mutation coverage, and provided an initial reasoning why this might be so. However, this research is limited by the extremely small number of programs sampled (**space.c**). We note that this study is the closest to the aims of our research.

Namin et al. [67] conducted a study very similar to our own where the influence of coverage and test-suite size was investigated. On a sample of seven C programs, the mutation score and coverage was measured, along with the size of test-suite. It was found that the size of test-suite and coverage independently impacted the mutation score. However unlike in our study, they found that a model combining test-suite size and coverage to be the best predictor.

Gligoric et al. [1] and later Sharma [68] compares various coverage criteria with the aim of choosing the best criteria to approximate mutation score. This research suggests that branch-coverage is the best criteria, followed by AIMP, a variation of path coverage.

Inozemtseva et al. [69] found that there is a low to moderate correlation between coverage and mutation score when size of test-suite is discounted, but its strength is variant between projects. Further, they also found that the correlation with mutation score remains similarly high for all coverage measures investigated.

F. Improving mutation analysis

The other major area of research has been in the reduction and elimination of equivalent mutants and redundant mutants. Equivalent mutants are mutants which differ from the correct program syntactically, but not semantically. Similarly Redundant mutants are mutants that differ from another mutant syntactically, but are semantically equivalent. Offutt et al. [70] suggests that 9% of the total mutants are equivalent mutants. Schuler et al. [71] looked at coverage as a means of weeding out equivalent mutants. They were able to achieve an accuracy of 75% in classifying equivalent mutants using coverage, and found that in a manual sample, 45% of all undetected mutants were equivalent, and about 7.39% of all mutants were equivalent.

Just et al. [72] shows that avoiding redundant mutants can reduce runtime by 34%, and they can inflate mutation score by as much as 10%. Amman et al. [73] provides a theoretical foundation for minimal mutation sets avoiding redundant mutants.

Belli et al. [74], [75] approaches mutation analysis from a different direction. They convert the program under test into a model using event sequence graphs and then run mutation analysis on this model, which reduces the number of mutagens to just two. They also find that when considering this model of the original program, the coupling hypothesis does not hold true.

Harman et al. [76] suggests that there are several myths associated with mutation analysis. They categorize these myths as

1) Real Fault Representation : The myth that mutation analysis produces faults similar to those that real programmers make.
2) Unscalability : The myth that mutation analysis is unscalable due to number of mutants produced
3) Equality of Mutagens: The myth that there is no order of priority among mutants to be killed,
4) Global Mutagens: The myth that a defined set of mutagens can be used for all kinds of programs
5) Competent Programmer Hypothesis : The myth that faults are first order or close
6) Syntactic Semantic Size : The myth that syntactic proximity suggests semantic proximity.
7) Coupling Hypothesis Extension : The myth that higher order mutants are not useful.

They further suggest that search based engineering and higher order mutations can together improve the quality of mutation analysis.

G. Alternatives to Mutation Analysis

One of the most promising alternatives to mutation analysis is *Checked Coverage* [77] where the actual properties checked by oracles are taken into account for coverage ("The ratio of statements that contribute to the computation of values that get later checked by the test suite"). While intuitive, an evaluation with real faults is lacking for this work. Another recent research [78] compared interaction coverage of test-suites and mutation scores to faults, and found that mutation analysis of the input model has a higher correlation to the faults than interaction coverage (Interaction coverage is the number of parameter interactions that are exercised by a test-suite. The input model consists of the parameters themselves, and the constraints applied on them). Rajan et al. [79] showed that MC/DC metric was very sensitive to simple semantics preserving changes such as inlining of conditionals in code. This suggests that MC/DC may not be a reliable measure for test-suite quality.

H. Fault patterns

Duraes et al. [80] collected 500 faults, and classified them according to ODC methodology, which was further classified into three groups — missing, extra, and wrong constructs. It was used to create G-SWFIT (Generic Software Fault Injection Technique) — a mutation engine. The major differences from our work are that, we investigate a much larger set of projects and commits, and secondly, unlike our research, G-SWFIT is oriented towards fault emulation rather than mutation analysis. Hence the direction of mapping differs. While we mapped the existing mutagens to the faults observed for validation of current mutation operators, Duraes et al. came up with mutation operators that emulated observed faults as closely as possible.

A similar research by Pan et al. [81] classified the bugs found from several opensource projects to 27 patterns (they do not mention the number of bugs thus classified). While their procedure is broadly similar to our study, their aims are similar to Duraes et al., and does not target mutagens.

Research by Elbaum et al. [82] finds that small code changes of the order of 1% statements can have an impact of as much as 10% in the coverage produced (and 2% had an impact of 20%). This suggests that small syntactical changes can have disproportionate semantic influence.

III. Research Overview

Our primary emphasis is to improve the current state of the art in judging quality of test-suites. Since mutation analysis is the de-facto method of measuring quality of a test-suite, we wanted to ensure that the measurements thus obtained were correct,

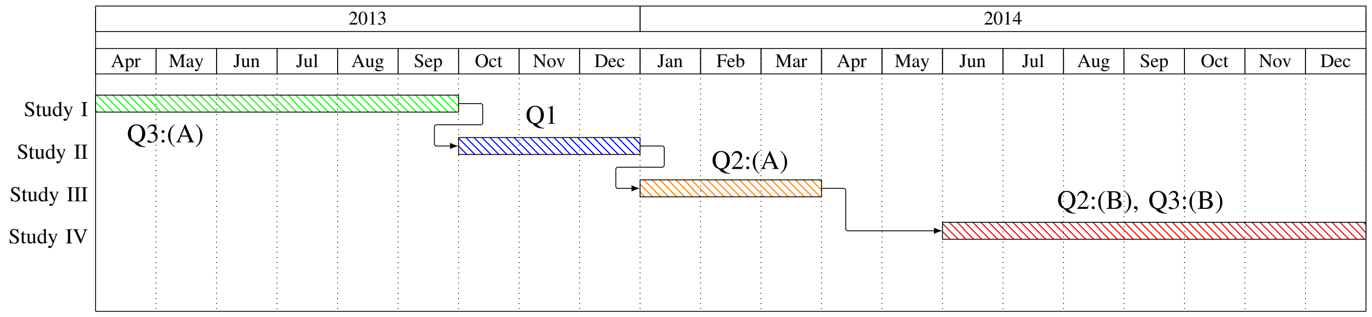

Fig. 1. Mapping between studies and research questions

and using it would not lead to aimless optimization towards a nonsensical target. We found that *the coupling effect*, one of the main assumptions of mutation analysis was well-studied, while the other main assumption, *the competent programmer hypothesis* was lacking in large-scale studies oriented towards mutation analysis. This is especially important in presence of factors such as differing programming languages with different paradigms. Hence, one focus of our research was in verifying the competent programmer hypothesis, and quantifying it in the presence of different programming languages.

Another area of focus was the accuracy of mutation analysis results in predicting quality of a test-suite. A few previous results indicate that mutation analysis can provide an estimate of quality of a test-suite. However, the previous research is limited by small sample size, making wider inferences difficult. Our research also intends to find whether different mutagens and mutants contribute equally to the fault detection capability, and if other metrics can be used to improve the accuracy of prediction of fault detection capability.

Finally, we felt that it is important to have a cheap and fast method of evaluating the adequacy of a test-suite while in development. For this reason, we also attempt to find if other coverage measures and metrics can be used to predict the fault detection capability with sufficient accuracy.

To recap, our research questions were

Q1 Explore and quantify competent programmer hypothesis, especially with respect to effect of programming languages.

Q2 Validate mutation analysis, and quantify the effects of different mutagens in predicting real faults in real projects on a very large-scale, and identify ways to improve accuracy of prediction.

Q3 Investigate alternatives to mutation analysis that can predict fault detection capability without significant loss in accuracy.

The mappings between the research questions and studies conducted are given in Figure 1.

Our emphasis in each of these studies was to find empirical evidence from a large number of real-world projects. This is in contrast with the previous studies which were mostly limited to small sets of well known programs. While relying on a well known set of programs is useful as a way of limiting variability, and can help in understanding particular results in the scope of the given programs, we believe that it is ultimately harmful as it leaves the results open to bias from non-representativeness. Hence, in our research we strove to analyze as many projects as possible, and gave preference to real-world data to ensure the validity of our results in the real-world. We have spent considerable amount of effort in ensuring that we have *real projects*, *real test-suites* and *real bugs* rather than giving in to the temptation to generate these as has been often done in the past.

Our studies are summarized in the following sections.

STUDY I

Code coverage for suite evaluation by developers[7]

Motivation: As Gligoric et al. [1] notes, while there exist a large body of research that focuses on comparing adequate test-suites, few attempts have been made to compare non-adequate test-suites. However, the results based on adequate criteria may not translate into results on non-adequate criteria.

Our research was motivated by previous research from Gligoric et al. [1] which suggested that branch coverage and path coverage could conceivably be used to estimate the mutation score of a test-suite.

However, their research attempts to compare different coverage criteria for the purpose of comparing different test-suites. That is, they compare a large number of test-suites for same set of programs, assessing which criteria provides a better approximation to mutation score. While this is indeed useful for researchers, it falls short of requirements of developers who usually want to know when their particular test-suite is good enough. Secondly, their research samples a small set of programs which limits the statistical inference that can be drawn.

We had two main contributions from this study. First, the availability of open source programs from Github and the presence of a uniform build system allowed us to undertake a large-scale unbiased study of real-world projects in various stages of growth, by very different authors, under varying styles of development, and of different levels of maturity and size. Our

[7]Published in ICSE 2014

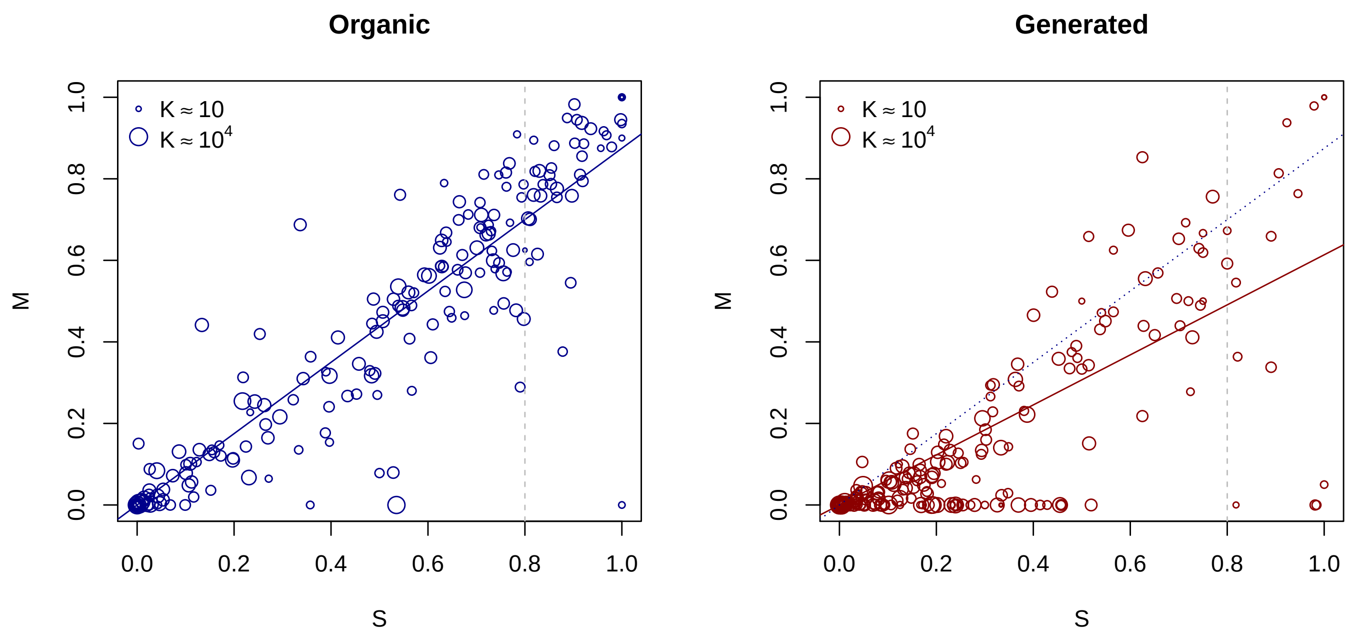

Fig. 2. Relation between *statement coverage* and *mutation score*. The circle represents the magnitude of project size.

study is much larger in scale than previous studies considering similar questions. This allowed us to draw strong statistical conclusions. Secondly, our results provide a directly applicable result for the developers to evaluate the adequacy of their tests with regard to fault detection ability.

Finally, the entirety of our research including the subject programs, the framework to conduct the evaluation, and the data we collected are available over the web for any researchers to download, replicate, and extend. We hope that this will foster further research in this area.

***Questions**:* Our focus in this research was to investigate how best to approximate the fault detection capability of test-suites cheaply, which addresses the need for *variety* (Q3).

A secondary aim was to ensure that our research was as widely applicable as possible while ensuring that the statistical inferences drawn was significant.

***Methodology**:* As part of the effort to ensure that our analysis was as widely applicable as possible, we started with 1733 projects from Github that used Maven build framework. These were reduced to 1254 after eliminating project aggregations. We found that only 729 of these had test-suites. For the initial set of 1254 projects, we also generated test-suites using Randoop [83] using the default settings. After eliminating pathological projects with missing dependencies, compilation errors etc. we obtained a total of 259 projects with organic test-suites, and 243 projects with generated test-suites that compiled and ran successfully with Pitest [84]. All test-suites that did not pass were discarded since Pitest can not process test-suites with failures.

For both groups, we analyzed the source code metrics including the size of the projects, and the cyclomatic complexity of the projects that were not included, to ensure that the projects remaining were representative of the larger population.

Next, we collected various coverage measures of these projects using different tools, trying to ensure that there were more than one tool that provided the same measure, and one tool measured more than one metric. We then verified that the measures reported by the different tools were in reasonable agreement with each other, thus accounting for any debilitating bugs in the tool. We used Emma [85], Cobertura [86], Codecover [87], Mockit [88], and Pitest [84] to collect statement coverage, basic block coverage, branch coverage, path coverage, and mutation coverage.

We built statistical models using each of these measures, along with the size of the project and average cyclomatic complexity. We ignored the project test-suite size which showed high correlation with project size to avoid multicollinearity. Our initial equation was

$$\mu\{M|K,C,S\} = 0 + \beta_1 log_2(K) + \beta_2 C + \beta_3 S$$

where K was the project size, C the cyclomatic complexity, and S the statement coverage. After removing non-significant components step by step and using R^2 to compare models, we found that statement coverage alone was the significant (and highest) contributor to mutation score, giving the equation.

$$\mu\{M|S\} = 0 + \beta_1 S$$

***Results**:* Our results showed $R^2 = 0.94$ for organic and $R^2 = 0.72$ for generated test-suites. The scatter plots between mutation score and statement coverage for both organic test-suites, and generated test-suites are given in Figure 2.

Similar analysis was conducted for branch coverage, and path coverage. The scatter plots for both organic test-suites and generated test-suites for branch coverage are given in Figure 3.

The scatter plots for both organic test-suites and generated test-suites for path coverage are given in Figure 4.

The correlation between mutation score and statement, branch and path coverage are given in Table I. We also provide Kendall τ_{beta} for comparison. All the experiments were significant with $p < 0.001$.

TABLE I. CORRELATION COEFFICIENTS (MUTATION)

	R^2(O)	τ_β(O)	R^2(G)	τ_β(G)
$M \times S$	0.94	0.82	0.72	0.54
$M \times B$	0.92	0.77	0.65	0.52
$M \times P$	0.75	0.67	0.62	0.49

This suggests that statement coverage can be a good proxy for the mutation score of a project. This finding is significant because statement coverage, unlike mutation coverage can be found from a single run of the test-suite, and hence cheap in computational terms. Further, statement coverage is one of the most common measures used by software engineers, which makes it very a relevant measure.

STUDY II

Mutations: How close are they to real faults[8]

***Motivation**:* The success of mutation analysis depends on the mutants produced being similar to the real faults. While there exist some studies that investigated the correlation between mutation analysis and defect detection [14], [6], [30], [9], except for the study by Just et al. others are based on results from very few programs and hence can not be considered statistical evidence in favor of mutation analysis. The study by Just et al., while statistically significant, stopped at observing the correlation between mutation analysis and fault detection rate, and did not attempt to verify the similarity between mutations and real faults. Even the studies previously quoted stopped at the similarity between the two in the ease of detection, and the error trace produced.

However, researchers often assume on the basis of competent programmer hypothesis that the mutations produced are representative of real faults [89], [7], [90], [53], [37][9]. This suggests that there is a need for more clarity in this regard, especially whether the mutations produced are representative of real fault patterns.

Our research undertakes to validate this assumption by examining bug fix patterns, which are in effect the reverse of fault patterns. We use revision histories from the large body of opensource programs in Github to quantify bug fix patterns. We also examine the effect of different languages on fault patterns, providing guidance for mutagen design.

Our research has the following major contributions. First, we quantify the magnitude of change patterns. Secondly, we identify the effect of different languages in the fault patterns, which tells us about the possible interaction between different mutagens and different languages.

***Questions**:* Our primary objectives were to quantify and *validate* (Q1) the common assumption that mutagens captured real fault patterns on the basis of *the competent programmer hypothesis*. The second objective was to investigate the effect of language on the distribution of bug fix patterns.

***Methodology**:* Our analysis was based on 5000 opensource projects belonging to Java, C, Python, and Haskell from Github. We applied statistical clustering mechanisms to identify bug-fixes from revision histories, and achieved an accuracy of classification of 73.19% for acceptance of bugs.

Next, we normalized each commit by pretty printing, removing differences due to white space, and passing through a token differencing algorithm. This generated multiple chunks per commit. We analyzed each chunk to identify the size of change, and distribution of mutagens involved.

Finally, we ran two-way ANOVA to determine the contribution of different mutagens, and influence of language.

***Results**:* We found that bug fixes were identical to feature updates as far as the syntactical patterns were concerned. Secondly, an average change involves about eight tokens addition or removal, increasing to ten tokens if we want to include at least 80% of the changes. This result strongly suggests that at least in terms of magnitude of changes, mutations produced by current tools are dissimilar to real faults. The distribution of cumulative token changes is given in Table III[10]

TABLE II. MEAN DIFFERENCE FOR AVERAGE TOKENS CHANGED BETWEEN DIFFERENT LANGUAGES ($p < 0.05$ EXCEPT C $\times$ JAVA)

	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

We found that the patterns of C and Java were very similar to each other with no significant difference between them. Python was (statistically) different, with Haskell having the largest difference. However, as we see in Table II, Haskell was closer to Python than to C or Java. We speculate that this may be due to functional programming patterns encouraged in Python programming, and the functional programming paradigm of Haskell language.

Secondly, our regression analysis showed that interaction of language and change patterns had a statistically significant impact in the change distribution. The Figure 5 shows that the interaction between language and mutation operators irrespective of whether only the bug fix changes were considered

[8]Targeted ISSRE 2014

[9]These articles do not explicitly call upon *the competent programmer hypothesis*, but we believe that their use of mutant-faults instead of a fault seeding approach based on fault distributions is based on *the competent programmer assumption of mutation analysis*

[10]max_ϵ is the largest percentage detected in the remaining token bins

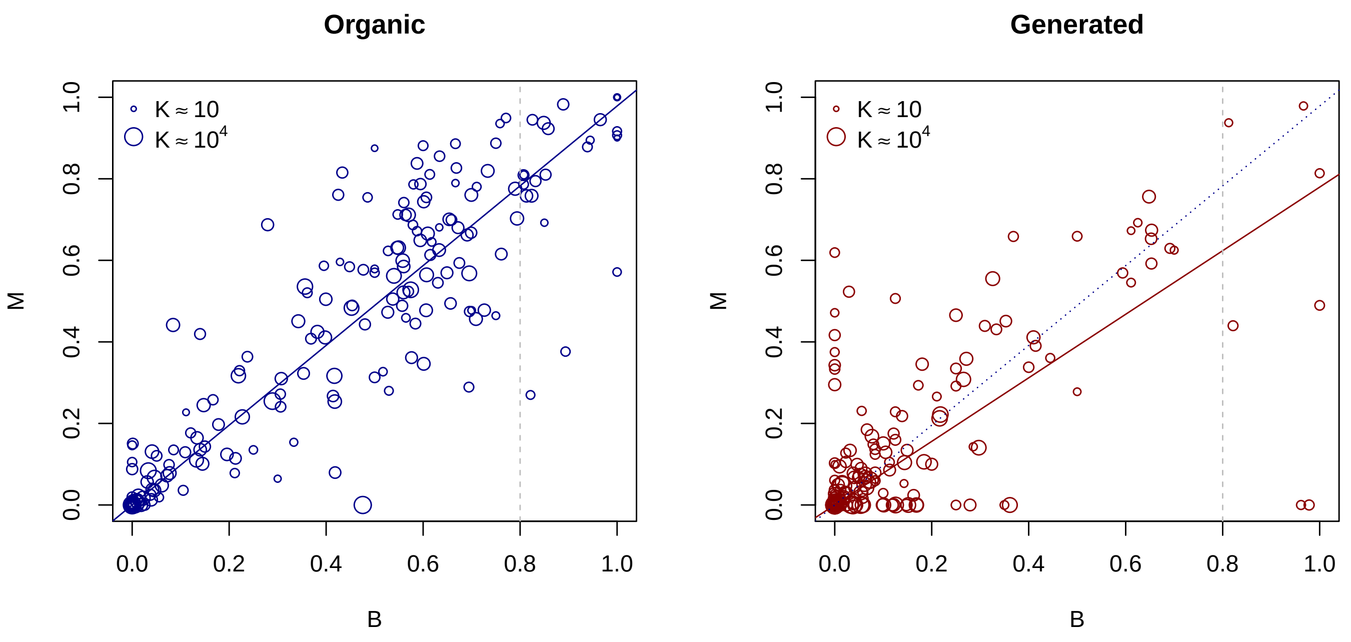

Fig. 3. Relation between *branch coverage* and *mutation score*. The circle represents the magnitude of project size.

TABLE III. CUMULATIVE DENSITY(%) OF AVERAGE TOKEN CHANGES

	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

or all commits (full distribution) were considered. All our experiments were significant at $p < 0.05$.

This suggests that we need to consider the effect of language when designing mutagens, and that the results in selective mutation from one language may not carry over to another.

Our research shows that if we need to include a large majority of faults (say 80%), we need to consider changes of at least 10 tokens[11], which suggests that a majority of real faults can not be characterized as simple faults, and relying on *the competent programmer hypothesis* for similarity of bugs and mutants may be risky. We also observe that quite a large number of changes observed could not be related to any mutagen. This suggests a need for further study on what kinds of bugs come under mutation analysis, and a need to rethink the current mutagens.

STUDY III

An Empirical Comparison of Mutant Selection Approaches[12]

Motivation: A major area of research is in finding ways to reduce the number of mutants, replacing the full mutation analysis with the score from a representative set of mutants [17]. This approach is included in the *do fewer* category of mutation reduction methods by Offutt [18].

The approaches under this category include constrained mutation by Mathur et al. [38], [39] using two mutants (selected for good coverage), selective mutation advocated by Offutt et al. [40], and sampling strategies which has recently been investigated by Zhang et al. [51] who showed that sampling based on program elements was more accurate than other approaches.

While these approaches tend to reduce the mutation analysis requirements, a large-scale comparative study is lacking [51]. Our research attempts to rectify this lacuna by taking up a large comparative study of different sampling approaches, and selective approaches.

[11]In the table, the addition or removal of one token is taken as $1/2$, and change of one token (one token removed, and another added) is taken as 1

[12]Submitted to FSE 2014

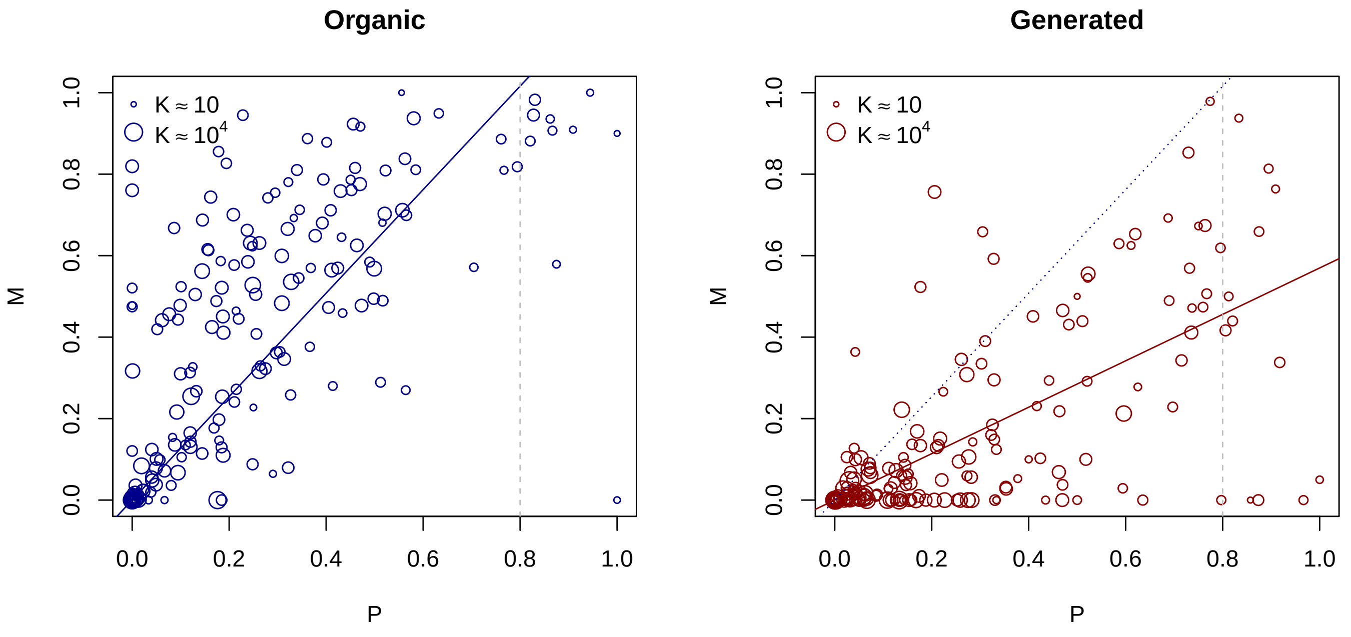

Fig. 4. Relation between *path coverage* and *mutation score*. The circle represents the magnitude of project size.

We undertook to study the effectiveness of the *mutagen selection*[13] approaches mentioned in the literature, such as constrained mutation [38], [39], E-selective mutation [41], Javalanche mutagens [91], variable reduction operators given by Namin [44], N-selective approach pioneered by Offutt [40], and statement deletion recently researched by Deng et al. [47].

We also compared the effectiveness of various *mutant sampling* approaches including *x% selection* [20], different ways of sampling over program elements [51] and *x% per mutagen* as given by Wong et al. [39]. Each sampling was conducted on decreasing fractions through $1/2, 1/4, 1/8, 1/16, 1/32, 1/64$ of the original set of mutants. This allowed us to judge the effectiveness of sampling criteria at different reductions. These were then compared to the full mutation score to determine the best mutation reduction strategy. We also noted the standard deviation of the results (stability) to ensure that the results were not project specific.

Our study is the largest so far in terms of comparisons between different mutation and mutagen selection methods, both for methods compared, and also the number of projects analyzed.

Questions: Our aim in this study was to compare various *mutagen selection* and *mutant sampling* criteria, and identify the best criteria for predicting the final mutation score (Q2).

Methodology: We collected a large number Java projects using Maven build framework from Github, eliminating very small projects, projects with very small test-suites with little coverage, pathological projects that were prone to oscilate in coverage metrics, or took an inordinate time to complete test runs. From the set of projects thus chose, we sampled 110 projects for our mutation analysis runs.

We collected the mutation analysis results of multiple *mutation selection* techniques including simple x% selection, sampling over program elements, sampling restricted to number of lines, sampling a single mutant per program element, and x% selection per mutagen. We also collected the results of various mutagen selection techniques including constrained mutation, E-selective mutation, Javalanche mutages, using variable reduction, using N-selection, and statement-deletion. Next, we compared the effect of increasing powers of reduction $(1/2, 1/4, 1/8, 1/16, 1/32, 1/64)$, and tabulated the result. A visualization of simple mutant sampling approach results from different scopes is shown in Figure 6.

Results: We found that simple sampling — without regard to the scope or location of the mutation produced — provided the best estimate of full mutation score. Further, simple sampling was also able to achieve the best mutant reduction ratio out of all methods compared. It was able to use only 1.37% of the original mutants to produce an estimate of full mutation score within 0.97% accuracy (using R^2), with minimal deviation of 6.36%. Finally, all mutant sampling methods seem to do better than mutagen selection methods in terms of reduction achieved and also the stability of the score. This suggests that at least in predicting the full mutation score, current techniques for mutation and mutagen selection fall short, and indicates the need for further research on this subject.

[13]selective mutation

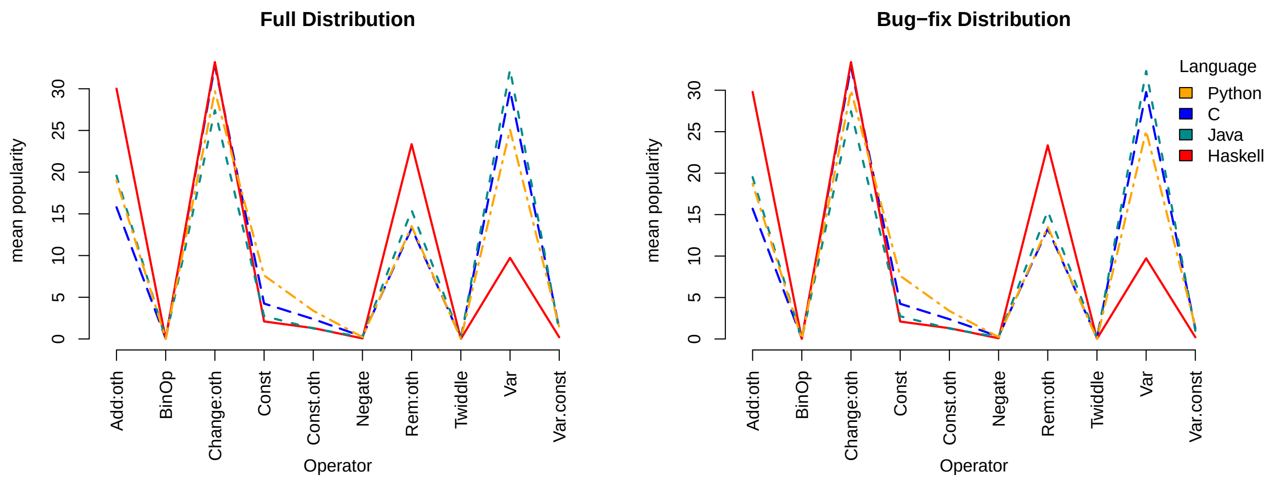

Fig. 5. $Op \times Language$ interaction for the complete commits (full distribution) and bug fixes only

A. Summary

We have conducted three large-scale studies so far. The first study compared different coverage measures against mutation score, and found that statement coverage can best approximate mutation score. This suggests that statement coverage merits consideration as an alternative to mutation analysis, which can be accomplished cheaply (Q3).

The second study looked at the change patterns across different languages, and found that the change patterns are dependent on programming language used, and further, the average magnitude of change is about ten tokens, which is much larger than what was expected. This ties in with our research objective of validating assumptions of mutation analysis (Q1).

The third study considered different variations of selecting mutagens and mutants, and found that none of the current techniques can provide a better approximation than random selection of mutants. If a particular selection of mutagens or mutations can approximate fault detection capability better than the full mutation score, it will improve the accuracy of mutation analysis. This is in line with our research objectives (Q2)

IV. Proposed Research

STUDY IV

Mutation analysis and real faults: A large-scale empirical analysis[14]

Motivation: While researchers have used mutation analysis as a measure for the fault detection capability of test-suites, there exist few large-scale studies that measures the correlation between mutation analysis and fault detection capability of test-suites. The largest study undertaken so far by Just et al. [9] investigated 357 bugs from five opensource projects. While the number of bugs examined is reasonably large, it is somewhat unsatisfying because it fails to provide a clear statistical relation between the fault detection capability and mutation score, which is necessary to provide a good stopping rule. Secondly the authors concentrated on just five opensource projects which lays their study open to biases due to individual characteristics of these projects.

This suggests a need for large-scale studies over a larger set of projects oriented towards providing a clear statistical relation between the fault detection capability of a test-suite and its mutation score, which can be used to decide when a test-suite is good enough.

A second important concern is to determine if the fault detection rate can be approximated using cheaper coverage measures. Such a metric, if available, would provide a quick measure of test-suite adequacy, which would be suitable for development environments. Our intent is not to replace mutation testing since such a measure would only have high correlation to fault detection capability. A measure to replace mutation analysis would also need to provide a sound theoretical reasoning as to why it measures the fault detection capability to be considered seriously as an alternative[15].

Questions: Our primary objective is to verify mutation analysis — to see if there exist a significant statistical relation between mutation analysis and fault detection capability. We would also look for any other model that includes mutation analysis and other coverage measures, or source code metrics that is able to achieve a higher accuracy in predicting the bug detection accuracy. Finally, we also propose to devote some attention to finding whether fault detection capability can be cheaply and accurately approximated using a combination of other coverage measures and code metrics.

A. Methodology

The following steps will be used to obtain a measure of fault detection capability. The procedure followed by Just et al. [9]

[14]Proposed

[15]The reason we insist on a theoretical basis is to avoid confounding variables which may cause developers to aim for nonsensical objectives.

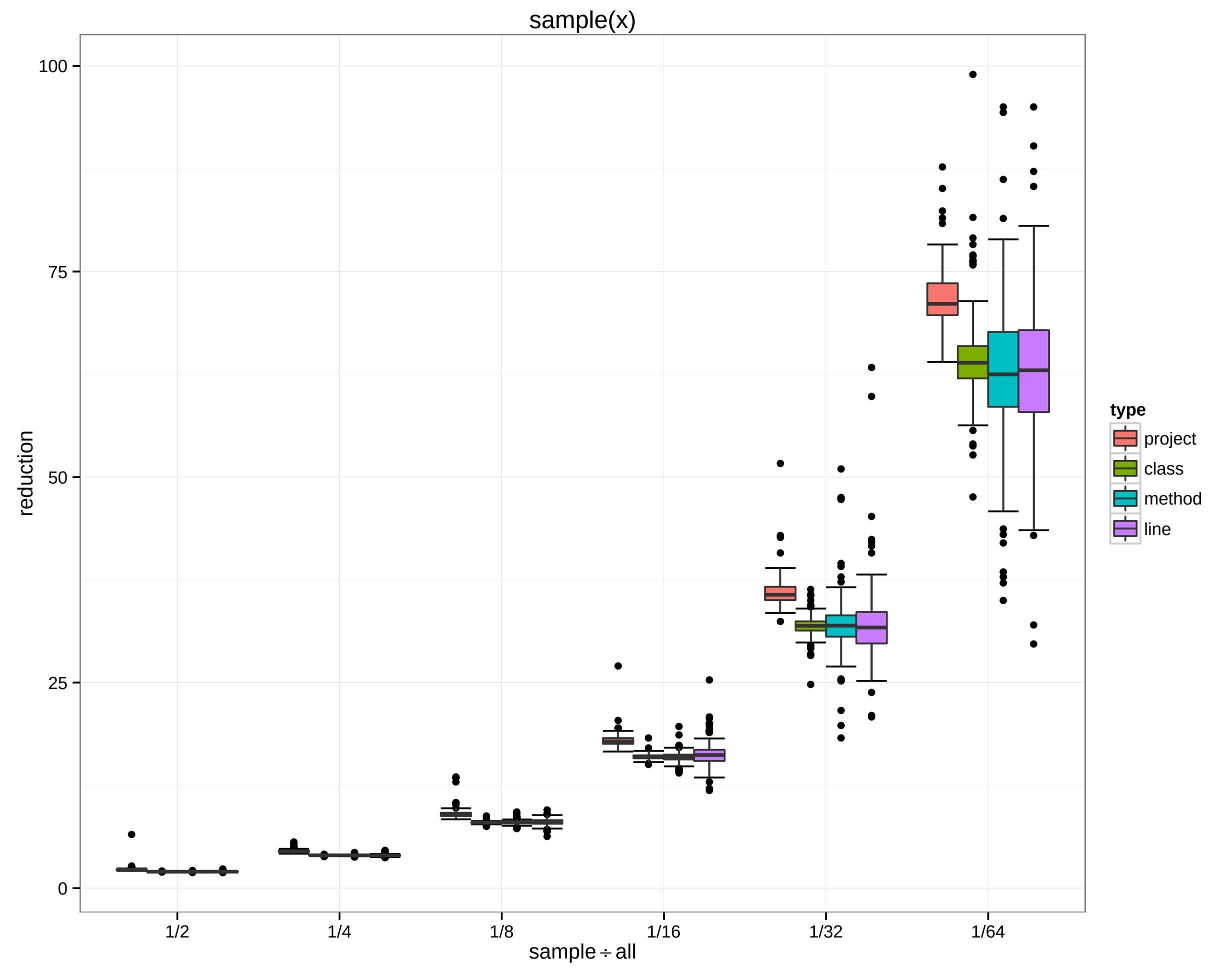

Fig. 6. The effect of sampling reductions, $R^2 > 0.90$. Notice the hierarchy of spread within boxplot for similar sampling ratios (÷) project < class < method < line i.e. project is better

was adapted for this work. The difference here is the use of statistical techniques for classification of commits into fixes and features which allows a much larger scaling of the work and hence conclusions drawn. Secondly, this work also keeps track of the history of a project. This allows our research to build a statistical regression model for use by software developers.

1) Choosing Projects: First, a large sample of opensource projects in Java, which utilize the popular Maven build framework will be sampled from Github, along with their revision histories. We expect to collect at least 1000 Java projects (maximum provided by Github search[16]), and to obtain at least 200 projects that have all dependencies met, have a good, passing test-suite, and excluding pathological projects that take an inordinate time to complete a test run.

2) Classifying Commits: Following the previous work, first a training sample of about 5000 commits will be created, with classification as fix and feature. Next, this will be split into 4000 samples for training and 1000 samples for cross validation. We expect to have an accuracy rate of about 72% for rejection, as obtained previously in our research. Next, the entire set of commits for each projects would be classified based on this learned statistical model.

3) Collecting test-suites: In this phase, the original test-suites T of each project will be collected. Next, the bug fixes identified in previous phase is removed one at a time. For each bug fix removed, a few test cases can be expected to fail. Whenever a test-suite fails, the failing tests are removed and a new related test-suite with a slightly smaller fault detection capability than the first test-suite is created. Let T^n be the new test-suite, where n is the number of fixes removed to get to it. Ultimately, this will result in a minimal test-suite with the least amount of capability to detect bugs after removing N fixes from the project. The number $N - n$ provides the fault detection capability of the test-suite in question.

For each project, the process of removing a random bug at a time to get to a minimal test-suite will be repeated multiple

[16]We will try to reuse the projects from our previous study

times to ensure that the results are not biased by any particular ordering. The results from same project would be blocked together to account for similarity.

4) Prediction of fault detection capability with mutation analysis: Our aim here is to find the best statistical model that predicts the residual defect score, which can be used to find the fault detection capability. To do this, We run the full mutation analysis, individually accounting for the scores generated by different mutagens and also including the interaction with complexity and size of the project.

Using linear regression, we will attempt to find the factors that contribute significantly towards predicting the residual defect score, and also the effect sizes for each. This result can be used to model the fault detection capability more accurately than it is possible with the mutation score alone. Further, it can be used to prioritize mutagens in terms of their contribution, and can also be used to sample the mutation instances based on their effect sizes.

5) Prediction of fault detection capability without mutation analysis: We use the same test-suites as before, but here, we remove the mutation scores, and instead, substitute them by test coverage measures such as statement coverage, branch coverage, and path coverage, and also include factors such as size of the project and average cyclomatic complexity of the project. The aim here is to be able to estimate the fault detection capability without taking recourse to mutation analysis.

B. Expected Results

We expect to fit the regression model in Equation 1 first, and report on the coefficients, correlation (R^2) obtained, and the *p-value*. In Equation 1, D is the fault detection rate, and M the full mutation score.

$$\mu\{D|M\} = \beta_0 + \beta_1 M \tag{1}$$

This will provide us with an estimate of how closely the full mutation score tracks the fault detection rate.

Next, we will fit the regression model in Equation 2, and report on similar statistics.

$$\mu\{D|M_e, M_s, M_c\} = \beta_0 + \beta_1 M_e + \beta_2 M_s + \beta M_c \tag{2}$$

Here M_e, M_s, and M_c indicate mutation scores of expressions, statements and constant mutagens respectively[17]. The results from this equation can answer the question whether different mutagens contribute differently to fault detection rate.

Next, the model given by Equation 3 will be fitted. Note that the equation is an outline. We intend to fit the model with terms representing each of the mutagens (M), coverage measures (C), and metrics measured (K).

$$\mu\{D\} = M_{e,s,c} + C_{s,b,p} + K_{l,d,c} \tag{3}$$

The idea is to see if the fault detection rate can be predicted with better accuracy than using just mutation analysis, which may lead to newer insights on the nature of defect distribution.

[17]This assumes a simple mutation analysis system with just three mutagens. It will be adapted to the full complement of mutagens provided by our tool of choice

Finally, in the previous model, mutation coefficients are removed, and analyzed again to find the closest approximation of fault detection rate using coverage measures other than mutation (Equation 4).

$$\mu\{D\} = C_{s,b,p} + K_{l,d,c} \tag{4}$$

The result of this model, if it has a high correlation with fault detection capability will provide a cheaper alternative to mutation analysis.

C. Discussion

A positive result from this study that suggests that mutation score is highly correlated with defect detection capability of a test-suite will help end the ambiguity of usefulness of the mutation analysis in measuring test-suite quality. It would give a fillip to the ongoing efforts to improve mutation analysis.

On the other hand, if mutation analysis proves to be not strongly correlated with defect detection capability, it would help in turning the testing world away from a mediocre measure, and focus the efforts on finding better measures for test-suite quality.

V. Timeline

This thesis calls for one more major research to be completed, which can conclusively show whether mutation analysis is an adequate measure for defect detection capability of test-suites. The time line for defense is visualized in Figure 7. It is as follows:

1) *June 2014 — August 2014* Literature survey
2) *September 2014 — December 2014* Collect the required projects, and work on enabling parallel execution of data collection framework.
3) *January 2015 — March 2015* Run the data collection framework, collect results, and preliminary statistical analysis of results.
4) *April 2015 — June 2015* First draft of thesis
5) *July 2015 — September 2015* Final draft of thesis
6) *Defence*

VI. Threats to Validity

While we have taken utmost care to avoid errors and biases, our results are subject to various threats. First, we have chosen samples from a single source — the Github opensource repository. While we expect the samples to be representative of the wider world, this may be a source of bias if projects in Github are biased in any way. Secondly, our samples were limited to opensource projects, and it could be argued that different developmental procedures followed may cause closed source programs to have different characteristics than opensource projects, and may be a source of bias. Finally, the selection mechanisms of Github itself may be a source of bias, favoring projects based on some internal criteria.

However, we believe that the large number of projects sampled more than adequately addresses this concern.

Another source of error may be mis-classifications in bug fixes and feature updates. However, in our initial research, we

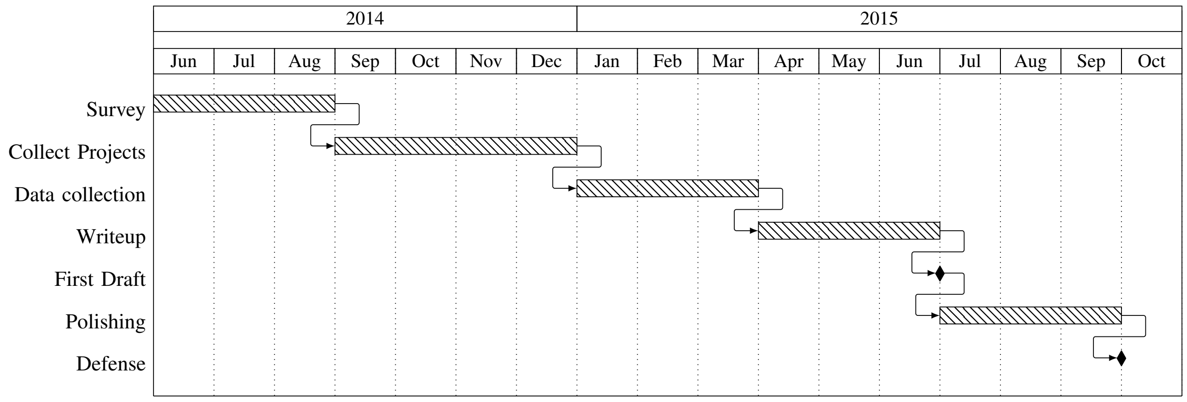

Fig. 7. Timeline to defence

were able to achieve an accuracy of classification comparable to the current research, and hence we believe that using the same tools would help us reduce mis-classifications.

VII. Conclusion

In software testing, it is important to have a way to measure the quality of a test-suite, and mutation analysis is the most common method used by testers for this purpose. However, we found that there is a lack of large-scale studies that targeted mutation analysis assumptions, which is worrisome.

In our research, we set out to improve three different aspects of measurement of test-suite quality by running large-scale studies on widely available opensource programs. The first was *validation* of mutation analysis assumptions, especially *the competent programmer hypothesis*. The second was to *verify* that mutation analysis was really measuring the fault detection capability of a test-suite, and finally, we wanted to provide *variety* to the software engineers, allowing them to use metrics other than mutation analysis to judge the quality of a test-suite, with comparatively small loss of accuracy.

Our proposal would pave way for further predictive accuracy, and ease of use of mutation analysis by reducing its resource requirements. This can help software engineers write better tests, and hence improve the quality of software produced.

By publishing the data [10], environment, and research materials [11] (all materials necessary for replication of our work, including the statistical steps used, and paper source [12] in Knitr, according to reproducible research guidelines [92]) we also hope to encourage robust research, and improvements to mutation analysis and other test-suite adequacy measures.

References

[1] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, “Comparing non-adequate test suites using coverage criteria,” in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2013.

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

[3] A. P. Mathur and W. E. Wong, “An empirical comparison of data flow and mutation-based test adequacy criteria,” *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[4] A. Offutt and J. Voas, “Subsumption of condition coverage techniques by mutation testing,” GMU, Tech. Rep., 1996.

[5] N. Li, U. Praphamontripong, and J. 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*. IEEE, 2009, pp. 220–229.

[6] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?[software testing],” in *International Conference on Software Engineering*. IEEE, 2005, pp. 402–411.

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

[8] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. 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*. ACM, 1980, pp. 220–233.

[9] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?” University of Washington, Tech. Rep. UW-CSE-14-02-02, 2014.

[10] R. Gopinath, “Code coverage and metrics,” in *http://hdl.handle.net/1902.1/22193 UNF:5:u0boYDAo3hrpukL5XZE0oQ==*. Harward Dataverse Network V1, 2013-09.

[11] ——, “Replication environment for code coverage for suite evaluation by developers,” http://eecs.osuosl.org/rahul/icse2014/.

[12] ——, “Source for code coverage for suite evaluation by developers,” https://bitbucket.org/rgopinath/coverage-icse-2014.

[13] J. C. Munson and A. P. Nikora, “Toward a quantifiable definition of software faults,” in *International Symposium on Software Reliability Engineering*. IEEE, 2002, pp. 388–395.

[14] M. Daran and P. Thevenod-Fosse, “Software Error Analysis: A Real Case Study Involving Real Faults and Mutations,” in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, New York, USA: ACM Press, 1996, pp. 158–171.

[15] L. J. Morell, “A Theory of Fault-Based Testing,” *IEEE Transactions on Software Engineering*, vol. I, no. 9036264, pp. 844–857, 1990.

[16] A. J. Offutt, “Investigations of the Software Testing Coupling Effect,” *ACM Transactions on Software Engineering and Methodology*, pp. 1–21, 1992.

[17] Y. Jia and M. Harman, “An Analysis and Survey of the Development of Mutation Testing,” *IEEE Transactions on Software Engineering*, pp. 1–31, 2010.

[18] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," in *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, 2000.

[19] A. P. Mathur, "Foundations of Software Testing," in *Foundations of Software Testing*, 2012.

[20] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale University, New Haven, CT, USA, 1980, aAI8025191.

[21] P. Ammann and J. Offutt, *Introduction to Software Testing*, 1st ed. New York, NY, USA: Cambridge University Press, 2008.

[22] P. G. Frankl and E. J. Weyuker, "Provable improvements on branch testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 10, pp. 962–975, 1993.

[23] R. A. DeMillo, A. P. Mathur, W. E. Wong, P. Frankl, and E. Weyuker, "Some critical remarks on a hierarchy of fault-detecting abilities of test methods [and reply]," *IEEE Transactions on Software Engineering*, vol. 21, no. 10, pp. 858–863, 1995.

[24] R. DeMillo, "Program Mutation: An Approach to Software Testing." Georgia Institute of Technology, Tech. Rep., 1983.

[25] Edward A Youngs, "Human Errors in Programming," *International Journal of Man-machine Studies*, vol. 6, no. 3, pp. 361–376, 1974.

[26] T. A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Informatica*, vol. 18, pp. 31–45, 1982.

[27] A. J. Offutt, "A Mutation Carol : Past , Present and Future," *Information and Software Technology*, pp. 1–26, 2011.

[28] A. J. Offutt and J. H. Hayes, "A Semantic Model of Program Faults," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1996, pp. 1–12.

[29] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Criteria and Controlflow-Based Test Adequacy," in *International Conference on Software Engineering*, 1994, pp. 191–200.

[30] A. S. Namin and S. Kakarla, "The use of mutation in testing experiments and its sensitivity to external threats," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, New York, USA: ACM Press, 2011, p. 342.

[31] R. Gopinath, C. Jensen, and G. Alex, "Code coverage for suite evaluation by developers," in *International Conference on Software Engineering*, 2014.

[32] Y. Wei, B. Meyer, and M. Oriol, "Is branch coverage a good measure of testing effectiveness?" in *Empirical Software Engineering and Verification*, B. Meyer and M. Nordio, Eds. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 194–212.

[33] K. S. H. T. Wah, "A theoretical study of fault coupling," *Software Testing, Verification and Reliability*, no. March 1999, pp. 3–45, 2000.

[34] K. How Tai Wah, "An analysis of the coupling effect I: single test data," *Science of Computer Programming*, vol. 48, no. 2-3, pp. 119–161, Aug. 2003.

[35] A. J. Offutt, "The Coupling Effect : Fact or Fiction?" *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131–140, Nov. 1989.

[36] V. Debroy and W. E. Wong, "Insights on fault interference for programs with multiple bugs," in *International Symposium on Software Reliability Engineering*. IEEE, 2009, pp. 165–174.

[37] N. DiGiuseppe and J. A. Jones, "Fault interaction and its repercussions," in *IEEE International Conference on Software Maintenance*. IEEE, 2011, pp. 3–12.

[38] A. Mathur, "Performance, effectiveness, and reliability issues in software testing," *Computer Software and Applications Conference.*, pp. 0–1, 1991.

[39] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *The Journal of Systems and Software*, vol. 1212, no. 94, pp. 185–196, 1995.

[40] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," *International Conference on Software Engineering*, pp. 100–107, 1993.

[41] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM Transactions on Software Engineering and Methodology*, pp. 1–23, 1996.

[42] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for c," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 113–136, 2001.

[43] J. C. Maldonado, E. F. Barbosa, A. M. R. Vincenzi, and M. E. Delamaro, "Evaluating N-selective mutation for C programs," *Mutation Testing for the New Century*, 2001.

[44] A. S. Namin and J. H. Andrews, "Finding Sufficient Mutation Operators via Variable Reduction," in *Workshop on Mutation Analysis*, 2006.

[45] A. S. Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient Mutation Operators for Measuring Test Effectiveness," in *International Conference on Software Engineering*, 2008, pp. 351–360.

[46] R. H. Untch, "On reduced neighborhood mutation analysis using a single mutagenic operator," in *ACM Annual Southeast Regional Conference*, 2009, pp. 1–4.

[47] L. Deng, A. J. Offutt, and N. Li, "Empirical Evaluation of the Statement Deletion Mutation Operator," in *International Conference on Software Testing, Verification and Validation*, 2013.

[48] M. E. Delamaro, A. J. Offutt, and P. Ammann, "Designing Deletion Mutation Operators," in *International Conference on Software Testing, Verification and Validation*, 2014, pp. 1–10.

[49] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies : An empirical study," *Software Testing, Verification and Reliability*, pp. 1–22, 1999.

[50] L. Zhang, S.-s. Hou, J.-j. Hu, T. Xie, and H. Mei, "Is Operator-Based Mutant Selection Superior to Random Mutant Selection ?" *The Journal of Systems and Software*, 2010.

[51] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-Based and Random Mutant Selection : Better Together," *IEEE/ACM ASE*, pp. 92–102, Nov. 2013.

[52] M. Patrick, M. Oriol, and J. A. Clark, "Messi: Mutant evaluation by static semantic interpretation," in *International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 711–719.

[53] M. Patrick, R. Alexander, M. Oriol, and J. A. Clark, "Probability-based semantic interpretation of mutants," in *Workshop on Mutation Analysis*, 2014.

[54] R. Kurtz, P. Ammann, M. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *Workshop on Mutation Analysis*, 2014.

[55] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, Oct. 2009.

[56] ——, "Constructing subtle faults using higher order mutation testing," in *IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2008, pp. 249–258.

[57] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the cost of mutation testing with second-order mutants," *Software Testing, Verification and Reliability*, vol. 19, no. 2, pp. 111–131, 2009.

[58] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 2010, pp. 300–309.

[59] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, pp. 371–379, Jul. 1982.

[60] I. Moore, "Strong vs weak mutation testing," http://ivan.truemesh.com/archives/000571.html.

[61] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *Software Testing, Verification and Analysis*. IEEE, 1988, pp. 152–158.

[62] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation Analysis Using Mutant Schemata," in *ACM SIGSOFT Software Engineering Notes*, 1993.

[63] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[64] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using conditional mutation to increase the efficiency of mutation analysis," *International Workshop on Automation of Software Test*, p. 50, 2011.

[65] M. A. Cachia, M. Micallef, and C. Colombo, "Towards incremental mutation testing," *Electronic Notes in Theoretical Computer Science*, vol. 294, pp. 2–11, 2013.

[66] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 331–341.

[67] A. S. Namin, P. St, and J. H. Andrews, "The Influence of Size and Coverage on Test Suite," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2009, pp. 57–67.

[68] R. Sharma, "Guidelines for coverage-based comparisons of non-adequate test suites," Thesis, University of Illinois Urbana-Champaign, 2013.

[69] L. Inozemtseva and R. Holmes, "Coverage Is Not Strongly Correlated With Test Suite Effectiveness," in *International Conference on Software Engineering*, 2014.

[70] A. J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1998.

[71] D. Schuler and A. Zeller, "Covering and Uncovering Equivalent Mutants," *Software Testing, Verification and Reliability*, no. April 2012, pp. 353–374, 2013.

[72] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do Redundant Mutants Affect the Effectiveness and Efficiency of Mutation Analysis?" in *International Conference on Software Testing, Verification and Validation*. Ieee, Apr. 2012, pp. 720–725.

[73] P. Ammann, M. E. Delamaro, and A. J. Offutt, "Establishing Theoretical Minimal Sets of Mutants," in *International Conference on Software Testing, Verification and Validation*, 2014.

[74] F. Belli, C. J. Budnik, and W. E. Wong, "Basic Operations for Generating Behavioral Mutants," *International Symposium on Software Reliability Engineering Workshops*, pp. 9–9, Nov. 2006.

[75] F. Belli, N. Güler, and A. Hollmann, "Model-Based Higher-Order Mutation Analysis," *Advances in Software Engineering*, pp. 164–173, 2010.

[76] M. Harman, Y. Jia, and W. B. Langdon, "A Manifesto for Higher Order Mutation Testing," in *International Conference on Software Testing, Verification and Validation Workshops*. Ieee, Apr. 2010, pp. 80–89.

[77] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *International Conference on Software Testing, Verification and Validation*, ser. ICST '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 90–99.

[78] M. Papadakis, C. Henard, and Y. L. Traon, "Sampling program inputs with mutation analysis: Going beyond combinatorial testing," in *International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2014.

[79] A. Rajan, M. W. Whalen, and M. P. Heimdahl, "The effect of program and model structure on mc/dc test adequacy coverage," in *Proceedings of the 30th International Conference on Software Engineering*, ser. International Conference on Software Engineering. New York, NY, USA: ACM, 2008, pp. 161–170.

[80] J. Duraes and H. Madeira, "Definition of software fault emulation operators: A field data study," in *International Conference on Dependable Systems and Networks*. IEEE, 2003, pp. 105–114.

[81] K. Pan, S. Kim, and E. J. Whitehead Jr, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[82] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2001, p. 170.

[83] C. Pacheco and M. D. Ernst, "Randoop random test generation," http://code.google.com/p/randoop.

[84] H. Coles, "Pit mutation testing," http://pittest.org/.

[85] V. Roubtsov and Others, "Emma - a free java code coverage tool," http://emma.sourceforge.net/.

[86] M. Doliner and Others, "Cobertura - a code coverage utility for java." http://cobertura.github.io/cobertura.

[87] R. Schmidberger and Others, "Codecover - an open-source glass-box testing tool," http://codecover.org/.

[88] R. Liesenfeld, "Jmockit - a developer testing toolkit for java," http://code.google.com/p/jmockit/.

[89] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *Software Engineering, IEEE Transactions on*, vol. 38, no. 2, pp. 278–292, 2012.

[90] R. Guderlei, R. Just, C. Schneckenburger, and F. Schweiggert, "Benchmarking testing strategies with tools from mutation analysis," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2008, pp. 360–364.

[91] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 297–298.

[92] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, "Ten simple rules for reproducible computational research," *PLoS computational biology*, vol. 9, no. 10, p. e1003285, 2013.