Evaluating Static Mutant Selection Techniques for Accurate Mutation Score Approximation

Magdalene Ashong*, Thomas Laurent, and Anthony Ventresque
School of Computer Science and Statistics, Trinity College Dublin, Ireland & Lero Research Centre
ashongm@tcd.ie, tlaurent@tcd.ie, anthony.ventresque@tcd.ie
*corresponding author

Abstract—Mutation analysis is known for its effectiveness in assessing the quality of test suites. However, it is a costly approach, as it generates many mutants even for small programs. Generating, compiling, and executing these mutants is a slow and resource-intensive process. Many mutant selection techniques have been proposed to reduce the number of mutants considered and thus lower the cost of mutation analysis. Yet, the effectiveness of all these techniques has not been systematically compared to understand the advantages of each technique and when they should be used. This work focuses on static mutant selection techniques (i.e., those that do not require executing tests against the mutants to select them) and compares their effectiveness in approximating the mutation score of a test suite. Using a dataset of 15 Java projects of different sizes and application domains and the LittleDarwin mutation tool, we compare the performance of ten state of the art static mutant selection techniques under different settings. Results show that no one technique provides better results than the others in all situations, i.e., across all projects and mutants sampling rates. Still, we found that stratification based selection techniques mostly outperform the other techniques (in up to 129 out of 135 of the studied settings). In particular, stratified sampling based on the source file in which the mutants appear provided the best approximation of the mutation score in nearly half the cases considered in our experiments (up to 68/135).
Additionally, we found that the quality of a project's test suite had a noticeable influence on the selection techniques' performance. Indeed, for lower quality test suites, the selected mutants performed worse and strongly under-estimated the mutation score.

Keywords–software engineering; software testing; mutation analysis; mutant selection; mutation score.

1. Introduction

Delivery of high quality software that meets specifications and user expectations is essential in software development environments. While testing software does not guarantee it is entirely fault-free, it is an essential part of the software development process and can identify errors and faults introduced and helps ensure a strong level of quality and reliability. Tests that are properly designed are more effective at revealing faults in a system.

Mutation analysis, a powerful fault-based test criterion, has proven its potential in assessing the quality of test suites [1]. It involves creating mutants, i.e., modified versions of the original program by injecting syntactic changes using mutation operators. Mutation operators are rules that define a single syntactic change to a program's source code, creating a simulated fault [2]. Mutants can contain multiple changes. The number of changes in a mutant is called its order. Mutants composed of one change are First Order Mutants (FOMs) and mutants containing multiple changes are called higher order mutants. A mutant is said to be killed when at least one of the tests in the test suite fails against it, i.e., the test detects the difference introduced by the syntactic change. The ratio of killed mutants to the overall number of mutants generated is the mutation score, serving as a measure of the test suites' thoroughness. A higher mutation score suggests that the test suite is more effective at detecting faults. Mutation analysis relies heavily on automation tools to generate and execute mutant programs. By analysing the results of mutation analysis, we can strengthen our test suites and reveal faults in the implementation code.
Even though mutation analysis has been proven to be very effective at assessing test suites' quality, it has not gained much prominence in the software industry. This stems from its computationally expensive nature, as many mutants are created and tests must be executed against these mutants. For example, a simple program such as the Triangle program [3] with 41 lines of source code can generate 51 first order mutants [4]. Besides the concern of generating many mutants, executing test suites against each mutant is also time consuming. Although some recent work [5], [6] has tried to use mutation analysis in industry settings, the large number of generated mutants still remains a core challenge. Indeed, these works suggest not using all mutants, either by focusing on mutants related to recent changes made to the code in the context of CI/CD pipelines [6], [7], or by limiting the number of mutants generated per line of code in the case of Google's experiments, a context where computational resources are abundant [5]. For this reason, researchers have proposed mutant selection techniques, which minimise the number of mutants to be executed. In this paper, we focus on the mutation score and we consider that a good subset of mutants gives a good approximation of the mutation score obtained using all mutants. However, we do not consider whether mutants that would have led to the addition of a new test to the suite are selected or not.
Previous studies on mutant selection techniques [8] have

explored ways to select a subset of mutants that are expected to evaluate the quality of a test suite as effectively as the complete set of mutants. This work focuses on static mutant selection techniques that do not require the tests to be run against mutants. In surveying the literature around mutant selection techniques, we found that Random Mutant Sampling (`RMS`) and Operator-based selection are two static techniques often used as a baseline for newly suggested techniques [9]–[11].

Although many studies have examined static mutant selection techniques, to the best of our knowledge, these techniques have not all been compared systematically. Inspired by the works of Gopinath et al. [12] and Zhang et al. [10], we carry out a systematic empirical study on ten static mutant selection techniques, using 15 Java projects and the Littledarwin mutation analysis tool [13]. In this paper, we focus on the performance of the ten static selection techniques when compared to each other in different settings. We use the accuracy of the mutation score obtained using selected mutants in approximating the one obtained using the full set of mutants as a proxy for the quality of the selected sets of mutants. We further evaluate the properties of a project (e.g., size, quality of the test suite) that influence the selection techniques' performance.

The contributions of this paper are outlined below: We compare ten static mutant selection techniques that optimise mutation analysis efficiency by selecting a subset of mutants from the original set. Results from the experiments conducted on 15 real-world Java projects show that stratified selection techniques [10], [14] are more likely to select mutants that represent the full set well and yield a mutation score that closely approximates the actual mutation score. Our experiments also show that how well a project is tested and the proportion of mutants selected is important in determining which selection technique performs best for that particular project.

The remainder of this paper is organised as follows: Section 2 gives an overview of the concepts underlying this work. Section 3 describes the experimental design of our study. Section 4 presents and analyses the findings from the experiments. Section 5 reviews related work. Section 6 covers the limitations and steps taken to address them. Finally, Section 7 concludes the paper and suggests future research directions.

2. Background Overview

This section provides background on mutation analysis, the mutation score, and mutant selection techniques.

2.1 Mutation Analysis

Mutation analysis is a very strong fault-based test criterion [1] that has been the topic of much work [15]–[18]. It involves introducing changes (mutations) to source code to create faulty versions (mutants) of the software. In practice, mutations are small syntactic changes, like changing a comparison from $<$ to $<=$. In mutation analysis, mutants can be classified based on test results: a failing test indicates a killed mutant while surviving mutants pass all tests. Surviving mutants suggest potential deficiencies in the test suite, prompting developers to strengthen their tests. The theoretical mutation score serves as a metric of the test suite's fault-detection capability and is defined as:

$$MS = \frac{K}{M - E} \times 100 \tag{1}$$

where K is the number of killed mutants; M is the total number of mutants generated and E is the number of equivalent mutants. The number of equivalent mutants – mutants that behave like the original program and cannot be killed, thus artificially lowering the mutation score – is impossible to know in practice and the mutation score is approximated by $\frac{K}{M}$.

Mutation analysis can be applied in various ways. The most common way to use mutation analysis, and the one this work focuses on, is to use the mutation score as a measure of the quality of a test suite. The mutation process can also be used to drive the creation of new tests by trying to create tests that kill mutants that are not currently killed by the test suite. This process is known as mutation testing [19]. Finally, mutation can also be used for test reduction, removing redundant tests in a suite that kill the same mutants as others in the suite [20] or for test prioritisation [21], indicating which tests are more likely to catch faults.

2.2 Static Mutant Selection Techniques

Despite mutation analysis being known for its effectiveness in assessing test suite quality, it faces the issue of high computational cost due to the large number of mutants generated. Generating and managing a large number of mutants is impractical and time consuming as each mutant must be compiled, executed and assessed by the test cases [22]. The mutants generated are not equally useful in improving the test suite's quality, some are easy to kill and multiple mutants can give the same information.

Several studies have explored ways to reduce the cost of mutation analysis by using fewer mutants. Notable among these techniques are "static" techniques such as Random Mutant Sampling (`RMS`) [23], [24], and selective mutation [25], [26] which do not require mutants to be executed. `RMS` randomly selects mutants from the entire set of mutants. Wong and Mathur [27] conducted an empirical study on this technique using the Mothra tool [28]. They found that examining only a small percentage of the mutants can be a practical way to evaluate test sets.

Operator-based selection, or selective mutation [25], [29], [30] is another approach to reduce the computational cost of mutation analysis. It involves choosing a specific subset of mutation operators and using only the mutants generated by these operators. Different sets of operators have been proposed [25], [29]–[31] for different languages or contexts to make mutation more efficient without sacrificing effectiveness.

Stratified mutant sampling [10], [14] first aggregates mutants into groups (strata) according to some given features and then randomly samples the different strata to perform the selection. An example of stratified sampling is to equally sample mutants created by the different mutation operators. This ensures that each operator is adequately represented in the selected set of mutants. Stratified sampling can also be

done across other dimensions such as line numbers where the mutations occurred, methods, functions, and classes of the program.

Other techniques such as search-based [32], [33] or machine learning [34] techniques have also shown promising results. However, these techniques rely on many test executions, either to select the mutants (e.g., subsumption-based fitness functions) or to learn the characteristics of strong mutants. These techniques are thus not static selection techniques and fall outside the scope of this study.

3. Empirical Study Design

This section describes the experiments conducted to compare the effectiveness of different mutant selection techniques when approximating the mutation score of projects using fewer mutants. First, it defines the research questions (RQs) under investigation in Section 3.1, then it describes the subjects and tools used for the experiments in Sections 3.2 and 3.3. Lastly, it details the selection techniques and the experimental procedure followed to answer the RQs in Sections 3.4 and 3.5.

3.1 Research Question

This work focuses on the following research questions:

RQ1 **Does a particular selection technique consistently perform the best?**
This RQ compares how the different selection techniques perform (how well the set of mutants they select approximated the mutation score obtained with the whole set of mutants) under the different settings of the experiment (programs mutated and sampling rate of mutants). The objective of this RQ is to understand whether a particular technique should be considered as the "default" static mutation techniques as it always provides better performance than the other techniques, or if practitioners should more carefully choose which technique to use.

RQ2 **What factors influence the performance of the techniques?**
This RQ examines the factors influencing the performance of the different selection techniques. It considers attributes of the system under test such as its size, or how well it is tested and how they impact the performance of the selection techniques.

3.2 Subject Projects

We based our experiments on a diverse set of Java projects used in other works [35]–[37]. Table I reports details on the selected projects and the developer-written tests they provide: the number of lines of code (LOC) and test cases (#Tests), the number of FOMs generated in the experiments (#FOMs), and the mutation score using all mutants (MS_a) for each project. For each project, the table contains a link to the commit used in our experiments. The number of test cases was reported by surefire [38] and the number of lines of code was measured using the MetricsReloaded plugin in IntelliJ [39]. These projects were chosen based on their different sizes and application domains. For example, Jodatime is a date and time manipulation library, and Net is a library that contains a collection of network utilities and protocol implementations. The number of lines of code (LOC) per project ranges from 442 to 48,115. The projects are all accompanied by a set of test cases designed to assess their functionality containing from 35 to more than 6000 tests.

TABLE I: Subject Projects' Details

Project	#LOC	#Tests	#FOMs	MS_a
Javancss	2,142	204	679	29.16
Jettison	3,987	115	1,654	42.02
Net	17,411	553	7,292	44.43
Jaxen	12,449	716	4,715	65.62
Javautil	16,527	6,825	8,709	72.99
Dbutils	2,978	321	1,527	73.35
Jgrapht-core	48,115	6,837	17,888	76.48
Antomology	442	35	246	76.83
Validator	7,899	990	3,498	78.82
Jodatime	29,950	4,232	16,170	83.96
Codec	9,527	1,723	5,652	85.21
Lang3	30,969	5,984	16,560	89.07
Text	10,601	1,392	6,302	90.75
Cli	2,151	438	1,200	92.33
Csv	2,179	849	1,089	97.25

3.3 The LittleDarwin Mutation Analysis Framework

In this work, we used the LittleDarwin [13] mutation tool which is a mutation analysis tool for Java programs. Similarly to the outdated MuJava [40], LittleDarwin generates mutants by modifying the Java source code of a program, unlike tools such as PIT [4] which generate mutants by modifying the bytecode. While this negatively affects the performance of mutation analysis it also leads to fewer mutants that are easier to understand for practitioners. The version of LittleDarwin used in our study (0.10.9) supports 14 operators: nine default mutation operators that cover the classic set of operators defined by Offutt et al. [25], and five experimental mutation operators which simulate null type faults and remove methods. Table II describes all 14 operators. We used the 14 operators to create all the FOMs for each project, which we use as the full set of mutants, not including higher order mutants in the experiments.

3.4 Selection Techniques

We compared ten different static mutant selection techniques in the experiments:

- **Random sampling (**`RMS`**)**: We implemented random sampling of mutants as described in Section 2.
- **Stratified sampling (**`SO`**,** `SL`**,** `SC`**)**: We implemented three stratified sampling approaches (see Section 2) using:
  - the mutant's operator (`SO`),
  - the line of code that the mutant is on (`SL`), and
  - the Java source code file the mutation appears in (`SC`).

TABLE II: LittleDarwin Mutation Operators

Operator	Description
AOR-B	Replaces a binary arithmetic operator
AOR-S	Replaces a shortcut arithmetic operator
AOR-U	Replaces a unary arithmetic operator
COD	Removes a unary conditional operator
COR	Replaces a binary conditional operator
NIV	Replaces a method's object reference by null
LOR	Replaces a logical operator
NOI	Replaces a new statement with null
NRV	A method returns an object, replaces it by null
RM	Removes the contents of a method
RNC	Replaces a null check by true or false
ROR	Replaces a relational operator
SAOR	Replaces a shortcut assignment operator
SOR	Replaces a shift operator

This last method is used as an approximation of class based sampling, as LittleDarwin does not report on the class a mutant is in. This approximation only affects nested classes.

- **Selective Mutation** ($\texttt{OG}_1$, $\texttt{OG}_2$, $\texttt{OG}_3$): We created three mutation operator groups based on the operator categories defined by LittleDarwin [13]. First, we merged the null type and RemoveMethod operators as one group, operator group one ($\texttt{OG}_1$). Then, the default and RemoveMethod operators as operator group two ($\texttt{OG}_2$). Finally, the default and null type operators as operator group three ($\texttt{OG}_3$). Table III shows the operators in each group. Note that no operator group is a subset of another.
- **Hybrid Selection** ($\texttt{HG}_1$, $\texttt{HG}_2$, $\texttt{HG}_3$): We investigated three hybrid selection strategies which combine the operator-based selection and random sampling [10] techniques: $\texttt{HG}_1$, $\texttt{HG}_2$, and $\texttt{HG}_3$, i.e., $\texttt{OG}_1$, $\texttt{OG}_2$, and $\texttt{OG}_3$ followed by RMS respectively.

3.5 Experimental Procedure

To answer the RQs, we executed the seven selection techniques that integrated an element of random sampling using a sampling rate ranging from 10% to 90% in steps of 10%. We ran each technique on each project with each sampling rate independently 1000 times (i.e., $7 * 9 * 15 * 1,000 = 945,000$ runs in total) to address the inherent variability in randomly selected samples and to obtain a more reliable estimate of the methods' performance. Indeed, techniques involving stochastic elements should ideally be evaluated through multiple repetitions to ensure reliability of the results [41], [42]. The three operator-based techniques ($\texttt{OG}_1$, $\texttt{OG}_2$ and $\texttt{OG}_3$) do not have a sampling rate parameter and we did not run 1000 times as they are deterministic, i.e., they lead to 45 runs in total.

3.6 Evaluation metrics

The aim of these experiments is to evaluate how well the different sets of mutants produced by the mutant selection techniques approximate the mutation score of a test suite obtained using all mutants generated from a project. To evaluate this we focus on one metric, the relative mutation score error, defined as follows:

$$RE = \frac{|MS_p - MS_a|}{MS_a}$$

Where MS_p is the approximation of the mutation score obtained from the selected mutants, and MS_a is the mutation score obtained using all mutants generated for the project. The mutation score is computed following the definition in Equation 1. Considering the amount of mutants generated in these experiments and the considerable cost of determining if a mutant is equivalent or not [43], all surviving mutants are considered as killable (i.e., not equivalent) in these experiments, making $E = 0$. Equivalent mutants are out of scope in this study, as our goal is to evaluate how well selected sets of mutants approximate the mutation score obtained with the whole set, rather than evaluating their adequacy (e.g., in terms of subsumption or potential for test generation), and equivalent mutants' effect is consistent across full and the selected sets.

3.6.1 RQ1

RQ1 explores whether one method consistently provides a better approximation of MS_a than another. Thus, it is concerned with the ranking of the different techniques more than their actual performance.

To answer this research question, for each of the iterations on all the projects, and selection rates, we first ranked the selection methods based on the relative mutation score error they returned. For each iteration, the method that provided the smallest RE is ranked first, then the next, etc.

Note that the operator-based techniques are only considered for one of the sampling rates for each project, the closest bracket to the proportion of mutants they used, rounded down. For example, if a combination of operators selects 52% of mutants generated for a project, the technique is only ranked for selection rate 50% for this project. For each of the 1000 iterations, this operator-based technique has the same relative error.

For each (project, selection rate) pair this provides us with 1000 rankings of the different methods. We aggregate these rankings into a single ranking of selection techniques for each pair using the Borda count method [44], a heuristic method to aggregate full rankings. We use this method to determine the ranking of the methods in the different settings. Indeed, this RQ concerns the best method, and does not consider effect size.

In order to provide a more complete view of the results we also consider the best selection method using the MdRE metric introduced in Section 3.6.2 in the different settings. MdRE is used because it is more robust to outliers than the mean relative error.

TABLE III: Mutation Operator Groups

Name	# Operators	Mutation Operators
OG_1	5	NIV, NOI, NRV, RM, RNC
OG_2	10	AOR-B, AOR-S, AOR-U, COD, COR, LOR, ROR, SAOR, RM , SOR
OG_3	13	AOR-B, AOR-S, AOR-U, COD, COR, LOR, NIV, NOI, RNC, NRV, ROR, SAOR, SOR

3.6.2 RQ2

This research question focuses on the techniques' performance in the settings we consider, and what factors influence different techniques' performance.

To evaluate how well the ten selection techniques approximate the mutation score of the complete set of mutants, we use the median relative error (MdRE), i.e., the median of the RE across the 1000 repetitions of each technique in each setting. As with the rankings, the operator-based techniques' RE has the same value for each repetition. We use the median rather than the mean because of its robustness to outliers. Based on our definition of mutant selection quality, MdRE values close to zero indicate that the sets of mutants selected by a selection technique closely match the actual mutation score, suggesting the selection technique is effective in that setting. Alternatively, a higher MdRE signals that the selection technique's chosen mutants do not accurately represent the full set of mutants.

4. Results

This section presents the results of the experiments and the corresponding findings according to the research questions highlighted in Section 3.1. Full results and materials to reproduce them are made available online [45].

4.1 RQ1 - Best Performing Technique

Table IV shows, for all project/sampling rate pairs, the selection technique that achieved the best performance, as measured by the Borda count (Table IVa) and by the MdRE (Table IVb). When a tie occurred, it lists all methods.

In Table IVa, SC, SL and SO perform well, with SC providing the most accurate mutation score 67 times (50%) out of the 135 cases , SL 35 times (26%), and SO 27 times (20%) respectively. Results also show that none of the techniques is the best for a given project or a given sampling rate, except for Jodatime, for which SC consistently did better than the other techniques across all the sampling rates.

The selective mutation techniques ($\texttt{OG}_{1-3}$) rarely performed best, appearing only five times (4%) in Table IVa altogether. RMS appeared only once, and the hybrid selection techniques never performed best, which can be explained by the low number of mutants they use as they re-sample the selective mutation techniques' selections.

In Table IVb, SC, SL and SO also performed well, with SC providing the most accurate mutation score 68 times (50%) out of the 135 cases, SL 37 times (27%), and SO 25 times (19%) respectively. Again, the selective mutation techniques ($\texttt{OG}_{1-3}$) altogether and RMS rarely performed best, each appearing only four times (3%). As in Table IVa, the hybrid techniques never performed best.

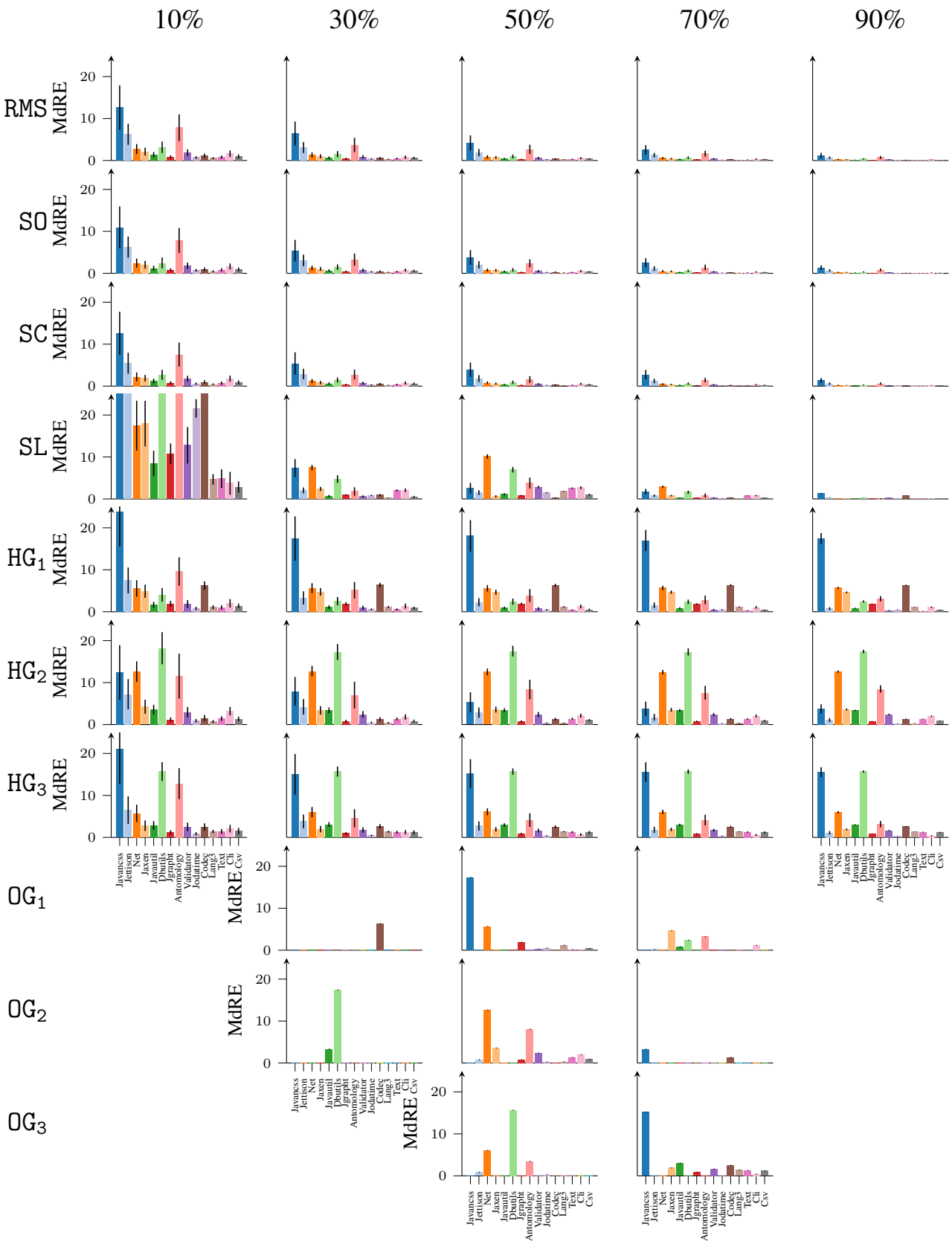
These results, shown by both aggregation methods, indicate that static selection techniques relying on stratification (SC, SL and SO) perform better than selective mutation ($\texttt{OG}_{1-3}$), random sampling RMS, and the hybrid selection techniques ($\texttt{HG}_{1-3}$) overall. However, this does not hold for all cases, with selective mutation and random selection sometimes providing the best results. Furthermore, the best technique among the stratified techniques varies between settings. RQ2 examines these variations and explores the factors that influence the selection techniques' performance.

4.2 RQ2 - Factors influencing the techniques' performance

Table IV shows that, for a given project, a different method might be the best depending on the chosen sampling rate. It also shows that at a fixed sampling rate, the best performing sampling technique varies per project. We thus investigated characteristics of the setting (sampling rate and project) that influence the methods' performance. Figure 1 displays the MdRE calculated for the ten selection techniques across the 15 projects for sampling rates 10%, 30%, 50%, 70%, and 90%. As described in Section 3.6.1, the MdRE for $\texttt{OG}_1$, $\texttt{OG}_2$ and $\texttt{OG}_3$ is displayed in the closest bracket to the proportion of mutants they selected. The sampling rate for the hybrid techniques is that of the random selection performed after the operator-based selection. Therefore, it does not correspond to the proportion of mutants in the project that the selection includes, but is higher than that. For example, the total number of mutants for Antomology using all operators is 246 while after applying $\texttt{OG}_1$ it is 199 (80%). Hence, $\texttt{HG}_1$ at 10% for Antomology uses 8% of all the mutants.

4.2.1 Testedness

Figure 1 shows that the different selection techniques struggle on projects like Javancss and Jettison, which are poorly tested compared to the other projects – having the lowest mutation scores of 29.16 and 42.02, respectively. This can be explained by the low number of killed mutants and thus selecting a percentage of the mutants highly selects mutants that survived, which leads to underestimation of the mutation score. Hence, this produces the higher MdRE for these projects. However, for well tested projects such as Cli and Csv – with the mutation scores 92.33 and 97.25 respectively – we observe lower MdRE for all the techniques as the mutant selections are more representative of the full set of mutants. However, the use of the relative error of the mutation score might skew these results. Indeed, given the same absolute error, projects with



(for SL, sampling rate 10%, MdRE = 114.3%; 100%; 100%; 100%; 48.57% for Javancss, Antomology, Dbutils, Jettison, Codec respectively, with $stddev = 0; 0; 0; 0; 3.63$)

Figure 1: MdRE (%) for the selection techniques on each project for different selection percentages with error bars (length = stderr)

TABLE IV: Best Selection Technique for each Project at each Sampling Rate

(a) Using the Borda Count

Project	Sampling Rate (%)								
	10	20	30	40	50	60	70	80	90
Javancss	SC	SC	SC	SL	SL	SL	SL	SO	SL
Jettison	SC	SL	SL	SL	OG_2	SL	OG_1	SL	SL
Net	SC	SC	SC	SC	SC	SC	SC	SC	SL
Jaxen	SC	SC	SC	SC	SL	SC	SC	SC	SL
Javautil	SO	SC	SC	SL	SC	SL	SL	SC	SL
Dbutils	SO	SC	SC	SC	SO	SC	SO	SC	SL
Jgrapht	SC	SC	SC	SC	SC	SL	SC	SL	SC
Antomology	SC	SC	SL	SC	SC	SL	SL	SC	SL
Validator	SC	SC	SL	SC	SC	OG_1	SL	SC	SC
Jodatime	SC	SC	SC	SC	SC	SC	SC	SC	SC
Codec	SO	SO	SO	SO	SL	SO	SO	SL	SO
Lang3	SC	SC	SL	SC	SC	SC	SL	SC	SL
Text	SC	SC	SO	RMS	OG_1	SO	SO	SO	SL
Cli	SO	SC	SO	SO	SO	SC	OG_3	SC	SL
Csv	SC	SO	SO	SO	SO	SO	SO	SO	SL

(b) Using MdRE values

Project	Sampling Rate (%)								
	10	20	30	40	50	60	70	80	90
Javancss	SO	SC	SC,SO	SL	SL	SL	SL	SO	RMS
Jettison	SC	SL	SL	SL	OG_2	SL	OG_1	SL	SL
Net	SC	SC	SC	SC	SO	SC	SC	SC	SL
Jaxen	SC	SC	SC	SC	SC	SC	SC	SC	SL
Javautil	SC	SC	SC	SL	SC	SL	SL	SC	SL
Dbutils	SO	SC	SC,SO	SC	SO	SC	SO	SC	SL
Jgrapht	SC	SC	SC	SC	SC	SL	SC	SL	SC
Antomology	SC	SC	SL	SL	SC	SL	SL	SC	SL
Validator	SC	SC	SL	SC	SC	OG_1	SL	SC	SC
Jodatime	SC	SO	SC	SC	SC	SC	SC	SC	SC
Codec	SO	SO	SO	SO	SL	SO	SO	SL	SO
Lang3	SC	SC	SL	SC	SC	SC	SL	SC	SL
Text	SC	SO	SC	RMS	OG_1	SO	SC	SO	SL
Cli	RMS,SO	SO	SC	RMS,SC	RMS	RMS,SC	SC	RMS,SO	SL
Csv	SC	RMS,SC,SO	SL	SO	RMS,SO	SL	SL	RMS	SL

smaller MS_a will yield higher relative errors (RE). Therefore, this aspect of the results will require further investigation in future work.

For projects that are not as well tested, however, other techniques, such as coverage-based test selection [46] might provide faster mutation analysis results while using all mutants. This would remove the need to approximate the mutation score but require the collection of coverage data, which could be complex depending on the way a project is built.

4.2.2 Project Size

The size of the projects, however, showed no distinct effect on the performance of the selection techniques. However, one noticeable exception was Antomology, which was the smallest project and contained only 246 mutants. The selection techniques yielded high error values, with the selected sets of mutants being very small, and a single mutant's status having a high influence on the mutation score. Furthermore, in such cases where projects are small enough, selection may be unnecessary. For example, mutation analysis for Antomology was completed in under an hour during our experiments.

4.2.3 Sampling rate

One would expect that the higher the sampling rate, the lower the error. Results mostly follow this pattern, however, that is not always the case. Indeed, we see a different trend in some cases. For example, for Jodatime, SL shows better performance at 30% ($\texttt{MdRE} = 0.849\%$) than at 50% ($\texttt{MdRE} = 1.508\%$) and better performance at sampling rate 70% ($\texttt{MdRE} = 0.158\%$) than at 90% ($\texttt{MdRE} = 0.173\%$). Antomology also shows better performance at 30% ($\texttt{MdRE} = 1.790\%$) than at 50% ($\texttt{MdRE} = 3.795\%$). We observe a similar surprising pattern with SL, and the hybrid selection methods on some projects. For $\texttt{HG}_1$ and $\texttt{HG}_2$, Antomology showed better performance at 70% ($\texttt{MdRE} = 2.806\%$) than at 90% ($\texttt{MdRE} = 3.103\%$) and at 30% ($\texttt{MdRE} = 7.029\%$) than 50% ($\texttt{MdRE} = 8.338\%$) respectively.

This result means that, for some projects, some techniques might produce a more representative selected set of mutants using a small sample size than using a larger sample size. When considering stratified selection based on line numbers (SL), this pattern might be the result of the unbalanced distribution of mutants across lines. For instance, Jodatime has a lot of strata with only one or two mutants and its biggest stratum contains 12 mutants. As the selection rate goes up, the lines with more mutants start being over-represented in the selected data, which can impact the approximated mutation score, For example, at a selection rate of 20%, a line with four mutants will contribute zero or one mutant, the same as a line with one mutant, while it could contribute three mutants at a selection rate of 70%, three times more than the line with one mutant.

A noticeable pattern is the bad performance of SL at 10% sampling rate, this is explained by the strata not only being unbalanced, but also being typically very small (each line of code contains only a few mutants), which means no mutants is selected from these strata at 10%. For example, the MdRE for Javancss is 114.33%. Interestingly for this setting the 1000 runs resulted in the same MS_p of 62.5%, despite the randomness of the process. When using SL, Javancss has only five strata containing more than five mutants, only one containing 42 mutants, two containing nine mutants and two containing 7 mutants. When using SL at 10%, four mutants are selected from the first stratum, and one from each of the other strata. Although the selected mutants vary, all mutants of each stratum are killed or survived (all mutants in a stratum have the same status), leading to the same approximated mutation score MS_p and error RE.

4.3 Global Analysis

Overall, the results demonstrate that the stratified techniques outperform the other techniques in our studied settings. However, since there are three distinct methods within the stratification selection and their performance varies depending on the setting, it is not clear which method is the best choice for a given project. This highlights the need for further analysis to better understand the project characteristics to consider when choosing which selection technique to use for a specific project. Also, for well tested projects, the selection techniques perform better, producing lower RE values than less well tested projects (as shown by MS_a in our experiments). Additionally, selecting too few mutants (i.e., using a low sampling rate) produces very imprecise and unreliable approximations of the mutation score and therefore should be avoided.

5. Related Work

Reducing mutation analysis' cost has lead to the development of various mutant selection approaches. This section presents other works comparing several state-of-the-art mutant selection techniques, focusing on their impact on both mutation score and efficiency.

Gopinath et al. found that random sampling techniques are more effective than operator selection methods [12] in approximating mutation scores and that only limited improvement is theoretically possible on its effectiveness [47]. In a subsequent study, they also found that the mutant selection techniques considered (many forms of operator-based and stratified sampling on operators and program elements) often yield minimal improvements in mutation effectiveness —- usually under 5% —- compared to random sampling [48]. They concluded that random sampling is a more reliable approach for assessing test suite quality and questioned the need for complex methods. Their work emphasises the limitations of current methods and suggests that simpler approaches may be more beneficial. However, their research was limited to random sampling, operator-based, and stratified (on operators and program elements) selection, leaving hybrid approaches [10] unexplored. Furthermore their work considered bytecode level mutants generated by PIT, while we consider source code mutants which are fewer in number and more readable by practitioners.

Zhang et al. [9], empirically evaluated three operator-based mutant-selection techniques [25], [29], [31] against random mutant selection to assess how good the selected mutants from these three operator-based techniques are on average and also how likely the selected mutants can be very bad. Their results showed that none of the three operator-based mutant selection techniques considered is superior to random mutant selection. This finding suggests that random mutant selection may be a better alternative to operator-based selection techniques. Nonetheless, their work did not address stratified mutant selection techniques, which we include in this work and can also leverage mutant's mutation operators (e.g., in SO).

Zhang et al. [10] investigated the effectiveness of combining operator-based mutant selection with random sampling (HG in this work) to enhance mutation analysis' efficiency. They also compared various stratified sampling strategies, including those based on program elements (e.g., the method the mutant is in, the statement, the class of the mutant) and mutation operators, across 11 real-world Java programs. Their findings on stratified sampling technique revealed that sampling across program elements is more effective than random sampling or operator-based selection methods. They further confirm that stratified sampling levels for certain program elements can be either overly fine grained or too coarse, making it select no mutant at all and this is also consistent with

our findings on stratified sampling over line numbers. Their findings also revealed that sampling as little as 5% of the mutants generated by operator-based selection could still yield highly precise mutation score estimations, particularly benefiting larger programs. Overall, the study concluded that random mutant sampling strategies based on program elements are more effective than operator-based selection strategies, especially for larger programs. These results confirm our own findings but were obtained using the Javalanche [49] mutation tool, which mutates bytecode rather than source code (which can lead to hard to interpret mutants) and contains different mutation operators to LittleDarwin.

6. Threats to Validity

This section describes threats to the validity of our work and the steps we took to limit them.

A first threat to the validity of this work is the limited dataset (15 Java projects). This limitation is due to the cost of mutation analysis. Even with this relatively small dataset our experiments considered more than 90,000 mutants and 20,000 tests across all projects. In order to limit this threat to the generalisability of our results we considered projects of different sizes, covering different application domains, and that were used in previous work.

Another threat to the generalisability of our results resides in the choice of LittleDarwin as the mutation tool used for these experiments. Results could differ if using another tool, as not all tools produce the same mutants, or even operate on the source code (some mutate the bytecode). Still, LittleDarwin's set of mutation operators covers most classic mutation operators [35], used in the majority of mutation analysis tools.

Finally, the metrics used in this work could be not adapted to the questions we seek to answer. This work focuses on the capacity of different static mutant selection techniques to accurately approximate the mutation score obtained without any selection of mutants. The relative error RE, which reflects the distance between the two thus is a relevant metric. Many of the techniques under study involving a degree of randomness, they required to be run multiple times. We used both the Borda count method and the median to aggregate these repeated results, two widely used methods.

7. Conclusion and Future Work

This work evaluates ten static mutant selection techniques and their ability to accurately approximate the mutation score obtained using all mutants on 15 open source projects, at nine different levels of sampling (i.e., considering different proportions of selected mutants). We measured how well the mutants selected by the different techniques approximate the mutation scores of the different projects using the relative error of mutation score. We aggregated results using the Borda count method and the median relative error.

Overall, the empirical results show that stratification based mutant selection techniques outperformed random sampling, selective mutation and the hybrid selection techniques. However, no one technique provided the best results in all studied cases.

Results also show that the quality of the test suite and the proportion of mutants selected impact the accuracy of the approximated mutation score. All techniques provided more accurate results on projects that were better tested, showing that the mutation score and mutant selection are better used on more mature test suites that are already (close to) adequate for less demanding coverage criteria. Furthermore, as a whole, using fewer mutants lead to less accurate results, although this trend was not always followed.

This work focuses on the accuracy of the mutation score approximation provided by the different mutant selection techniques. While this is a major factor to consider when using mutation selection, it does not reflect aspects such as fault coupling or test selection ability of the selected mutants. In future work, we plan to explore other metrics such as mutants' representativeness (i.e., the proportion of the minimal mutants set included in the selected mutants) to further explore the quality of the selected mutants sets. This will require information not yet provided by LittleDarwin which we are working on obtaining, in particular the complete list of tests that kill each mutant. Furthermore, we plan to expand our projects benchmark and explore the correlation between a project's features and the selection methods' performance in more depth to better understand when to use each technique.

Acknowledgment

This work was supported, in part, by Research Ireland grants 20/FFP-P/8818, and 13/RC/2094_P2 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Research Ireland Research Centre for Software (www.lero.ie). This work was also supported by RIT (Research IT, Trinity College Dublin).

References

[1] 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 *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 220–229.

[2] H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the c programming language," Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue ..., Tech. Rep., 1989.

[3] "GitHub - hcoles/triangle-example — github.com," https://github.com/hcoles/triangle-example, [Accessed 10-10-2024].

[4] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 449–452.

[5] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Practical mutation testing at scale: A view from google," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3900–3912, 2021.

[6] J. Örgård, G. Gay, F. G. de Oliveira Neto, and K. Viggedal, "Mutation testing in continuous integration: An exploratory industrial case study," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2023, pp. 324–333.

[7] M. Ojdanić, W. Ma, T. Laurent, T. T. Chekam, A. Ventresque, and M. Papadakis, "On the use of commit-relevant mutants," *Empirical Software Engineering*, vol. 27, no. 5, p. 114, 2022.

[8] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, p. 110388, 2019.

[9] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 435–444.

[10] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: Better together," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 92–102.

[11] J. A. P. Lima, G. Guizzo, S. R. Vergilio, A. P. Silva, H. L. J. Filho, and H. V. Ehrenfried, "Evaluating different strategies for reduction of mutation testing costs," in *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*, 2016, pp. 1–10.

[12] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "An empirical comparison of mutant selection approaches," *Journal of Systems and Software*, 2015.

[13] A. Parsai, A. Murgia, and S. Demeyer, "Littledarwin: a feature-rich and extensible mutation testing framework for large and complex java systems," in *Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers 7*. Springer, 2017, pp. 148–163.

[14] R. Gopinath, A. Alipour, A. Iftekhar, C. Jensen, and A. Groce, "Do mutation reduction strategies matter?" Oregon State University. Department of Computer Science, Tech. Rep., 2015.

[15] R. A. DeMillo, "Test adequacy and program mutation. software engineering, 1989," in *11th International Conference on*, 1989, pp. 355–356.

[16] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," *Mutation testing for the new century*, pp. 34–44, 2001.

[17] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[18] J. Offutt, "A mutation carol: Past, present and future," *Information and Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[19] B. H. Smith and L. Williams, "Should software testers use mutation analysis to augment a test set?" *Journal of Systems and Software*, vol. 82, no. 11, pp. 1819–1832, 2009.

[20] K. Kapoor and J. P. Bowen, "Ordering mutants to minimise test effort in mutation testing," in *International Workshop on Formal Approaches to Software Testing*. Springer, 2004, pp. 195–209.

[21] D. Shin, S. Yoo, M. Papadakis, and D.-H. Bae, "Empirical evaluation of mutation-based test case prioritization techniques," *Software Testing, Verification and Reliability*, vol. 29, no. 1-2, p. e1695, 2019.

[22] A. Ibias and M. Núñez, "Using a swarm to detect hard-to-kill mutants," in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2020, pp. 2190–2195.

[23] T. A. Budd, *Mutation analysis of program test data*. Yale University, 1980.

[24] A. T. Acree Jr, *On mutation*. Georgia Institute of Technology, 1980.

[25] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[26] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 205–232, 1999.

[27] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, 1995.

[28] R. A. DeMillo, D. S. Guindi, W. McCracken, A. J. Offutt, and K. N. King, "An extended overview of the mothra software testing environment," in *Workshop on Software Testing, Verification, and Analysis*. IEEE Computer Society, 1988, pp. 142–143.

[29] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 351–360.

[30] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *Proceedings of the 2013 international symposium on software testing and analysis*, 2013, pp. 224–234.

[31] 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.

[32] R. A. Silva, S. d. R. S. de Souza, and P. S. L. de Souza, "A systematic review on search based mutation testing," *Information and Software Technology*, vol. 81, pp. 19–35, 2017.

[33] A. S. Banzi, T. Nobre, G. B. Pinheiro, J. C. G. Árias, A. Pozo, and S. R. Vergilio, "Selecting mutation operators with a multiobjective approach," *Expert Systems with Applications*, vol. 39, no. 15, pp. 12 131–12 142, 2012.

[34] A. Garg, M. Ojdanic, R. Degiovanni, T. T. Chekam, M. Papadakis, and Y. Le Traon, "Cerebro: Static subsuming mutant selection," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 24–43, 2022.

[35] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. Le Traon, and A. Ventresque, "Assessing and improving the mutation testing practice of pit," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 430–435.

[36] D. Van Nho, N. Q. Vu, and N. T. Binh, "A solution for improving the effectiveness of higher order mutation testing," in *2019 IEEE-RIVF international conference on computing and communication technologies (RIVF)*. IEEE, 2019, pp. 1–5.

[37] J. Zhang, "Scalability studies on selective mutation testing," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 851–854.

[38] S. C. A. Ramirez, "Maven Surefire Plugin; Introduction — maven.apache.org," https://maven.apache.org/surefire/maven-surefire-plugin/, [Accessed 13-11-2024].

[39] Jetbrains, "MetricsReloaded - IntelliJ IDEs Plugin — Marketplace — plugins.jetbrains.com," https://plugins.jetbrains.com/plugin/93-metricsreloaded, [Accessed 02-11-2024].

[40] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: a mutation system for java," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 827–830.

[41] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[42] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[43] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability*, vol. 23, no. 5, pp. 353–374, 2013.

[44] S. Lin, "Rank aggregation methods," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 5, pp. 555–570, 2010.

[45] M. Ashong, T. Laurent, and A. Ventresque, "Companion repository," https://github.com/ComplexSoftwareLab/QRS_2025_Static_Mutant_Selection, 2025, [Accessed 19-06-2025].

[46] S. Vercammen, S. Demeyer, M. Borg, N. Pettersson, and G. Hedin, "Mutation testing optimisations using the clang front-end," *Software Testing, Verification and Reliability*, vol. 34, no. 1, p. e1865, 2024.

[47] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "On the limits of mutation reduction strategies," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 511–522.

[48] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Mutation reduction strategies considered harmful," *IEEE Transactions on Reliability*, vol. 66, no. 3, pp. 854–874, 2017.

[49] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 297–298.