PhD-FSTM-2023-054
Faculty of Science, Technology and Medicine

Dissertation

Presented on the 20/06/2023 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Miloš OJDANIĆ
Born on 23rd April 1996 in Cetinje, Montenegro

CODE-CHANGE AWARE MUTATION BASED TESTING IN CONTINUOUSLY EVOLVING SYSTEMS

Dissertation Defense Committee

Prof. Dr Michail Papadakis, Supervisor
University of Luxembourg, Luxembourg

Prof. Dr Yves Le Traon, Chairman
University of Luxembourg, Luxembourg

Prof. Dr Annibale Panichella, Co-Chairman
Delft University of Technology, Netherlands

Prof. Dr Gregory Gay, Member
Chalmers and the University of Gothenburg, Sweden

Prof. Dr Mauro Pezzè, Member
Università della Svizzera Italiana, Switzerland & Università di Milano Bicocca, Italy

Abstract

In modern software development practices, testing activities must be carried out frequently and preferably after each code change to bring confidence in anticipated system behaviour and, more importantly, to avoid introducing faults. When it comes to software testing, it is not only about what we are expecting; it is equally about what we are not expecting. Developers desire to test and assess the testing adequacy of the delta of behaviours between stable and modified software versions.

Many test adequacy criteria have been proposed through the years, yet very few have been placed for continuous development. Among all proposed, one has been empirically verified to be the most effective in finding faults and evaluating test adequacy. Mutation Testing has been widely studied, but its current traditional form is impractical to keep up with the rapid pace of modern software development standards and code evolution due to a large number of test requirements, i.e., mutants.

This dissertation proposes change-aware mutation testing, a novel approach that points to relevant change-aware test requirements, allows reasoning to what extent code modification is tested and captures behavioural relations of changed and unchanged code from which faults often arise. In particular, this dissertation builds contributions around challenges related to the code-mutants' behavioural properties, testing regular code modifications and mutants' fault detection effectiveness.

First, this dissertation examines the ability of the mutants to capture the behaviour of regression faults and evaluates the relationship between the syntactic and semantic distance metrics often used to capture mutant-real fault similarity. Second, this dissertation proposes a commit-aware mutation testing approach that focuses rather on change-aware mutants that bring significant values in capturing regression faults. The approach shows 30% higher fault detection in comparison with baselines and sheds light on the suitability of commit-aware mutation testing in the context of evolving systems. Third, this dissertation proposes the usage of high-order mutations to identify change-impacted mutants, resulting in the most extensive dataset, to date, of commit-relevant mutants, which are further thoroughly studied to provide the understanding and elicit properties of this particular novel category. The studies led to the discovery of long-standing mutants, demonstrated as suitable to maintain a high-quality test suite for a series of code releases. Fourth, this dissertation proposes the usage of learning-based mutant selection strategies when questioning how effective are the mutants of fundamentally different mutation generation approaches in finding faults. The outcomes raise awareness of the risk that the suitability of different kinds of mutants can be misinterpreted if not using intelligent approaches to remove the noise of impractical mutants.

Overall, this dissertation proposes a novel change-aware testing approach and provides insights for software testing gatekeepers towards more effective mutation testing in the context of continuously evolving systems.

It is nice to know the dictionary definition for the adjective "elegant", meaning "simple and surprisingly effective". Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better... If you deliver a lecture that is crystal clear from the beginning to end, your audience feels cheated and mutters while leaving the lecture hall "That was all rather trivial, wasn't it?"... In short: Two cheers for Elegance!

prof. dr. Edsger W. Dijkstra (On the nature of Computing Science)

And for Logic as the melody of Reason!

Acknowledgements

Well, this ought to be a rather lengthy acknowledgement, as the list of individuals that shaped me on this adventure deserve nothing less and even more than what my words can express. First and foremost, I owe gratitude to my supervisor Mike, for the opportunity and for investing time in me, hard work, profound scientific discussions that often went outside of my cognitive abilities, patience and pedagogical conversations to return me on track in times when I found myself lost in the darkness of self-doubt. Close to four years is a long time in the area of computer science, and the group of people that remained, left or visited our research lab deserves in-depth gratitude fitting into vastly longer paragraphs than what this short page can offer. Yet I would consider this dissertation vacant if I unintentionally miss mentioning even a single one of them; thus, I will not take that risk, but I desire to at least slightly express my gratefulness and make all my colleagues aware of their acceptance, inspiration, persistence, support, all knowledge they enriched me with and diverse - more often than not, deep - topics during all the lunches we had. On top of every group of researchers should stand a fine leader, at least near to the qualities of our professor. Prof. Yves Le Traon deserves an in-depth appreciation for the opportunity and devoted time, from the piece of wisdom to the piece of equipment he offers, treating everyone with care.

As the second part of this acknowledgement, and equally important, I owe an expression of love to my family and friends. My friends refused to be conquered by geographical distance and kept me close all these years; they succeeded in handling my curious - and what often can look like a restless - spirit, accepted me with all my flaws, and provided me nothing less but pure tenderness. My family counts many members, which I endlessly and equally treasure. I am very lucky to have them and no words can express vividly how close they are to my heart and repay them for all pieces of advice and support I received over the years. Finally, I want to acknowledge my brother Janko who keeps teaching me the art of heartfulness; my sister Ana who daily teaches the whole family the power of kindness; and my sister Sandra who, unaware, provides me invaluable lessons on the meaning of courage. From my father, Radomir, I have been learning about endurance in its purest, humble form. While my mother Zorka, taught me valuable lessons of devotion and the use of willpower in whatever I do. *"Of all things, just don't lose will Miloš"*, she used to say to me. If I can dedicate this dissertation to anyone, it would be my mother, which I believe still finds ways to protect me under her wing.

Miloš Ojdanić
Luxembourg City

Contents

List of Abbreviations

a.k.a	*Also Known As*
APFD	*Average Percentage of Fault Detected*
BLEU	*Bilingual Evaluation Understudy*
CI	*Continuous Integration*
CT	*Continuous Testing*
FDP	*Fault Detection Probability*
FOM	*First Order Mutants*
HOM	*High Order Mutants*
MS	*Mutation Score*
ML	*Machine Learning*
NLP	*Natural Language Processing*
NMT	*Neural Machine Translation*
RIPR	*Reachability, Infection, Propagation Reveilability*
RMS	*Relevant Mutation Score*
SOM	*Second Order Mutants*
SUT	*System Under Test*
TAC	*Test Adequacy Criteria*
TCP	*Test Case Prioritisation*
w.r.t.	*What Refers To*

List of Figures

List of Tables

1

Introduction

This chapter starts the dissertation with the context of the area in which the challenges are recognised and contributions offered. Foremost, it introduces software testing, more specifically mutation testing, while discussing the state of the technique in academia as in industry, where rapid incremental code-base evolution becomes rather standard of software development practices carrying specific challenges. This chapter clarifies the challenges identified and tackled by this dissertation. Afterwards, the chapter provides an overview of the contributions, whose detailed descriptions rest in further chapters. At the end of the chapter, the structure of the dissertation organisation is presented, aiming to ease the navigation and reading.

Contents

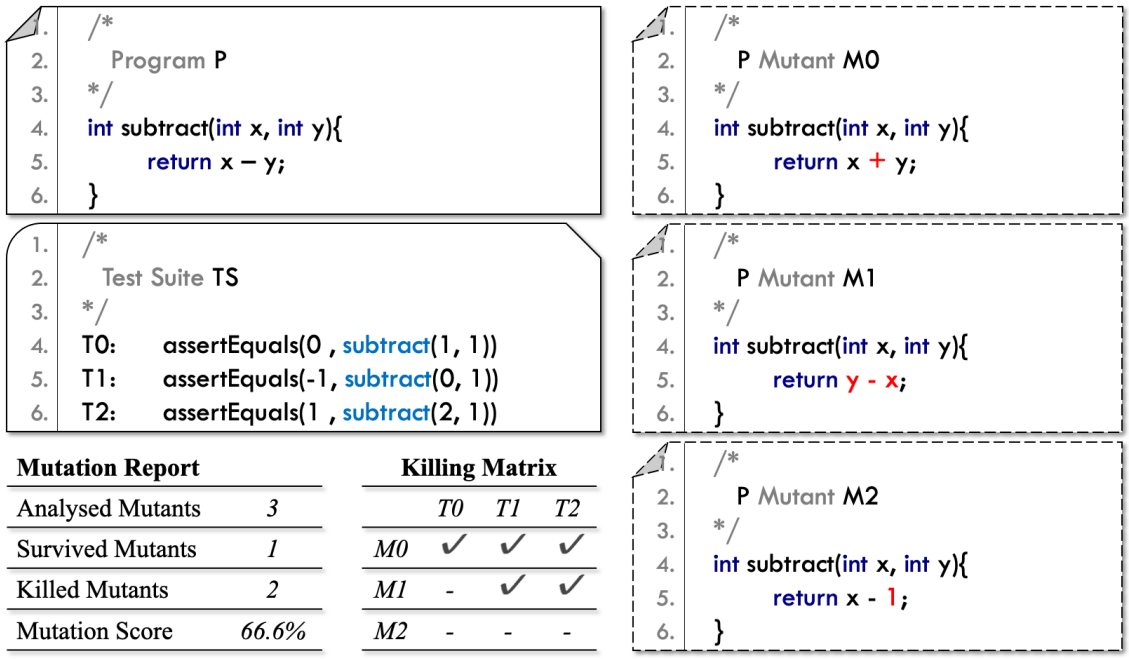
1.1 Introduction in Mutation Testing

On the doorway of software development stands a quality assurance gatekeeper that carries out software testing to preserve software correctness.

Among different software quality assurance activities, *software testing* is the mostly practice one [1, 2]. Software testing is performed by running/executing the corresponding software code with inputs and observing/evaluating whether the program, a.k.a system under test - SUT, behaves as expected. The aim is to uncover faults, i.e., unexpected behaviour that often occurs, early since if discovered late, the faults are often expensive to repair, e.g., resources invested in locating the fault or, as an even more hazardous scenario, the propagation of the fault has a potential to result in system failure leading to consequences for a human or nature [3, 4]. What sounds like a trivial process in practice is labour intensive task due to the complexity and diverse behaviours of the SUT, resulting in a very large system input space - if not infinite. To deal with the inability to perform exhaustive testing, the practitioners turn to different test objectives that lead to measurable feedback about the software's reliability, w.r.t., the likelihood of the software being correct [1,2]. Test objectives are often regarded as test adequacy criteria - TAC - and serve to guide the design of test suites, where a test triggers the execution of the program under certain test inputs and measures how well it covers/exercises code elements. Over the years, many different testing criteria have been thoroughly studied and include different forms of code coverage, building on the assumption that if the element of the code is covered by the test, that part of the code has been evaluated, i.e., its impact on the program output is taken into account. As mentioned, testing criteria are diverse, while some of the well-known, and broadly applied for evaluating the testing quality of the software measure exercised code statement coverage, branch coverage, i.e., leaving no code paths not executed, etc. The whole procedure is iterative, as it allows developers to continuously quantify the thoroughness of testing, and guides them to create new tests such as to lead to more coverage - evaluating diverse code behaviour - thus better approximating the correctness of SUT.

Among many criteria, mutation testing refers to using mutations, or slight code alteration, as TAC for test guidance and test quality evaluation [5]. These code alterations - better known as mutants - are often referred to as seeded artificial faults couple with real ones in a sense that test cases that reveal them also reveal real faults [6]. Thus, the test objective is to distinguish between the original program and a mutant. If a test is able to make this distinction, it is said to be valuable and of relative quality; otherwise, a mutant that escapes tests - survives - defines an objective that new tests should fulfil [7]. Therefore, test suites that distinguish more mutants are considered stronger than those that distinguish less mutants.

Mutation testing as a technique has been long present and recognised by the community [8]. The reason is that even being in its nature simple approach, which checks to what extent the tests capture seeded faults, it is appropriate to evaluate the adequacy of the test suites as mutations will result in numerous and various program outputs requiring evaluation. Hence, the technique is often seen through the lenses of a proverb, stating *"if there is a fault in the system, there will be a mutant corresponding to that fault"*. Indeed, over the years, many empirical evaluations confirmed that mutation testing is the strongest fault-revealing testing technique, leading to the strongest test suites [9]. The mutation testing is typically performed



Mutation Report

Analysed Mutants	3
Survived Mutants	1
Killed Mutants	2
Mutation Score	66.6%

Killing Matrix

	T0	T1	T2
M0	✓	✓	✓
M1	-	✓	✓
M2	-	-	-

Figure 1.1: An example of three mutants (M0, M1, and M2) generated out of the program P, with a test suite TS containing tests T0, T1 and T2 to assess P. The killing matrix depicts the ability of tests to distinguish between mutants and the original program, while the mutation report contains descriptive statistics of the analyses. Mutant M2 escapes tests from TS and requires a new test T3: `assertEquals(0, subtract(-1,-1)` to be distinguished and P adequately assessed.

systematically by altering simple language-grammar code elements since it is observed those form a coupling effect [6]. In short, the coupling effect declares that the software programs, in their nature, are "close to being correct", and test data that distinguishes all program versions differing from a correct one by only simple faults will be so sensitive that it also implicitly distinguishes more complex faults. Therefore, simple mutants faults can create erroneous behaviour as complex as those of real faults [10].

Simple mutation faults are introduced in the program under test using mutant operators, which serve as rules for how to transform an original program to a mutated faulty version. Each mutated faulty program version is a distinct instance of the original program whose behavioural difference needs to be identified by at least one test. The test suite is preferably augmented until the difference between every pair - original program, mutant - is recognised. For a developer to write a test that detects such a difference in the program output, it needs to understand the test requirement - the behaviour produced by the mutant - and the required test inputs to reach and recognise it. Hence, the goal of mutation testing is to evaluate test thoroughness by providing test adequacy criteria and guiding the generation of test cases against mutants. The metric often used is called mutation score and represents the ratio of mutants distinguished.

1.1.1 Mutation Testing through Examples

As previously mentioned, *mutants* are instances of the program under analysis containing simple syntactic modifications used to evaluate tests' ability to distinguish between what is original and what is altered [8].

For more practical representation, let us observe Figure 1.1, which illustrates mutants, $M0$, $M1$ and $M2$, respectively, and exemplifies the technique. These mutants represent the instantiated version of the original program P. In this particular case, mutant $M0$ performs addition instead of subtraction of parameters x and y. Mutant $M1$ alters the return statement, line 5, such as that instead of subtracting parameters x and y, it switches them and subtracts y and x, thus making original behaviour faulty. Mutant $M2$ replaces parameter y with a constant 1, such as to construct expression `x - 1`. The sole purpose of these particular mutants is to provide visualisation support of how expected behaviour diverges and how they serve to evaluate the adequateness of tests in identifying the inconsistency. In the example, the adequacy is evaluated on the provided test suite TS, which counts three tests, $T0$, $T1$ and $T2$, respectively. Each provided test asserts whether the program expected output equals actual output - `assertEquals(expected, actual)`. After analysing provided mutants by running tests on each, we create a killing matrix suggesting which tests distinguish, a.k.a., kill, which mutant. First, we observe that mutant $M0$ does not escape any tests - all tests identify differences in actual and expected behaviour. A similar scenario is observed when running tests on mutant $M1$; however, in this case, the mutant escapes test $T0$, which does not see it differently from the original program. The final observed mutant $M2$ is not identified by any tests - escapes all tests from TS. Therefore, we report that among three analysed mutants, TS kills two, while one mutant survives and escapes - making a mutation score *66.66%*. Hence, indicating the need for additional effort and the creation of tests that will identify overlooked potentially faulty behaviour under certain test inputs. From the example, we can conclude that one test case suitable to identify the remaining mutant is `assertEquals(0, subtract(-1,-1)` as its test input is suitable to distinguish between the mutant and original program. The mutant behaviour under these test inputs is -2 while the original program output is 0. Overall, with this new test, test suite TS gains on straight, testing diversity and thoroughness.

1.2 State of Mutation Testing in Academia

Mutation Testing has been extensively studied in the last decades with many advances and tools being developed [8]. Since its origin in the 70s, all up until today, the technique has attracted many practitioners and merged many communities as they all recognised that its potential reaches even beyond the field of software testing [8]. The ability of seeded faults to accurately characterise and mimic a set of potential real faults motivated researchers to employ mutants' versatile behaviours for extensive scientific experimentation. Consequently, this led to a high number of studies and contributions in domains such as test generation [11, 12], reduction and test case prioritisation [13, 14], test oracle creation [15], fault repair [16, 17], fault localisation [18], and many more. The impact of the technique still grows, and its best adaptable form is visible in the current well-studied area of machine learning, where a considerable number of studies emerge that use some form of mutation testing to evaluate artificial neural networks and the accuracy of their prediction [19, 20].

However, the state of mutation testing in academic circles is still mainly grounded in traditional software development. In traditional software development, the testing phase is regarded as one of the last phases of the development cycle to be completed before software release [21]. The whole testing phase is conducted with the assumption

that enough time and resources are allocated. Another assumption is that the development process will not suffer major disruption since software requirements are already pre-defined. Traditional mutation testing, built for this purpose, helps practitioners to evaluate the test adequacy of the entire code base in isolation utilising the power of diverse software behaviours brought by mutants. For this purpose, researchers proposed different mutation approaches, which further gave rise to the development of many mutation testing tools [22]. Yet nowadays, this is where the interest of academia and industry - the side that follows modern agile software development practices - diverges. Modern agile development practices encourage systems' continuous integration through building blocks of small chunks of code, highly dependent on user feedback and volatile market requirements [23]. Hence, testing activities in such dynamic environments are performed on the fly, adjusting to the user needs and optimising to minimise the lead time between opening a ticket for product change and the change being integrated with the software product that runs in production, providing its services. Therefore, in order to preserve the mutation testing impact, help the industry make use of its fault detection capability and bridge the gap between industry and academia interests, mutation testing needs to be adjusted to the development shift leading to the Integrated Evolving Systems.

1.2.1 Development Shift towards Integrated Evolving Systems

The progress of the technology related to overall software infrastructure, cloud-based and cloud-native software solutions made disruption of development software practices and rapidly spread in the majority of industry sectors. From the development of traditional monolithic and "functionally-complete" applications, the software engineering community is nowadays placed towards the development of software through agile build cycles that are best described as continuous [21]. The paradigm is to make and release a production-ready, minimally viable software product with initial services benefiting users as early as possible [23]. Subsequently, the product is revised through new features, integration, and testing; positioning its evolution to a continuous attempt to incorporate new requirements through code changes. In a remarkably short interval of time, this practice became a standard of software development, followed by many software solutions that help developers identify new change requests, develop required code faster, automate its integration and software testing, and minimise the time and effort required for deployment and release [21]. These chained activities allow for quick user feedback, accumulating requests for integration of new features, thus leading to advances in software scope and complexity; ultimately resulting in a constant battle of trade-offs between code change integration duration and code quality. Diverse existing solutions allow developers to track code changes and perform continuous testing, which stands as a gatekeeper to distinguish between stable code versions and error-prone versions impacted by the changes. Developers often build with the assumption that if all tests from the previous stable version successfully pass on to the newly altered version, the new code changes do not impact the stable program behaviour. Yet, to help developers reason about code change testing adequacy, continuous testing frequently incorporates test objectives as statement coverage - indicating whether the tests cover the statement on which code change occurs - accepting it as a means to prevent untested changes. However, covering changed lines does not imply adequately testing their impact. If we only

ask our test suite to validate what we want to test, i.e., what we are expecting, the question arises of what kind of new information about the changed system we acquire and what is the assurance of unexpected behaviour, i.e., developers desire confidence that the performed changes do not cause breakage of the existing behaviour and not to escape existing tests.

1.2.2 Mutation Testing Applications in Industry

Since almost the introduction of the technique, many studies have emerged from the industry of safety-critical systems reporting on the power of the technique to lead to the identification of real faults [24, 25]. Software development in this particular domain requires rigour quality evaluation and, due to the safety impact of the software, requires developers to spend additional time on activities such as testing - reasoning their effort through the proverb "go slower, go better". Although using mutation testing to specify testing objectives has been proven effective and unmatched, the reason its application has not been broadly recognised by the various industries that cherish the development of continuously evolving systems can be partly associated with the scope of mutations and their traditional untargeted nature. Yet, recently several reports emerged revealing the usage of technology and the potential benefits it brings in the incremental development practices, which often unofficially motivate builds through the proverb "go faster, go better". Moreover, Google reported the use of the technique to provide developers with a code mutation after each code change [26]. In this scenario, mutation locates on a changed code, serving the purpose of understanding the code, raising awareness of the potential faulty behaviour, and guiding it to generate tests that would identify the fault in case it actually occurred [27]. Following this practice helps developers augment the test suite and think in the direction of the impact of the change. On the other hand, Facebook, as another key member in the software industry, reported usage of mutation testing in the way of semi-automatically learning error-inducing patterns from their own code-base, and thus generating mutants which do not seem change-oriented but rather evaluating the strength of the entire test suite [28]. Moreover, they report and argue that in order to consider applying the technique on an industrial scale system, it is necessary to focus on small targeted deviations in the program, such as providing actionable signals to developers, which in the context of the evolving systems leads to the care of code change.

1.3 Challenges with Mutation Testing

Steadily and slowly, mutation testing is reaching its maturity. Although it provides effectiveness in its current traditional untargeted form, its potential for evolving systems must be surfaced since it has been long neglected due to challenging and unoptimised parts of the technique. This dissertation identifies such challenges and studies several specific characteristics intending to bring mutations one step closer to broad adoption in continuous software development. Figure 1.2 provides a visualisation of the challenges and serves as the dissertation storyline.

1.3.1 Mutants Behavioural Properties

At its core, mutation testing has been built on the premise of fault coupling, which states that small syntactic deviations lead to sensitive tests that identify complex real faults. Although popular, it has been criticised for producing unrealistic faults,

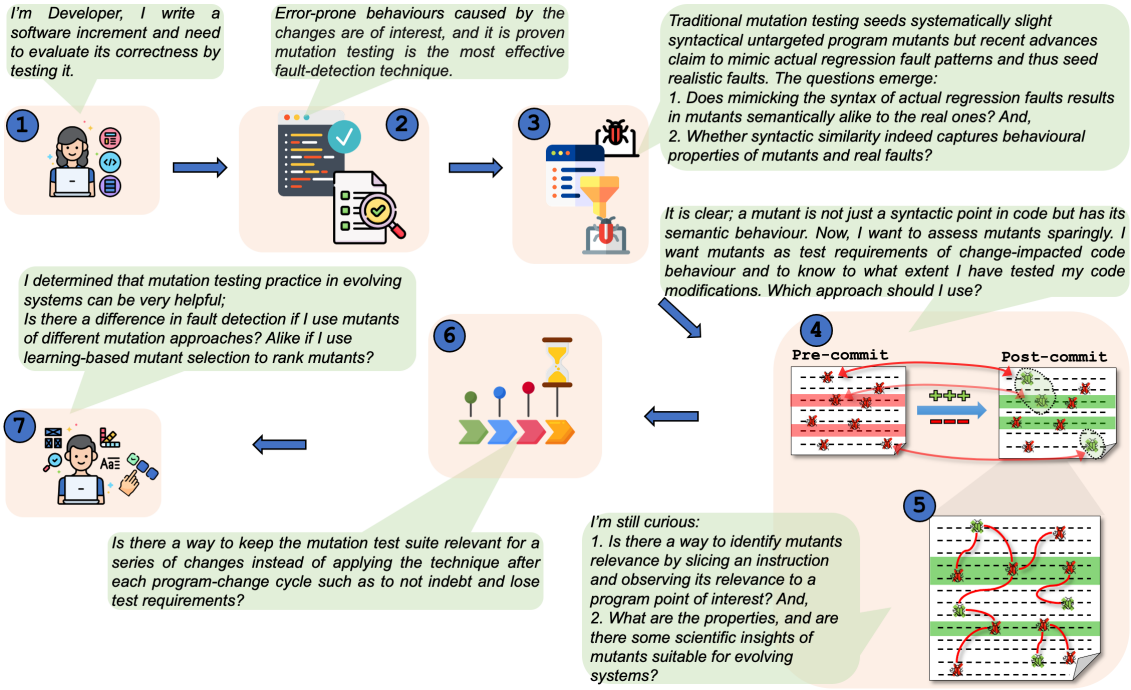


Figure 1.2: Opened questions, the storyline followed and challenges targeted in this dissertation. *Icon credits: Flaticon.com*

i.e., faults that are significantly different from real ones in terms of syntax, and as a result, numerous propositions have been made claiming to produce seeded faults that are syntactically similar to real ones. These propositions resulted in techniques that promise to form some sort of realistic faults and to improve fault seeding realism by defining and evaluating mutants with respect to non-semantic metrics, i.e., mainly syntactic-based metrics. Although these techniques are promising, they still treat mutants as points in the program under test instead of considering their behavioural properties - semantics - as defined in the RIPR model of fault-based test assessment techniques (see Section 2.2.2). Some mutation testing practitioners would agree and find it quite apparent that seeding syntactically close faults do not necessarily result in being semantically close to the real regression faults; yet a lack of evidence on the topic makes many practitioners diverge from the obvious, form and build around an assumption that seeding faults with frequent syntactic fault patterns that have similarities with a real fault will result in faults that are subtle or semantically similar to real ones. Therefore, the question of mutant behavioural properties and real fault representation requires significant improvement in understanding, and clarification of the role of the faults' syntactic nature with respect to program semantics and mutation assessment, raising awareness on the use of semantic and syntactic evaluation metrics in fault seeding studies and the usage of the semantic-based metrics in capturing mutants behaviour in the context of fault-based test assessment.

1.3.2 Testing Regular Code Modifications

Modern incremental software development practices produce software systems through regular modifications during the software life cycle. Modifications are usually made in order to maintain and improve the software - fixing bugs, refactoring, or

improving code quality - or to make it evolve by including new features. In either case, automated testing is used as gate-keeping, i.e., to establish confidence that the modifications did not break any of the previously developed program functionalities. In such scenarios, developers often assume that the previous - operational - version of the system was stable and correct. Therefore, they are interested in testing only the behaviour of the changes they perform. This implies they want to assess the delta of behaviours between their pre- and post-commit system versions. For such cases, developers need metrics quantifying the extent to which they have tested the error-prone program behaviours affected by their changes. Unfortunately, little research has been devoted to forming such change-aware test criteria. Change-aware test criteria would offer a viable, from a performance perspective, way of dealing with the continuous software modifications, as one would only focus on the particular program changes or commits. To devise such criteria forms a challenge for mutation testing, which has the unfortunate effect of blindly using all possible mutations without considering their relevance to the most recent changes as the task in question. Analysing all mutations under test is impractical due to narrow time intervals between builds; plus, it is even uncertain how aware of the change is traditional mutation score. In short, the question emerges whether performing traditional mutation testing with the entire set of mutants or with mutants located on the changed code is sufficient to assess how well subtle program changes have been tested despite being recognised as the most effective fault-revealing technique. This aspect is also missing in the software testing literature since it mainly focuses on using mutants as a proxy of faulty program versions independently of the program changes under test. Thus, by recognising and targeting this challenge, this dissertation strives to modernise mutation testing such as to capture behavioural differences caused by a code modification, since of particular importance in evolving systems is to utilise its fault-detection ability and early identify problematic regression issues arising from neglected change-aware test assessment.

1.3.3 Mutants Fault Detection Effectiveness

In the context of evolving systems, particularly in the phase of automated testing, the quantification metrics are viewed through the lenses of application effectiveness and cost-effectiveness. With the pace at which software evolves, the traditional way of mutant generation has been questioned. The grammar-based mutants generation is regarded to often put noise/cost in the fault detection activities due to low-quality mutants - a.k.a, trivial or equivalent mutants [29]. Recently, researchers and practitioners started arguing against solely relying on grammar transformations and see their use as challenging and questionable for fault detection of modern development practices. Consequently, several novel and fundamentally different mutation testing approaches have emerged. Building on top of standard predefined syntactic transformation rules, new approaches aim at seeding mutants by either learning from recurrent fault instances occurring in versioning history or by employing pre-trained code deep-learning language models to maximise the probability of employing the right mutant operator. Being fundamentally different from standards, the question emerges of how different and whether significantly more effective is the fault revelation potential of these techniques compared with traditional grammar-based mutation testing. In parallel, deep-learning language models have been suggested for mutant selection. These learning-based selection strategies aim to

reduce the cost by discarding redundant mutants and providing testers with mutants that will bring value to the testing process. Overall, proposed selection strategies have not been utilised with respect to the recently proposed mutation approaches, and the challenge that occurs regards whether the fault-detection effectiveness of different mutation techniques/tools when using mutation selection to reduce the application cost of the technique still holds; where application cost is modelled as such to detect faults as early as possible and deal with requirements of evolving systems.

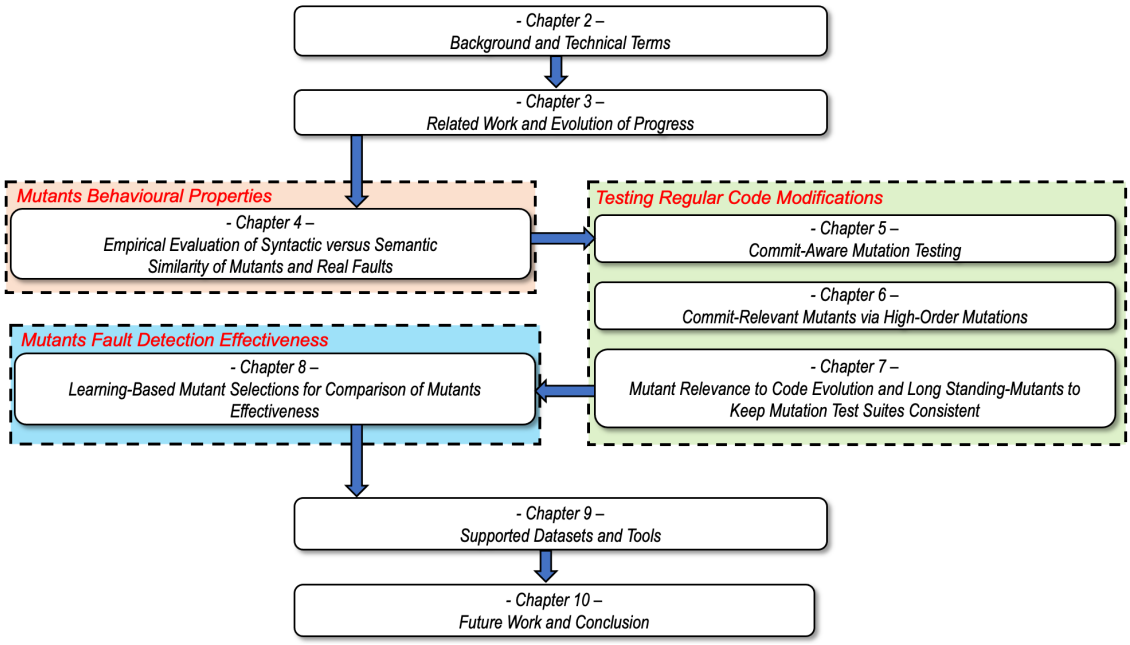


Figure 1.3: Organisation and outline of the dissertation.

1.4 Contributions of the Dissertation

This section describes the contributions presented in this dissertation to address the before-mentioned challenges, and it follows the organisation of the chapters presented in Figure 1.3.

Empirical Evaluation of Syntactic versus Semantic Similarity of Mutants and Real Faults (Chapter 4)

This chapter performs an empirical study aiming to evaluate whether seeding faults with frequent fault patterns have behavioural similarities with real faults and whether syntactic metrics can be used to capture the similarity between mutants and real faults. The present assumption is that fault patterns syntactically close to real ones are also semantically similar; indicating that mutants are the points in the program and rather neglect their RIPR model, which suggests that even slight changes have broad propagation and reachability. The study is conducted on Defects4J, which is, at the time, the most extended benchmark of real faults available to research. The results show a lack of evidence that syntactic similarity does reflect semantic similarity, indicating that syntactic distance cannot be used as an evaluation metric to capture mutants' behaviour in the context of mutation testing. This chapter further provides a view and insights into real faults-mutants

behavioural differences and the usage of semantic similarity measurement to represent and capture the similarity of mutants with real faults. After this first step which addresses the challenges related to mutant-fault behavioural properties, the next step is to study the mutation behaviour in relation to rapid regular code modification.

Commit-Aware Mutation Testing (Chapter 5)

This chapter presents, at the time, the first approach to allow change-aware mutation testing. Until this point, mutation testing had the unfortunate effect of blindly using all possible mutants without considering their relevance to the task or to the most recent changes in question. To allow such modification-focused testing, this dissertation introduces commit-relevant mutants, i.e., mutants able to capture unforeseen behavioural interaction between changed and unchanged code on which regression faults often occur. The study is conducted on both C and Java languages with the purpose of alleviating concerns of generalisation to a specific language and specific granularity level of a test suite. The study shows that this category of mutants, in terms of testing, represents change-relevant requirements and can be used to judge whether test suites are adequate in testing code commits and, if not, to provide guidance in improving them. Overall, the change-aware mutation testing shows 30% higher fault detection in comparison with baselines. After introducing the notation of commit-relevant mutants, the next step is to ease the constraints regarding their designation.

Commit-Relevant Mutants via High-Order Mutations (Chapter 6)

This chapter presents the first experimental approach for identifying commit-relevant mutants using the notion of observational slicing of a single program version. Naturally, the relevance of an instruction to a program point of interest, such as a program state or variable(s), can be determined by mutating instructions and observing their impact on the point of interest (changes in the target program state or variable). Since the aim is to identify mutants relevant to changed instructions, this chapter checks the impact of mutants located in the changed code on mutants located in unchanged code. In essence, the approach measures the impact of second-order mutants on the first-order ones, which captures the existence of implicit interactions between the changed and unchanged code parts. This formulation addresses the challenges of state-of-the-art and makes commit-aware mutation testing more general in evolving systems, resulting in the most extensive dataset of commit-relevant mutants up to date. The study shows that the commit-relevant mutants are prevalent; one in three is relevant and indicates the need for additional study to provide scientific insights into the nature and properties of commit-relevant mutants and their utility over time. Thus, as the next step, the next chapter studies the relevance of mutants to code evolution, investigates the predictability of commit-relevant mutants properties and, in line with the program evolving nature, proposes how to keep the mutation test suite consistent and relevant over software releases instead of different sequential versions.

Mutant Relevance to Code Evolution and Long Standing-Mutants to Keep Mutation Test Suites Consistent (Chapter 7)

The main objective of this chapter is to examine the properties, predictability, and utility of commit-relevant mutants, as well as subsuming commit-relevant mutants such as to quantify further the potential benefits of selecting relevant mutants during

project evolution. Furthermore, this study provides scientific insights concerning the cost and effectiveness of mutation analysis in testing evolving software systems. The study shows that a large proportion of commit-relevant mutants are located outside of the program changes and that selection of subsuming commit-relevant mutants reduces even further the number of mutants, by about 93%, on average. This chapter further argues that there is a remaining barrier to uptake: mutant consistency. A consistent set of mutants for a project is desirable so that test effectiveness can be consistently tracked against a common baseline over a series of project releases. The results show that identifying a high-quality suite of long-standing mutants allows the maintenance of mutant relevance over a series of releases. In light of the mutants' effectiveness in the presence of emerging diverse mutation approaches, the next chapter studies the suitability of learning-based strategies to select and compare the effectiveness of different kinds of mutants.

Learning-Based Mutant Selections for Comparison of Mutants Effectiveness (Chapter 8)

This chapter shows that mutants of different mutation testing approach lead to different conclusions on fault detection due to noise introduced by the high number of mutants with low quality. Therefore there is an inherited risk that their suitability can be misinterpreted, which brings the need to use learning-based selection strategies to alleviate the noise and focus only on the effective mutants produced by a mutation approach. The study is conducted such as to model the application cost that different mutation testing techniques entail and perform a controlled cost-effectiveness comparison under different cost models, which reflect the main efforts spent in mutation testing campaigns. We show that mutation testing techniques significantly improve their performance under the machine learning-based selection strategy. Additionally, this chapter shows that when comparing the effectiveness of mutants from different approaches or tools, it is imperative to account for a mutant selection technique suitable for this purpose.

Supported Datasets and Tools (Chapter 9)

This chapter describes three distinctive mutants datasets generated to support the studies in this dissertation. The chapter provides open-access links for each of the datasets to serve the benefit of the community since the generation of the mutants requires significant computational resources. The datasets follow the evolving context of the system under tests and contain mutants with clean test contracts, w.r.t., no change between the versions touches the test suite, a dataset of mutants generated from different mutation approaches and a dataset of high-order mutants. The dataset of high-order mutants is a by-product of the most extensive empirical study of commit-relevant mutants at the time, described in chapters 6 and 7. Specifically, the datasets contain 10,071,875 mutants and 288 commits extracted from five (5) mature open-source software repositories. The experiments took over 68,213 CPU days of computation. Each of the generated datasets used throughout this dissertation is described in detail in this chapter following step by step manner, holding information on the dataset metadata. Besides the datasets, this chapter described the tools used and frameworks built for the purpose of experimental evaluation. This chapter describes PiTestAssert, an extension tool built on top of PiTest with the purpose of studying mutants on and around a code change, their interaction and test suite instrumentation.

1.5 Organisation of the Dissertation

Figure 1.3 depicts the organisation and outline of this dissertation. The dissertation is organised to first familiarise the reader with the background, terminologies and definitions - in Chapter 2 - used throughout the chapters. Next, Chapter 3 offers insights on the solid grounds of related work on which this dissertation builds. Chapter 4 tackles the challenge of mutants' behavioural properties and provides a study on the *Empirical Evaluation of Syntactic versus Semantic Similarity of Mutants and Real Faults*. The following chapters deal with the challenges regarding Testing Regular Code Modifications. Moreover, Chapter 5 introduces novel *Commit-Aware Mutation Testing* approach. Chapter 6 describes and studies an approach to identify *Commit-Relevant Mutants via High-Order Mutations*. Chapter 7 studies *Mutant Relevance to Code Evolution and Long-Standing Mutants to Keep Mutation Test Suites Consistent*. The third identified challenge regarding Mutants Fault Detection Effectiveness is further described in Chapter 8 and tackled by the study on *Learning-Based Mutant Selections for Comparisons of Mutants Effectiveness*. Chapter 9 provides descriptions on *Supported Datasets and Tools* used throughout the dissertation. While final Chapter 10 provides the insight in the *Future Work* and concludes the dissertation.

2

Background and Technical Terms

This chapter introduces the reader to the technical background and familiarises with terminologies and definitions used throughout the dissertation. Its sole purpose is to rather guide the reader through the context in which this dissertation regards - ease the reading and reasoning.

Contents

2.1 Adequacy Criteria-based Software Testing

Software Testing measures via tests how well we exercise different parts of a software program or - to put it in another way - measures how the program behaves under a specific test input [1]. Software practitioners want to keep track of software execution to identify inconsistencies that will lead to unexpected software output/behaviour, a.k.a. bugs, faults, defects etc. The reason rests in the software complexity and numerous decision paths that are often too long and deep to apprehend. To maintain a bigger picture of the executed scenarios through tests, different Test Criteria are used to serve as test adequacy metrics and guide various testing activities. Test criteria metrics quantify the extent to which systems are tested [1]. They are based on the notion of test requirements, i.e., portraying and providing insights on a question: *"What should be covered by tests in the software system?"*. Different testing requirements gain information on how well a set of tests, a.k.a. *test suite*, covers various parts of software, e.g., execution of statements, conditional branching etc. [9]. The aim is that no code statement or program execution path leading to a program output is left unexposed and the behaviour assessed.

Depending on test requirements covered by a test suite, a test criterion defines a concrete value that reflects how well the program is tested w.r.t. its intended behaviour. To practitioners, the overall value serves as a coverage rate which synonymously relates to testing thoroughness, given the testing technique and test criteria used for software execution. Reaching a specific test criteria threshold can assess whether effort adequately enough has been invested into software testing activities to achieve confidence in software behaviour or rather extra testing needs to be conducted by executing the program under additional inputs, w.r.t., creating new tests. Thus, besides assessment - it is worth noting - test criteria drive different aspects of the testing process, such as test generation [30] or test selection [31] etc.

All-around, test requirements serve to assess the thoroughness of a test suite, indicate redundant tests, guide towards new tests, and decide if more effort should be devoted to testing or if reasonable confidence has been gained in the behaviour of the system. Test criteria are also used to assess other criteria for the reason of difference in effectiveness and subsumption between criteria - some are simply stronger than others [9, 32].

2.2 The Theory of Mutation Analysis

The previous subsection puts test requirements in the core of software testing practice as they guide the identification of unwanted program behaviour w.r.t., program faults. Mutation analysis is such a test adequacy criterion [33] that operates by estimating the capability of a test suite to detect artificially seeded program faults. These seeded faults are better and broadly known as *mutants* of a program under test. A mutant usually takes the form of a small syntactic change in the code, such as, for instance, changing the relational expression of the statement *"if ($a > b$)"* into *"if ($a \geq b$)"*; see Section 1.1.1 for a more practical example of mutants generation.

A set of mutants is often systematically generated, following a set of replacement rules called *mutation operators*. Different mutation operators can be used in order to tailor the creation of the mutant and, thus, the test requirements. This gives the tester freedom and allows focus on different aspects of the test suite effectiveness concerning the level of testing, testing aim or testing specific parts of the program;

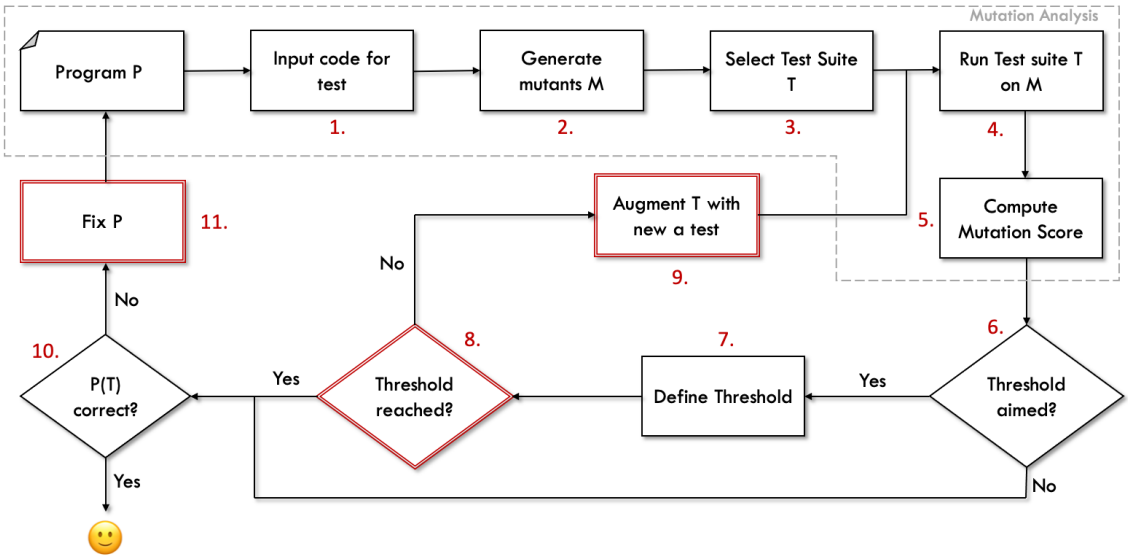


Figure 2.1: Mutation-based Testing Process

should the tester only want to focus on those. Once mutants, i.e., test requirements, are created as multiple faulty versions of a program under test, the test suite runs against the original and the mutants, aiming to differentiate their behaviour. The behaviour is usually represented by the output, triggered with test input and captured by test oracle, a.k.a., expected-actual values mechanism.

If a test triggers and captures different behaviours between the original program and a mutant, the mutant is considered to be *"killed"*, w.r.t., a test suite is able to distinguish this artificial fault - fulfilling a test requirement. A test killing a mutant not only demonstrates that the test covered the mutant but also that it is capable of detecting altered and propagated program states the mutant introduces. If, for all tests considered, the original program and a mutant behave the same, the mutant is said to *"survive"* tests, and its output escapes being identified. This process of assessing the ability and thoroughness of the test suite to differentiate between the program under test and generated mutants is called *mutation analysis*. The thoroughness is measured using the *"Mutation Score"* (MS), the ratio of mutants killed by test suites over all killable mutants created. Notice there may exist mutants that cannot be killed by any test since they are functionally *equivalent* to the original program [34]. On the other hand, the number of killable mutants does not necessarily represent the number of test cases since several mutants can be redundant; for instance, one test may also kill several mutants at the same time. Thus, the effort put into analyzing and executing redundant or not appropriate mutants for a particular use case is wasted; hence it is desirable to analyze only the mutants that add value.

2.2.1 An Example of the Process of Mutation Testing

Mutation Analysis makes a core of a mutation-based testing process. Mutation testing is often used interchangeably with mutation analysis, yet, there exist slight differences that are noteworthy to distinguish. As described in the previous subsection, mutation analysis is a methodology that encloses the generation of program mutations and the execution of tests to kill/distinguish them in order to obtain the metrics on test assessment. On the other side, mutation testing is a technique with a repetitive

process where the mutation-based analysis guides towards the generation of new tests for not killed mutants. Thus, the technique helps augment the test suite, conveys confidence by providing assurance of reaching a predefined testing threshold or guides towards fixing a program under test. The process adapter from the survey by Papadakis et al., this dissertation depicts in Figure 2.1, as it is the more recent process following current trends [8]. More curious readers are encouraged to refer to the first process defined by A. J. Offut and R. Untch [35].

First, the process starts with a program under test and its code as input for the test adequacy evaluation - *step 1*. On the input code, we generate the set of mutants M, which represent instantiated versions of the executable program - *step 2*. Once the mutants are instantiated, we determine a test suite T to run - *step 3*. In the case that no tests exist, we need to create tests, either manually or automatically, in support of some available tools [30]. Alternatively, all tests can be selected for execution or just those that cover mutants to save execution time. Naturally, a test that does not cover a mutant cannot kill it. Tests can be executed on a mutant until at least one test kills it; while executing all covering tests on all mutants still has its benefits, as it can measure the coverage and strength of tests and mutants through the whole execution matrix; this knowledge can be reused in the steps related to mutants selection, test prioritisation etc. The next step involves executing test suite T on the set of generated mutants M - *step 4* - and calculating a mutation score from the killing matrix as the product of the execution - *step 5*. In case the mutation score threshold is aimed - *step 6*, as often is the case, separate activity needs to be conducted to define the threshold - *step 7*. In a scenario where the threshold is not reached, *step 8*, further effort is needed to distinguish all remaining behaviours caused by the mutants. Thus, new tests are created based on surviving mutants - *step 9*. The test suite T is augmented with new tests, and the process repeats. If, in the next cycle, the mutation score reaches the predefined threshold suggesting sufficient thoroughness and sensitivity of the test suite T, the process continues, and a question comes to be whether the program P under test behaves correctly on all tests - *step 10*. Note that a mutant can also be killed if a test passes on the mutant but fails on the original program. In the case in which the program under test does not behave correctly in regards to T, its identified unwanted behaviour requires fixing - *step 11* - and repeating the whole process with the newly fixed version of the program. On the contrary, in the case when the program behaves correctly, and T is sensitive to different program outputs, testing quality is considered on the satisfactory level, thus bringing confidence in the output of P.

It is important to be aware that the process of mutation testing incorporates multiple actions or steps not included in the figure. For example, mutant selection and mutation reduction w.r.t., which mutants to use, test selection and prioritisation, w.r.t., which tests and in which order to perform execution on mutants, and many others. Note no step has less importance; however, for the purpose of this dissertation and for the purpose of simplicity, the figure depicts the basic steps, while some additional ones follow in subsequent subsections.

2.2.2 Fundamental Principles - Mutants Behaviour

The overall idea behind the process of mutation testing comes to be simple. Syntactic changes representing artificial faults called mutants are generated, and we observe whether tests can distinguish between the original program and mutants

under the same test input. Now, natural questions may tickle the mind of a curious reader: *How test execution leads to different behaviour for a mutant and original program? Why is it important to identify mutants with tests? And moreover, why is this approach effective?* There is more than one answer to satisfy each question. First, we must start with a simple broader explanation. Mutants are simple code-potential faults that a tester would target when evaluating the correctness of a program against the presence of real faults. Assume that there exists a set of complex faults in a program under test, such as that those faults produce an erroneous undesirable program behaviour. Therefore we desire our tests to trigger erroneous behaviour, and even more, we desire them to be sensitive enough to capture even the slightest change in the program behaviour caused by faults, since, accordingly, the tests will also be able to capture more complex faults. By creating slight faults, a.k.a. mutants, systematically in a program under test, a program is stressed on various locations, causing various program behaviours and allowing tests to reveal other types of faults often overlooked by the developers. Practically, mutants require test cases to be capable of propagating corrupted program states to the observable program state, at which point real faults have good chances of becoming observable as well.

Furthermore, it is important to understand that mutations represent common programmer mistakes [36]. More recent studies found that real faults include infection of only a couple of code elements, thus confirming once again the Competent Programming Hypothesis [10] on which grounds the mutation testing emerged. The hypothesis states that a programmer produces software close to being correct, with only slight syntactic variations of what comes to be incorrect program behaviours. Thus by mutating what is considered correct behaviour, we can evaluate whether incorrect behaviours escape being distinguished by tests and assess whether tests differentiate correct from incorrect.

Finally, to fully understand why the approach of mutation is effective, we need to understand that the behaviour of mutants is considered in relation to the expected observable - what is often considered correct - program behaviour of interest. In simple words, by leading to expected and unexpected observable program behaviour, mutants help in testing all corner cases that easily escape even the most competent programmers. To understand why mutants lead to different behaviour and why this behaviour is of interest, we need to consider the *RIPR fault model*, which defines the Reachability, Infection, and Propagation Revealability of real faults [1]. To find a fault in a program, the test case needs to reach the location of faulty syntax *(Reachability)*. Next, this faulty syntax needs to cause an infection in a program state, such as that the program state diverges from the expected or what is considered correct one *(Infection)*. This divergence or corruption of the program state needs to propagate to the output of the program since it only then impacts the program behaviour *(Propagation)*. In the end, the undesirable output needs to be intercepted and asserted by a test case *(Revealability)*. On the opposite, undesirable erroneous program behaviour escapes the attention of the tester and propagates to the end user with potential consequences. Therefore, in summary, when we use mutants as test requirements, we check for the perceptiveness of erroneous program states.

More often than not, mutation behaviour and, with it, mutation testing, can be summarised in a premise: *"If there exists a fault in a program, there will usually be a set of mutants that can only be distinguished with test cases that lead to detection of that fault"* [37].

2.2.3 Different Categories of Mutants as Quality Indicators

Questions about what drives a mutant interest or whether specific mutants are better than others came to be subjective and conditional on a use case. Many studies put effort into identifying the relationship between mutants and different testing quality indicators, giving birth to different mutant categories [8].

Foremost, killing all mutants is not feasible, as some mutants are semantically equivalent to the original program, i.e., they will behave the same way for all possible inputs, although they are syntactically different. These mutants are called *equivalent*, while mutants for which there exists an input for which their behaviour is different from the original program's, are said to be *killable*.

When using mutation analysis to measure the thoroughness of a test suite, we do not desire to take equivalent mutants into consideration, as even a perfect test suite will not kill them. Equivalent mutants have proven to be a major challenge in the area of mutation testing [8, 34], as identifying them is an undecidable problem [38]. Interestingly, many killable mutants are functionally equivalent to others, introducing skew in the mutation score. The studies of Papadakis *et al.* [39] and Kintis *et al.* [40] have shown this to be problematic and suggest getting rid of *duplicated* mutants w.t.t., mutants equivalent to others, in order not to count the same test requirement multiple times. Different categories of mutants exist when it comes to mutants operators, i.e., rules of their generation. Previous research concluded that mutants of specific types, e.g., Relational Operators, are more important as they encode test requirements not captured by other mutant types [41]. Research also suggests that a mutant quality and importance should be judged by how easy it is to kill it. Intuitively, if a mutant is so unreasonable that most or all tests distinguish its behaviour, a mutant is not of a particular value since it is *trivial*, a.k.a., easy test requirements. On the contrary, the researchers suggested that their stubbornness, i.e., how hard it is to generate a test required to distinguish it, should judge the mutants. Calculating *hard-to-kill* mutant is accomplished with the ratio of tests that identify its divergent output over how many tests cover it. A mutant is hard to kill if it is killed only by a small proportion of the tests that reach it. Another category defines the *fault-revealing* capability of mutants. It focuses on those mutants that reveal the fault, such as that a behaviour a mutant produces is truly what is expected from a program to behave, while an incorrect behaviour, in this case, is the one of the original program.

Moreover, blindly using available mutants will lead to overhead in testing when it comes to effort. A specific category of mutants that contributes to the overall computation expense but they do not contribute to the overall analysis of testing quality is called *redundant mutants*. The redundant mutant will always be redundant if there exist other mutants distinguished with the same test. Plus, redundant mutants provide noise to the overall mutation adequacy score, consequently providing the noise into a comprehensive observation of test suite quality. Redundant mutants contribute to the overall computation cost because they are numerous, and to analyze them, one would require additional tests to be executed. To deal with redundant mutants, the category of subsuming mutants is born.

2.2.3.1 Subsuming Mutants

A mutant is adequate to consider if it can be a subset of other mutants, w.r.t., does it have disjoint behaviour from other mutants. The measurement is conducted

by studying whether the mutants have similar killing conditions, which means distinguishing between a set of mutants leads to determining its subsumed set. Specific mutants that form a set and subsume all other mutants are a representative set of all mutants. The literature accepts a subsuming set of mutants as a minimal set of mutants bypassing mutant redundancy [32, 42, 43].

Following mutants subsuming relations aims at finding the minimal set of mutants required to cover all (killable) mutants [44]. Intuitively, this set of mutants has minimal redundancies and represents a nearly optimal mutation testing process with respect to cost [29, 32]. A mutant M_1 subsumes a mutant M_2 if killing M_1 implies killing M_2, i.e., if fulfilling the requirement represented by M_1 means fulfilling the requirement represented by M_2. More formally, let us consider that M_1, M_2, and T be two mutants and a test suite, respectively. Consider also that $T_1 \subseteq T$ and $T_2 \subseteq T$ are the set of tests from T that kill mutants M_1 and M_2, respectively, where $T_1 \neq \emptyset$ and $T_2 \neq \emptyset$ indicating that both M_1 and M_2 are killable mutants. We say that mutant M_1 subsumes mutant M_2, if and only if, $T_1 \subseteq T_2$. In case $T_1 = T_2$, we say that mutants M_1 and M_2 are indistinguishable. The set of mutants that are both killable and subsumed only by indistinguishable mutants are called *subsuming mutants*. For instance, assuming that $T_1 = \{t_1, t_2\}$ and $T_2 = \{t_1, t_2, t_3\}$, one can notice that every time we run a test to kill mutant M_1 (i.e., t_1 or t_2) we will also kill mutant M_2. While vice versa does not hold since if we kill mutant M_2 by t_3, we will not kill mutant M_1. In this case, we say that M_1 subsumes M_2.

2.2.3.2 High-Order Mutants

Depending on the number of mutation operators we apply to the one instance of the original program, we can categorize the mutants by the number of simple changes one has to introduce in the original program to form one mutant. That is, *first-order mutants* (FOM) are obtained by making only one simple syntactic change to the original program. Second-order mutants (SOM) are obtained by making two syntactic changes to the original program (or applying one additional mutation to first-order mutants). In the general case, higher-order mutants (HOM) [45] are produced after the successful application of n mutations to the original program.

At the time of the inception of the technique, using higher-order mutants in mutation testing was not considered viable because of the *Coupling Effect* proposed by DeMillo et al. [10]. It stated that *"Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors"*. However, later on, Offut *et al.* [37] defined first-order mutants as simple faults while characterizing higher-order mutants as complex artificial defects able to capture more complex internal program state infection.

Determining the right location and type of mutants to apply, and deciding if mutants are relevant for the testing activity is part of ongoing research, and typically is up to a software tester and the tools selected to make a decision.

2.2.4 Mutant-Fault Resemblance

In the program development iterations, developers often go through many cycles of implementation carving. As previously said, programmers are competent that have a rough idea of what errors are most likely to occur.

Besides systematically generating code-grammar mutations, research has proposed to mimic and learn a fault pattern observed in some error-occurring program instances, seeking to bring program behaviours even more, closer to real faults. Advances in

data mining proposed techniques to mimic real faults, and advances in machine learning proposed techniques to learn how to seed faults based on bug fixes, such as to craft "natural" faults by replacing code tokens based on deep learning language models code embedding of what caused real faults. An argument often used in favour of "natural" faults is the tester's ability to understand them better.

The number of possible mutations to generate has been regulated differently based on the utilised techniques and accompanied fault-seeding tools. The question that often occurs is how to deal with design choices, e.g., when to stop the generation of mutants, and at which mutants are the most similar to real faults.

Two types of metrics are usually used to decide on the mutants' quality and evaluate the resemblance of a mutant to a real fault: syntactic and semantic similarity. Intuitively, *syntactic similarity* refers to the distance between the text representations of the mutant and the real faulty code, while *semantic similarity* to the program *behaviour* similarities between the mutant and the real fault.

Note that mutants' syntactic and semantic similarities vary from 0 to 1. Similarity 0 represents a mutant completely syntactically dissimilar to a fault or a mutant that semantically does not lead to a fault. While score 1 illustrates that the mutant is syntactically the same as a fault or semantically recreates an actual fault. By selecting semantically similar mutants, a developer would write a test that would find a fault mutant recreates.

2.2.4.1 Syntactic Similarity

One of the widely used scores to compute the syntactic similarity between two sequences of tokens, w.r.t., code elements, is Bilingual Evaluation Understudy (BLEU) score [46], which is often used for quantifying machine-translated text in Natural Language Processing (NLP) [47–50]. Given a candidate and reference code, the BLEU score takes the candidate text, breaks it into n-grams, w.r.t., characters or sub-strings, and computes how many n-grams appear in the reference text. The geometric mean of all n-grams up to 4 is often used in the literature [51]. Besides the BLEU score, syntactic similarity can be captured with additional metrics. One example is Cosine and Jaccard Similarity Coefficient. Cosine similarity[1] is a metric used to determine how similar a reference and a candidate text are, irrespective of their size. The metric requires sets of the word counts of two input texts and measures the cosine of the angle between two vectors projected in a multi-dimensional space. The Jaccard Coefficient metric[2], measures similarity between two observed sample texts by calculating the size of the intersection tokens divided by the size of the union of the sample sets.

2.2.4.2 Semantic Similarity

To compute the semantic similarity, researchers often resort to dynamic test executions since capturing all program behaviours is an undecidable problem. The most popular approach is to compute the similarity of the passing and failing tests. Figure 2.2 depicts the semantic similarity between a fault and a set of mutants. The Ochiai coefficient [52] calculation is a common practice in many different lines of work to represent semantic similarity, including mutation testing [45, 53, 54] and goes as far as to program repair [55] and code analysis [56] studies. The Ochiai semantic

[1] $https://en.wikipedia.org/wiki/Cosine\_similarity$
[2] $https://en.wikipedia.org/wiki/Jaccard_index$

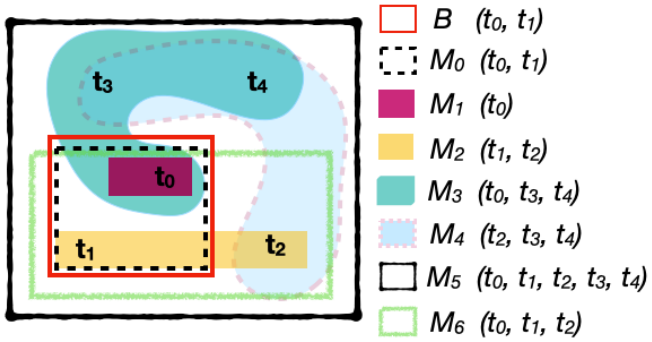


Figure 2.2: Semantic similarity. Mutant M_0 perfectly resembles (semantically) fault B, while M_1, M_2 and M_6 resemble it partially. M_6 resembles B better than M_2 and M_1 since it correctly detected 2 out of 3 cases that detect either B or $M6$, while M_2 detected 1 out of 3 cases and M_1 1 out of 2. M_1 underestimates test effectiveness as it does not capture t_1, M_2 does not capture t_0 and overestimates effectiveness as it mistakenly captures t_2, while M_6 mistakenly captures t_2.

similarity coefficient compares the behaviour given two program versions and their accompanied reference test suite.

The Ochiai coefficient represents the ratio between the set of tests that fail in both versions over the total number of tests that fail in the sum of the two. More formally, let P_1, P_2, fTS_1 and fTS_2 be two programs and their respective set of failing tests, then the Ochiai coefficient between programs P_1 and P_2 is computed as $Ochiai(P_1, P_2) = \frac{|fTS_1 \cap fTS_2|}{\sqrt{|fTS_1| \times |fTS_2|}}$, where $|\cdot|$ denotes the set size. A mutant M *resembles* fault B, if and only if its semantic similarity is equal to 1, i.e., $Ochiai(B, M) = 1$.

Another metric often used to approximate semantic similarity is defined as FDP, w.r.t., *fault detection probability*. This metric is also a probabilistic form of fault coupling, as it is a target metric of fault seeding that measures the subsumption of a real fault by a mutant, in the context of mutation-guided testing. Table 2.1 provides the overview of *fault detection probability*, where a real fault B is identified by the tests that detect a seeded fault M. The metric is, therefore, computed as the ratio of the number of tests detecting both M and B to the number of tests detecting M. Precisely, $FDP(B, M) = \frac{|fTS_B \cap fTS_M|}{|fTS_M|}$, where fTS_B and fTS_M denote the set of tests detecting the fault and killing the mutant, respectively.

Table 2.1: Semantic similarity between the real fault and the mutants captured from Figure 2.2. Mutant M_0 perfectly resembles (semantically) fault B.

	Tests	t_0	t_1	t_2	t_3	t_4	$Ochiai(B, M_i)$	$FDP(B, M_i)$
	B	✓	✓				-	-
Seeded Faults	M_0	✓	✓				1.00	1.00
	M_1	✓					0.70	1.00
	M_2		✓	✓			0.50	0.50
	M_3	✓			✓	✓	0.40	0.33
	M_4			✓	✓	✓	0.00	0.00
	M_5	✓	✓	✓	✓	✓	0.63	0.40
	M_6	✓	✓	✓			0.82	0.66

Example of Semantic Similarity provided in Table 2.1. Let B be a real fault, $M = \{M_0, \ldots, M_6\}$ a set of mutants and $T = \{t_0, \ldots, t_4\}$ a set of test cases. Figure 2.2 depicts the mutant killings of T and M. We observe that tests t_0 and t_1 detect fault B. Particularly, mutant M_0 is killed by the same tests, t_0 and t_1, resulting in a semantic similarity with the fault B equal to 1. Mutant M_1, is killed by test t_0 that also finds fault B, but is not killed by test t_1, and thus its semantic similarity is $Ochiai(B, M_1) = |\{t_0\}|/\sqrt{|\{t_0, t_1\}| \times |\{t_0\}|} = 1/\sqrt{2 \times 1} = 0.71$. Mutant M_2 is killed by tests t_1 and t_2, so its semantic similarity is $Ochiai(B, M_2) = 1/\sqrt{2 \times 2} = 0.50$. The semantic similarity between mutant M_3 and the fault B is 0.40 ($Ochiai(B, M_3) = 1/\sqrt{2 \times 3} = 0.40$). Semantic similarity of M_4 is 0, since all tests killing mutant M_4 do not detect the fault. Mutant M_5 is killed by all tests $(t_0, \ldots, t_4)$ and has a semantic similarity of $Ochiai(B, M_5) = 2/\sqrt{2 \times 5} = 0.63$. Notice that mutant M_6 is killed by tests t_0, t_1 and t_2, where 2 of them also find the fault, leading to a semantic similarity of $Ochiai(B, M_6) = 2/\sqrt{2 \times 3} = 0.82$. Table 2.1 summarises the Ochiai coefficient between the mutants and the real fault B.

2.2.5 Mutation Testing Tools

Grammar-Based Mutation Testing Tools

Over the years, many grammar-based mutation testing tools have been proposed to generate mutants by either mutating source code or lower-level intermediate code representation. All those tools have been identified, in detail described and compared in different empirical studies and surveys [8, 22, 57]. Yet, one mutation testing tool has been predominantly used.

PiTest (PIT) [58] has been continuously recognised as a state-of-the-art mutation testing tool for Java language that works by analyzing bytecode sequences and by looking for a possible location, i.e., instruction, to seed faults, using syntactic transformation rules (aka mutant operators). The mutation operators are categorized into 29 task-specific distinct groups. Examples of groups include Conditionals Boundary and Return Value mutators, which seed variations concerning relational operators and method call return values. PiTest has over 120 mutant operators, among which are many experimental mutants used for scientific purposes.

Learning-Based Mutation Testing Tools

Recently, many different approaches and associated tools emerged in pursuit of mimicking real faults and in an attempt to learn faulty patterns, see Section 2.2.4.

IBIR [54] is, at the time of writing, a state-of-the-art fault seeding tool that uses an information-retrieval-based fault localization model (IRFL) combined with automatic program repair inverted fix-patterns. It favours the generation of few but realistic natural mutants. It takes as input the git repository of the program to mutate and a bug report, written in natural language and seeds faults (introducing multiple faulty versions) that emulate the fault described in the bug report.

IBIR starts by analysing the given bug report using IRFL [59] to identify locations that are likely to be related to the features impacted by the corresponding fault. It then applies fault patterns on the identified locations, which are inverted fix-patterns used in pattern-based automated program repair approaches [54]. As the fix patterns are crafted from real bug-fixes, their inverse would induce faults similar to real faults. IBIR repeats this process until it exhausts all pre-defined patterns.

μBERT [60] is a mutation testing tool that uses a pre-trained language model (CodeBERT) to generate mutants by masking and replacing tokens. μBERT takes a Java class, extracts tokenized expressions, which are then masked for token replacement (mutation), e.g., for binary expressions μBERT masks the binary operator, and invokes CodeBERT to complement the masked sequence. For instance, in sequence `int mid = (low + high) / 2;` μBERT mutates the variable name expression `low` by feeding CodeBERT with the masked sequence `int mid = (<mask> + high) / 2;`. CodeBERT predicts the 5 most likely tokens to replace the masked one, e.g., it predicts `low`, `mid`, `Low`, `high`, and `medium` for the given masked sequence. μBERT uses these predictions to generate mutants by replacing the masked token with the predicted ones (5 mutants are created per masked token). μBERT discards non-compilable mutants and those that are syntactically the same as the original program, which are the cases in which CodeBERT predicts the original masked token (aka duplicated mutants [40]).

DeepMutation [61] generates mutants by employing Neural Machine Translation [51] a.k.a., NMT, which is also used by many recent studies [48, 62–64]. It uses an NMT model trained on a large corpus (∼787k) of existing bug-fixing commits mined from GitHub repositories. It takes a Java method as input and outputs a mutant.

In particular, every method is abstracted, in which the user-defined variable names and literals are replaced by predefined identifiers to obtain an abstracted code representation. These abstracted code representations are then input into the trained NMT model to produce abstracted mutants. The user-defined variable names and literals are restored to obtain source-code mutants.

The studies under this dissertation use the publicly available trained model of DeepMutation [65] to generate the mutants and `src2abs` [66] tool to perform the abstraction process. We followed the guidelines [61] and used beam search to generate 10 mutants per method.

2.3 Testing Evolving Systems

In modern software development, many developers contribute, w.r.t., commit code changes to the common-code base through the practice called Continuous Integration (CI). This contribution is shared among many developers through automatic chronological procedures of merging, building and testing software, eventually resulting in the development life cycles of a Continuously Evolving System [21].

Software evolves for many reasons, e.g., due to new features, bug fixes, code refactoring etc. Moreover, software systems evolve frequently due to ever-emerging program requirements that continuously adapt to user needs and the never-ending attempts to capture complex real-world environments. Hence, it is pertinent to provide strategies and approaches to analyze the impact of the program changes. Program changes result in behavioural changes usually captured through test suites. *Regression testing* helps in this respect by re-running the test suite on the altered version of the code to ensure that the developed functionality behaves as expected. Simultaneously, developers' attention rests on whether will any test case fail since it will indicate that altered/new functionality breaks what was until that point considered a stable system. Hence, after appending a change to the shared code base, every developer's primary concern and interest is whether modifications behave correctly not to cause breakage scenarios, plus whether modifications are tested enough such that in the future they do not be the cause of breakage scenarios.

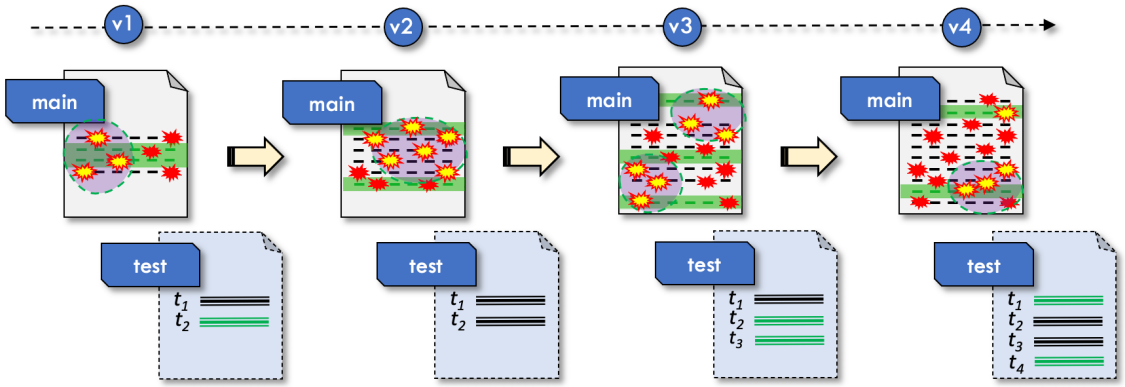


Figure 2.3: Typical evolution of software and its test suite depicted through four versions (v_1 to v_4) of a program (`main`) and its test suite (`test`). The green portions of the program (`main`) symbolize the *program changes* (e.g., a commit-change), and the explosions symbolize the *mutants* injected into the program. In the test suite (`test`), t_i symbolizes a *test case* i, and the green rectangles represent changes in the test suite (i.e., addition and modification of tests). The test suite and source code evolve as the program evolves through versions. As the size of the program increases, we can observe that the number of mutants increases as well. This eventually leads to a substantial number of irrelevant mutants that result in a waste of effort since the change does not impact their behaviour. In the figure, red mutants are irrelevant, while yellow mutants are impacted by the committed change. Notice that focusing only on commit-relevant mutants reduces the number of requiring attention and potentially leads to significant cost reductions. Additionally, the set of commit-relevant mutants can potentially quantify and capture the extent to which practitioners have tested the program behaviours affected by the change.

All-around, with every potential new code version, developers assume the previous version is operational, stable, and has the correct behaviour if all tests were passed successfully. Therefore, developers are interested in testing the delta between versions and establishing confidence that the system behaves as expected. Unfortunately - and what comes to be continuously surprising - regression issues arise from the unforeseen interaction between changed and unchanged code.

This dissertation argues that in testing evolving systems *it is not only about what we are expecting; it is also about what we are not expecting.* New tests need to stress new code increments, and an existing test suite needs to be evaluated on the premise of how well it can distinguish wanted and unwanted software behaviours impacted by the program code changes. The change-aware criteria would be a desirable solution, such as to quantify test suites' ability to test the error-prone program behaviours affected by code changes. That being said, this dissertation strives to utilize the power of Mutation Testing and versatile erroneous mutation output as a robust fault-based testing criterion and bring this testing technique to change-level testing, a.k.a. commit-level, suitable for Evolving Systems.

2.3.1 Continuous Testing

As the software evolves, the test suite also evolves. Concretely, as the program changes, new tests are added, or old tests may be modified to exercise those changes. Figure 2.3 illustrates the evolution of a program and its test suite during a typical software development process, showing changes in four versions of the program source code and the test suite. Each version can incorporate changes like new functionalities,

modified functionalities, code refactorings or bug fixes etc. These alterations, a.k.a commit changes, locate in code source files (`main`), and they are usually - at least they should be - followed by extended or altered test source files for testing introduced commit changes (the figure refers to them as *test*).

It is important to understand; how to test the program change, if it is enough to test only the changed lines, as well as how many test requirements and test cases are needed to analyze. Previous works [67, 68] have shown that many severe integration issues arise from unforeseen interactions triggered between the introduced change and the rest of the software. Notably, developers are burdened with the challenge of testing evolving systems, specifically, how to effectively analyse the difference in the program behaviours induced by their changes. A common approach to address these challenges is to leverage code coverage information, i.e., analyzing changed code statements or code elements having some sort of dependency on what is changed. Yet, the coverage information does not quantify to which extent the change has been tested and does not disclose unexpected behaviour impacted by the change, which is often overlooked. These are the main challenges of regression testing.

Therefore, there is a need for change-aware test metrics to guide effective regression testing and allow developers to quantify the extent to which they tested the error-prone program behaviours affected by their changes. This dissertation plans to use mutation testing to capture these interactions by targeting suitable mutants that demonstrate an (implicit) interaction between the changed lines and the unmodified part of the program (i.e., the code outside the change). These mutants form the change-relevant requirements and should be used to determine whether test suites are adequate and provide guidance in improving the test suite.

As the software evolves, the codebase becomes more complex and prolonged. We can observe in the figure this linear relationship between the lines of code and the number of mutants generated for each program version (referred to as *explosions* in the figure). When software approaches maturity, the number of potential mutants is very high. For visualization, observe versions 2 and 3 in the figure compared to version 1. If we assume that in the provided depiction, the interval of changes between versions is reasonably short, analyzing all mutants per version becomes costly. The number of mutants (both red and yellow in the figure) is independent of the program changes; it is actually dependent on the size of the program version and increases as the size increases. Hence, traditional mutation testing will be costly in general, since it uses more mutants than required. More importantly, analyzing mutants that are not relevant to what is actually changed in the code introduces noise in the change-aware testing activities.

2.4 Mutation Selection

The mutation testing process incorporates the activities of mutation-selection aiming to reduce the number of potential mutation candidates. To avoid a large number of mutants, if not infinite, the mutation testing process requires specifying a set of mutation operators to instantiate mutants. Designing and deciding on mutant operators greatly depends on the programming language, application in question, type of expected software defects, code, data or interface testing etc. Many different tools for code mutation have been released (see Section 2.2.5), and each cherishes its own set of mutant operators. Some shared operators target code expressions concerning Conditional Operators, Arithmetic Operators, Relational Operators and

many others. Due to a plethora of mutants operators, one of the first occurring and well-studied mutant selection strategies is based on the mutants operators. The idea is that some types are more important than others and that the mutant generation should always be performed by selecting a specific set of mutant operators from which to generate mutants [41].

Depending on the application of mutation testing, whether performed for test assessment, or guidance for test generation, further reduction in the number of mutants is usually required - yet, keeping in mind to not lose mutation effectiveness by narrowing the scope of mutation. Naturally, one of the most used approaches to perform reduction is to randomly sample mutants w.r.t., sampling 10%, 20%, 30%, 50% etc. of generated mutants. Although it looks simple, the approach loses the mutation testing fault detection effectiveness [69]. However, studies showed that randomly sampling 5% sufficiently represents the population in terms of mutation score [70]. Note that randomly sampled mutants are evenly distributed, which is reported to be vulnerable to the subsuming mutants, as a category of mutants that bypasses the noise of traditional mutation score and emphasises the importance and quality of mutants that represent a.k.a., subsume, all the others [68].

However, opinions on selection strategies are still divided up to this day, as some argue that the selection of mutants requires to be guided by anticipated quality and that mutant seeding should be in the program parts that are likely to be faulty or can influence program output, e.g., branch statement, return variables etc. Another broad opinion, especially in industry settings and concerning testing evolving systems, is to create and analyse mutants only on the lines where the code has been changed, thus hoping to capture the interaction between changed and not changed code.

Overall, all selection strategies lead to attempts at some sort of ranking of mutants based on their specific importance. By ranking mutants, practitioners will customise the analysis on only a certain number of mutants taking into consideration needs, usually related to available time and resources. More recently, many approaches emerged that integrate intelligence into the selection and ranking of mutants, such as utilising the power of machine learning approaches.

2.4.1 Learning-based Selection

Selecting mutants of high quality is not a trivial task. Yet, with the advances in machine learning, several mutation selection approaches emerged that, by learning the labelled ground-truth of mutants' characteristics, are able to predict the same in the future. Learning-based approaches attempt to rank the mutants by their probability of being representative of the set of mutants on which ML models, a.k.a., algorithms are trained. The ranking is often directed by an engineered set of different categories of static code features that characterise a mutant with respect to code, e.g., type, line, statement complexity, branch block etc. After representative features are fed into multi-dimensional latent space, a non-linearity can be drawn, classifying mutants of interest over those of no interest. Machine learning model power is to learn this distinction during the process of training by seeing characteristics of all categories of mutants used for the training. Following the trial-error-adjust practice, the learning-based approaches bring mutants of a category of interest close together in the latent space. After the training, when the learning is finished, a model is ready to associate a probability to a mutant being close to a certain category of mutants of interest based on its characteristics used for training the model.

2.4.2 Developer Work Simulation

To compare different types of mutants, e.g., in correspondence with fault detection, or to compare different mutation testing techniques, the practice of simulating real developer work has often been conducted. This practice includes standard developer work of selecting a mutant and writing a test to kill it. It is simulated since, due to effort required in time, it is unrealistic to assume the human feedback on the large scope [8]. Such simulation has been recognised by the community and has repeatedly been utilised as real work simulation. More precisely, it starts with an initial - usually empty - test set and the set of mutants not detected by those tests. The next step is to select a mutant - usually with high priority given by some strategy - together with a randomly selected test (without replacement) which is added to the test suite, and the mutant is discarded as satisfying the test requirement. The simulation is repeated until meeting the guiding function of detecting all mutants. Some mutants cannot be killed as they are functionally *equivalent* to the original program; thus, they are discarded by the developer. Hence, the thoroughness of a test suite is measured in terms of its mutation score, computed as the ratio of killed mutants over the total number of generated ones, or in the case of fault detection, computed as a ratio of tests identifying existing faults over several repetitions to remove the threat of randomness.

2.5 Additional Definitions

The studies conducted in this dissertation make use of different kinds of statistical tests and bivariate analyses to quantify and, with confidence, measure whether data support a particular hypothesis. In particular, this dissertation uses Wilcoxon non-parametric statistical tests to measure whether the values of one group sample are different from the values of another group sample. The test outputs the p-value, which is a measurement that helps decide whether there is no real effect/difference between the compared groups (a.k.a null hypothesis). The value ranges between 0 and 1, where a lower p-value indicates that there is a statistically significant difference between the compared groups and attributes confidence that the difference is not due to chance alone. The significance level often used to decide on the difference is 0.05 (a.k.a, alpha), leading to a confidence level of 95%. Alpha is usually defined before the analysis starts. Depending on the data types and the analysis, this dissertation also uses the Mann-Whitney U test to decide whether there is a statistically significant difference and whether one observed group has higher or lower values than another observed group. In contrast, this test observes independent samples of data, meaning an observation from one sample is not paired with the associated observation from another sample. This dissertation uses the same alpha levels, 0.01 and 0.05, to calculate significance.

To calculate whether two observed sets of data show any mutual correlation or relationship, in this dissertation, we use Kendall rank coefficient τ (Tau-a), Pearson product-moment (r) and Spearman correlation coefficient. Each of those correlation coefficients quantifies and summarises the strength and direction of a relationship between two variables. Values of the analysis are in the range from -1 to 1, where values close to both ends represent negative and positive correlations, respectively. While values in a range of absolute 0.2 around zero denote absence and insignificant correlation, while values higher than absolute 0.2 indicate a moderate or strong

relationship between two observed variables. In all cases, we also use the p-value to measure the significance of the results.

To evaluate the magnitude of difference between observed groups, we calculate the Vargha and Delaney A12 effect size [71]. A12 values over 0.56, 0.64 or 0.71 indicate a small, medium or large difference between the two populations, respectively.

3

Related Work and Evolution of Progress

This chapter contains information on the solid grounds of related work on which this dissertation builds. Special care was taken to thoroughly cover all related published work up to the time of conducting and writing the studies of this dissertation.

Contents

3.1 Mutants as Real-Faults representation

In this section, we discuss the empirical studies related to mutants-real faults syntactic similarity and mutants' behavioural closeness with the real faults. This section mainly relates to the first contribution described in Chapter 4.

Mutation Testing is widely used in experimental studies as a way to compare and assess testing techniques since it serves as a valid substitution for real faults occurring for a program under test [8]. The technique assumes that seeded faults include properties that are in some sense similar to real ones [72]. Interestingly, it introduces faults that are syntactically simple and are quite different from real faults that are in their majority more complex [8, 36]. In particular, the study of Gopinath et al. [36] provided empirical evidence showing the misalignment between seeded and real faults that are produced by traditional mutation operators and concluded that real faults are rarely equal, syntactically, to mutant faults.

To deal with this observation and motivated with the aim to bring seeded faults closer to programmers understanding, Brown et al. [73] proposed inferring fault seeding patterns, w.r.t, mutation operators, by using historical fault-fixing commits. The idea was to form - syntactic - fault patterns that resemble - in terms of syntax - real historical faults. The results show that syntactic fault patterns can be mined from code versioning systems, and these differ - syntactically - from those used by modern mutation testing tools. Roughly since the time when Gopinath et al. [36] provided the empirical evidence, until this moment of writing, clear trend and practitioners' positioning can be observed towards mutants-faults similarity [28, 51, 61, 73–78] - see Table 6.1 for a tabular view of the clear trend in proposed approaches and techniques. Motivated by advances in data mining and machine learning, practitioners lean towards the assumption that the more syntactically similar the seeded faults are to the real ones, the more they resemble them semantically. Suggesting the relationship between these metrics.

In short, DeepMutation [61], a neural machine translation technique [51] that automatically infers fault patterns from historical fault-fixing commits, was proposed. It was shown that DeepMutation resembles exact matches of 45% of real faulty cases while achieving relatively good syntactic similarity scores in most of the cases. SemSeed [77] aims to infer faulty patterns from bug fixes and to generalize them by appropriately adapting them to the particular local code, i.e., context. More recently, mutation monkey [28] was built by mining frequently occurring faults from complex changes that caused operational issues at Facebook [28]. The analysis of these faults indicated they were good at finding holes and missing tests in the systems under test. Consequently, language models pre-trained on code stirred tasks such as code mutation while using syntactic matching to derive the context and evaluate code manipulation [74,75]. Interestingly, the above studies aim at mimicking - syntactically - real faults, and as a result, they have been evaluated with "static" syntactic-nature metrics such as syntactic similarity.

Overall, the observed trend raises the question of whether such syntactic-distance metrics are suitable to capture mutants' behaviour and guide tasks in dynamic analysis, such as those in mutation testing, i.e., incorporating realistic semantic fault properties, and moreover, how they compare with traditional mutation testing. All these form open questions that this dissertation investigates in the scope of its first contribution related to the real faults-mutants behavioural properties.

3.1.1 Fault Coupling

Since its origin, traditional mutation testing has aimed at seeding faults using simple syntactic changes [10, 79, 80]. Showing empirical evidence of the coupling effect, DeMillo et al. [10] state that simple faults subsume almost all the complex ones [37]. Meaning that a test that identifies a simple fault, at the same time, is able to recognise a more complex fault [72]. Offut et al. provided a general assumption about the "size" of faults [81], suggesting that seeded faults, even often with small syntactic distance from the original program, introduce semantic deviations which form valuable test requirements [82] and have a high fault revealing potential [29].

While traditionally mutants are seeded by performing simple syntactic changes in the programs, real faults are in their majority more syntactically complex [36, 83]. As described in the previous section, recent studies aim to introduce mutants that are complex and look like being similar to real faults, holding the assumption that such techniques produce mutants of high quality, i.e., that have high real fault detection ability. In particular, it was proposed to form mutation operators based on fault-fixing commits [61, 73] or recurrent real fault instances [28, 54].

Knowing the portion of mutants with high fault detection probability provides additional insights into different techniques, mutants' fault-related utility and opens further directions for their studies. Over the years, we witnessed several mutation testing approaches competing for the state-of-the-art throne. Those approaches gave birth to tools that can be further categorised as Grammer-Based Mutation Testing Tools and Learning-Based Mutation Testing tools (see Section 2.2.5).

The relation between mutants generated by different grammar-based mutation testing tools has also been studied [8,84], and showed that PIT, which this dissertation uses for its studies, is the most effective tool today and produces mutants of the highest quality [57]. However, previous studies compare the effectiveness of mutation testing tools and draw conclusions under the assumption that every mutant has the same probability of being selected, that is, under a standard mutant selection policy. Therefore, the coupling between seeded faults has also been considered a source of bias in mutation testing studies as it introduces large overlaps between the seeded fault instances [32, 45, 68]. Recently, learning-based approaches emerged to question how to optimally select subsuming mutants and discard that subsumed [63]. Those studies have resulted in powerful techniques for cost-effectively selecting mutants, i.e., by avoiding the analysis of redundant mutants (basically, equivalent and subsumed ones). This makes it a promising technique to remove the noise of each of the techniques and allow focusing only on the mutants of the high-quality and their corresponding fault detection ability.

Overall, raising questions about how the effectiveness of mutants - arriving from different approaches - is compared when using learning-based approaches for their selection. This dissertation studies these questions in detail as a part of the contribution found in Chapter 8.

3.2 Regression Mutation Testing

The field of regression testing investigates how to automatically evaluate evolving software systems to avoid regression bugs occurring due to code change. Researchers have been proposing approaches in this field for a considerable interval of time [31]. Indeed, applying mutation during continuous testing has long been proposed but,

unfortunately, hasn't been studied thoroughly. In particular, among a few studies that occurred, Zhang *et al.* [85] proposed Regression Mutation Testing, a technique that speeds up mutant execution on evolving systems by incrementally calculating the mutation score (and mutant status, killed/live) by only considering mutants on execution traces affected by changes. As such, they assume that testers should use the entire set of mutants when testing evolving software systems. Another study that takes into consideration evolving nature of modern software is the study of Predictive Mutation Testing [86, 87]. The goal of predictive mutation testing is to estimate the mutation score without mutant execution using machine learning classification models trained on different static features [87].

Furthermore, existing mutation testing tools, such as Pitest [58], include some form of incremental analysis in order to calculate the mutation score (and mutant status, killed/live) of the entire systems or class under test.

Overall, it should be clear that a few existing techniques are either targeting the entire set of mutants or those located in the modified code areas. All proposed approaches are focused on speeding up test execution and approximating mutation score computation for testing evolving software systems. In contrast, this dissertation - Chapter 5 - focuses on identifying the test requirements - mutations - relevant and impacted by the program changes. Moreover, such a set of mutants provides a refined and exact change-aware mutation testing score that provides developers with mutation-based test assessment to capture changed program behaviours and thus evaluate to what extent code modification is tested.

3.2.1 Diff-based Commit-aware Mutant Selection

Following a similar line of work, Cachia *et al.* [88] proposed an Incremental Mutation Testing that limits the scope of mutant generation to strictly changed code regions since the last mutation run. Therefore, holding the assumption that in order to test a code change, it is sufficient to test lines impacted by the change. Similarly, Petrovic and Ivankovic [26] proposed a diff-based probabilistic mutant selection technique that focuses only on the mutants located within the program changes. The mutations within the code change are used for the purpose of the code-review phase by randomly picking some mutants located in the altered code areas. These approaches differ from the approaches introduced as main contributions in this dissertation as they ignore the program dependencies between the committed changes and the unmodified code by design, plus they lack to provide a quantifiable metric that captures the testing change-aware effort. The approaches described in this dissertation - Chapter 6 - account for the dependencies between program changes and unmodified code when identifying commit-relevant mutants to demonstrate the extent to which using mutants from modified code can help.

3.3 Static and Dynamic analysis based Approaches

In this section, we discuss motivating related work in the areas of change impact analysis, program slicing, interface mutation, and test augmentation.

3.3.1 Change-Impact Analysis

To accomplish the aim of this dissertation, it was necessary to refer to the works that study how to analyze and test the impact of program changes on evolving software systems. To this end, researchers have proposed several automated methods

to assess the impact of program changes on the quality of the software, e.g., in terms of correctness and program failures [31]. For instance, researchers have employed *program analysis* techniques to identify relevant coverage-based test requirements, specifically by analyzing the impact of all control and data dependencies affected by the changed code to determine all tests that are affected by the change [89,90]. Unlike these works, contributions in this dissertation focus on performing dynamic change impact analysis employing commit-aware mutation testing. In particular, aiming to empirically evaluate the properties, distribution, prevalence and effectiveness of - subsuming - commit-relevant mutants.

3.3.2 Interface Mutation

Delamaro *et al.* [91] proposed an inter-procedural mutation analysis approach for generating mutants that are relevant for integration testing, i.e., suitable for testing the interactions between program units. In particular, Interface Mutation aims at testing component integrations, it injects mutants on the component contracts (interfaces) and pairs of them at the component call sites and related uses of the interface parameter inside the components bodies in order to capture potential interactions between the caller and called components. This mechanism of capturing dependencies through pairs of mutants is, to the extent, the most similar to the approaches proposed in Chapters 5 and 6, and relates to the identification of commit-relevant mutants via High Order mutants, but more restrictive as it targets interfaces - call sites and method parameter uses. The work presented by Delemaro *et al.* demonstrates that inter-procedural program slicing is applicable for mutation analysis, particularly for integration testing. In contrast, commit-aware mutation testing aims at identifying relevant dependencies between changed and unchanged code and not between components.

3.3.3 Program slicing

A related line of work regards the formulation of dynamic or observation-based slicing [92–94]. These techniques aim at identifying relevant program statements and not mutants [95, 96]. Though, they could be used in identifying relevant mutant locations, in which every located mutant could be declared as relevant.

For instance, Guimaraes *et al.* [97] proposed the use of dynamic program slicing to determine the subsumption relations among mutants, in order to detect redundant mutants and reduce the number of tested mutants. In their evaluation, the authors demonstrate that using dynamic subsumption relation among mutants reduces mutation testing time by about 53%, on average.

3.3.4 Change-Aware Test Augmentation

This line of research aims to automatically generate additional test cases to improve the fault-revealing ability of test suites and, thus, quality. This area is particularly important and provided additional motivation for this dissertation due to rapid software system changes requiring the generation of new tests that exercise the program changes; as a vital practice to evaluate anticipated and unexpected software behaviours. Researchers have proposed several test augmentation approaches to trigger program output differences [98], increase coverage [99] and increase mutation score [100, 101]. Some test augmentation approaches have been developed to address code coverage problems using propagation-based techniques [102–105]. Other

approaches employ symbolic execution for test augmentation by generating tests that exercise the semantic differences between program versions by incrementally searching the program path space from the changed locations and onwards; this includes approaches such as differential symbolic execution [106], KATCH [107] and Shadow symbolic execution [108]. These techniques rely on dependency analysis and symbolic execution to decide whether changes can propagate to a user-defined distance by following the predefined propagation conditions. Hence they are considerably complex and computationally expensive, for instance, because they are limited by the state explosion problem of symbolic execution. These studies are complementary to the work done in this dissertation since the aim is to generate additional tests to improve existing test suites. However, the presented contributions in this dissertation focus on identifying suitable mutants, i.e., test requirements or objectives such as lead to test augmentation that exercise code changes, albeit using mutation testing.

3.4 Mutation Cost Reduction Solutions

In this section, we discuss motivating related work regarding the use and usefulness of subsuming mutants, different categories and different strategies utilised for mutant selection.

3.4.1 Useful Mutants Selection

Several researchers have studied the impact and prevalence of subsuming mutants for traditional mutation testing [29, 32, 43, 97, 109, 110]. For instance, Alipour *et al.* [109] demonstrated that subsuming mutants could reduce traditional mutation testing efforts. In particular, their empirical study on mutation test reduction found that subsuming mutants can reduce the number of mutants requiring analysis by up to 80%. Their study demonstrated the importance of subsuming mutants in traditional mutation testing, emphasizing that there is strong inter-dependency among mutants. In their empirical evaluation of *traditional mutation testing* involving four C projects and thousands of mutants, they found that test case reduction based on N mutants can reduce mutation testing effort (in terms of the number of mutant test executions) by 33 to 80% [109]. Likewise, Guimarães *et al.* [97] empirically demonstrated that identifying dynamic subsumption relations among mutants reduces traditional mutation test execution time by 53%. More recently, Kaufman *et al.* introduced a new measure of mutants' usefulness named TCAP, which stands for test completeness advancement probability [111]. TCAP is estimated with the probability of the expected number of tests to be written in order to achieve mutation adequacy. Due to the strong linear relationship between subsuming mutants and test completeness, TCAP can be considered a dominator score and closely related to subsuming mutants targeting the reduction of cost in traditional mutation testing.

However, despite the evidence of the impact of subsuming mutants on traditional mutation testing and integration testing, their impact on commit-aware mutation testing remains unknown. Thus, as one of the contributions, this dissertation studies the prevalence and distribution of mutants relevant to a committed change and the extent to which subsuming relations are maintained.

3.4.2 Machine-Learning Selection

Among the first papers that utilise the power of machine learning for static mutation selection is the work that introduces *FaRM*, a technique for selecting fault-revealing mutants [112]. The work proposes that fault-revealing mutants, which are killable and lead to test cases that uncover real faults, can be characterised and learnt by the set of static code features. By learning to select and rank mutants, the paper shows that with respect to mutants selection *FaRM* is able to identify 23% to 34% more faults than any of the baseline methods.

Recently, a few papers emerged that aim to learn the behaviour of mutants with respect to their subsuming characteristics. Moreover, Kaufman *et al.* [111] uses linear and random forest machine learning models to learn TCAP scores of mutants based on their static features context similar to the approach of *FaRM*. The reported results suggest that the prediction model can lead to the practical use of TCAP as a mutant selection-stopping criterion. Next, Garg *et al.* [63] proposed a machine-translation approach named *Cerebro* that learns the surrounding context of a mutant instead of its static manually engineered features. The results show that the approach is able to lead to strong tests able to kill 2 times a higher number of subsuming mutants than the corresponding baselines using manually engineered mutant features. Plus, the approach outperforms state-of-the-art by requiring 90% fewer test executions.

Yet, the closest recent work that builds on the contributions of this dissertation presents a machine learning and static analysis-based approach for predicting commit-relevant mutants called `Mudelta` [113]. The technique illustrates the importance of commit-aware mutation testing, particularly its ability to reduce mutation testing effort and reveal commit-related faults while drawing attention to the ability of the ML models to learn the mutant features in the C programming language on which the study is conducted. In comparison to the random mutant selection, `Mudelta` reveals 45% more commit-relevant mutants and achieves 27% higher fault-revealing ability in fault-introducing commits. Unlike this work, this dissertation defines and formalises commit-relevant mutants, evaluates their relationship with traditional mutation test criteria, emphasises the importance of commit-aware mutation testing, conducts an in-depth empirical study to understand the characteristics of commit-relevant mutants to shed more light on their properties and provide scientific insights for future research in commit-aware mutation testing.

3.5 Mutation Testing in Practice

The motivating related work for this dissertation likewise came from several reports and work from the industry. Google [26] applies mutation testing at scale, providing 6.000 developers with diff-based mutants after each code change. The company considers the effects of mutants as useful in attracting developer attention and in assisting the code-review process. A study by Petrovic et al. [26] at Google includes 13.000 code authors, the authors of the study report that after "70.000 commit diffs, testing 1.1 million mutants, the mutants surfaced 150.000 actionable findings during code review". Moreover, they document that 75% of surfaced findings developer feedbacks reported as useful, while many mutants caught actual bugs in the code and improved tests suites such as to kill them. These results led Petrovic et al. to more thorough studies related to the question of whether mutation testing

would improve the testing practices at Google [26] and what are the challenges, lessons and research directions for future industrial application of mutation testing. Their work reports that after analysing how mutants influence developers over time, they observed that it helps them to write more tests which consequently resulted in the active improvement of the test suites. Moreover, the work also concludes that the mutants relate to the real faults, as some of the real-high priority faults coupled with the mutants, while some of the faults would be prevented if a mutant existed to capture a code behaviour impacted by the change. Based on the study with 30.000+ developers and 1.9 million committed changes, the authors also report no significant overhead to the software development process, while they emphasise and point out the challenges caused by unproductive mutants, i.e., mutants that do not represent a relevant test requirement to a code change, concluding that in studied context achieving traditional mutation adequacy is, in their words, "neither practical nor desirable".

Unlike all before-mentioned work, this dissertation delivers a novel change-aware mutation testing approach that introduces relevant mutants as a separate category of mutants that represents a set of productive, practical and desirable requirements to evaluate the impact of a code change; plus, this dissertation is also arguing against the noise introduced by the traditional mutation testing in the context of evolving system by proposing the usage of learning solution to rank and prioritise mutants based on their fault detection ability.

4

Empirical Evaluation of Syntactic versus Semantic Similarity of Mutants and Real Faults

The relationship between using syntactic or semantic distance metrics for evaluating seeded artificial faults, a.k.a. mutants, in the context of mutation-test assessment has not been well established due to a lack of empirical evidence. Researchers increasingly make the assumption that seeding faults with syntactic patterns that are similar to a real fault, results in mutants that are semantically similar to real faults and, thus, more realistically capture their behavioural properties. We show this is not the case and find no evidence suggesting any link between syntactic and semantic similarity of real and seeded faults. Our results indicate that the semantic similarity is uniformly distributed project-wise, exemplifying that behavioural properties of real faults cannot be approximated by syntactic similarity, and establish that semantics is independent of similar and dissimilar mutants w.r.t., significant syntactic changes do not imply significant semantic changes, and vice-versa.

This chapter is based on the work published in the following journal article:

- Milos Ojdanic, Aayush Garg, Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis and Yves Le Traon, *"Syntactic vs. semantic similarity of artificial and real faults in mutation testing studies."*, 2023, *IEEE Transactions on Software Engineering.*
  `https://doi.org/10.1109/TSE.2023.3277564`

Contents

4.1 Introduction

Fault seeding techniques, such as mutation testing, are extensively used in controlled studies to evaluate and compare testing techniques [8,32]. These techniques allow researchers to seed faults under experimentally controlled conditions and thus perform reproducible test assessments. In a sense, by comparing the number of seeded faults revealed by test methods, researchers can form a proxy metric that approximates the fault-revealing potential of the performed testing [53, 72, 114].

Although popular, such techniques have been criticised for producing unrealistic faults [36, 51, 73, 77], i.e., faults that are significantly different from real ones in terms of syntax [36], and as a result, numerous propositions have been made claiming to produce seeded faults that are syntactically similar to real ones. The most recent research, in particular, motivated by the code naturalness hypothesis [115][1], aims at forming realistic faults that are, in fact, artificial faults that have some form of syntactic similarity to real ones, i.e., usually following particular syntactic fault patterns. We call this line of work as *fault mimicking* approaches.

Table 4.1 lists a set of recent fault-mimicking techniques that aim, in diverse forms, at generating (syntactically) realistic faults. By inspecting the table, the research trend becomes evident as these techniques seek the realism of fault-seeding, which is defined and evaluated by some form of non-semantic metrics, i.e., mainly syntactic-based metrics (number of tokens changed, BLEU score, etc.) from real faults. This means that many studies are solely guided by syntactic metrics and not semantic ones. Nevertheless, such approaches may indeed succeed at generating some exact matches of targeted real faults and may indeed be effective in their domain. However, since those fault mimicking methods are guided by non-semantic metrics, a key question that remains is whether they are suitable for fault-based test assessment [8], as is the typical use of mutation testing in research studies [8, 32, 72].

Mutation testing is based on the basis that fault seeding should be performed using untargeted program syntactic changes [10,35,37]. These changes are defined using the programming language grammar and are completely unaware of any fault semantics. The key assumption is that simple syntactic changes, although syntactically dissimilar to complex faults, result in semantic deviations that are coupled with complex and real faults[2] [9, 37] and can be used for test assessment [72].

In contrast, the key strategy followed by fault mimicking is to identify program locations where fault opportunities emerge and perform relevant changes, following a pattern observed in some fault instances, that alter the program behaviours similarly to real faults. This implies an underlying assumption that *seeding faults with frequent syntactic fault patterns that have similarities with a real fault will result in faults that are subtle or semantically similar to real ones.* Similarly, another assumption is that *seeding faults that are syntactically dissimilar to real ones results in unrealistic faulty semantics*, i.e., the seeded fault semantics are quite different from those of real faults.

These assumptions may appear intuitive but have absence of evidence, except, of course, in the case where seeded faults match exactly real ones. Early research on the coupling effect [37] stated that “simple faults can cascade or couple to form

[1]Naturalness hypothesis states that programs exhibit properties similar to text and thus, natural language process techniques can be used to support code analysis techniques.

[2]In this study, we use the term “real faults" to refer to the set of reproducible curated faults provided at the Defects4J dataset [116]. Therefore, our results reflect the similarities between the artificial faults and their corresponding curated fault.

Table 4.1: Fault Mimicking Techniques

Approach	Year	Aim	Evaluation metric	Venue
Bug Creation for Neural Bug Detectors [74]	2022	Derive contextual mutation operators to inject more realistic faults	Syntactic match	ICST
A Study of Codex, Pre-Trained Language Model on Code [75]	2022	Evaluate code manipulation and code generation tasks such as code mutation	Exact syntactic match and manual analysis	arXiv
Self-Supervised Bug Detection and Repair [76]	2021	Produce and detect hard-to-detect faults	Syntactic match	NeurIIPS
SemSeed: Token Embeddings [77]	2021	Derive syntactic patterns that are syntactically similar to real faults	Exact syntactic match	ESEC/FSE
Mutation Monkey [28]	2021	Deriving common fault syntactic patterns	Detection Ratio	ICSE
A SBST Framework of Source Code Embedding [78]	2021	Generate adversarial code snippets that can fool a downstream task	Number of tokens changed	ICST
DeepMutation: Learning-based Mutations [61]	2020	Produce mutants syntactically similar to real faults	Syntactic distance from real faults	ICSE
Learning-based Mutations [51]	2019	Derive syntactic patterns from bug-fixes	Syntactic distance from real faults	ICSME
Wild-Caught Mutations [73]	2017	Deriving simple syntactic patterns from bug-fixes	Token similarity, Compilability	ESEC/FSE
Analysis of real faults and mutants [36]	2014	Syntactic similarities of bug-fixes and mutants	Number of tokens changed	ISSRE

other emergent faults“, implying that fault instances couple independently of their pattern. Additionally, recent studies report large semantic overlaps between simple and complex faults [32, 45, 68], questioning the role of the syntactic-based metrics.

This raises the question of whether syntactically similar, or dissimilar, faults are also semantically similar, or dissimilar. More generally, a question of whether the use of such techniques results in faults that: *a) are semantically similar to real faults, b) resemble (semantically) more faults than the dissimilar ones, and c) are subsumed by simple untargeted syntactic deviations as done by mutation testing, i.e., whether they form a useful addition to mutation testing.*

We answer the above questions by employing four fundamentally different fault-seeding techniques. These include PiTest [58], a popular mutation testing tool [57], that uses simple syntactic patterns, IBIR [54], a mutation testing tool with manually crafted fault patterns, DeepMutation [61], a deep learning-based tool that derives patterns from real bug-fixes [51], and μBERT [60], a mutation testing tool that uses a pre-trained language model (CodeBERT [117]). Hence, we investigate the ability of all faults produced by these techniques to form similar semantic deviations as the real faults of Defects4J V2 [116] and check their potential utility within mutation-based test assessment.

Perhaps surprisingly, our results show that syntactic similarity does not reflect semantic similarity, indicating that syntactic distance cannot be used as an evaluation metric in the context of mutation testing. Additionally, our results show that the real faults of Defects4J V2 can be semantically resembled and subsumed by μBERT, PiTest, IBIR and DeepMutation faults, respectively.

Moreover, we also show that simple faults introduced by IBIR subsume almost all faults introduced by other tools, being complemented in $\approx 2\%$ by PiTest and μBERT. Furthermore, when controlling the number of seeded faults, we find that μBERT resembles similar number of real faults as PiTest, while IBIR keeps a significantly higher ratio compared with the rest of the tools in $\approx$10%. Additionally, we find that other techniques probably subsume DeepMutation, whose technique produces significantly fewer mutants which are, at the same time, easier to kill (on average, DeepMutation's mutants are killed by 10 more tests compared to other tools).

Overall, we aim to raise awareness of the use of semantic and syntactic evaluation metrics in fault seeding studies. Key contributions expose the use of syntactic metrics and provide evidence related to the utility of recent fault seeding advances in the test assessment context. The findings significantly improve our understanding on the role of the faults' syntactic nature with respect to program semantics and the use of the semantic-based metrics in the context of fault-based test assessment.

4.2 Motivating Example

We demonstrate the potential differences between syntactic and semantic deviations in fault seeding by using an example from the work of Tufano et al. [51]. Consider the following example[3]:

```
//Original (abstracted) code in abstract representation (representation used by
    Tufano et al.)
public TYPE_1 remove ( int index ) {
  TYPE_2 < TYPE_1 > VAR_1 = this . VAR_2 . remove ( index ) ;
  return null != VAR_1 ? VAR_1 . get ( ) : null ; }
```

In this example, the `remove` method first accesses one of the attributes of the invoking object (`this.VAR_2`) and invokes the method `remove` recursively, saving the result in variable `VAR_1`. Then, it returns `null` in the case that `VAR_1` was `null`, otherwise, it returns the result of invoking `VAR_1.get()`. Tufano et al. in their work seed a fault that resembles exactly the real faulty instance, which is the following:

```
//Successful fault seeding by Tufano et al. (the fault resembles exactly the real
    fault)
public TYPE_1 remove ( int index ) {
  return this . VAR_2 . remove ( index ) . get ( ) ; }
```

The fault is caused because of the conditional check that is skipped and, indeed, resembles a real fault made by developers [51]. In particular, this fault removes the check on whether the result of the recursive call is `null`.

Consider now a particular fault seeded by "traditional" mutation testing, using simple syntactic changes (e.g., generated by the REMOVE_CONDITIONALS [4] operator from PiTest [58]):

```
//Fault seeded using mutation testing, simple syntax-based mutation
public TYPE_1 remove ( int index ) {
  TYPE_2 < TYPE_1 > VAR_1 = this . VAR_2 . remove ( index ) ;
  return true ? VAR_1 . get ( ) : null ; }
```

This mutant replaces the condition `null != VAR_1` by `true`, causing the guarded statements (i.e. `VAR_1.get()`) to be executed irrespective of the condition.

Interestingly, by comparing the two faulty instances, one can easily observe that they are syntactically different despite being semantically equivalent. One can also observe that a simple syntactic transformation, such as the one used by mutation testing, perfectly matches the complex transformation learned by Tufano et al. To make the differences concrete, we can compute the BLEU scores (syntactic similarity between seeded and real fault), i.e., the evaluation metric used by Tufano et al. [51], and see that the returned scores are *1* and *0.48*, respectively. However, as the mutants are equivalent and resemble a real fault, their semantic similarity is 1 despite the large difference in the BLEU scores.

The above example clearly shows that seeded faults do not necessarily need to be similar to real faults in order to resemble them. At the same time, the above example demonstrates the *fault coupling* [37], i.e., simple syntactic transformations, such as those used by mutation testing, couple to more complex faults. In this particular case, the transformation performed by mutation is significantly smaller than Tufano et al. as it has a BLEU score (syntactic similarity from the original code) of 0.85, while Tufano et al. has 0.39.

[3] this example was taken from [51, Figure 2] and demonstrates a successful case where the fault seeded by Tufano et al. matches exactly a real fault.

[4] https://pitest.org/quickstart/mutators/#REMOVE_CONDITIONALS

4.3 Research Questions

We start our analysis by recording the syntactic and semantic similarity between seeded and real faults. We perform this analysis to understand the general relation between seeded and real faults and check if there are any associations between these two variables. The existence of such a relationship will provide evidence that fault seeding techniques, instead of using grammar-based (simple) transformations as is traditionally done in mutation testing, should attempt to form frequent fault patterns and design fault seeding techniques guided by actual fault instances, in a sense follow a similar path to static code analysis [118, 119]. Therefore, we ask:

RQ1 *How semantically and syntactically similar are seeded and real faults?*

The answer to this question will provide evidence on the use of syntactic distance metrics in evaluating fault seeding methods in the context of mutation-based test assessment. More precisely, whether seeded faults with small (or big) syntactic distance from the actual faults are indeed semantically close (or far) to actual faults (at least closer than those not syntactically similar).

Syntactic evaluation metrics are used by recent research (Table 4.1), and there is no empirical evidence of their suitability in test assessment. This means that we want to check whether the techniques of Table 4.1 could be used in mutation testing studies and whether syntactic distance metrics are appropriate in this context.

Answering the above question aims to investigate general trends among seeded and real faults. However, it does not say much about the extent to which real faults are resembled by seeded ones and does not provide quantitative evidence on the real faults that can be resembled (have high semantic similarity) by syntactically close and far-seeded faults. Thus, we ask:

RQ2 *How many real faults can we (semantically) resemble by using syntactically similar and dissimilar seeded faults?*

In case we find many syntactically similar seeded faults being semantically similar to real ones, we have evidence that syntactic distance actually leads to "True Positives" and may be used in mutation testing. On the contrary, if we find many syntactically similar seeded faults that are semantically dissimilar to real ones, we have evidence that syntactic distance leads to many "False Positives". Similarly, if we find many syntactically dissimilar seeded faults that are semantically similar to real ones, then we have evidence that fault-mimicking techniques produce many "False Negatives". By putting all cases together, we have evidence on how effective fault-mimicking techniques are.

While we investigated the relationship between seeded and real faults, we have not said much about how the faults from different seeding techniques differ, w.r.t., the resemblance of real faults by different techniques. Hence we ask:

RQ3 *How do the employed techniques compare to each other in resembling real faults?*

Knowing how different techniques compare provides evidence in support of semantic fault resemblance. In particular, we check whether there is a compliment in fault resemblance or subsumption between fundamentally different approaches.

Overall, answering these questions raises awareness on the use of semantic and syntactic metrics in fault seeding and provides evidence on fault resemblance by fault seeding techniques.

4.4 Fault Seeding

PiTest (PIT) [58] is a state-of-the-art mutation testing tool that works by analyzing bytecode sequences and by looking for a possible location, i.e., instruction, to seed faults using syntactic transformation rules (aka mutant operators). The mutation operators are categorized into 29 task-specific distinct groups. Examples of groups include Conditionals Boundary and Return Value mutators, which seed variations concerning relational operators and method call return values. PiTest has over 120 mutant operators, among which are many experimental mutants used for scientific purposes. For this study, we take into consideration all mutants generated by PiTest.

IBIR [54] is a fault seeding tool that uses an information-retrieval-based fault localization model (IRFL) combined with automatic program repair inverted fix-patterns. It favours the generation of few but realistic mutants (similar to real ones). It takes as input the git repository of the program to mutate and a bug report, written in natural language and seeds faults (introducing multiple faulty versions) that emulate the fault described in the bug report.

IBIR starts by analysing the given bug report using IRFL [59] to identify locations that are likely to be related to the features impacted by the corresponding fault. It then applies fault patterns on the identified locations, which are inverted fix-patterns used in pattern-based automated program repair approaches [54]. As the fix patterns are crafted from real bug-fixes, their inverse would induce faults similar to real faults. IBIR repeats this process until exhausts all pre-defined patterns. In this study, we run IBIR on the classes changed by the bug-fix on Defects4J to exclude the mutants from other classes, and we apply all pre-defined patterns exhaustively on every location, instead of mutating only the lines predicted by the IRFL. This will allow us to explore more faulty patterns and study more broadly their relationship with real faults. Under this setting, IBIR results in producing a large number of mutants for the studied subjects.

DeepMutation [61] generates mutants by employing Neural Machine Translation [51] aka NMT, which is also used by many recent studies [48, 62–64]. It uses an NMT model trained on a large corpus (∼787k) of existing bug-fixing commits mined from GitHub repositories. It takes a Java method as input and outputs a mutant. In this study, we use beam search to generate a maximum of 10 mutants per method, which provides us with a more thorough study of the correlation with real faults.

In particular, every method is abstracted, in which the user-defined variable names and literals are replaced by predefined identifiers to obtain an abstracted code representation (as shown in Section 4.2). These abstracted code representations are then input into the trained NMT model to produce abstracted mutants. The user-defined variable names and literals are restored to obtain source-code mutants.

We use the publicly available trained model of DeepMutation [65] to generate the mutants and `src2abs` [66] tool to perform the abstraction process. We followed the guidelines [61] and used beam search to generate 10 mutants per method.

μBERT [60] is a mutation testing tool that uses a pre-trained language model (CodeBERT) to generate mutants by masking and replacing tokens. μBERT takes a

Java class, extracts tokenized expressions, which are then masked for token replacement (mutation), e.g., for binary expressions μBERT masks the binary operator, and invokes CodeBERT to complement the masked sequence. For instance, in sequence `int mid = (low + high) / 2;` μBERT mutates the variable name expression `low` by feeding CodeBERT with the masked sequence `int mid = (<mask> + high) / 2;`. CodeBERT predicts the 5 most likely tokens to replace the masked one, e.g., it predicts `low`, `mid`, `Low`, `high`, and `medium` for the given masked sequence. μBERT uses these predictions to generate mutants by replacing the masked token with the predicted ones (5 mutants are created per masked token). μBERT discards non-compilable mutants and those that are syntactically the same as the original program, which are the cases in which CodeBERT predicts the original masked token (aka duplicated mutants [40]).

4.5 Experimental Setup

Real Faults

We used Defects4J [116] v2.0,0, which contains over 800 faults with supporting build infrastructure and forms one of the largest collections of reproducible real faults for Java programs.

Every fault in the dataset consists of the faulty and fixed versions of the code, a developer's test suite accompanying the project, and information regarding the commit modified classes and the patches produced to fix the fault. The faults have been manually minimized, so every irrelevant change to the fix has been removed. The dataset also includes at least one fault-triggering test that fails in the faulty version and passes in the fixed one.

For the purpose of this study, we consider the following projects and the number of faults. We refer to these curated faults (mined through a systematic process) as real faults. We consider Defects4J faults as a good sample since it consists of real, systematically mined faults that have been built independently of the present study. From Apache Commons [120] family, consisting of a collection of projects of Java utility classes, we include commons-cli (39), commons-codec (18), commons-compress (33), commons-csv (16), commons-math (101), commons-lang (63), commons-collections (4), commons-jxpath (22). We also include projects from the Jackson [121] family, which is a suite of data-processing tools for Java, we include jackson-core (26), jackson-databind (102), and jackson-dataformat-xml (6). Additionaly, we include faults from: Mockito (28), one of the most popular mocking frameworks in Java; Jsoup (90), a Java library for HTML parsing; Gson (18), a Java library for JSON parsing and generation from and into java objects; and joda-time (26), a project for the Java date and time classes.

Defects4J faults span more than a decade of development history, making it hard to apply all faulty versions with all the studied tools. Therefore, due to unsatisfied build requirements caused by technical constraints, we do not consider certain faulty versions. The technical issues we encountered included obsolete dependencies not supported by studied tools, old testing frameworks (for example, some faults contain JUnit 3 while the tools work on JUnit4+), Java language versions (some of the tools require java 1.8+ to apply faulty patterns while the project Jfreechart (number of faults 26) and Closure-compiler (174) contain 1.5 or 1.6). Furthermore, five versions of the Math project also fall under this category. Additionally, at the time

Table 4.2: Mutants used

Fault Seeding Tool	# of Analysed Faults	# of Mutants
DeepMutation	530	119,017
IBIR	382	1,094,493
μBERT	481	286,763
PiTest	508	1,120,719

of conducting this study, we found that 5 faults from the Jsoup project were not compilable due to technical reasons, as already reported [122]. Overall, some of the studied tools have been recently developed and are versions specific, not being able to satisfy all the reported building requirements. In total, we analyzed 592 faults from 15 projects and generated a significant number of artificial faults that portray a representative dataset for our investigation.

Artificially Seeded Faults

For each selected faulty project version from Defects4J, we start by identifying the modified classes between the faulty and fixed versions. Next, we generate mutants by employing the selected mutation testing tools for the fixed version of each modified class.

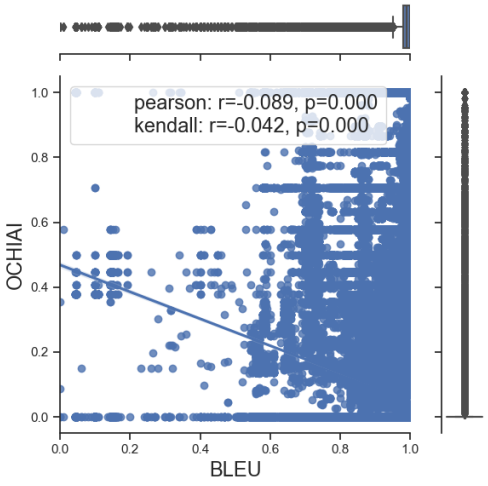
Table 4.2 records the number of faults analysed by each tool and the number of mutants generated. Overall, PiTest generated 1,120,719 mutants for the 508 faults that it was successfully applied to. μBERT was successfully applied on 481 faults and produced a set of 286,763 mutants. DeepMutation produces ten mutants per method, and thus, it produced 119,017 mutants for the 530 faults that it was successfully applied. After applying all faulty patterns from IBIR, it produces 1,094,493 mutants, per bug report, for the 382 faults that it operates.

After generation, in the mutant detection phase, we execute relevant tests from Defects4J, as those tests are carefully filtered by the framework to leave out flaky tests. We use Defects4Js predefined *compile* and *test* scripts.

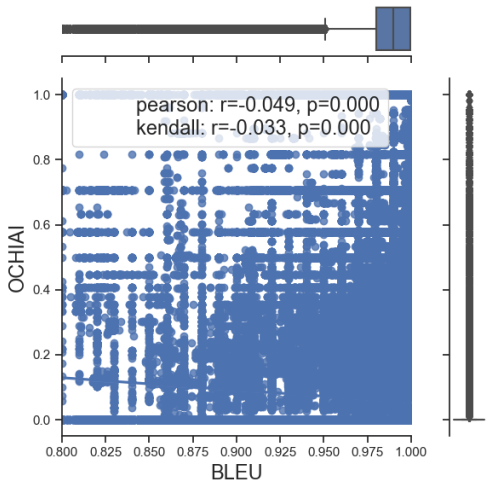
Experimental Procedure

We start by executing every generated mutant using the Defects4J framework, thus, recording the set of failing tests distinguishing (killing) each mutant. After, we proceed to compute syntactic and semantic similarities between the mutants and the corresponding faults, relying on the metrics defined in Section 2.2.4. Thus, the syntactic similarity between the mutant (artificially seeded fault) and the real fault will be measured in terms of the BLEU score, while the semantic similarity will be characterised by the Ochiai coefficient between the mutant and the fault. It is worth mentioning that, since PiTest produces the mutations at the bytecode level, we perform the syntactic similarity computation between the bytecode instruction sequences corresponding to mutants and faults.

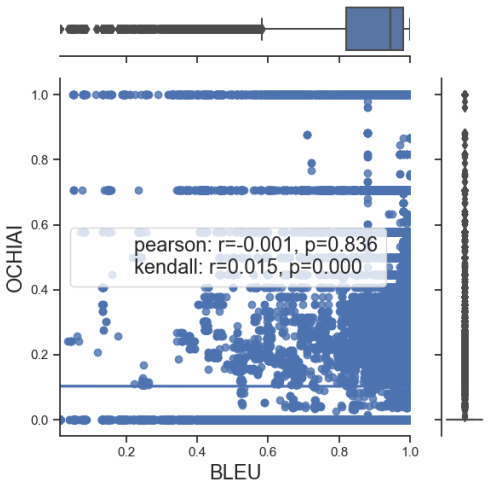
To answer RQ1, we check the existence of correlations among the syntactic and semantic similarity of the seeded and real faults. We consider all the mutants created by all studied tools and analyse several cases, when mutants located in the project classes and when mutants located on the same methods modified by the related fault fixing patch, according to the information given by Defects4J (modified-methods mutants). In all cases, we aim for general trends that indicate a relationship between syntactic and semantic, over different percentages, similarity (e.g., values greater



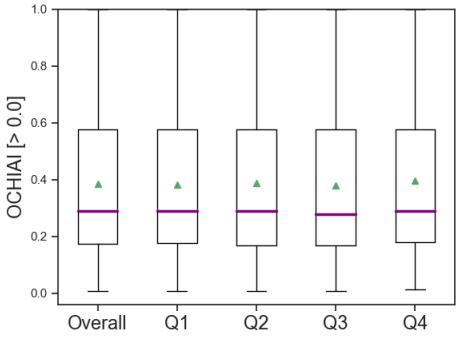
(a) At class granularity level. Weak link between syntactic and semantic similarity.



(b) When syntactic similarity is greater than 80%. We observe no link between syntactic and semantic similarity.



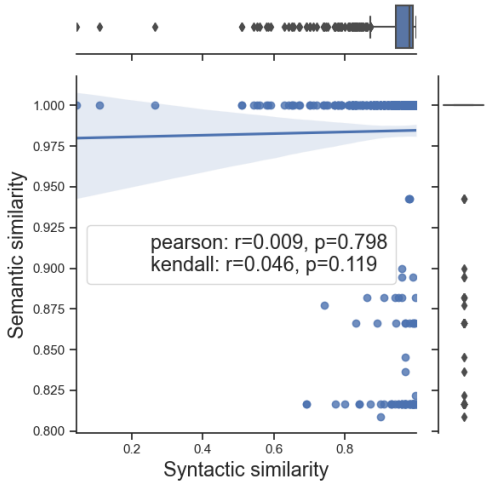
(c) At method granularity level. Faults on the same method have diverse semantic similarity.



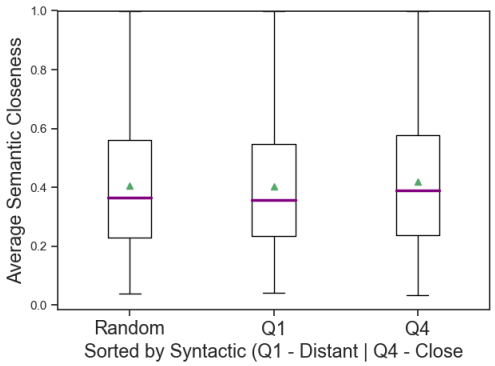
(d) Distribution of Semantic similarities across different levels of syntactic similarities. Medians depicted with lines, Means with triangles

Figure 4.1: Syntactic and Semantic Similarity between Seeded and Real Faults (RQ1)

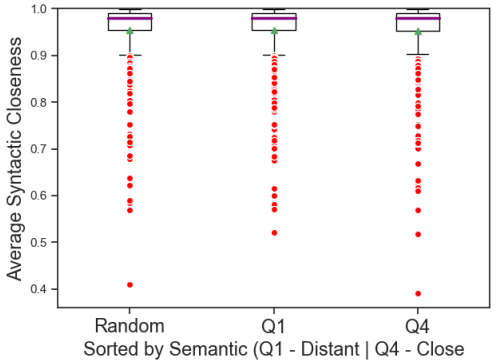
than 80%). We also check whether high scores for syntactic similarity (i.e., seeded and real faults are syntactically similar) imply high scores for semantic similarity (i.e., seeded and real faults behave the same), and whether low scores for syntactic metrics imply low scores for semantic metrics. To perform this, we sort mutants in ascending order according to their syntactic similarity. Thus, we organise them into four sorted quartiles Q_1, Q_2, Q_3 and Q_4, where Q_1 represents the most syntactically dissimilar mutants, w.r.t., the fault (lowest syntactic scores), while Q_4 represents the most syntactically similar mutants w.r.t. the fault (highest syntactic scores). For the mutants in each quartile, we also analyse their semantic similarity w.r.t. the fault, aiming to observe if there is any evidence that more syntactically dissimilar mutants behave very differently than the faults and whether syntactically similar mutants behave the same as the faults. To avoid potential threats in the quartiles composition, it is worth noting that we do not consider mutants which are extreme cases and introduce noise, such as those with Ochiai equal to zero and those that



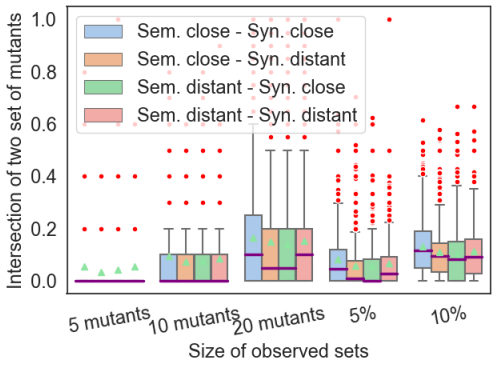
(a) Syntactic similarity of the semantic closest mutants. We observe no link between syntactic and semantic similarity.



(b) Average Semantic Closeness, per studied fault, of the most syntactically similar mutants (Q4), the most syntactically distant mutants (Q1), and a quartile of randomly sampled mutants.



(c) Average Syntactic Closeness, per studied fault, of the most semantically similar mutants (Q4), the most semantically distant mutants (Q1), and a quartile of randomly sample mutants.



(d) The average intersection size, per fault, between sets of mutants with closest and distant syntactical and semantical similarity.

Figure 4.2: Syntactic and Semantic Similarity between Seeded and Real Faults (RQ1)

syntactically exactly match the real faults.

To further strengthen our analysis, we examine whether there are faults that do not have syntactically close mutants that are semantically close. Similarly, we examine if there are faults that do have syntactically distant mutants that are semantically close. We consider semantically close mutants to be the ones with Ochiai $> .8$, and syntactically close or distant, if they belong to previously defined Q_4 or Q_1, respectively. In particular, we count the number of faults with at least one mutant that is semantically close, among all mutants that are syntactically close and distant to the fault. We also do the same count among randomly picked mutants of the same sample size as the syntactically close and distant sets of mutants. This means that a relationship between the two metrics exists if there are many more faults with high semantic similarity to the mutants that are syntactically similar than to either those that are syntactically distant or to the randomly picked ones. Furthermore, we also analyze the extreme cases (i.e., the set of semantically and syntactically most close and most distant mutants). To do so, we repeat the previous analysis, but now from the set of semantically closest mutants, per fault,

we select semantic and syntactic relations between the syntactically closest and the syntactically most distant mutant. We plot the data points on a scatter plot and further evaluate their relationship with statistical tests. This process allows us to check whether there is any potential trend in the very extreme case of syntactic similarities. Additionally, we also measure the average semantic or syntactic closeness of the faults and check whether there are any statistical differences among the sets of semantically/syntactically close or distant mutants. Therefore, we study the average semantic closeness of the most syntactically similar mutants (Q_4) and the most syntactically distant mutants (Q_1), plus the average syntactic closeness of the most semantically similar mutants (Q_4) and the most semantically distant mutants (Q_1).

Finally, we check the number of mutants that are in the intersection of the closest and distant syntactically and semantically similar mutant sets. In particular, we compute the average size of the intersection set between the most semantically close and most syntactically close mutants and compare it with that of the most syntactically distant and most semantically distant ones. Suppose the intersection of both sets – semantically close but syntactically distant mutants and vice versa – is similar to the intersection of mutants semantically and syntactically close. In that case, we can reason that even if exists a small set of mutants semantically and syntactically close, there also exist mutants which are semantically similar but syntactically dissimilar (or vice versa), indicating that semantic similarity exists independently of syntactic similarity. We also measure the average size of the intersection, since a high intersection would suggest that for the small sets of mutants, high syntactic similarity encloses high semantics and vice versa. While the low intersection would further confirm our hypothesis around the absence of a relationship between semantic and syntactic similarity. To reduce the impact of selecting an arbitrary mutant sample size on our obtained results, we repeat this analysis with the closest and most distant 5, 10 and 20 mutants as well as the 5% and 10% closest mutants w.r.t. both metrics.

To answer RQ2, we measure the ratio of real faults for which at least one mutant has semantic similarity equal to 1. We focus the analysis on the same quartile split as done for RQ1 to observe whether syntactically similar or dissimilar mutants yield higher semantic similarity over different projects.

In RQ3, we analyse the percentage of real faults that each tool can resemble. Plus, we study the ability of the tools to resemble different faults. It is noted that we consider the intersection of faults for which each tool can generate mutants. To make a fair comparison, we are controlling for the number of seeded mutants. We, thus, study tool pairs based on the number of mutants each tool generates by randomly selecting and controlling the set of mutants for each tool and calculating the tool's mean ratio to produce a mutant that resembles the real fault. We do this to avoid bias because each tool generates a different number of mutants. To avoid coincidental results, we repeated the experiment 100 times.

The dataset of generated mutants and results are publicly available in the accompanying website [123].

Statistical Analysis

To study the relationships between semantic and syntactic properties, we use a correlation metric since it analyses any statistical relationships between variables, whether causal or not. In particular, we use the Kendall rank coefficient (τ) (Tau-a) and Pearson product-moment correlation coefficient (r). In both cases, we use the

0.05 significance level. Each correlation coefficients measure similarity, taking values from -1 to 1. Values close to both ends represent negative and positive correlations, respectively. While values in a range of absolute 0.2 around zero denote absence and insignificant correlation. In our case, it refers to the degree to which a pair of variables are related. Concretely, the two variables we study characterize the syntactic and semantic similarity between faults and mutants. Particularly, we use the BLEU score as a syntactic similarity metric and the Ochiai coefficient as a semantic similarity metric (later on, during the discussion and threats to validity sections, we also include other syntactic and semantic metrics). Therefore, correlation measures whether the two variables are related and indicate a predictive relationship that can be exploited in practice, i.e., aiming at syntactic similarity instead of semantic as done by many approaches, e.g., DeepMutation. To evaluate the magnitude of difference between observed groups, we calculate the Vargha and Delaney A_{12} effect size [71]. A_{12} values over 0.56, 0.64 or 0.71 indicate a small, medium or large difference between two populations, respectively.

4.6 Experimental Evaluation

4.6.1 RQ1: Syntactic and Semantic Similarity between Seeded and Real Faults

Figure 4.1a shows the syntactic and semantic similarity values of the mutants created with different tools. Interestingly, we notice that while many of the mutants have high syntactic similarity, their semantic similarity is scattered from 0 to 1. This seems to imply that the relationship between the two metrics is weak. Figure 4.1b depicts syntactic and semantic similarity values for all mutants with a syntactic similarity greater than 0.8. We notice that the mutants behaving as faults (obtaining Ochiai 1) are both syntactically similar and dissimilar to the faults (see the plots' top values, y-axis). We also observe that most mutants that are syntactically close to real faults (BLEU near 1) behave very differently (Ochiai near 0), indicating that the relationship is weak even when seeded faults are syntactically close to real ones.

These results are on the class granularity level, and therefore their syntactic and semantic changes may be impacted by the "size" of the seeded faults. We, thus, analyse the results at method-level granularity as well. Figure 4.1c shows the syntactic and semantic similarities for the mutants that reside on the same methods as the real faults. In this case, we see a similar trend with the class-level results, i.e., both syntactically similar and dissimilar mutants behave exactly like real faults. Additionally, when syntactic similarity is close to 1, the semantic similarity is scattered from 0 to 1.

To further analyze this relationship, we investigate whether seeding faults with small syntactic distance results in semantically close faults. The key objective is to check whether there is some effect when we have high syntactic similarity as well as high semantic similarity.

Figure 4.1d shows the distribution of semantic similarities when we group mutants according to their syntactic similarity. We observe that semantic similarity is uniformly distributed between mutants that are syntactically similar and dissimilar to the real faults. This observation is visible even when considering only mutants with very close semantic similarity. This evidence that smaller syntactic transformations do not imply smaller semantic changes; at the same time, bigger syntactic changes

do not imply bigger semantic changes.

Additionally, we find that 39% of studied faults do not have syntactically close mutants that are semantically close (Ochiai > .8). This percentage is roughly the same as that of syntactically distant mutants that are semantically close (41%). This observation indicates no relation between metrics since analysing either syntactically close or syntactically distant mutants, leads to the same number of faults (w.r.t., mutants being semantically similar independently of their syntactic similarity). Figure 4.2a) shows the same data for the semantically closest mutants. From these data, we observe no relationship even in the extreme cases of the most syntactically close and distant mutants. By observing this plot, it becomes evident that there is no relationship between the two variables (note Pearson and Kendall correlation coefficients to be both < .1, indicating no relationship; plus, via Wilcoxon statistical test, we find no sign that higher syntactic similarity leads to higher values of semantic similarity ($p < 0.05$).

We also studied what is the average closeness of the semantically and syntactically closest fault mutant pairs (respectively, the figures 4.2b and 4.2c). Our results show that the average semantic closeness of the syntactically closest fault mutant pairs is 0.4192; the closeness of the syntactically distant mutant pairs is 0.4020, while the closeness of the randomly picked mutants is 0.4058. The difference between these averages is negligibly small to suggest a link between syntactic and semantic similarity metrics. Wilcoxon statistical test further confirms these observations by showing no statistically significant difference between the average semantic closeness of syntactically close and the average semantic closeness of syntactically distant mutants ($p<0.05$), plus, no statistically significant difference between the average semantic closeness of syntactically close and randomly sampled mutants. Similar results are observed, leading to the same conclusions when calculating the average syntactic closeness of the semantically closest (0.9550) and semantically distant mutants (0.9530) and randomly sampled mutants (0.9543).

Figure 4.2d depicts a follow-up analysis in which we study the intersection of a set of semantically closest and a set of syntactically closest mutants. Our results show that the intersection between the top five syntactically and semantically closest mutants is of only 5%, refuting any implication between both metrics in 95% of the cases. We observe the same pattern and small semantic and syntactic similarity overlapping for the top 10, 20 mutants, and 5 and 10 percent of mutants, 10%, 17%, 8% and 13%, respectively. Our results discover a small difference (2%) when comparing semantically and syntactically closest mutants overlapping with the overlapping of the semantically closest and syntactically distant ones (and vice versa). This analysis again confirms our previously shown evidence, i.e., even for a set of mutants with the metrics closest to real faults, semantics behaves independently of syntactic and vice versa. By examining the faults that do not have syntactically close mutants that are semantically close and the average closeness of semantically and syntactically closest fault mutant pairs, we confirm the previous results and find no indication of a pattern or relationship that would suggest that syntactic measurement leads to the semantic closeness of mutants and real faults.

Here, it is important to mention that the plots used throughout this section only consider BLEU score since it showed to be more sensitive to small changes, suitable for quantifying small mutations. Please refer to the supplementary material for the Jaccard and Cosine scores.

Table 4.3: RQ2 Percentage of resembled real faults - Quartiles represent mutants sorted by syntactic similarity

Project	Faults	∃ Semantic Mutant	Q1	Q2	Q3	Q4
Cli	39	71.79	53.84	61.53	48.71	58.97
Codec	18	72.22	44.44	44.44	44.44	50.0
Collections	4	75.0	75.0	25.0	50.0	50.0
Compress	33	93.94	81.81	90.90	78.78	81.81
Csv	16	81.25	37.50	75.0	62.5	43.75
Gson	18	72.22	50.0	61.11	38.89	66.67
JacksonCore	26	88.46	73.07	65.38	73.07	76.92
JacksonDatabind	102	77.45	60.78	57.84	53.92	62.74
JacksonXml	6	83.33	83.33	83.33	50.0	83.33
Jsoup	90	12.22	11.11	12.22	7.77	7.77
JxPath	22	68.18	54.54	54.54	59.59	63.63
Lang	63	68.25	58.73	63.49	64.49	60.31
Math	101	67.32	54.45	52.47	59.40	59.40
Mockito	28	60.71	39.28	32.14	50.0	46.42
Time	26	65.38	50.0	53.84	50.0	57.69
Total/Average	592	70.51	55.19	55.55	52.67	57.96

Many seeded faults behave similarly to real faults (high semantic similarity), while they have low syntactic similarity to real faults. We find no evidence suggesting any link between syntactic and semantic similarity, except in the cases of exact matches.

4.6.2 RQ2: Semantically Resembling Real Faults

Table 4.3 summarizes the results related to the percentage of resembled faults, i.e., having at least one mutant that semantically resembles the fault. The column Faults refers to the number of faults studied per project, and column ∃ Semantic Mutant refer to the percentages of faults with at least one semantically similar mutant. We sort mutants based on their syntactic similarity and group them into 4 buckets/quartiles (columns Q1, Q2, Q3 and Q4 in increasing order). Table 4.4 records the ratio of mutants that are semantically similar per fault (column Ratio) and the ratio per syntactically similar bucket.

For the 592 real faults, we observe that seeding techniques can produce at least one artificial fault that is semantically similar to the real one for 417 of them (70.51%). From the distribution of the results over different quartiles, we see the absence of trends suggesting that higher syntactic similarity does imply higher semantic similarity. For example, the project with the highest number of faults studied (JacksonDatabind – 102 faults) has quite similar ratios among the quartiles, i.e., 60.78%, 57.84%, 53.92%, 62.74%. Overall, on average, across all studied faults, the distribution of faults that can be resembled is 55.19%, 55.55%, 52.67%, 57.96%.

Table 4.4 shows a similar distribution across quartiles. On average, the percentage of seeded mutants with semantic similarity is 11.77%, while over different levels of syntactic similarity, the distribution is 3.01%, 2.95%, 2.59% and 3.20%, respectively.

Our results indicate that real faults are resembled by artificially seeded faults independently of their syntactic similarity.

Table 4.4: RQ2: Mean ratio of mutants resembling real faults - Quartiles represent mutants sorted by syntactic similarity

Project	Ratio	Q1	Q2	Q3	Q4	Exact Matches
Cli	12.79	2.61	2.82	3.15	4.05	0.13
Codec	5.54	1.33	0.94	1.38	1.61	3.74
Collections	12.75	6.25	1.15	3.0	2.0	0.17
Compress	12.39	2.72	3.48	2.36	3.72	2.72
Csv	9.12	0.93	4.3	1.93	2.0	0.25
Gson	7.94	1.38	3.0	1.05	2.61	0.09
JacksonCore	19.46	4.88	4.23	4.42	6.00	0.54
JacksonDatabind	14.20	3.17	3.79	3.48	3.71	0.32
JacksonXml	11.66	4.33	2.33	2.33	3.0	0.37
Jsoup	5.13	1.12	1.68	1.01	1.12	0.86
JxPath	13.77	3.40	4.04	2.86	3.31	0.02
Lang	22.06	5.23	5.28	5.49	5.96	1.29
Math	12.16	2.98	2.61	2.78	3.71	0.10
Mockito	8.42	2.60	1.42	1.75	2.67	0.58
Time	9.26	2.15	2.76	1.92	2.23	0.08
Total/Average	11.77	3.01	2.95	2.59	3.20	0.75

Table 4.5: RQ3: Mean ratios of mutants resembling real faults.

Fault Seeding Tool	Overall	Q1	Q2	Q3	Q4
DeepMutation	1.99	0.39	0.42	0.56	0.59
IBIR	2.79	0.53	0.58	0.55	1.05
μBERT	2.94	0.69	0.54	0.74	0.89
PiTest	2.66	0.47	0.54	0.65	0.93

4.6.3 RQ3: Comparing Seeding Techniques

Table 4.5 records the mean ratios of mutants that resemble the real faults. We observe that, on average, between 1.99%-2.94% of mutants resemble the real faults, independently of their syntactic similarity. Table 4.6 records the percentage of real faults that were resembled by at least one mutant produced by each tool. We observe that PiTest resembles 65.11% of the real faults, while μBERT resembles 61.39%, DeepMutation 9.76% and IBIR 76.44%. Interestingly, μBERT and PiTest resemble a similar number of the real faults, while μBERT identifies 0.6% of real faults not identified by other tools (Figure 4.3), while PiTest identifies 1.7%. IBIR resembles 76.44% of faults, and 5.2% is not identified by the other tools. However, when we compare the performance when controlling the number of seeded faults (Table 4.7), we observe that when generating a number of mutants equal to the number of mutants generated by DeepMutation, μBERT, IBIR and PiTest perform similarly, resembling real faults with 43.84%, 45.34% and 41.72%, respectively. When generating the same number of mutants with μBERT, we observe that both IBIR and μBERT outperform PiTest by around 10%. And when the number of seeded faults is higher, (equal to what PiTest produces), IBIR outperforms PiTest for 8.05% of real faults resembled. We observe that PiTest performance is the lowest indicating a large number of redundancies. Regarding exact matching, IBIR successfully resembles 1.98% of the faults, outperforming the rest of the tools, which only managed to match around 1%. DeepMutation resembles real faults also resembled by other tools, indicating that it is probably subsumed by them.

Table 4.6: RQ3 Percentage of resembled real faults - Quartiles represent mutants sorted by syntactic similarity.

Fault Seeding Tool	Total	Q1	Q2	Q3	Q4	Exact Matches
DeepMutation	9.76	5.11	6.04	6.04	5.11	1.07
IBIR	76.44	38.60	46.04	59.06	59.65	1.98
μBERT	61.39	39.06	34.41	41.39	45.11	1.24
PiTest	65.11	42.32	45.11	46.97	56.27	0.01

Table 4.7: RQ3 Percentage of resembled real faults when the number of mutants is controlled - different tools used as a baseline

Fault Seeding Tool	Mutant Selection Control Groups			
	DeepMutation	μBERT	PiTest	IBIR
DeepMutation	9.76	-	-	-
μBERT	43.84	61.39	-	-
PiTest	41.72	51.48	65.11	-
IBIR	45.34	62.34	73.16	76.44

IBIR resembles 76.44% of the real faults, PiTest (65.11%), μBERT (61.39%) and DeepMutation (9.76%). When seeding the same number of mutants, IBIR shows better performance than all other tools (≈10%).

4.7 Discussion

4.7.1 Use Cases of Fault Seeding

Over the years, fault seeding has served multiple purposes, e.g., testing, dependability analysis, debugging etc., as found by the survey work of Papadakis et al. [8]. The survey also identifies that fault seeding is primarily used in research for *a) mutation-based test assessment*, i.e., empirical and experimental evaluation of test techniques (seeded faults are used as a means to compare test techniques based on the number of faults they detect), and *b) mutation-guided testing*, i.e., guiding testers to write test cases (by using seeded fault as objectives to be covered).

These two use cases are often confused and considered as being equal [9], while in fact they are different—though related. This is not only because of the different underlying processes but also due to the involved assumptions.

Process: in mutation-based test assessment, a) case, tests are independently produced by external parties, while in mutation-guided testing, b) case, tests aim at detecting specifically targeted faults. This implies an untargeted case (case a)) that starts from independent test cases and aims at estimating their fault detection potential versus a targeted one (case b)) that starts from seeded faults then goes to tests that aim at finding real faults.

In terms of injected faults, this difference means that in case a), one needs seeded faults that are as close (semantically) as possible to the real ones in order to estimate their test potential, while in case b), one needs seeded faults that lead to tests that detect faults. In essence, real faults in case a) should be detected *by every test case that detects a seeded fault*, while in case b) should be detected by *the subset of test*

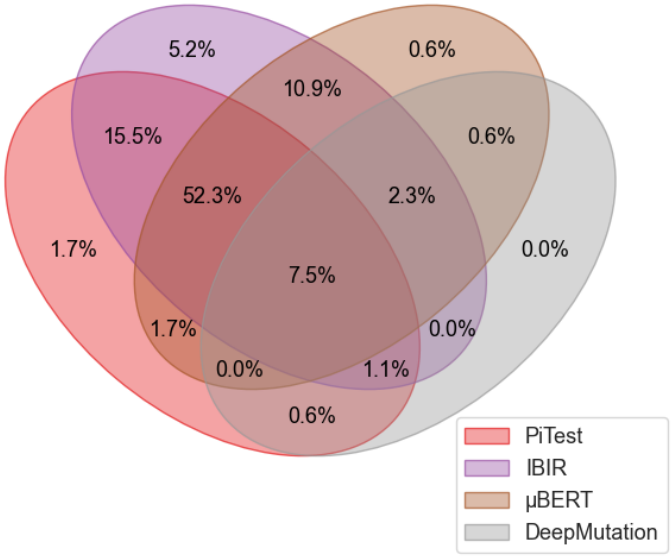


Figure 4.3: Real faults with at least one semantically similar mutant by each tool. PiTest - IBIR - μBERT - DeepMutation

cases that detect a seeded fault. For example, consider the seeded faults M_0 and M_1 from Figure 2.2 that both are detected by tests that also detect the real fault B. This means that for the case b) both M_0 and M_1 are equally useful since they can both lead to a test that detects the real fault. However, for the case a) (test assessment), M_0 is better than M_1 since it does not underestimate the fault detection potential of t_1.

Assumptions: in mutation-based test assessment, a) case, it is assumed that the tested/asserted program behaviour is correct, while in mutation-guided testing, b) case, this is not the case (testers judge the observed behaviour). These assumptions often imply differences in both the definition of fault detection (deciding whether mutants are killed) and the used artifact (by experimental studies). The difference in the definitions is usually that, in a) case, the behaviour of seeded faults is contrasted with that of the specifications (through test assertions) while, in b) case, that is contrasted with that of the program under test, which may or may not be correct. Similarly, experiments targeting case a) are applied on program versions where test suites pass, while in case b) experiments are applied on buggy program versions where test suites fail, i.e., detect some real faults.

In essence, the differences in the assumptions necessitate a different treatment in the way seeded faults are detected and used. For instance, it is unclear what causes a test failure when executing a test in a buggy program version where a fault has been seeded. Typically this case is treated by considering the behaviour delta between the buggy and the faulty seeded versions (seeded on the buggy version) [9]. However, this behaviour delta between the buggy and the seeded fault is different from the behaviour delta between the specifications and the buggy version as has been demonstrated by Chekam et al. [9].

Similarly, the seeded faults on the fixed and the buggy program versions differ. Consider, for example, the case of an omission fault where an if condition is missing. This fault is easy to emulate if one seeds faults in the non-buggy version by simply deleting the related condition. However, the fault is hard to detect if one seeds faults

in the buggy version since the related code is not there (it is hard to seed a fault that can detect such a bug).

The above discussion aims at detailing the differences among the two main use cases of fault seeding and motivating the need for appropriate metrics that fit well with the envisioned application use cases. In the following subsection we demonstrate the importance and appropriateness of using semantic similarity, as opposed to fault detection estimates, used by previous studies [29, 112], in the context of test assessment.

4.7.2 Semantic similarity Vs. Fault Detection Probabilities in Test Assessment

Mutation testing has long been based on the notion of fault coupling [10, 37] that assumes couplings among different types and (syntactic) sizes of faults. This assumption has been validated by recent studies that report large semantic overlaps between simple and complex faults [32, 37, 45]. Undoubtedly, faults that couple with real ones are the most important when one performs mutation-guided testing (the b) use case of fault seeding that was described in the previous section) since the starting point is the seeded faults. However, this is not necessarily the case for the test assessment (the a) use case of fault seeding that was described in the previous section) since we want accurate estimations of test effectiveness.

Previous studies [29, 112] have defined *FDP*, as a probabilistic form of fault coupling, as a target metric of fault seeding that measures subsumption of a real fault by a mutant, in the context of mutation-guided testing. In essence, the metric form an approximation of the *fault detection probability*, w.r.t. a real fault B, of the tests that detect a seeded fault M. The metric is, therefore, computed as the ratio of the number of tests detecting both M and B to the number of tests detecting M. Precisely, $FDP(B, M) = \frac{|fTS_B \cap fTS_M|}{|fTS_M|}$, where fTS_B and fTS_M denote the set of tests detecting the fault and killing the mutant, respectively.

To illustrate this concept, let us consider the example of Figure 2.2. The mutant killing matrix is presented in Table 2.1, together with the semantic similarity and fault detection probabilities between the mutants and the fault (columns $Ochiai(B, M_i)$ and $FDP(B, M_i)$, respectively).

An interesting observation from Table 2.1 is that the FDP of mutants M_0 and M_1 is 1, since all the tests killing them also find the fault, but the *Ochiai* coefficients (semantic similarity metric) distinguish between these mutants ($Ochiai(B, M_0) = 1$ but $Ochiai(B, M_1) = 0.70$). This example shows that in the case of mutation-guided testing, both M_0 and M_1 are of equal value (since targeting either of these faults leads to tests that detect the real fault). However, in the case of test assessment, M_0 is preferable over M_1 since it does not underestimate the test potential of t_1. Consider, for example, 10 combinations of test suites of two tests *(*$C(|\{t_{0...4}\}|, 2) = \binom{|\{t_{0...4}\}|}{2} = 10$*)*. M_1 mistakenly evaluates the fault detection potential of $t_1 - t_2$, $t_1 - t_3$, $t_1 - t_4$ (3 out of 10) as being non-effective, while M_0 correctly evaluates them all. Similarly, M_6 is better than M_1 since it mistakenly evaluates the fault detection potential in 2 out of 10 cases, i.e., considers that $t_2 - t_3$, $t_2 - t_4$, are effective while they are not.

The above reflects the differences between the metrics of Table 2.1. The *Ochiai* coefficient for mutant M_6 is 0.82, being the second top-ranked fault, making it preferable over M_1. While FDP would prefer M_0 and M_2 over M_6, which is actually

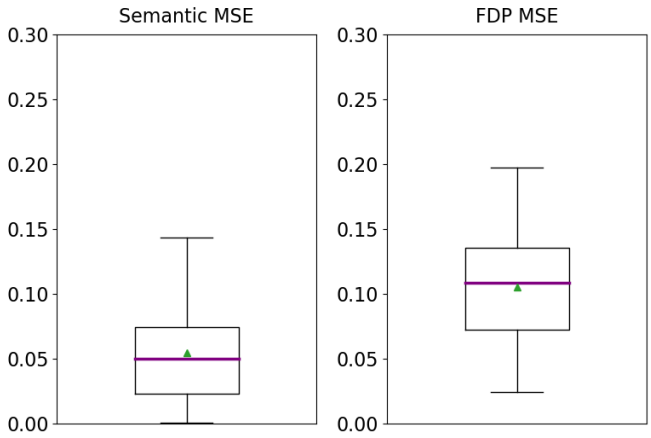


Figure 4.4: Semantic Similarity yields significantly lower Mean Squared Errors in its assessments than Fault Detection Probability. The results are statistically significant with 99% of confidence and with a high effect size of 0.82

the case if one is guided by the faults. Another difference occurs between mutants M_2 and M_5; while *Ochiai* for M_5 is higher than for M_2, the opposite happens when we use FDP.

These examples aim at demonstrating the use of the metrics in the fault injection context. By considering these examples and the differences outlined in the previous section, it should be clear that both metrics are closely related, meaning that one could approximate the other, but semantic similarity (*Ochiai*) is a better fit for test assessment, while fault detection estimates (FDP) are a better fit for mutation-guided testing.

To empirically demonstrate these differences, we design a related test assessment experiment, reflecting the example given above. We thus, treat semantic similarity and fault detection estimates as estimators of the actual test assessment potential of the mutants and compute their related error. To measure this, we use the Mean Squared Error (MSE) of the estimators with respect to the actual ratios of fault detection potential of test suites, i.e., the ratio of test suites that detect both the seeded and the real fault over the number of test suites that detect either of them. The MSE is typically used as a quality indicator of the estimated values, in this case, semantic similarity and fault detection estimates, and aims at reflecting the associated risk of using them.

In this analysis, we randomly pick 100 test sets of equal size, determined by the ratios of tests detecting the real faults, in order to have a balance between failing and passing sets. We then computed the MSE values on the faults of our dataset, which have more than one failing test, 114 faults in total, of both semantic similarity and fault detection estimates for all available mutants. We selected cases with more than one failing test because the metrics are almost the same in all other cases.

Figure 4.4 depicts the MSE errors of both estimators. These results show that both metrics have low error rates, with semantic similarity yielding statistically significantly lower errors with sizable differences, i.e., A_{12}, indicating that semantic similarity is better suited for test assessment.

When exploring further, we observe that despite the clear relationship between both semantic metrics, they prioritize mutants differently. For instance, Figure 4.5

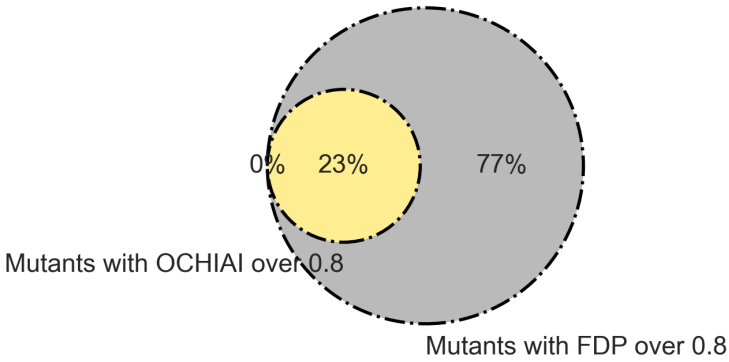


Figure 4.5: Overlapping between the set of mutants with Ochiai coefficient greater than 0.8, and the set of mutants with FDP greater than 0.8. The figure shows significantly low overlapping between mutant sets, indicating that the metrics are appropriate for different use cases even though they are strongly related.

shows two sets of mutants - one with high Ochiai and one with high FDP - and provides evidence that the overlapping is significantly low; only 23% of the mutants with a FDP greater than 80% have also an Ochiai greater than 80%. This may explain why the sets of mutants are distinct for different use cases, and particularly the preference of the semantic similarity metric (Ochiai) for test assessment.

Nevertheless, we also study if there is any relationship between syntactic similarity and FDP (fault subsumption) as a semantic similarity metric, instead of using Ochiai. We find that there is no relation and the message conveyed is the same as when using Ochiai as a semantic similarity metric, that real faults are subsumed independently of their syntactic similarity. On average all tools subsume 80.39% of studied faults, while different quartiles $Q1$, $Q2$, $Q3$, $Q4$ show 66.41%, 66.40%, 63.02% and 67.78%, respectively. We also find that different tools (PiTest, IBIR, μBERT, and DeepMutation) subsume 74.41%, 85.58%, 71.16%, 12.09% of real faults, respectively - keeping the same distribution as we report studying semantic similarity. Overall, the answer to the studied research questions would be similar if we would adopt a related but different semantic metric such as FDP. We provide further figures and tables regarding the subsumption of faults with FDP on the complementary web page: `https://mutationtesting-user.github.io/bugs_vs_mutants/`

4.7.3 Sensitivity to Program Locations

One may wonder how sensitive the syntactic and semantic similarity metrics are with respect to the seeded faults' locations. In other words, these metrics may reflect the utility of the locations and not of the faults. Thus, we study the variance of the syntactic and semantic similarity of mutant pairs generated from the same location (we do not consider DeepMutation since it creates only one mutant per method). Figure 4.6 records the distances between the syntactic and semantic similarity of mutant pairs, taken from a) the same randomly picked locations and b) from the bug-fixing locations. We observe that while there is almost no syntactic difference between mutants from the same location, the semantic similarity varies significantly. There are a few outliers in which syntactic similarity varies up to 16% between mutants from the same location. Some PiTest mutations remove a complete line or replace entire boolean conditions with true (as shown in the motivating example), affecting the bytecode generated. Figure 4.6 records a similar trend for all studied tools. Please refer to the accompanying website for results related to additional

studied syntactic metrics. [123].

Overall, these results support the conclusion that there is no link between syntactic and semantic similarity. Interestingly, even small syntactic changes in the same instruction can have a large and diverse impact on the program semantics.

Seeding Faults with DeepMutation

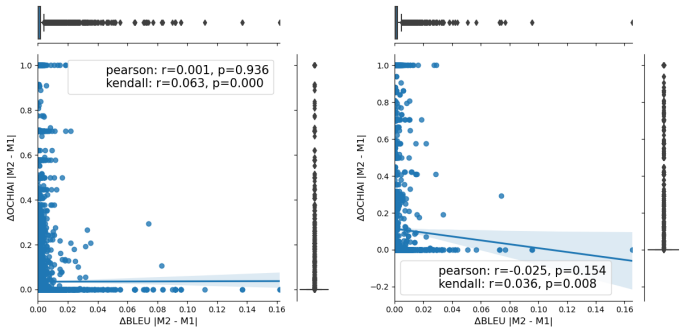
Guided by intuition, one would assume that DeepMutation - as a mutant generation tool based on deep learning - shall be able to generate more complex (stronger) faults and thus complement and subsume other tools. However, we observed that the tool seeds faults which are easy to kill or, in other words, the tests cannot miss behaviour produced by those kinds of seeded faults as they are overly complex, i.e., replacing too many code elements and thus significantly changing code logic. In particular, we found that DeepMutation mutants are, on average, identified by 10 tests more than the mutants from other tools. Moreover, we analysed seeded faults and found that from all seeded, 46% does not compile. In contrast, 29% are duplicates or not killed — leaving 25% of seeded faults suitable for mutation analysis. Out of those seeded faults, no fault can resemble faults that other tools cannot (Figure 4.3), making the tool subsumed by other tools.

```
// Defects4j JxPath project - Bug version 7
private int compare(Object l, Object r) {
    double ld = InfoSetUtil.doubleValue(l);
    double rd = InfoSetUtil.doubleValue(r);
    return ld == rd ? 0 : ld < rd ? -1 : 1;
}
```

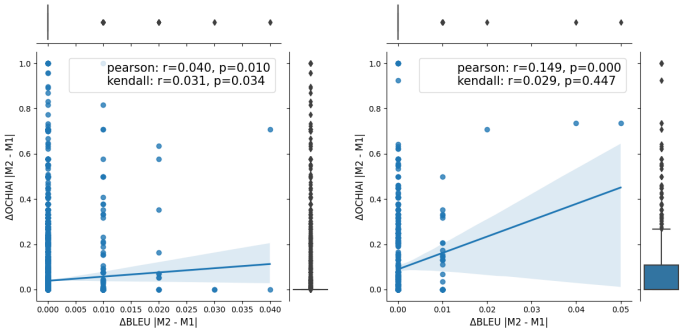
```
// DeepMutation mutant - does not compile due to the return type
private int compare(Object l , Object r) {
    double ld = InfoSetUtil.doubleValue(l);
    double rd = InfoSetUtil.doubleValue(r);
    return ld == rd ? 0 : ld;
}
```

Additionally, to further provide qualitative remarks on seeded faults, we provide two examples in which seeded faults indicate the technique's potential, even though the mutants are considered weak. In the example of a mutant in a method taken from project JxPath, the mutant alters the ternary operator condition, which is a syntactically adequate location for a bug; however, the mutation does not consider the return type, which results in a compilation error. In the second example, where we observe a mutant from the Mockito project, the technique removes the complete conditional check of whether an object is a valid instance, resulting in a weak mutation that cannot escape a test suite. Instead, the conditional check should be altered instead of removed, as those faults are subtle and represent a mistake that a programmer would make. However, the mutant from the second example alters the core logic of the code, which makes it unlikely to escape the majority of test cases (Ochiai metric is ≈ 0.02.

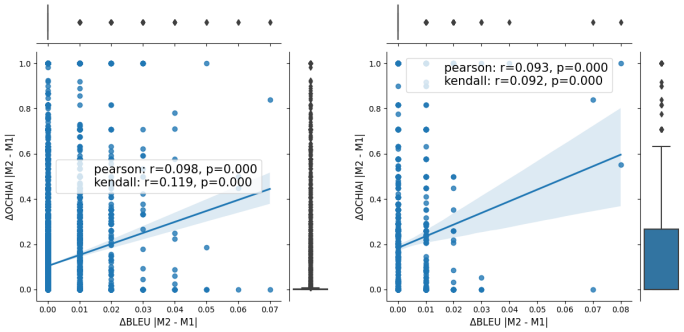
```
// Defects4j Mockito project - Bug version 20
public MockHandler getHandler(Object mock) {
    if (!(mock instanceof MockMethodInterceptor.MockAccess)) {
        return null;
    }
    return ((MockMethodInterceptor.MockAccess)
        mock).getMockitoInterceptor().getMockHandler();
}
```



(a) PiTest - Random lines (b) PiTest - Changed lines



(c) μBERT - Random lines (d) μBERT - Changed lines



(e) IBiR - Random lines (f) IBiR - Changed lines

Figure 4.6: Sensitivity of mutants from the same location ($\Delta BLEU\,|M_2 - M_1|$ over $\Delta Ochiai\,|M_2 - M_1|$). Small syntactic changes lead to diverse semantic changes.

```
// DeepMutation mutant - mutant compiles but has Ochiai 0.02 and it cannot escape
    any test
public MockHandler getHandler(Object mock) {
    return ((MockMethodInterceptor.MockAccess)
        mock).getMockitoInterceptor().getMockHandler();
}
```

Moreover, using semantic similarity for reinforcement metrics for learning algorithms should bring more practical artificial faults than using syntactic metrics. This knowledge can provide practitioners with insights and pave the way to discover more fine-grained metrics to approximate semantics over existing ones.

4.7.4 Implications for practice

Our key finding regards the mismatch of syntactic and semantic similarity. This implies that research studies should not attempt to approximate semantic similarity

through syntactic similarity (as currently done by many methods). Therefore, researchers should focus on measuring the semantic sensitivity of their results and perhaps attempt learning based on semantic features rather than solely syntactic ones. For instance, DeepMutation aims at mimicking syntactically real faults thereby resulting in being relatively weak and probably subsumed by traditional mutants.

Additionally, our results shed light on the semantic similarity aspect of real and seeded faults that has not been researched by the mutation testing literature. Therefore, we believe our work can improve our understanding of this fundamental relationship and offers a starting point for future research on the semantic approximation of real faults.

Moreover, the Ochiai score is a metric used by previous work in order to approximate the semantic similarity between a bug and a seeded fault. We thus expect it to diverge from the true similarity due to the following two reasons. First, there is some noise due to the incompleteness of the test suit used, and second, due to the coarse granularity level of the test failures, i.e., we consider test failures in our approximation and not the exact program output. We aimed at mitigating both factors by using mature and strong test suites, augmented with automatically generated tests and manually estimated the level of error due to the application of semantic similarity at the test failure level(found it $\approx 2.8\%$ as reported in Section 4.8).

In our work, we also report on the existence of a specific category of seeded faults - the seeded faults that are syntactically dissimilar but semantically similar to real bugs. This category can provide implications for practice towards increased test assessment by providing targeted diversity represented through code comprehension.

4.8 Threats to Validity

To reduce external validity threats, we selected a new and significant benchmark of faults that have not been used by previous studies. We excluded some faults for technical reasons, making our study with 592 faults from 15 mature and well-tested open-source real-world projects. Nevertheless, we do not exclude the generalization threat in other domains. As we already discussed, while conducting our experiments, we could not compile or run all the faulty program versions available in Defects4J.

One may argue that Defects4J is not the most representative dataset of all real faults. We acknowledge this threat as the scope of real faults is, in theory, infinitive. However, we believe that Defects4J is the most representative dataset available to research and the most studied, which reduces the risk that the results will not generalise to other real faults. Moreover, upon release of the new datasets in future, we motivate practitioners to replicate our study.

Internal validity threats emerge from the tools' specificity and configuration, such as the number of mutants they generate and the source-code locations they are applied to. For instance, DeepMutation produces fewer candidate mutants than any other tools, while μBERT, IBIR, and PiTest generate mutants everywhere, in a brute-force way. To mitigate this threat, we analyze the effectiveness of the tools under *the same number of mutants and the same locations* and observe a similar trend.

Unfortunately, we did not manage to compile and run the latest master-branch [124] version available of DeepMutation. We thus had to handle the tool from the resources and pre-trained artefacts provided in the repository.

Additional threat mitigation actions involved the analysis of mutants at different

granularity levels (class, method, and location of patch). We also restricted the scope of analysis to the artefacts where the bug fixes were available to reduce noise from irrelevant mutants and tests. We ensured that all mutants reside on the same class/method/statement as the target faults. Thus, we compare different mutated versions w.r.t. their similarities and distances from the corresponding fixed and faulty version.

To measure semantic similarity, we used the well-known Ochiai score that has been regularly used in the fault-seeding community as a representative metric to capture the semantic similarity between a seeded and real fault. The metric takes into consideration test execution output and neglects the lower level of granularity, i.e., whether the test crashed due to error or due to failure, which may result in a divergence of behaviour between a bug and a mutant.

To study this threat, we conducted a manual follow-up analysis. In the manual study, due to the inability of Defects4J to provide fine-grained test outputs, we sample randomly from the mutant pool and analyze them in isolation. By taking around 4000 mutants that obtained an Ochiai coefficient equal to 1, which gives a confidence level of 99% with a confidence interval of 2%, we found that just approximately 2.8% of mutants show potentially different behaviour than the real fault (one triggers a failure while the other triggers an error), even though the same test captures them. This percentage of mutants does not impact the message we want to convey with our study; nevertheless, we found this concern necessary to inform practitioners.

In addition to BLEU scores, for measuring syntactic similarity, we also used Cosine [125] and Jaccard [126] similarity coefficients [123]. The results did not show any significant differences w.r.t. the ones of BLEU scores. It is worth noting that these metrics appear less often in the literature, and in an attempt to keep the story clear and concise, we provide results on these metrics on an accompanying website. Nevertheless, please refer to the accompanying website [123] for additional details on using Cosine and Jaccard similarity.

Overall, our study aims to raise awareness of using semantic and syntactic evaluation metrics in fault seeding studies since understanding is shaken by the rapid integration of "intelligence" in the current software testing practices. In the spirit of discarding all misinterpretations, we declare that it is far from our intention to generalize the studied approaches and raise the claims regarding their future usage as we believe in very much needed future studies on their utilities and effectiveness that will undoubtedly result in new tools. Our study targets the current state-of-the-art tools and embedded underlined approaches to shed more light on the studied area and pave the way for future work.

4.9 Conclusion

We investigated the link between syntactic and semantic similarity of seeded and real faults in the context of mutation-based test assessment. Our results showed that many seeded faults behave similarly to real ones (they have high semantic similarity), while at the same time having low syntactic similarity (to real faults). We also observed the opposite case, i.e., faults with high syntactic similarity having low semantic one. This means that we found no evidence suggesting any link between syntactic and semantic similarity, except, of course, in the case of exact matches. When considering the ability of fault injection tools to resemble real faults, we found that 65.11%, 76.44%, 61.39% and 9.76% of the real faults in Defects4J

V2 are semantically resembled by PiTest, IBIR, μBERT and DeepMutation faults, respectively. For further inquiry about our data, figures and examples, please refer to the webpage of the study:

`https://mutationtesting-user.github.io/bugs_vs_mutants/`

5

Commit-Aware Mutation Testing

The questions concerning change-aware test criteria have received very little attention from research. In the current trend of continuous software development w.r.t., building software artefacts through increments, this is very unfortunate since it would allow practitioners to focus on the particular changed program behaviours a.k.a., the delta of behaviours between the stable and modified software version, instead of a whole program. This chapter recognises the potential and proposes commit-aware mutation testing that allows commit-relevant mutation-based test assessment to capture changed program behaviours and thus evaluate to what extent code modification is tested. In particular, the proposed approach characterises commit-relevant mutants as a novel category of mutants representing change-aware test criteria suitable to capture test requirements affected by a code change and, thus, in turn, accurately and adequately guide testing of program modifications. Among different empirical findings, in detail described in this chapter, our results reveal that commit-relevant mutants have a 30% higher fault-detection ability over other strategies.

This chapter is based on the work published in the following journal article:

- Milos Ojdanic, Wei Ma, Thomas Laurent, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. *"On the use of commit-relevant mutants."* 2022, *Empirical Software Engineering Journal.* 27, 5 (Sep 2022). `https://doi.org/10.1007/s10664-022-10138-1`

Contents

5.1 Introduction

Software systems are subject to regular modification during their life-cycle. Modifications are usually made in order to maintain and improve the software (fixing bugs, refactoring, or improving code quality), or to include new features. In either case, automated testing is used as gate-keeping, i.e., to establish confidence that the modifications did not break any of the previously developed program functionalities.

In such scenarios, developers often assume that the previous (operational) version of the system was stable and correct. Therefore, they are interested in testing only the behaviour delta of the changes they performed. This means that they want to assess the delta of behaviours between their pre- and post-commit system versions. For such cases developers need metrics quantifying the extent to which they have tested the error-prone program behaviours affected by their changes. Unfortunately, little research has been devoted to forming such change-aware test criteria. Change-aware test criteria would offer a viable, from an economic perspective, way of dealing with the continuous software modifications, as one would only focus on the particular program changes or commits.

Mutation testing has long been established as one of the strongest test criteria [8]. It operates by measuring the extent to which test suites can distinguish the behaviour of the original program from that of some slightly altered (syntactically altered) program versions, which are called mutants. Testers can use mutants to design strong test cases, likely to be fault revealing [1, 9] and to perform test assessment as it effectively quantifies the test suites' strengths [127].

Mutation testing research assumes a static nature of software, and thus it is focused on making the mutation score metric accurate with respect to all possible mutants that one can generate, by using a predefined set of mutation operators, in a given piece code. Thus, existing research is focusing on using specific mutant types [128]; on detecting equivalent mutants [40, 129], i.e., mutants that cannot be killed by any test case because they are semantically equivalent to the original program; or on eliminating redundant mutants [32, 42, 68], i.e., mutants that are killed "collaterally" whenever other mutants are killed [42] (subsumed by the subsuming mutants).

This strategy has the unfortunate effect of blindly using all possible mutants without considering their relevance to the task or to the most recent changes in question. To allow such focused testing, one should use only what we call commit-relevant mutants, i.e., mutants interacting with the changed program behaviours. These mutants are relevant to the program changes, meaning that they are killed by tests that exercise the committed code and its integration to the rest of the program under test. In terms of testing, these mutants form the change-relevant requirements and can be used to judge whether test suites are adequate in testing commits and, if not, to provide guidance in improving them (by creating tests that kill commit-relevant mutants).

In an attempt to form such commit-relevant mutants one could use the entire set of mutants or those that are located on the modified code, assuming that mutant locations reflect their utility and relevance. Unfortunately, such solutions are imprecise since they either include large volume of noise (irrelevant mutants), or are insufficient to cover all possible interactions between the unmodified and changed code. We argue that covering all interactions between unmodified and modified

code is particularly important because problematic regression issues arise from such unforeseen interactions [103, 130]. This is demonstrated by our results, which show that the majority of the altered program behaviours is captured by mutants located on unmodified code parts. In fact the majority of the altered program behaviours are captured by mutants located on unmodified code parts.

Overall, the contribution of the study regards the definition of the commit-relevant mutants and the related commit-relevant mutation-based test assessment. Intuitively, a mutant is commit relevant if it defines a test requirement (a mutant fault) that depends on the commit, i.e., the test cases that cover this requirement (detect this mutant fault) exercise the program behaviour altered by the commit. To ensure a testable link between the mutants and the commits, we require the existence of an "observable dependence" between the mutants and the committed code. This means that the presence and absence of the mutant and the commit code imply an observable behavioural change under some test execution.

We also show that by identifying those commit-relevant mutants one can accurately and adequately test program changes. Perhaps more importantly, we also demonstrate that mutation testing performed with the entire set of mutants or with the mutants located on the committed code is insufficient to assess how well subtle program changes have been tested. This implies that relevant mutants also enable the study of commit-aware fault detection assessment, in a sense using relevant mutants as a proxy for fault introducing commits. This aspect is missed by the software testing literature since it mainly focuses on using mutants as proxy of faulty program versions independently of the program changes under test. We showcase such a case by using commit-relevant mutants to evaluate regression test prioritisation techniques.

Taken together, the key research contributions of this present study can be summarised as follows:

- We define commit-relevant mutation testing, which is based on the notion of commit-relevant mutants, i.e., mutants capturing the interactions between modified and unmodified code.
- We show that commit-relevant mutants are a distinct class of mutants, i.e., it differs significantly from the other mutant classes (subsuming and hard-to-kill mutants) [29].
- We investigate the extent to which mutation-based test assessment metrics such as a) the mutation score (score that includes the entire set of mutants), b) the delta of mutation scores between pre- and post-commit, c) the mutation score of mutants located on the committed code, correlate with the commit-relevant mutation score. Our results show that all three metrics have relatively weak correlations (less than 0.4), indicating the need for a commit-relevant test assessment metric.
- We further examine the potential guidance given by commit-relevant mutation testing by comparing the gains and losses of strategies that use the entire set of mutants, the mutants located on the committed code and the commit-relevant mutants. Our findings suggest that commit-relevant mutants have 30% higher fault revelation ability (w.r.t. real commit-introduced faults) than the other strategies when analysing the same number of mutants.
- We illustrate a possible application of commit-aware mutation testing as a metric to evaluate test case prioritisation.

5.2 Commit-Relevant Mutants

Informally, a commit-aware test criterion should reflect the extent to which test suites have tested the altered program behaviours. This means that test suites should be capable of testing and making observable any interaction between the altered code and the rest of the program. We argue that mutants can capture such interactions by considering both the behavioural effects of the altered code on mutants' behaviour and visa versa. This means that mutants are relevant to a commit when their behaviour is changed by the regression changes. Indeed, changed behaviour indicates a coupling between mutants and regressions, suggesting relevance.

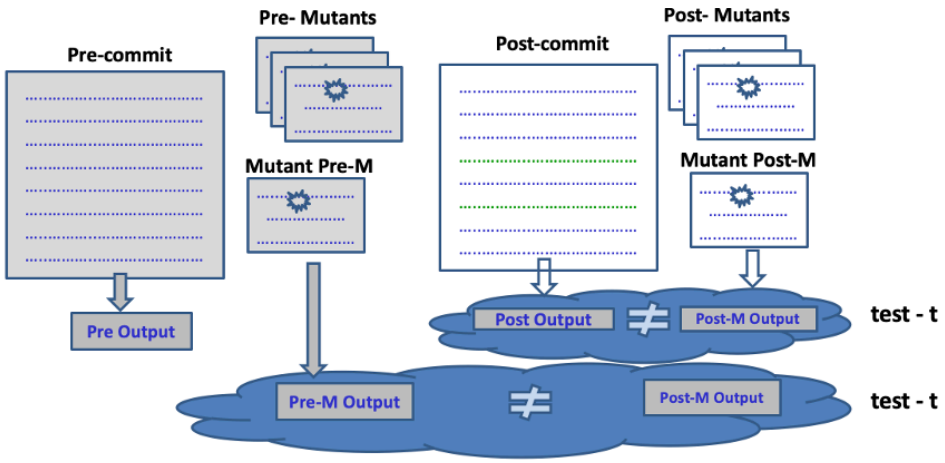


Figure 5.1: A mutant is relevant if it impacts the behaviour of the committed code and the committed code impacts the behaviour of the mutant. This means that there is at least one test case (test - t) that can distinguish both the behaviours of Pre-M from Post-M and Post from Post-M.

5.2.1 Rationale Behind Commit-Aware Mutation Testing

Since relevant mutants form commit-aware test requirements they should be killed by tests that exercise/test the committed code and its integration to the rest of the program. This means that relevant mutants should be killed by tests that are capable of detecting, i.e., making observable, any potential fault that depends on the commit.

To identify such mutants we check, for each mutant, whether there is at least one test case that can make observable any behavioural difference between the mutant and:

1. the program version that includes only the mutant (mutant in the pre-commit version).

2. the program version that includes only the committed changes (post-commit version).

These two conditions ensure the presence of "observable dependencies" between the mutant and the committed code since the removal of either of them impacts (changes) the behaviour of the program under the same test execution. Figure 5.1 illustrates the use of the above conditions. In particular, given the pre- and post-commit versions, and a mutant located on lines unmodified by the commit,

denoted as pre-M and post-M, we can identify relevant mutants by checking whether there is any test case (if there exists at least one) that can make observable the differences between pre-M and post-M and between post-commit and post-M.

More formally:

- let m be a mutant of the post-commit version of the program under analysis.
- let t be a test case from a set T of all possible test cases for this program.
- let $O_v(t)$ be an execution function of a test t on a program version v. Where v takes format of:
  - *post* - the post-commit version of the program.
  - m_{post} - m mutated post-commit version of the program.
  - m_{pre} - m mutated pre-commit version of the program.
- let denote A as a set of commit non-relevant mutants.
- let denote B as a set of commit-relevant mutants.

Definition 1. *Commit Non-Relevant mutant*

$$m \in A := \forall\,(t) \in \{T\} : Om_{post}(t) = O_{post}(t) \vee Om_{post}(t) = Om_{pre}(t) \tag{5.1}$$

Definition 2. *Commit Relevant mutant*

$$m \in B := \exists\,(t) \in \{T\} : Om_{post}(t) \neq O_{post}(t) \wedge Om_{post}(t) \neq Om_{pre}(t) \tag{5.2}$$

5.2.2 Demonstrating Example

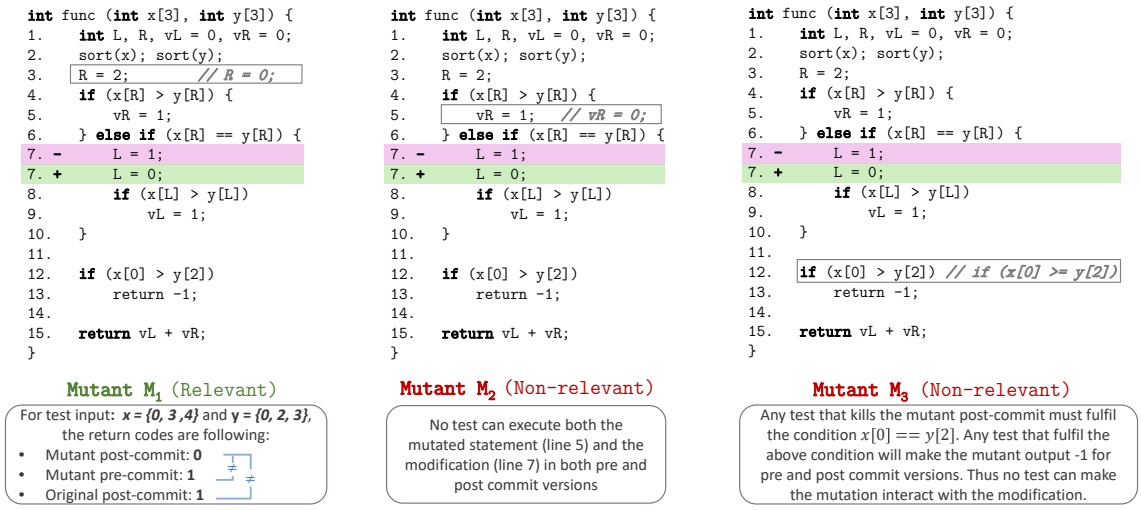


Figure 5.2: Example of relevant and non-relevant mutants. Mutant 1 is relevant to the committed changes. Mutants 2 and 3 are not relevant.

Figure 5.2 illustrates the concept of relevant mutants. The example function takes 2 arguments (integer arrays x and y of size 3), sorts them, makes some computations, and outputs an integer. The commit modification alters the statement at line *7* by changing the value assigned to the variable L from *1* to *0*, denoted with the pink-highlighted line (starting with '-') for the pre-commit version and green-highlighted line (starting with '+') for the post-commit version.

The sub-figure on the left side shows mutant M_1. M_1 is characterised by the mutation that changes the statement $R = 2$ into $R = 0$ in line 3 (the C language style comment represents the mutant's statement). We observe that, with an input t such that $t : x = \{0, 3, 4\}, y = \{0, 2, 3\}$, the original program post-commit has an output value of 1, the mutant M_1 pre-commit outputs 1 and the mutant M_1 post-commit outputs 0. Based on the definition of relevant mutants, M_1 is relevant to the commit modification.

The sub-figure in the center shows mutant M_2 (mutation changes the statement $vR = 1$ into $vR = 0$ in line 5). We observe that the mutated statement (in line 5) and the modification (in line 7) are located in two mutually unreachable nodes of the control-flow graph. Thus, no test can execute both the changed statement and M_2. M_2 is not relevant to the commit modification.

The sub-figure on the right side shows mutant M_3 (mutation changes the expression $x[0] > y[2]$ into $x[0] >= y[2]$ in line 12). We observe that some tests execute both the commit modification and the mutated statement. However, no test can kill M_3 in the post-commit version and at the same time differentiate between the outputs of the pre-commit and post-commit versions of mutant M_3. The reason is that any test that kills M_3 in the post-commit version must fulfil the condition $x[0] == y[2]$. Any such test makes both the pre- and post-commit versions of M_3 to output -1, thus, not fulfilling the condition to be relevant. Since, there exists no such test, M_3 is not relevant to the commit modification.

Note that in case a modification inserts statements, all killable mutants (in the post-commit version) located on these statements (new statements) are relevant to the modification. In case of deletion (modifications remove statements), the mutations located on these statement do not exist in the post-commit version, and thus, are not considered.

5.3 Experimental Setup

5.3.1 Research Questions

We start our analysis by recording the prevalence of commit-relevant mutants in code commits. Thus, we ask:

RQ1: (Mutant distributions) What ratio of mutants is relevant, is located on changed code, and is located on non-changed code?

Answering this question will help us understand the extent of "noise" included in the mutation score and will provide a theoretical upper bound on the application cost of commit-aware mutation testing.

As we shall show, the majority of the mutants are irrelevant to the committed code, indicating that using all mutants is sub-optimal in terms of the application cost. Perhaps more interestingly, using such an unbalanced set could result in a score metric with low precision. Therefore, we need to check the extent to which the mutation score is adversely influenced by irrelevant mutants. Thus, we investigate:

RQ2: (Metrics relation) Does the mutation score (MS), computed based on all mutants, on mutants located on the committed/modified code, and the delta of the pre- and post- commit MS correlate with the relevant mutation score (rMS)?

Knowing the level of these correlations can provide evidence in support (or not) of the commit-aware assessment (i.e., the extent to which the mutation score reflects the level at which the altered code has been tested). In particular, in case there is a strong correlation, we can infer that the influence of the irrelevant mutants is minor. Otherwise, the effects of the irrelevant mutants may be distorting.

While the correlations reflect the influence of the irrelevant mutants on the assessment metric, they do not say much about the extent to which irrelevant mutants can lead to tests that are relevant to the changed behaviours (in case mutants are used as test objectives). In other words, it is possible that by killing random mutants (the majority of which is irrelevant), one can also kill relevant mutants. Such a situation happens when considering the relation between mutants and faults, where mutant killing ratios have a weak correlation with fault detection rates but killing mutants significantly improves fault revelation [53]. Hence we ask:

RQ3: (Test selection) To what extent does the killing of random mutants result in killing commit-relevant mutants?

We answer this question by simulating a scenario where a tester analyses mutants and kills them. Thus, we are interested in the relative differences between the relevant mutation scores when testers aim at killing relevant and random mutants. We use the random mutant selection baseline as it achieves the current best results [68, 112]. We compare here on a best effort basis, i.e., the commit-relevant mutation score achieved by putting the same level of effort, measured by the number of mutants that require analysis. Such a simulation is typical in mutation testing literature [9, 68] and aims at quantifying the benefit of one mutant selection approach over another.

Answering the above question provides evidence that killing relevant mutants yields significant advantages over killing of random mutants. While this is important and demonstrates the potential of killing commit-relevant mutants in terms of relevance, still the question of actual test effectiveness (actual fault revelation) remains. This means that it remains unclear what the fault revelation potential of killing commit-relevant mutants is when the commit is fault-introducing. Therefore we seek to investigate:

RQ4: (Fault Revelation) How does killing commit-relevant mutants compare with killing of random mutants w.r.t. to (commit-introduced) fault revelation?

To answer this question we investigate the fault revelation potential of killing commit-relevant mutants based on a set of real fault-introducing commits. We follow the same procedure as in the previous research question (RQ3) in order to perform a best effort evaluation. While answering the question about mutant's fault revelation ability, and showing their usefulness and practicality in finding faults, we would like to know whether commit-aware mutants can be found through different classes of mutants. If the majority of the relevant mutants are also part of other mutant classes, it indicates that the other classes can be used as a proxy to relevant mutants. This is important since previous research [89, 131–133] heavily relied on other mutant classes to evaluate regression testing techniques. Moreover, by investigating the relationship with other mutant classes we can better understand the nature of relevant mutants and their fundamental differences (and similarities) with other classes. In particular, by investigating the relationship with Subsuming mutants, we can see how many

subsuming mutants are relevant, which represents the relevant behaviours captured by the mutants over all behaviours, i.e., ratio of relevant over all mutants after minimising the noise from redundant mutants. This allows us to have a better understanding of the discrepancies and potential wasted effort caused by irrelevant mutants. Similarly, the comparison with hard-to-kill mutants can show whether relevant mutants are not that difficult to kill and somehow distinct from the other classes. Thus, we are engaged in knowing:

RQ5: (Mutants Classes) How different the relevant mutants are to the classes of subsuming and hard-to-kill mutants?

We answer this question by investigating the relationship and overlap among the three sets of relevant mutants, hard-to-kill mutants, and subsuming mutants. As a reminder for a reader, hard-to-kill mutants are the set of mutants killed by a few tests.

Overall, answering the above questions will improve the understanding of the potential of the cost-effectiveness application of commit-aware mutation testing.

5.3.2 Analysis Procedure

We performed mutation testing on the selected subject using all the mutation operators supported by Mart [134] and Pitest [58] (the mutation testing tools we use). For the C programs, before conducting and running any experiment we discarded all the trivially equivalent mutants (including the duplicated ones), using the TCE method [39, 40] in order to reduce their impact on our results. This is an important step in order to avoid influence from trivially equivalent mutants that could anyway be reduced using the TCE method. Therefore, we applied our analysis on the resulting sets of mutants i.e., those that are not trivially equivalent.

Identifying relevant mutants requires excessive manual analysis, thus we approximate them based on test suites (this is a typical experimental procedure [8,44,68]). To do so we composed large test pools, which approximate the input domain. The pools are composed of the post-commit version developer tests (mined from the related repository). For C programs we augment the pools with automatically generated tests, similarly to the process followed by Kurtz *et al.* [68] and Papadakis *et al.* [8].

Using the test pools, we execute all the mutants (on both pre- and post-commit versions) and construct the mutation matrix that records the test execution output of each test on each mutant. For C programs, the output is the standard output produced when running the test, while for the Java programs it is the status (pass/fail) of the test run.

By using this information, i.e., test execution outputs on every related version, we approximate the relevant mutant set based on Algorithm 2. In the algorithm, the function calls *postCommitOrigOutput*, *postCommitMutOutput* and *preCommitMutOutput* compute the output of the execution of test case 'test' on the post-commit original program, post-commit version of mutant 'mut' and pre-commit version of mutant 'mut', respectively. In particular, in C the function calls *postCommitOrigOutput*, *postCommitMutOutput* and *preCommitMutOutput* return the exact concrete value of a test case output when executed on a mutant or the original program. While in Java they provide standard unit-level oracle pass/fail output. The generated output of the execution of test case 'test', on each version of a software (post-commit version,

mutant M generated on the pre- and post-commit version) is compared between the versions to identify a difference in behavior between mutant pre-M and post-M.

Besides the relevant mutant set, we also extract the modification mutant set, made of mutants that are located on a statements modified or added by the commits. This set is computed by extracting the modified or added statements from the commit *diff* and collecting the mutants that mutate those statements. Note that, by definition, the killable modification mutants are also relevant mutants, as their pre-commit output is not defined, and thus different from their post-commit output.

Therefore, we have a set of all the mutants generated on a post-commit version of a program (post-M), which can be divided into two subsets; those that are identified as commit-relevant by our approach (commit-relevant) and those that are identified as commit-irrelevant (non-relevant). The set of post-commit mutants located on statements modified or added by regression changes forms the (modification) subset of mutants. As already mentioned the modification set of mutants includes both relevant and irrelevant mutants. In RQ2, we want to know the correlations between the mutation scores of the aforementioned mutant sets. To do so, we select arbitrary test sets of various sizes and record the mutation scores on each mutant set and compute their correlations.

Algorithm 1: Approximate Relevant Mutants Set

```
Data: TestSuite, Mutants
Result: Relevant Mutants
RelevantMuts ← ∅;
for mut ∈ Mutants do
    for test ∈ TestSuite do
        origV2 ← postCommitOrigOutput(test);
        mutV2 ← postCommitMutOutput(test, mut);
        mutV1 ← preCommitMutOutput(test, mut);
        if origV2 ≠ mutV2 ∧ mutV2 ≠ mutV1 then
            RelevantMuts ← RelevantMuts ∪ {mut};
            break;
        end
    end
end
return RelevantMuts ;
```

In RQ2 we arbitrary pick sets of tests representing 10%, 20%, ..., 90% of the test pool. As these sets are randomly sampled we selected multiple sets (500 for C and 100 for Java) per size considered and per program commit (each subset of test can be seen as a testing scenario). For every test set, we computed the mutation score for each of the three mutant sets. We name as *MS*, *rMS* and *mMS* the mutation scores for the whole mutant set, relevant mutant set and modification mutant set, respectively. The mutation scores are computed on the post-commit versions and using the mutation matrix. Thus, for each commit and each test size, we have three statistical variables (*MS*, *rMS* and *mMS*), whose instances are the corresponding mutation scores for each test set.

Having collected the data for the statistical variables *MS*, *rMS* and *mMS*, we compute the correlations between *rMS* and *MS* as well as the correlation between

rMS and *mMS*. If the correlation between *rMS* and *MS* (*mMS*) is high, it means that *MS* (*mMS*) can be used as a proxy fo *rMS*. Otherwise, *MS* (*mMS*) is not a good proxy for *rMS* and thus, *rMS* should be targeted directly.

We also computed, for each test set, the mutation score in the pre-commit version. Then we compute the absolute change of mutation score (named *deltaMS*), on the analyzed mutant set, incurred by a commit modification ($delatMS = |MS_{post-commit} - MS_{pre-commit}|$), and we compute the correlation between *rMS* and *deltaMS*. A strong correlation would mean that the absolute change of mutation score between versions is a proxy for *rMS*. Weak correlation would mean that *rMS* cannot be represented by *delatMS*.

In RQ3, we simulate a scenario where a tester selects mutants and designs tests to kill them. This is a typical evaluation procedure [8, 68] where a test that kills a randomly selected mutant (from the studied mutant set) is selected from the test pool. This test is then used to determine the killed mutants, which are discarded from the studied mutant set. The process continues (by picking the next live mutant) until all mutants have been killed. If a mutant is not killed by any of the tests, we treat it as equivalent. This means that our effort measure is the number of mutants picked (either killable or not) and effectiveness measure is the relevant mutation score. Since we perform a best-effort evaluation we focus on the initial few mutants (up to 50) that the tester should analyse in order to test the commits under test. We repeat this process (killing all mutants) 100 times and compute the relevant mutation score.

For RQ4, we repeat the same procedure as in RQ3. However, instead of computing the relevant mutation score, we compute the fault revelation probability.

For RQ5, we calculate the overlap of three different categories of mutants: relevant mutants, hard-to-kill mutants, and subsuming mutants. We compute the set of subsuming mutants following the standard subsumption theory described in Section 2.2.3. As typically performed, we use the available tests to compute the subsumption relationships [8, 68]. Based on these relationships, we determine the subsuming mutants set. When it comes to the set of hard-to-kill mutants, we consider as hard-to-kill any mutant killed by less than 2.5% of covering tests, i.e., a mutant M is hard-to-kill if and only if less than 2.5% of the tests that cover M also kill M. For analysis of our results, we calculate relation between the corresponding sets of mutants, following percentage formula: sample/population $\times$ 100.

5.3.2.1 Statistical Analysis

We perform a correlation analysis to evaluate whether the mutation score, when considering all mutants, correlates with the relevant mutation score. To this end, we use two correlation metrics: *Kendall rank coefficient (τ)* (Tau-a) and *Pearson product-moment correlation coefficient (r)*. In all cases, we considered the 0.05 significance level.

The Kendall rank coefficient τ, measures the similarity in the ordering of the studied scores. We measure the mutation score MS and the relevant mutation score rMS when using test suites of size 10%, ..., 90% of the test pools. The Pearson product-moment correlation coefficient (r) measures the covariance between the MS and rMS values. These two coefficients take values from -1 to 1. A coefficient of 1, or -1, indicates a perfect correlation while a zero coefficient denotes the total absence of correlation.

Table 5.1: C Test Subjects

Benchmark	#Programs	#Commits	# Mutants	#Test cases
CoREBench [83]	6	13	154,396	8,828
Benchmark-1	13	34	338,390	11,866

To evaluate whether the achieved mutation scores MS and relevant mutation scores rMS are significantly different, we use a Mann-Whitney U Test performed at the 0.05 significance level. This statistical test yields a probability called p-value which represents the probability that the MSs and rMS are equal. Thus, a p-value lower than 0.05 indicates that the two metrics are statistically different. We use paired and two-tailed U test, to account for the different commits and programs.

5.3.2.2 Program Versions Used

To answer RQs 1-3 we used the C programs of GNU Coreutils[1], used in many existing studies [82, 108, 135]. GNU Coreutils is a collection of text, file, and shell utility programs widely used in Unix systems. The whole code-base of Coreutils is made of approximately 60,000 lines of C code[2]. In order to obtain a commit benchmark of Coreutils programs we used to following procedure to mine recent commits from the Coreutils github repository. (1) We set the commit date interval from year 2012 to 2019. This resulted in 5,000 commits considered. (2) Next, we filtered out the commits that do not alter source code files. This resulted in 1,869 commit remaining. (3) Then, we only kept the commits that affect only the main source file of a single program (This enable better control of test execution, because other programs of Coreutils are often used to setup the test execution of a tested program). (4) After that, we filtered out commits that are very large (commits whose modification has an edit actions of more than 5 according to GumTree [136]). This resulted in 218 commits. (5) Due to the large execution time of the experiments, approx. 2 weeks of CPU time per commit, we randomly sampled 34 commits among the remaining commits for the experiments. This constitutes our Benchmark-1.

In order to further strengthen our experiment and answer RQ4, we also use 13 commits from the CoREBench [83] that introduce faults. We selected these commits to validate the fault revelation ability of relevant mutants. Since we approximate relevant mutants, we needed commits where automated tests generation frameworks could run. Thus, we limit ourselves to the 18 fault introducing commits of Coreutils that we can run with Shadow symbolic execution [108]. Among these faults, two were discarded due to technical difficulties in compiling the code (the build system uses very old versions of the build tools). Three faults were discarded due to the excessively high required execution time to run the mutants (we stopped after 45 days). Table 5.1 summarizes the information about the C language benchmarks used in the experiments.

To answer RQs 1-3, we also consider a set of commits from well-known and well-tested Java programs. We extract these commits from projects in the Apache

[1]https://www.gnu.org/software/coreutils/

[2]Measured with cloc (http://cloc.sourceforge.net/)

Table 5.2: Java Test Subjects

Project	# Commits	# Mutants	# Test cases
commons-cli	9	61,419	3,247
commons-collections	5	323,584	55,076
commons-io	3	105,181	3,972
commons-net	6	345,130	1,478
joda-time	5	561,782	20,962
jsoup	8	330,125	4,985

Commons Proper repository[3], a set of reusable Java component projects, from Joda Time[4], a time and date library, and Jsoup[5], an HTML manipulation library. For each of the projects, we manually gathered the most recent commits meeting the following conditions from the project's history: (1) only source code is modified, no modification to configuration files, (2) the commit introduces a significant change, not a trivial one such as a typo fix, (3) test contracts are not modified, in order to meaningfully compare pre- and post-commit outputs and (4) both pre- and post-commit versions of the project build successfully. Overall, we gathered 36 commits, Table 5.2 summarises information about the commits used from each project.

5.3.2.3 Mutation Mapping Across Versions

As mutation testing tools generate mutants for a given program version instead of regression pairs, we need to identify the common mutants between the two versions. In other words, we need to map each mutant from its pre- to post-commit version of the program.

To establish such a mapping in the case of C programs, we unify the commit modifications into a single program, as done in the literature [108], and apply any standard (unmodified) mutation tool to generate the mutants. The code unification of the commit modification is done through annotation that has no side-effect. The annotations are made through a special function called *"change"* that takes 2 arguments/values (the arguments are the value of the pre-commit and post-commit versions, respectively) and return one of the two values.

The annotations are manually inserted in the program, according the semantics presented in previous studies [108].

Note that the statement insertion can be annotated by wrapping the inserted statement with $if(change(false, true))$; and a statement deletion can be annotated by wrapping the deleted statement with $if(change(true, false))$.

The choice of the version to use, for each mutant, is decided at runtime (by specifying the version to use through an environment variable recognizable by the *change* function).

For the Java programs, we perform the mapping of mutants from both sets of mutants of pre- and post- commit versions and the commit diff. First we start by generating the mutants for both pre- and post-commit versions of the program using

[3] `https://commons.apache.org/`

[4] `https://github.com/JodaOrg/joda-time/`

[5] `https://github.com/jhy/jsoup`

the mutation tool. We then map pre- and post- commit line numbers by parsing the commit diff, such as that we can identify which lines have been altered between the versions. Then, we use this mapping of altered line numbers to map pre- and post-commit mutants: using the line number, bytecode instruction number and mutation operator of the mutants to match both sets. We adopt this way for the Java programs in order to avoid making drastic changes on Pitest (the mutation testing tool we use).

5.3.2.4 Mutation Testing Tools and Operators

As test suites are needed in our experiment, we use the developer tests suites for all the projects that we studied. These were approximately 4,194 tests in total for C programs.

To strengthen the test suites used in our study, we augment them in two phases. First, we use KLEE [135], with a robust timeout of 2 hours, to perform a form of differential testing [137] called shadow symbolic execution [108], which generates 234 test cases. Shadow symbolic execution generates tests that exercise the behavioural differences between two different versions of a program, in our case the pre-commit and the post-commit program versions.

In order to also expose behavioural difference between the original program and the mutants, we used SEMu [82], with a robust timeout of 2 hours, to perform test generation to kill mutants in the post-commit program versions. SEMu generates 17,915 test cases.

These procedures resulted in large test suites of 22,343 test cases for C programs in total. Since we compare program versions, we use the programs output as an oracle. Thus, we consider as distinguished or killed, every mutant that results in different observable output than the original program.

We use Mart [134], a mutation testing tool that operates on LLVM bitcode, to generate mutants. Mart implements 18 operators (including those supported by modern mutation testing tools), composed of 816 transformation rules.

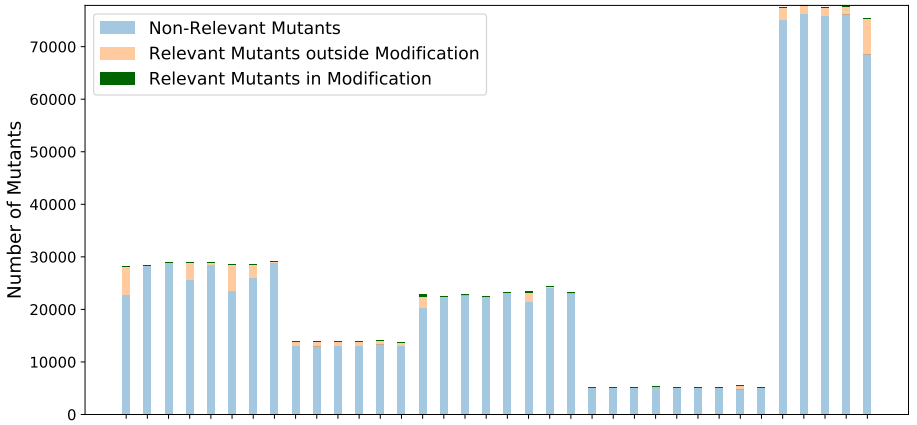
To reduce the influence of redundant and equivalent mutants, we enabled Trivial Compiler Equivalence (TCE) [39,40,138] in Mart to detect and remove TCE equivalent and duplicate mutants.TCE detected 13,322 and 460,072 equivalent and redundant mutants.

For the Java programs, we use the developer test suites available. We perform mutation analysis using Pitest [58], a state of the the art mutation testing tool that mutates JVM bytecode. We use all mutation operators available in Pitest, which are described in [139].

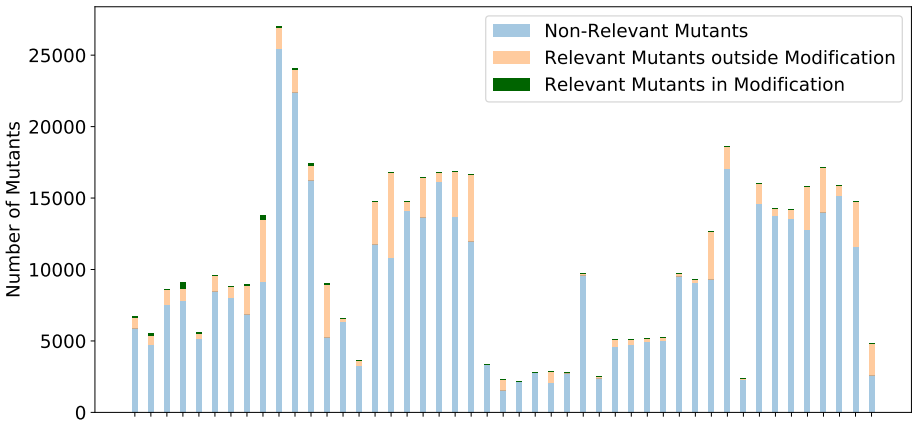
5.4 Experimental Evaluation

5.4.1 RQ1: Relevant mutant distribution

We start our analysis by examining the prevalence of commit-relevant mutants, i.e., mutants that affect the altered program behaviours. Figure 5.3 records the distribution of the relevant and non-relevant mutants among the studied commits. Based on these results we see that only a small portion of the mutant population produced by the selected mutation operators is actually relevant. This portion ranges from 0.5% to 47%, among which 3.6% is located on the changed program lines, while



(a) Java programs



(b) C programs

Figure 5.3: The distribution of killable, non-relevant, relevant outside the modification and relevant on the modification mutants among the studied commits.

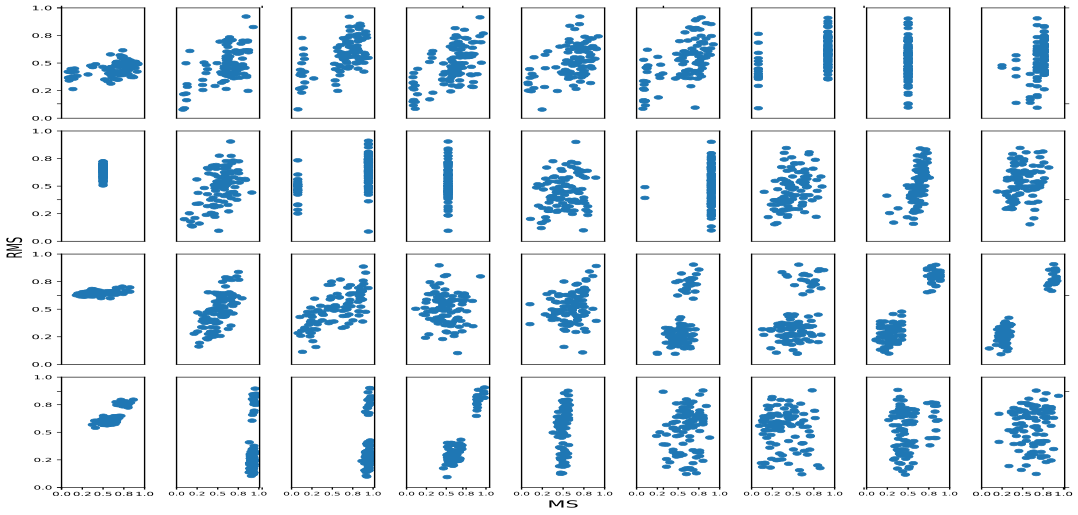
the rest is located on the rest of the code. For the large portion, it is possible to happen when the source code is not large, and the change is located in the crucial position.

Interestingly, the presence of so many "irrelevant" mutants, can have major consequences when performing mutation testing. Such consequences are a distorting effect on the accuracy of the mutation score, and a waste of resources when executing and trying to kill non-relevant to the commit mutants. We further investigate these two points in the following sections.

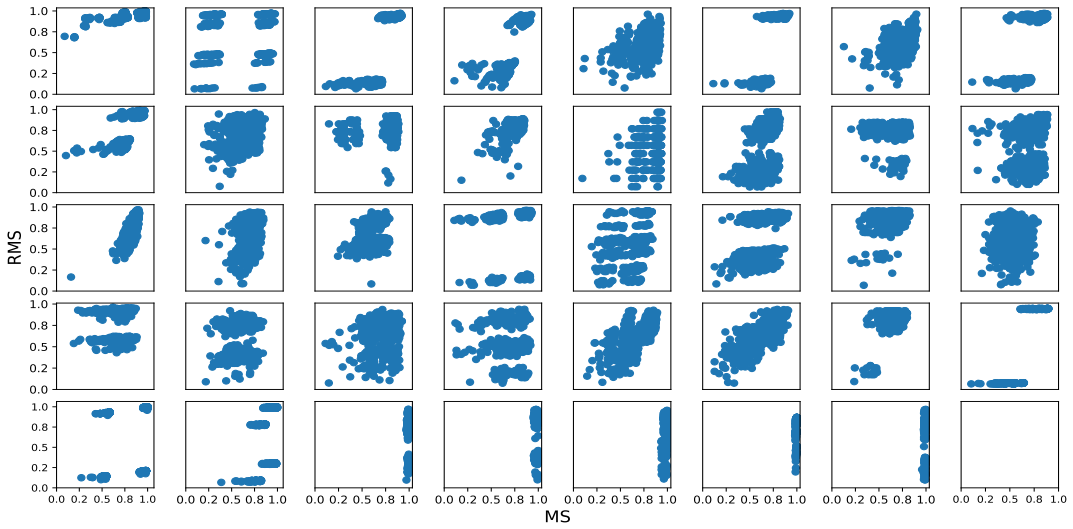
5.4.2 RQ2: Relevant mutants and mutation score

Figure 5.4 visualizes our data; each data point represents the mutation score and relevant mutation score of a selected test suite. As can be seen from the scatter plots, there is no visible pattern or trend among the data. We can also see that there is a large variation between mutation scores and relevant mutants scores in almost all the cases. These observations indicate that the examined variables differ significantly. In other words, one cannot predict/infer one variable using the other one. To further explore the relationship between mutation score and relevant mutation score within our data we perform statistical correlation analysis.

Finding a strong correlation would suggest that the two metrics have similar behaviours (an increase or decrease of one implies a relatively similar increase or decrease of the other). Figure 5.5 displays the results for the two correlation



(a) Java programs



(b) C programs

Figure 5.4: The relationship between Mutation Score and Relevant Mutation Score.

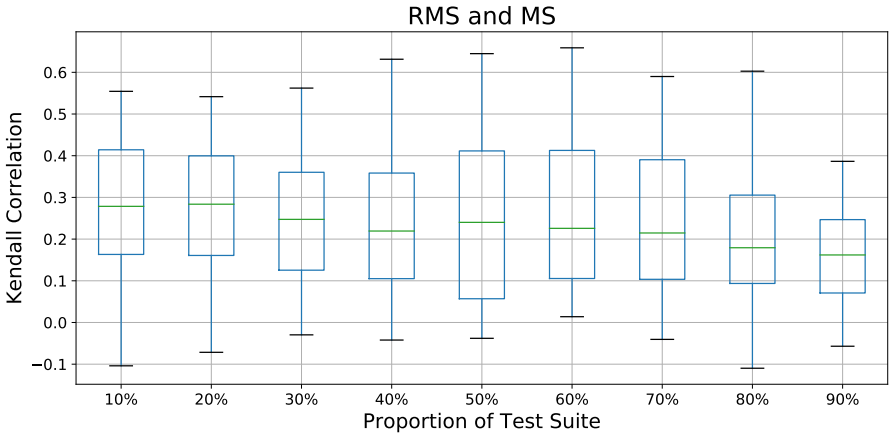
coefficients that have statistically significant values for randomly selected test suites (from our test suite pool) of different sizes[6]. Interestingly, we observe that most of the correlations are relatively weak with their majority ranging from 0.15 to 0.35. Additionally, we see that both coefficients we examine are aligned, indicating a weak relationship when either ordering test suites or considering their score differences.

One may assume that the relevant mutation score may be well approximated by the mutants that are located on the modified code, assuming that mutants' location reflects their utility and relevance. Similarly, one may assume that the commit-relevant score could be approximated by the delta of the pre- and post-commit mutation scores. We investigate these cases and find that most of the correlations are relatively weak with their majority ranging from -0.1 to 0.1.

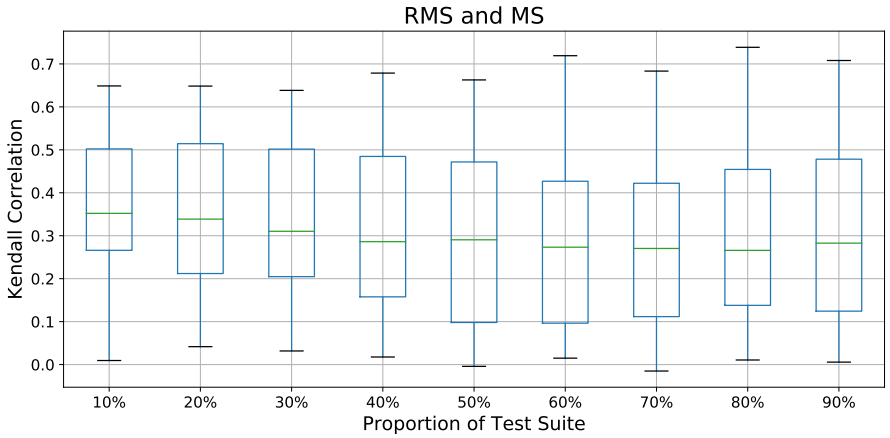
Overall, our results indicate that irrelevant mutants have a major influence on the mutation score calculation, and that using the overall mutation score does not reflect the actual value of interest, i.e., how well the altered behaviours are tested,

[6]We observe similar trends with Pearson correlation. For the sake of saving space, Pearson correlation results can be found on the accompanying website.

which is represented by relevant mutation score (rMS). Approximating the rMS using either the deltaMS or the mutants of the altered lines is also not sufficient. Hence, our results suggest that MS and other direct metrics are not good indicators of commit-related test effectiveness. We envision that future research should develop techniques capable of identifying relevant mutants at testing time, i.e., prior to any test generation and mutant analysis, in order to support testers.



(a) Java programs



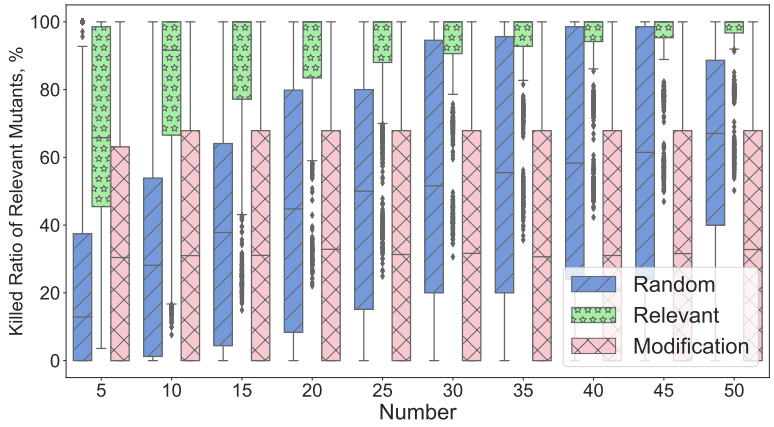
(b) C programs

Figure 5.5: Correlation between Mutation Score and Relevant Mutation Score for different test suite sizes on different languages.

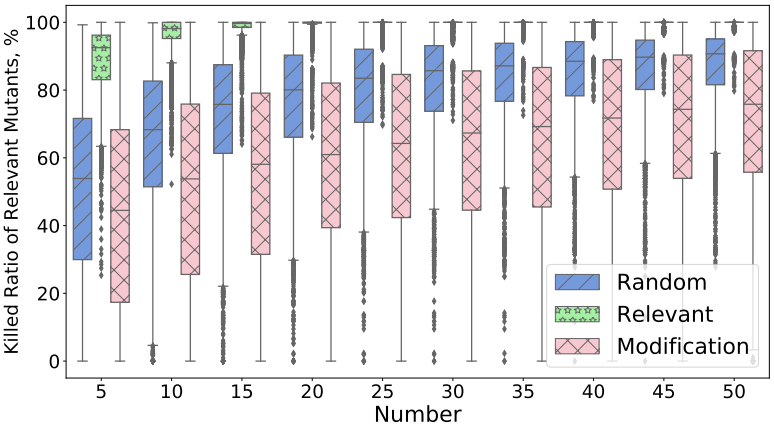
5.4.3 RQ3: Test Selection

Recent research has shown that mutation testing is particularly effective at improving test suites and revealing faults (guiding testers to design test cases that reveal faults), while at the same time mutation score is weakly correlated with fault detection [53]. In view of this, it is possible that despite the weak correlations we observe in our case, traditional mutation could successfully guide testers towards designing tests that collaterally kill relevant mutants.

Results are recorded in Figure 5.6 for the first 1-50 mutants to be analysed by the tester. We observe a large divergence (approximately 50%-60%) between the random, commit-based and relevant mutants. This suggests that by analyzing mutants in interval of 5, from selected 5 mutants to selected 50 mutants at random, one would miss approximately 60% and 50% of commit-relevant mutants for C and Java programming languages, respectively. This difference is statistically significant



(a) Java programs



(b) C programs

Figure 5.6: Test suite improvement of mutation-based testing with random (traditional mutation) and relevant mutants.

Table 5.3: $\hat{A}_{12}$. rMS when aiming at Relevant, Random and Modification related mutants.

#Mutants	**5**	**10**	**20**	**30**	**40**	**50**
Relevant-Random	0.90	0.95	0.98	0.98	0.98	0.97
Relevant-Modification	0.89	0.96	0.99	0.99	0.99	0.99

and with large effect size (Effect Size values are recorded on Table 5.3). Moreover, what we can observe that as we start increasing the number of analyzed mutants (5-50 mutants), the difference between the killed ratio of relevant mutants decreases. This is expected since putting more effort essentially results in selecting more mutants thereby increasing the chances to select some relevant. Taking together the weak correlations we found in the previous section with these results, we conclude that traditional mutation testing is sub-optimal and cannot be used to assess or guide (in a best-effort basis) the testing of committed code. Therefore, to support practitioners, future research should aim at identifying and using commit-relevant mutants. Similarly, controlled experiments should be based on relevant mutants when aiming at assessing change-aware test effectiveness.

Table 5.4: $\hat{A}_{12}$. Fault revelation when aiming at Relevant, Random and Modification related mutants.

% Relevant mutants analysed	10%	20%	50%	75%	100%
Relevant-Random	0.55	0.59	0.64	0.66	0.64
Relevant-Modification	0.57	0.59	0.69	0.73	0.70

5.4.4 RQ4: Fault Revelation

To demonstrate the importance of commit-aware mutation testing, we further compare the ability of the traditional mutants and commit-relevant mutants to reveal commit-introduced faults (real faults). We follow the same procedure as in the previous section but evaluate w.r.t. to the rate of faults revealed by the selected test suites.

The fault revelation results are depicted in Figure 5.7. From this data, we can see that a significant fault revelation difference (approximately 30-40%) between the compared approaches can be recorded. This difference is statistically significant with large effect size (Effect Size values are recorded on Table 5.4). Here it must be noted that these results can be achieved by an effort equivalent to analysing 0.4% of the mutants, which is 27 mutants per commit (on average).

Overall, our results demonstrate that by aiming at relevant mutants one can achieve significant fault revelation benefits (approximately 30%) over the traditional way of using mutation testing.

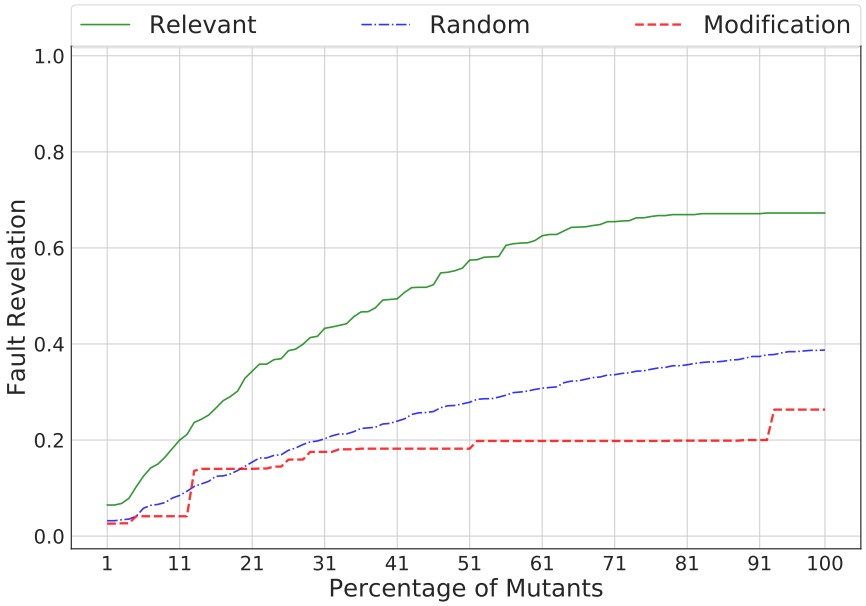
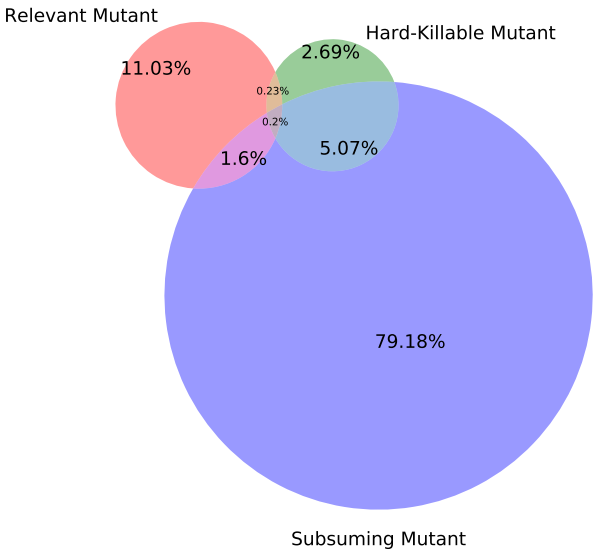


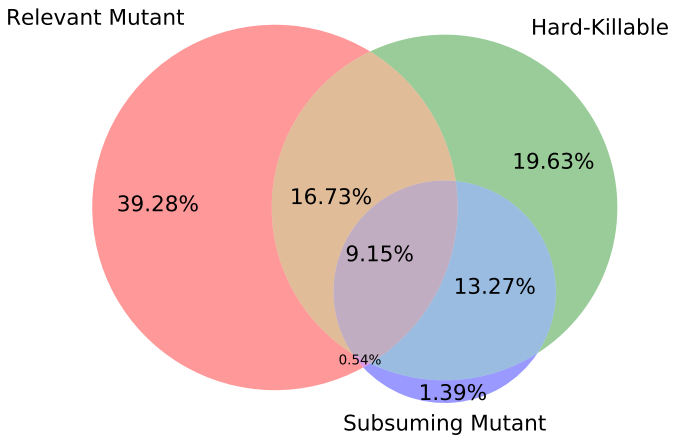
Figure 5.7: Fault revelation of mutation-based testing with random (traditional mutation) and relevant mutants.

5.4.5 RQ5: Mutant Classes

Figure 5.8 shows the overlap among the relevant, subsuming, and hard-to-kill mutant classes for the C and Java benchmarks. From these results we can conclude:



(a) Java programs



(b) C programs

Figure 5.8: Relevant Mutants intersection with Subsuming and Hard-to-Kill Mutants

Commit-Relevant vs. Subsuming

Our results show that most of the relevant mutants are also non-subsuming, more precisely 59.79% and 84.45% (11.03 / (11.03+0.2+0.23+1.6) * 100) for both C and Java benchmark. Suggesting that relevant mutants have many redundancies, similarly to other mutants. Now, if we measure overlapping just between those two categories, commit relevant and subsuming mutants, we see an overlap of 11.38% and 0.21% for both languages, respectively. This overlap is small implying an imbalanced case, i.e., by targeting subsuming mutants one wastes significant resources than if targeting commit relevant mutants.

An interesting finding here is that most of the commit-relevant mutants are also non-subsuming, meaning that relevant mutants have many redundancies, similarly to other mutant classes. This is important since it indicates that mutant selection may also benefit and be improved by emerging work on subsuming mutant selection [63, 129, 140].

Commit-Relevant vs. Hard-To-Kill

The results of the comparison with Hard-to-kill mutants shows that relevant mutants are not that difficult to kill and somehow distinct from the other categories. Among the union of mutants, 19.63% and 2.69% represent hard-to-kill mutants that do not fall under other classes, both for C and Java, respectively. More precisely, 33.39% and 32.84% of mutants from the hard-to-kill set do not overlap with other classes, while we can observe the overlap of 17.06% and 1.12% for both languages. The overlap is small because the committed changes are in relevantly easy-to-reach points of the programs.

In conclusion, relevant mutants is a distinct class of mutants that is hardly approximated by other mutant classes. This means that if one would like to use mutation testing to assess change-relevant fault detection (simulating faults introduced by commits), will need to rely on relevant mutants since any other form is inherently different.

5.5 Analysis of Commit-Relevant mutants

Our relevance definition include the set of mutants that can be impacted by the committed changes by at least one test case. Strictly speaking this definition allows the inclusion of mutants that may be killed by tests irrelevant to the committed code. Though, we consider them as interesting as these tests exercise code parts, the parts where these mutants are located, that depend on the changed introduced by the commits. Therefore, these tests indirectly exercise the committed code. In view of this, one can define different levels of relevance by considering the strength of the dependence between the mutants and the commits. We can thus define a strong relevance relationship by mandating an observable difference between pre-M and post-M by every test case that kills mutant M. In such a case we can define a weak relevance relationship, w.r.t., complete relevance relationship, as we do in this paper, by mandating an observable difference between pre-M and post-M by at least one test case that kills mutant M.

Formally:

- let m be a mutant of the post-commit version of the program under analysis.
- let t be a test case from a set T' of all test cases that kill mutant m.
- let $O_v(t)$ be an execution function of a test t on a program version v. Where v takes format of:
  - m_{post} - m mutated the post-commit version of the program.
  - m_{pre} - m mutated the pre-commit version of the program.
- let denote D as a set of strong commit relevant mutants.

Definition 3. *Strong commit relevant mutant*

$$m \in D := \forall\,(t) \in \{T'\} : Om_{post}(t) \neq Om_{pre}(t) \tag{5.3}$$

This definition allows selecting mutants that always lead to commit-relevant tests, i.e., tests that directly exercise the committed code. Informally, the mutant relevance leans on the strength of dependency between a mutant and commit changes, expressed through the number of tests that observe the dependency over the number of tests that kill the mutant. The value of relevance lies between 0 and 1. When relevance takes value 0 there is no observable behavioural difference on test outputs

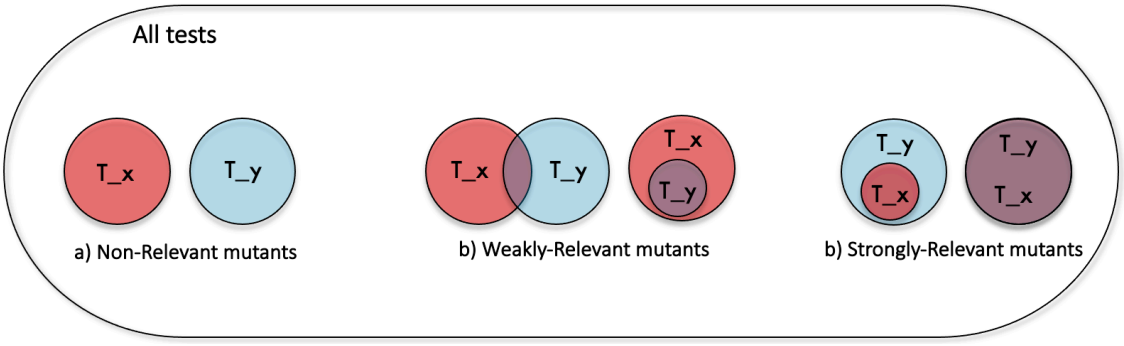


Figure 5.9: Illustration of different levels of relevance. The outer rounded rectangle represents all tests of the program under test. Set of tests T_x (red circle) includes all tests t that make observable the differences between post-commit and mutated post-commit version of a program under test (Definition 2). Set of tests T_y (blue circle) includes all tests t that make observable the differences between mutated post-commit version and mutated pre-commit version (Definition 2). We identify three cases, a) Non-Relevant mutants, i.e., no test t belongs to both T_x and T_y, b) weakly-relevant mutants case with least one test t that belongs to T_x and not to T_y, c) strongly-relevant mutants where every test t that satisfies T_x also satisfies T_y.

impacted by code-change. As the value of relevance increases, the mutant is assessed to be more relevant, until the value equals 1, in which case the behavioural difference can be observed with every test, making a mutant strongly relevant.

Formally, let's consider the same notations as for the definition of strong commit relevant mutants. Let t be a test case from a set T of all possible test cases for a program under analysis. Thus, formal definition of mutant relevance level can be defined as follows:

Definition 4. *Relevance*

$$relevance(m) = \frac{|\forall\,(t) \in T : t \in T' \wedge Om_{post}(t) \neq Om_{pre}(t)|}{|T'|} \tag{5.4}$$

Furthermore, in RQ5 (Mutants Classes) we witnessed that most of the relevant mutants are non-subsuming. This phenomenon suggests that many relevant mutants are redundant. This means that one could envision an optimised scenario where redundancies among relevant mutants are minimised. Thus, we can define subsuming commit relevant mutants, which is the set of relevant mutants that subsumes the set of all relevant mutants. Formally, let M be a set of mutants $\{m_0, ..., m_n\}$ for post-commit version of the program under test. Let RM be a set of commit-relevant mutants, whereas $RM \subset M$. Let $subsuming(M)$ be a function that returns the subsuming mutant subset of M [44, 45]. Thus, the set of subsuming commit-relevant mutants SRM can be defined as:

Definition 5. *Subsuming commit-relevant mutants*

$$SRM = subsuming(RM) : SRM \subset RM \subset M \tag{5.5}$$

5.6 ShowCase the use of Relevant Mutants in assessing Regression Test Prioritisation methods

5.6.1 Test Case Prioritisation

Testing is a key process in Software Engineering, but can also be a very expensive one. Test case prioritisation aims at ordering the different tests that are executed against the system in order to achieve some desired goal more efficiently. Tests are ordered to reveal a fault as early as possible in the execution of the test suite, providing faster feedback to testers and developers [31].

Much work has been done on test case prioritisation in the context of regression testing [31, 133, 141]. In this context, tests can be ordered based on their contribution to some test criterion on the previous version of the system, either totally or additionally [131]. Examples of test criteria used for regression test case prioritisation include code coverage [142], logic coverage [143], and also mutation score [14].

Mutation analysis has also been used as a metric to evaluate and compare different test prioritisation methods [133]. The different tests orderings are then compared based on how much each new test execution improves the mutation score, i.e., how fast the best possible mutation score is achieved by an ordering.

5.6.2 Demonstrating ShowCase

The primary purpose of regression testing is to validate whether the changes performed to the software break any of the unchanged program functionality. Therefore, test prioritization is used to support the regression testing of commits, giving rise to the question of how well they perform against commit-relevant faults. We, therefore, use commit-relevant mutant score as a metric to evaluate the ability of test prioritization techniques to detect change-relevant faults. Our motivation is to showcase the use of relevant mutants in a regression scenario where mutants serve as a proxy for the introduced faults (from the commits) and use them to assess test case prioritization approaches, i.e., assess how well test case prioritization techniques reveal faults introduced by the commits. This is important since previous research [89, 131–133] heavily relied on other mutant classes to evaluate regression testing techniques.

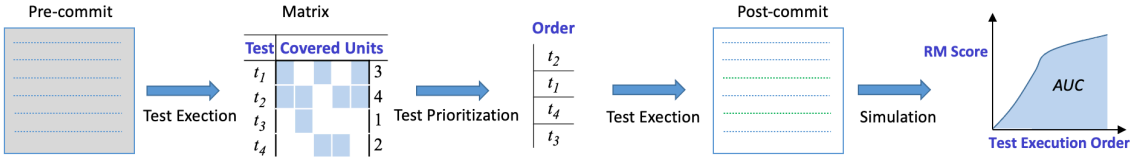


Figure 5.10: Test Prioritisation Pipeline

To demonstrate the use of commit-relevant mutants, we illustrate their application in evaluating regression test case prioritisation techniques. We thus, apply popular test case prioritisation techniques to the commits that we used in our experiments and evaluate their performance w.r.t commit-relevant mutation score. In particular, for every commit, we collect the statement-, branch-, and mutant-coverage information of the available tests (the same tests used in Section 5.3) on the pre-commit version of the programs. Each test is then run in isolation and produces a trace of units (statements, branches, mutants) covered. These traces direct the different test

prioritisation methods, which are listed in Table 5.5. This study considers the most popular priorization techniques, i.e., total incremental coverage test prioritisation methods [131], as described in the related literature [144].

On the post-commit version of the programs, we execute different test orderings generated from each pre-commit coverage. Each new test, executed following the ordering, may kill new commit-relevant mutants and thus increase the commit-relevant mutation score. This evolution of the relevant mutation score along the test ordering is recorded and represent a curve. The area under that curve divided by the total number of test cases represents the average Average Percentage of Fault Detected (APFD) [144]. Note that in our context, the (artificial) faults are the commit-relevant mutants. The APFD shows the average commit-relevant mutation score achieved across all possible numbers of tests, taken according to the orderings. The APFD shows how well the ordering prioritises tests killing commit-relevant mutants, i.e., tests that are relevant to the commit. The higher the APFD, the more commit-relevant tests are prioritised.

Figure 5.10 visualises the process of the use case we conduct.

Table 5.5: Test Prioritisation Criteria

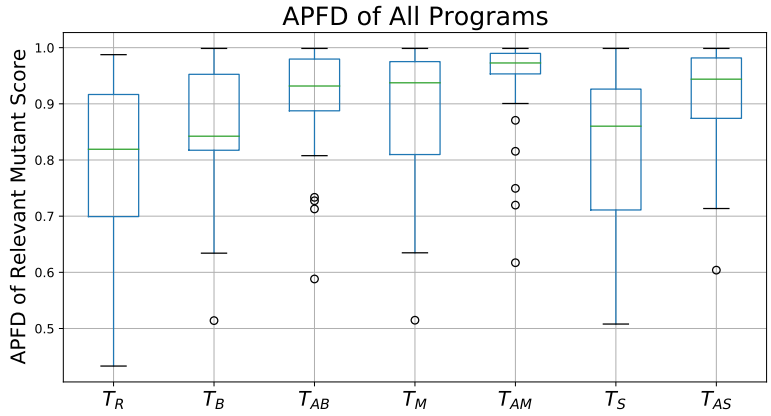
Acronym	Name	Prioritisation Objective
T_R	*Random*	Cover by randomised ordering
T_B	*Total Branch*	Cover the maximum number of branches
T_{AB}	*Additional Branch*	Cover the maximum number of uncovered branches
T_M	*Total Mutant*	Cover the maximum number of killed mutants
T_{AM}	*Additional Mutant*	Cover the maximum number of mutants not yet killed
T_S	*Total Statement*	Cover the maximum number of statements
T_{AS}	*Additional Statement*	Cover the maximum number of uncovered statements

Figure 5.11 shows the experimental results, aggregated across commits for both the C (Figure 5.11b) and Java (Figure 5.11a) benchmarks. For each test prioritisation technique (see Table 5.5), it records the APFD values achieved by the method.

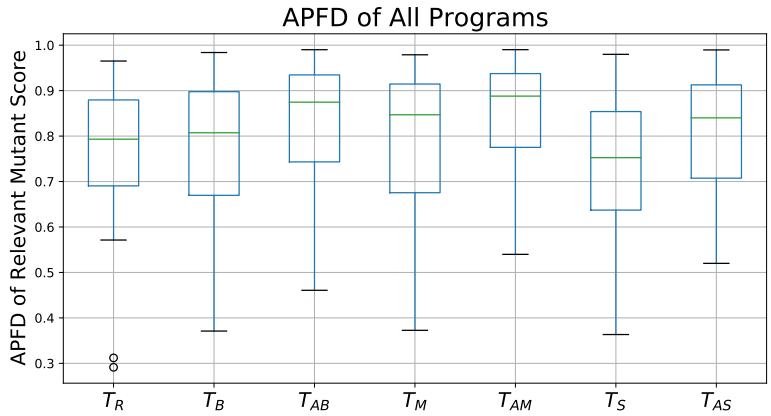
The results on C programs, shown in Figure 5.11b, do not indicate significant benefits from the total coverage methods (T_S, T_B, T_M) over random selection (T_R). Additional coverage methods (T_{AM},T_{AS}) result in higher APFD values, showing that the test orderings produced by these methods better detect commit-relevant mutants. However, the improvements are relatively small, indicating that further research, perhaps change-aware test prioritisation should be considered when testing such cases.

The results on Java programs, shown in Figure 5.11a, however, show clear benefits from all coverage-based test prioritisation techniques over the random test prioritisation, as well as more difference between the different criteria. Mutation-based prioritisation performs best in terms of the APFD values, while statement-based prioritisation performs the worst. Similar to the results shown for the C programs, additional coverage based methods perform better than total coverage based methods. Additional mutant coverage prioritisation T_{AM} performs best, achieving over 0.9 APFD for most commits.

Furthermore, what is also interesting to observe in the box plot is the high APFD value for τ_{AM}. It indicates that mutation-based prioritisation remains robust in the presence of the committed changes.



(a) Java programs



(b) C programs

Figure 5.11: Test Prioritisation.

Overall, we have shown that the approaches featuring the "total" strategy perform worst, in contrast to the additional strategy which offers more robust test prioritisation. This conclusion conforms with the one in [133], which shows that the best approaches reach the APFD median of approximately equal to 87%. In our case, our best additional approaches reach APFD median of approximately 95%, while the best approach T_{AM}, reaches the average value above 95% for 75% of commits and above 90% for 100% of commits. One key insight out of the above is that test prioritisation offers relatively small improvements, indicating that further research, should be directed towards change-aware test prioritisation.

5.7 Threats to validity

External validity: We selected commits that do not modify test contracts. Such commits are common in industrial CI pipelines [145] but rare in open source projects. To mitigate this threat, we performed our analysis on a relatively large set of commits given the computational limits posed by mutation analysis. In C, our

experiment required on average approximately 2 weeks of CPU time to complete, per commit studied (executions performed using Muteria [146]). In addition, we used an established research benchmark (CoREBench [83]) where we found similar results. Unfortunately, we consider fault introducing commits only in C as the Java datasets do not adhere to our non-changed test contract requirement.

Another threat may relate to the mutants we use. To reduce this concern we used a variety of operators covering the most frequently used language features including the operators adopted by the modern tools [139], in both C and Java.

Another threat may relate to the occurrence of flaky tests. We believe that we bypassed this threat by running 5 times all test cases of each project and its corresponding version. However, we consider more than one reason why flaky tests should not change conclusions related to our results. First, we worked with open-source software that does not contain solid environmental dependencies, one of the leading root causes of flakiness [147]. Second, all the programs we used as a benchmark for our study are well-studied projects with a reliable test suite with no previous reports on the occurrence of flaky tests. And as third, we consider that we study versatile and various projects for both C and Java programming languages. Thus, we have reduced any potential external validity related to the flaky tests.

Internal validity: Such threats lie in the use of automated tools, the way we treated live mutants and non-adequate test suites. To diminish these concerns, we used KLEE, a state of the art test generation tool and strong mature developer test suites. Nevertheless, the current state of practice [26] relies on non-adequate test suites, so our results should be relevant to at least a similar level of practice. To ensure our results, we carefully checked our implementation and performed a manual evaluation on a sample of our results. Moreover, we use established tools also employed by numerous studies.

To deal with randomness and minimize stochastic effects, we repeated our experiments 100 times and used standard statistical tests and correlations.

Construct validity: Our effort related measurement, number of analysed mutants, essentially captures the manual effort involved in test generation. Automated tools may reduce this effort and change our best-effort results. Still, we used the current standards, i.e., TCE [40] to remove all trivially equivalent mutants before conducting any experiment and KLEE (including a mutation-based test generation approach [82]). In test generation, we acknowledge that automated tools may generate test inputs that kill mutants, but we note that they fail to generate test oracles. Therefore, even if such tools are used, the test oracles will still require human intervention, i.e., introduce some effort. Here it should be noted that we consider the mutant execution cost as negligible since it is machine time and our focus is on the human time involved when performing mutant analysis. Moreover, existing advances [86] promises to reduce this cost to a practically negligible level.

Overall, we believe that our effort measurements approximate well (in relative terms) the human effort involved. All in all, we aimed at minimizing potential threats by using various metrics, well-known tools and benchmarks, real and artificial faults and following methodological guidelines [8]. Additionally, to enable reproducibility and replication we make our tools and data publicly available[7].

[7]The study presents a subset of our results. Our data and results are openly accessible on the following Github link: `https://github.com/relevantMutationTesting`

5.8 Conclusion

We proposed commit-aware mutation testing, a mutation-based assessment metric capable of measuring the extent to which the program behaviors affected by some committed changes have been tested. We showed that commit-aware mutation testing has a weak correlation with the traditional mutation score and other regression testing approximations (such as the delta on mutation score between the pre- and post-commit versions and mutants located on modified code), indicating that it is a distinct metric. Furthermore, we investigated and concluded that the relevant-mutants set is a distinct mutant set that cannot be found or expressed through proxies in different mutant classes. Our results also showed that traditional mutant selection is non-optimal for evolving and commit-oriented systems as it loses approximately 50%-60% of the commit-relevant mutants when analyzing 5-25 mutants. Moreover, we demonstrated that by focusing attention on commit relevant mutants, over randomly selected ones and the mutants occurring on a modification, one has 30% more chances of revealing commit-introducing faults. Additionally, to provide further evidence of the importance and diversity of commit-relevant mutants' applicability, we demonstrate a potential use case of the commit-relevant mutants and illustrate their application in evaluating regression test-case prioritization techniques. We show that commit-relevant mutants can be used to evaluate test case prioritization techniques.

In the next chapter, we plan to study relevant mutants and their occurrence through commit-history. The exploratory study of relevant mutants will shed more light on the properties of this particular category of mutants and their usability. Moreover, we want to study the properties of mutants in combination with commit-changes properties to identify potential correlations that can lead to more autonomous techniques and the development of machine learning models for automatic commit-relevant mutant selection.

6

Commit-Relevant Mutants via High-Order Mutations

The question arises whether we can determine commit-relevant mutants by releasing the conditions of the state-of-the-art approach, which captures behavioural differences of two consecutive chronological program versions. This chapter presents an approach for identifying commit-relevant mutants using the notion of observational slicing on a single program version. Naturally, the relevance of an instruction to a program point of interest, such as a program state or variable(s), can be determined by mutating instructions and observing their impact on the point of interest (changes in the target program state or variable). Since we aim to identify mutants relevant to changed instructions, we check the impact of mutants located in the changed code on mutants located in the unchanged code. In essence, with this approach, we measure the impact of second-order mutants on the first-order ones, which captures the existence of implicit interactions between the changed and unchanged code parts. In this chapter, we provide a detailed description of the proposed approach, with the up-to-date, most extensive empirical study of commit-relevant mutants in the context of the evolving system, counting more than ten million mutants over around three hundred commits and five projects. Our analysis studies the distribution and location of commit-relevant mutants. At the same time, our results show that by focusing on a special category of subsuming-commit-relevant mutants, a.k.a, the minimal set of mutants that represent all the others, we can reduce the number of mutants for selection by about 93%, on average through commits.

This chapter is based on the work published in the following journal article:

- Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. *"Mutation Testing in Evolving Systems: Studying the Relevance of Mutants to Code Evolution."* 2023, *ACM Transactions on Software Engineering and Methodology.* 32, 1, Article 14 (January 2023), 39 pages. `https://doi.org/10.1145/3530786`

Contents

6.1 Introduction

Software systems evolve and are typically developed through program evolution cycles that involve frequent code modifications [21]. Therefore, when software evolves, the program modifications need to be tested to avoid introducing faults and ensure the expected program behaviour. To test an evolving program, developers need to perform *regression testing*, i.e., assessing the impact of the change on the program by generating additional test cases targeting the change and its dependencies [31]. Typically, developers have to write or generate test cases to exercise the changes, stress their dependencies, and check that the program changes behave as intended [102].

Mutation testing is an established software testing technique [8]. It is typically applied to reveal faults in a program by modifying the program (aka injecting mutants) and generating tests to reveal the faults (i.e., kill the mutants) in the modified program. Mutation testing is an effective approach to improve the test suite's strengths by ensuring that it is adequate and diverse enough to kill all injected mutants. In the last decade, mutation testing has focused on selecting or reducing the number of executed mutants to ensure that mutation testing is feasible and scales in practice. To this end, researchers have proposed mutation testing with a specific type of mutants [128], mutant reduction by detecting equivalent mutants [40, 112] or by focusing on a particular category of mutants such as subsuming mutants[1] or hard-to-kill mutants [29, 42, 68].

Traditional mutation testing involves injecting mutants into the entire code base of the software. However, mutation testing of evolving programs is challenging due to the *scale of the required mutation analysis, the complexity of the program, and the difficulty of determining the impact of the dependencies of the program changes.* The sub-field of mutation testing addressing these issues by targeting the mutation testing program changes is referred to as *commit-aware mutation testing* [148].

A few commit-aware mutation testing approaches have been proposed to tackle the challenges of mutation testing of evolving software systems [26, 88, 113, 148]. These approaches suggest that mutation testing of evolving systems should focus on the program changes rather than the entire program. Recent studies have also indicated that commit-relevant mutants can be found on unchanged code due to unforeseen interactions between changed and unchanged code [113,148]. However, no scientific insights into the nature and properties of commit-relevant mutants and their utility over time have been provided. For instance, it is necessary to understand the distribution and program location of commit-relevant mutants over a representative span of a program history in order to confidently aim to identify, select, or predict commit-relevant mutants effectively.

In this study, we address this challenge by conducting an exploratory empirical study to investigate the properties of, at the time of writing, the most extensive dataset of commit-relevant mutants. Specifically, we examined the distribution, location and prevalence of commit-relevant mutants, as well as subsuming commit-relevant mutants[2]. To achieve this, we propose an experimental approach for identifying commit-relevant mutants using the notion of observational slicing [94], i.e., the relevance of an instruction to a program point of interest (such as a program state or

[1]Subsuming mutants [45] or disjoint mutants [42] is a set of mutants that has no mutant that is killed by a proper subset of tests that kill another mutant.

[2]Subsuming commit-relevant mutants is a set of commit-relevant mutants that has no commit-relevant mutant killed by a proper subset of tests that kill another commit-relevant mutant

variable(s)) can be determined by mutating instructions and observing their impact to the point of interest (changes on the target program state or variable). Since we aim to identify mutants relevant to changed instructions, we check the impact of mutants located on the changed code, as performed by observational slicing, on mutants located on unchanged code. In essence, with this approach, we measure the impact of second-order mutants on the first-order ones [149, 150], which captures the existence of implicit interactions between the changed and unchanged code parts (*for a detailed explanation of the notion of higher-order mutants, please refer to the chapter of background, Section 2.2.3).*

Overall, our formulation of the commit-aware mutation testing addresses the limitations and challenges of the state of the art [113, 148], in particular, making it more general and applicable for most evolving systems (*see Section 6.3.1*). Using this approach, we elicit the properties of commit-relevant mutants and study the advantage of commit-relevant mutant selection in comparison to random mutant selection or mutants located on program changes.

To the best of our knowledge, this is the most extensive empirical study of commit-relevant mutants. Specifically, our evaluation contains 10,071,875 mutants and 288 commits extracted from five (5) mature open-source software repositories. Our experiments took over 68,213 CPU days of computation. The main objective of this work is to provide scientific insights concerning the application of mutation analysis in testing evolving software systems. The main findings of this paper are summarized as follows:

- *Commit-relevant mutants are prevalent.* In our evaluation, *30%* of mutants are commit-relevant, on average. Hence, by reducing the number of mutants (by around 70%) and concentrating merely on those representing change-aware test requirements, considerable cost reductions can be achieved.
- *Selecting subsuming commit-relevant mutants significantly reduces the number of mutants.* Selection of *subsuming commit-relevant mutants* reduces even further the number of mutants, by about 93%, on average.
- *A large proportion of commit-relevant mutants are located outside of the program changes.* The majority of the commit-relevant mutants are located outside the changed methods (*69%*).

6.2 Commit-Aware Mutation Testing via HOMs

6.2.1 Definition

Intuitively, *commit-relevant* mutants are those that are linked with (capture) changed program behaviour, by the committed changes.

These mutants are those that a) are killable and are located on the changed lines because they capture behaviour relevant to the committed changes, and b) those that are killable, are located on unchanged lines and affect the change, by the commit, program behaviour, because they capture the interaction of the changed and unchanged code. This is approximated by a special form of observational slicing that uses higher-order mutants (*for a detailed explanation of the notion of higher-order mutants, please refer to the chapter of background, Section 2.2.3).* The idea is that mutants located on unmodified code, that impact the behaviour of mutants located on modified code, are commit-relevant because they depend/interact with the changed code. Consider two first-order mutants M_X and M_Y, such that M_X is located on

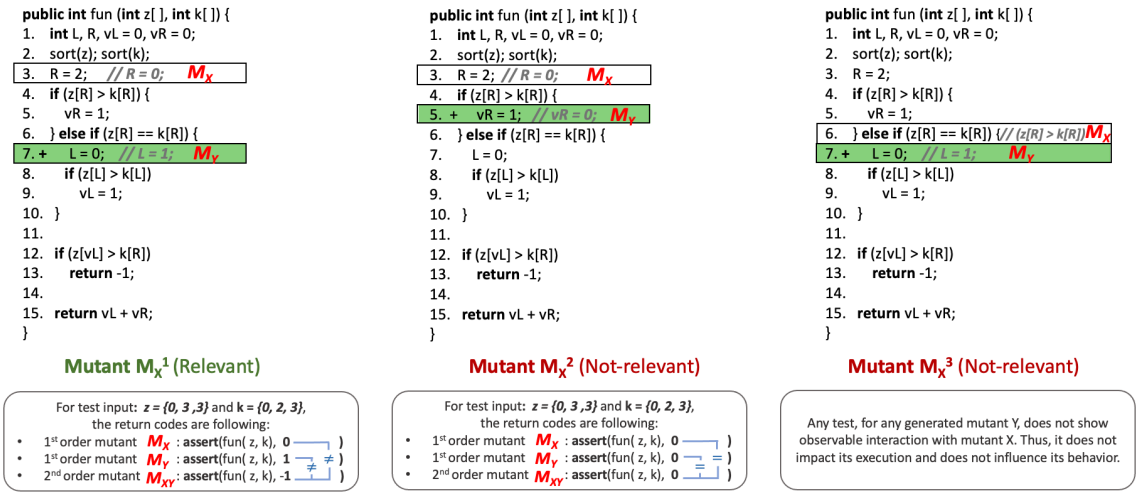


Figure 6.1: Example of relevant and not-relevant mutants. Left Sub-Figure: Mutant M_X^1 is relevant as mutant M_Y impacts its behaviour. Center Sub-Figure: Mutant M_X^2 is non-relevant as mutant M_Y does not impact its behaviour. Right Sub-Figure: mutant M_X^3 is not relevant since there is no behavioural difference for every possible M_Y.

the changed code and M_Y is located within the changed code. Then, the higher order mutant (M_{XY}) is the one created by combining M_X and M_Y. We say that M_X is *commit-relevant* if the higher-order mutant (M_{XY}) has a different program behaviour from the first-order mutants M_X and M_Y. That is, M_X is commit-relevant if (M_{XY} != M_Y) and (M_{XY} != M_X). Formally, the definition of *commit-relevant mutant* can be formed as:

Definition 6 (Commit-relevant mutants)**.** *A mutant M_X is relevant to a commit-change if a) it is killable and is located on the changed code, or b) there is a second order mutant M_{XY} (formed by the mutant pair of M_X, located outside the change, and M_Y, located on the change) that has different behaviour from the two first-order mutants M_X and M_Y that it is composed of.*

6.2.2 Motivating Examples

Simple Example

Figure 6.1 describes three simple scenarios illustrating commit-relevant mutants on a toy code example. In the code snippet on the left, we observe the example function *fun* that takes two arguments (integer arrays of size *3*). It starts by sorting the arrays' elements, then makes computations, and returns an integer as a result.

The green rectangle on line seven (*7*) represents the line that has been modified in the code. Using Java comments (symbols "*//*") on line three (*3*) we represent mutant outside the change M_X, and the mutant on the change M_Y on line seven (*7*). Mutant M_X changes the value of variable "*R*" to zero (*0*), while the mutant M_Y changes the value of variable "L" to one (*1*).

Consider that our test suite is confirmed just by one test that invokes function *fun* with the following arguments: $z = \{0, 3, 3\}$ and $k = \{0, 2, 3\}$. The output of the corresponding input value is observed from the inside of an atomic assertion as the input's actual value. We can see that after comparing obtained values after running each mutant in isolation, given the same test input, the mutant's behaviour is different.

Following our definition, this suggests that M_X is relevant to the modification since the actual execution value output (*fun(z, k)*) for mutant M_X is *0*, which is different from mutant M_Y whereas *fun(z, k) = 1*, and M_{XY} *fun(z, k) = -1*.

The code snippet in the middle of the figure presents the scenario in which a mutant is not relevant to the modification. Precisely, let us consider the same mutant M_Y on the change on line *5.+* as before, but now mutant M_X located outside the change is on line *3*. Mutant M_X modifies the assignment statement into $R = 0$. Given the same test input as before (i.e., $z = \{0, 3, 3\}$ and $k = \{0, 2, 3\}$), and following the mutants execution behaviour, we can observe that mutants show no observable interaction. Therefore, mutant M_X is not considered relevant for this particular change.

The code snippet on the right side shows an additional example of a non-relevant mutant. However, in this example, we observe two mutants that are unreachable from each other. These two mutants, for any test input, do not show observable differential interaction. Therefore, mutant M_X is considered to be non-relevant to test the corresponding change.

Real Example

Figure 6.2 presents an excerpt of a program from the Apache commons-io[3] project, version `81210eb`. The figure shows an evolution of the program in which the function `read` was modified in line 142 (from `org.apache.commons.io.input.BoundedReader.java`). The program change adds a constraint on the function's return value, suggesting that it should return a negative one (-1) in case the buffer does not contain any more values, and otherwise, it should return the index of the current iteration (i.e., `i`). Specifically, the previous version of the program always returned `i` in line 142, but the modified version either returns -1, when `i` is equal to 0, or `i` otherwise.

The function `read` takes three parameters, namely, an array of chars `cbuf`, and two integers `off` and `len`. Intuitively, the function `read` aims at modifying a certain number of characters (`len`) of array `cbuf`, starting from the given offset position `off`.

The function starts by reading a new character from a different buffer (see built-in `read()` invocation in line 140), then it proceeds to update `cbuf` array with the new character, and finally it returns the number of updated characters[4] Notice that the `read()` invocation (line 140) returns the fed character as an integer in the range between zero (0) to 65535 (0x00-0xffff), or it returns negative one (-1) if the end of the buffer has been reached.

As an example, consider a testing scenario that executes function `read` with the following inputs:

```
read([‘X’, ‘X’, ‘X’, ‘X’], 1, 2);
```

and the buffer accessed by the `read()` call in line 140 is as:

```
[‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘0’];
```

For this test case, both versions of the program, the previous one and the recently changed version, will return the same output (i.e., `len = 2`). Moreover, both versions of the program will produce the same modifications into array `cbuf` given as input,

[3]https://github.com/apache/commons-io

[4]For more information about the implementation of this method, please refer to official implementation documentation page: [151]

```
127 127     /**
128 128      * Reads into an array, @see java.io.Reader#read(char[], int, int)
129 129      *
130 130      * @param cbuf The buffer to fill
131 131      * @param off  The offset
132 132      * @param len  The number of chars to read
133 133      * @return the number of chars read
134 134      * @throws IOException If an I/O error occurs while calling the underlying reader's read method
135 135      */
136 136     @Override
137 137     public int read( char[] cbuf, int off, int len ) throws IOException {
138 138         int c;
139 139         for ( int i = 0; i < len; i++ ) { // M2: Inline Constant len -> 0  M3: Unary Insertion i -> ++i
140 140             c = read();    // M4: Remove statement
141 141             if ( c == -1 ) {
142                     return i;
    142                 return i == 0 ? -1 : i;  // M1: Negate Condition == -> !=
143 143             }
144 144             cbuf[off + i] = (char) c;
145 145         }
146 146         return len;
147 147     }
148 148 }
```

Figure 6.2: Method (`read()`) excerpted from the `BoundedReader.java` program in the from Apache commons-io project (version `81210eb`)

resulting in:

$$[\text{`X'}, \text{`0'}, \text{`1'}, \text{`X'}]$$

This is an example of a test case that does not exercise the program changes since the change (line 142) is never executed for this test case. Hence, the test does not show any behavioural difference between the previous version of the program and the current modified version of it.

Commit-aware Mutation Testing

Now, consider that during the mutation testing analysis, four mutants ($M1$ to $M4$) are injected into the function `read`, as it is shown in Figure 6.2 via Java comments ("`//`"). Particularly, *Mutant* $M1$ is located on the modified statement in line 142, i.e., it is a mutant within the program change, and it replaces the condition `==` (equal) with `!=` (not equal). *Mutant* $M2$ is located on line 139 (outside the change) and replaces the variable `len` with a constant value zero (0), mutating the condition of the `for` loop `i < len` to `i < 0`. *Mutant* $M3$ is also injected on the same statement (line 139) but it uses a unary insertion (`++i`) to update variable i within the condition check of the `for` loop, such that condition (`i < len`) is mutated to (`++i < len`). Finally, *Mutant* $M4$ removes the statement located on line 140 (i.e., `c = read()`).

Then, by using our HOM-based approach, we can create higher-order mutants by pairing all four mutants. Precisely, we pair the mutants located outside the change with the mutants on the commit-change (line 142), and we obtain three higher-order mutants $M12$, $M13$, and $M14$. Table 6.1 illustrates the behaviour (outputs) of the function `read` under its previous version, its current changed version, and all the mutants.

Consider now a different testing scenario in which the input buffer accessed by the built-in function `read` in line 140 is empty (i.e., `[]`). This testing scenario shows the behavioural difference between the previous version and the modified version

Table 6.1: Test output observation for Figure 6.2 showing the program behaviour (outputs) of the original program, the changed program, and the first and second-order mutants of the program. The test observations are performed using input `read(['X','X','X','X'], 1, 2)` and an empty buffer (`[]`) fed to the built-in function `read` (in line 140)

	Program Versions	Program Changes	Code line	Test output	Commit-Relevance
Pre-commit	Old version `51f13c84`	`i`	*142*	0	N/A
Post-commit	New version `81210eb`	`i == 0 ? -1 : i`	*142*	-1	N/A
First-order mutant	M1	`==` ⇒ `!=`	*142*	0	*Relevant*
First-order mutant	M2	`len` ⇒ `0`	*139*	2	*Non-Relevant*
First-order mutant	M3	`i` ⇒ `++i`	*139*	1	*Relevant*
First-order mutant	M4	`delete statement`	*140*	2	*Non-Relevant*
Second-order mutant	M12	`==` ⇒ `!=` ∧ `len` ⇒ `0`	*142* ∧ *139*	2	N/A
Second-order mutant	M13	`==` ⇒ `!=` ∧ `i` ⇒ `++i`	*142* ∧ *139*	-1	N/A
Second-order mutant	M14	`==` ⇒ `!=` ∧ `delete statement`	*142* ∧ *140*	2	N/A

of the program since it executes the change (in line 142). We observe that, while the execution of the previous version of the program returns zero (0), the modified version returns negative one (-1).

Table 6.1 highlights the behaviour (i.e., output) of each mutant. First, according to the traditional definition of commit-relevant mutants, $M1$ is a commit-relevant mutant, since it is located on the program change [88]. Additionally, according to our extension of the definition of commit-aware mutation (*see* subsection 6.2.1), we compare the output of the second-order mutants and their isolated first-order mutants. We observe that the second-order mutant $M13$ is also a commit-relevant mutant. This is because the second-order mutant ($M3$) has a different behaviour from the isolated first-order mutants (i.e., $M13 \mathrel{!}= M3$, and $M13 \mathrel{!}= M1$). Meanwhile, the other second-order mutants, i.e., $M12$ and $M14$ are not commit-relevant because they have similar behaviours as the isolated first-order mutants (i.e., $M12 == M2$, and $M14 == M4$).

Commit-aware Criteria

Let us illustrate the importance of the strict constraint employed in our approach to compare the behaviours of first-order and second-order mutants (e.g., the need to ensure $M13 \mathrel{!}= M3$ *AND* $M13 \mathrel{!}= M1$) using a counter-example. Specifically, we will discuss the rationale for this constraint and why considering a less strict constraint does not suffice (and mutants like, for instance, $M2$ are not commit-relevant, despite the fact that $M12 \mathrel{!}= M1$ but $M12 == M2$). Let us consider the example program in Figure 6.2 and the output behaviour observed in Table 6.1. Even though $M1$ and $M12$ have different outputs, inspecting the behaviour of $M12$ on the program change using the provided test cases, we observe that the behaviour of the second-order mutant $M12$ is not different from that of $M2$. In fact, $M2$ does not execute the program change nor the mutant within the change. Indeed there is no input that can force mutant $M2$ to execute the changed line (in line 142) since the `for` loop condition will always evaluate to false. This implies that using a less strict constraint (e.g., an "OR" operation) in our check, will lead to such miss-identification of commit-relevant mutants, implying that mutants that can never lead to the execution of the program change (e.,g., $M2$) can be miss-classified as relevant to the program change. Thus, it is important to ensure that the behaviour of the second-order mutant and that of

the isolated first-order mutants are indeed different.

Subsumption Relation

Let us illustrate the subsumption relation of commit-relevant mutants. Consider the two commit-relevant mutants in our example, i.e., *mutant* $M3$ that we have identified as commit-relevant, and *mutant* $M1$ located on the changed statement (commit-relevant by default). In our example (Figure 6.2), both mutants are killed by the initial test input (in Table 6.1). Let us consider that the test suite has one additional test, in particular, the following test:

```
read(['X','X','X','X'], 1, 1)
```

Given this new test input, we observe that *mutant* $M1$ is killed by this test input (output zero (0)), but *mutant* $M3$ is not killed by the test (output one (1)). Following the definition of mutant subsumption [44], we can observe that a test that distinguishes *mutant* $M3$, will also distinguish *mutant* $M1$, but *mutant* $M1$ can be distinguished by a test that cannot distinguish *mutant* $M3$. In this situation, we say that $M3$ *subsumes* $M1$, making $M3$ a *subsuming commit-relevant mutant*. This example illustrates a scenario where a subsuming commit-relevant mutant is located outside the program change. The mutant residing outside the change subsumes a mutant residing on the program change, which makes the test requirement of the mutant on the change redundant. We can satisfy both mutants ($M1$ and $M3$) by writing test requirements to identify the subsuming commit-relevant mutant located outside of the committed change (i.e., $M3$), which is the commit-relevant mutant also identified by our approach.

6.2.3 Design Requirements

Our commit-relevant mutation approach aims to fulfil certain requirements to ensure we gather and study a vast number of commits and commit-relevant mutants. These design requirements address some of the limitations and challenges of state-of-the-art [26, 113, 152]. In particular, we address the following:

- ***Location of Commit-relevant Mutants:*** In this work, we focus on identifying commit-relevant mutants within and outside the program changes, i.e., within the commit-change as done in prior work [26], and the unmodified program code. In particular, we are interested in revealing behavioural interactions induced by the program changes on the rest of the unmodified program code. We achieve this by identifying the commit-relevant mutants outside of the program changes. In particular, we employ second-order mutants; we analyze the impact of second-order mutants on the behaviour of the evolving program (*see Section 6.2.4*).
- ***Test Contract:*** Our experimental approach employs only the test suite from the post-commit program version. In previous work [152], the experimental design requires the execution of test suites across the pre-commit and post-commit versions of the program. Plus, it implies that the number of tests does not increase or decrease across versions, i.e., the test contract is intact. Therefore, the approach observes the delta between versions by comparing mutants test suites from pre- and post-change commits. This assumption can be challenging to address since the test suite also evolves as the program evolves, e.g., when implementing new features or fixing bugs. In this work,

we observed that this assumption is not significantly common in practice. In particular, in our study, the proportion of commits where the test contract is preserved is less than 40%.

- ***Commit Patches and Hunks***: In this approach for commit-aware mutation testing, we require the commit patches and commit hunks for empirical evaluation and analysis. Indeed, commit properties are vital for commit-aware mutation testing and are commonly used by state-of-the-art techniques [113,152]. In this work, we employ commit properties (i.e., patches and commit hunks) for both commit-relevant mutant detection and experimental analysis. Particularly, this approach requires the commit patches (i.e., the delta between pre-commit and post-commit versions) to identify the interaction of the mutants within the change and the mutants outside the change. We also employ commit hunks in our experimental analysis, to identify the number of individual code-chunks structures present in a commit. Involving commit hunks in our analysis sheds light on the relationship between altered statements in the commit hunk and the mutants residing outside the commit hunk.
- ***Post-Commit Version:*** Proposed experimental approach requires *the post-commit version of the program to be compilable, executable, and testable.* These requirements are vital for the dynamic analysis of our approach and they only apply to the *post-commit version* of the program. In contrast, our previous work [152] requires two program versions (pre-commit and post-commit) and *assumes a green test suite, no build failures and no compilation errors for both program versions.* In our evaluation setup, we find that these conditions are irregular (less than 40% of the cases). To address this concern, we ensure our approach requires the post-commit version of the program, without the need for the pre-commit version. This allows collecting significantly more commits for our study and allows us to evaluate a vast amount of commit-relevant mutants.
- ***Test Oracle:*** In this work, we employ test assertions of system units as our test oracle. This is a fine-grained test oracle used by unit tests. Here it is important to note that when we refer to assertions these are *test assertions*, and *not program assertions* and are used for checking the observable behaviour of the program units, as mandated by strong mutation. In practice, we use test assertions to define the mutant behaviour and study the impact of mutants and changes on program behaviour.
- ***Number of Commits:*** Our empirical study characterized commit-relevant mutants and required a substantial number of commits and commit-relevant mutants. Dues to the flexibility of our experimental approach, in this study, we analyzed significantly (10x) more commits and mutants than in our previous work [152]. We addressed the significant limitations and assumptions of prior work, in which strict design constraints could contain gathering a sufficient number of commits and commit-relevant mutants. For instance, as stated in the paper Ma et al. [148], it is challenging to find commits in open-source projects that do not break the test contract, i.e., keep the test suite intact. This challenge further inhibits our goal of automatically studying the characteristics of relevant mutants. Addressing the concerns above allows us to gather and study more commits than previous studies.

Our experimental approach aims to target the requirements above to ensure that we gather many commits and cover several realistic corner cases for evolving

software systems. Overall, fulfilling these requirements and addressing these concerns enables us to collect significantly more commits and identify significantly more commit-relevant mutants for our study. In particular, our study involved 10x more commits and 6x more commit-relevant mutants than previous studies.

6.2.4 Approach Overview

Our study aims at investigating the existence and distribution of commit-relevant mutants in evolving software systems. Specifically, we study the relationship between the lines of code changed in a commit hunk and the mutants residing in program locations outside the commit hunk under consideration. Intuitively, we want to study the interaction between two program locations, where one location is part of the commit hunk, and the other is outside the change. We plan to employ high-order mutants (second-order, to be more precise) and simulate potential changes in a commit hunk and the mutants outside the commit hunk. This study aims at providing scientific evidence of the relationship and relevance of mutants (test requirements) outside commit hunks that need to be taken into account when testing evolving systems.

To determine if a mutant is relevant for a commit hunk, we plan to observe whether the commit changes affect mutants' behaviour. Intuitively, suppose a change in a location in the commit hunk (produced by a mutation) affects the outcome of the mutant outside the commit. In that case, we have evidence that there exists an interaction between these two locations, indicating that the mutant is *relevant* for the commit. The absence of interactions indicates either the existence of equivalent mutants [149, 150] or the absence of dependence/relevance. To account for the case of equivalent mutants and ensure the relevance of observations, we sample multiple mutants per statement.

Mutants' behaviour is (partially) determined by observing their covering test set. Implying that if we want to observe the interaction between mutants in different locations, the test set should make any difference in the mutant's behaviour whether they are run in isolation or combined. More precisely, if we can observe that the behaviour of two mutants M_X and M_Y run in isolation differs from the behaviour of the second-order mutant M_{XY} (obtained by combining both mutations M_X and M_Y), then we can conclude that mutants M_X and M_Y influence each other. Consider a situation where the test set $\{t_0, t_1\}$ is able to observe that M_{XY}'s behaviour differs from M_X and M_Y's behaviour. For instance, test t_0 passes on mutants M_X and M_Y but fails on mutant M_{XY}. Thus, we can conclude that locations in which mutations M_X and M_Y were applied to interact with each other.

Following a similar idea, consider generating one of the mutants outside the change (M_X) and the other one on the change (M_Y), and their combination makes a second-order mutant (M_{XY}) suitable for observing if there exists an interaction between them. To determine if mutant M_X is relevant for the commit change, we can iterate this process by exploring different high-order mutants M_{XY} by varying mutant on the change M_Y, with the aim of finding one combination that evidences their interaction.

To compare mutants' behaviours, first, we need an intersection set of tests covering mutants M_X, M_Y, and M_{XY}. Second, we proceed to run these tests to observe a difference between the mutants. Instead of considering only passing and failing output as a standard unit-level testing oracle, we instrument tests and contained

assertions to obtain and compare actual assertion values. For instance, an assertion like `assertEquals(0, Z)` can be violated by a (potentially) infinite number of values for `Z`, all of them violating the assertion. Suppose after executing mutants M_X, M_Y and M_{XY}, the value of `Z` is different. In that case, we can observe a difference in their execution, allowing us to determine if there exists an interaction between these mutants, concluding that mutant M_X is relevant for the commit change. Section 6.3.6.3 describes the implementation details on how we instrument test executions to obtain actual assertion values.

Figure 6.3 illustrates our approach to detecting interactions between mutants by comparing their behavioural assertion values. It depicts that after executing each first-order mutant M_X and M_Y in isolation (assertions that cover them), we compare the output values with the value obtained after running second-order mutant M_{XY}. If running M_X and M_Y in isolation differs from running M_{XY}, we determine that mutant M_X is relevant for the commit change.

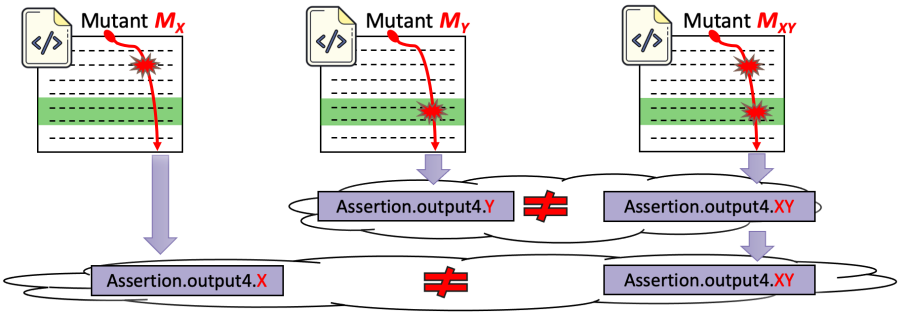


Figure 6.3: A mutant M_X is relevant to a commit-change, if any higher order mutant M_{XY}, shows different behaviour from M_X and M_Y executed in isolation

6.2.5 Algorithm

To perform an empirical study toward distinguishing relevant mutants, we generate the first-order mutants located around and on the commit change (i.e., M_X and M_Y, respectively). The second-order mutants (i.e., M_{XY}) are a combination of the previous two. The *mutant-assertion matrices* were obtained by executing the mutants against developer-written and automatically generated test pools. Note that test run status is pass/fall for Java programs; therefore, to observe behavioural differences produced by mutants, we need to focus on test assertions and record assertion execution actual value output of each test on every mutant. Precisely, for every mutant and every test assertion, a mutant-assertion matrix stores the assertion values obtained after running a mutant against a test. As noted, this study performs mutation analysis on commits from Java programs, using PiTest[5] as the Java mutation testing tool and EvoSuite[6] as the state-of-the-art test case generation tool. Section 6.3.3 provides further details regarding mutants test case generation and test assertions instrumentation.

After computing mutant-assertion matrices, we proceed to approximate which mutants are relevant to the change, according to our Definition 6 following the steps incorporated in Algorithm 2. The algorithm summarises the previously described

[5]http://pitest.org/
[6]https://www.evosuite.org/

process, where functions *MutantsonChangeMutantOutput*, *aroundChangeMutantOutput*, *highOrderMutantOutput* return the output of a specific test *assertion* execution per specific *mutant*. Finally, Algorithm 2 returns a set of relevant mutants for a particular commit change.

This algorithm has a worst-case polynomial time complexity of $O(n^4)$ due to the four nested `for` loops ($O(n * n * n * n)$). For each of the three inputs fed to the algorithm (*TestSuite*, *MutantsOnChange* and *MutantsAroundChange*), there is a linear-time complexity ($O(n)$). Additionally, there is a linear-time complexity ($O(n)$) for evaluating each test assertion corresponding to the test cases. Overall, the performance of the algorithm depends on the number of mutants in the change, the number of mutants injected in the modified code, the size of the test suite and the number of assertions in each test. Specifically, to derive higher-order mutants, we consider every pair of mutants within and outside the change; we also execute all test cases corresponding to these mutants and evaluate all test assertions in each test case. This algorithm can be optimized by improving the number of evaluated tests, assertions or pairs of mutants.

The complexity of this algorithm can be reduced to $O(log(n) * n^3)$ via a binary search on the pair of mutants (outside the change) that exposes a behavioural difference. Likewise, the complexity can be reduced to cubic complexity ($O(n^3)$) by executing a constant number of test cases/assertions ($O(1)$). For instance, an improvement is achievable by selecting and executing only the most relevant tests for the changes, e.g., from historical test executions in the CI. A reduction is also achievable if only one test assertion is evaluated for each test case, e.g., executing only the assertion that captures the interaction between the pair of mutants has a constant time complexity ($O(1)$).

Algorithm 2: Approximate Commit-relevant Mutants Set

Data: TestSuite, MutantsOnChange, MutantsAroundChange
Result: Relevant Mutants

```
1  RelevantMuts ← ∅;
2  for X ∈ MutantsAroundChange do
3      for Y ∈ MutantsOnChange do
4          for test ∈ TestSuite do
5              for assertion ∈ test do
6                  Yval ← onChangeMutantOutput(assertion, Y);
7                  Xval ← aroundChangeMutantOutput(assertion, X);
8                  XYval ← highOrderMutantOutput(assertion, Y, X);
9                  if Yval ≠ XYval ∧ Xval ≠ XYval then
10                     RelevantMuts ← RelevantMuts ∪ {X};
11                     jump to line 2 and take next mutant X;
12                 end
13             end
14         end
15     end
16 end
17 return RelevantMuts ;
```

6.3 Experimental Setup

6.3.1 Goals

Besides introducing the new approach to determine commit-relevant mutants, the main goal of this study is to investigate the prevalence and characteristics of commit-relevant mutants in evolving software systems in terms of their distribution, program location and correlation with program changes and the number of mutants outside the changes. We also study subsuming commit-relevant mutants and what is the proportion of those that are sufficient to test and cover all other commit-relevant mutants. Specifically, our empirical goal is to achieve the following three main goals:

1. Study the *prevalence, distribution and location* of commit relevant mutants (RQ1 and RQ2);
2. Examine the *correlation* between commit-relevant mutants located within and outside the program changes (RQ3);
3. Investigate to what extent we can even further reduce the scope of commit-relevant mutants by targeting the ones with *subsuming properties* (RQ4).

Overall, our study aims at providing insights into the properties of commit-relevant mutants and demonstrate their importance and effectiveness in testing evolving systems.

6.3.2 Research Questions

As we aim to assess the potential of mutation testing in evolving systems, we investigate the following research questions (**RQs**).

RQ1 Prevalence What is the *prevalence* of "commit-relevant mutants" among the whole set of mutants?

RQ2 Location Are commit-relevant mutants located within or outside the developers' committed changes?

RQ3 Correlation Is there any correlation between the number of commit-relevant mutants located within program changes and the number of commit-relevant mutants outside the changes?

RQ4 Subsumption What is the *proportion* of "subsuming commit-relevant mutants", i.e., the number of commit-relevant mutants that subsumes other commit-relevant mutants, such that testing only these subsuming mutants is sufficient to test all other commit-relevant mutants?

RQ1 aims at improving our understanding of the prevalence, i.e., distribution, of relevant mutants in relation to committed changes. The answer to the question allows having a rough view of the relevant mutant's distribution over different projects and their associated commits. By answering RQ2, we aim to show how many commit-relevant mutants are within or outside the committed changes. The answer promises insights on how to perceive testing of the developer changes on the impacted program effectively, w.r.t., it is important to not only test within

the committed changes but also all the unforeseen dependencies that interact with the changed code. Previous work [32] has shown that redundant mutants inflate mutation scores with the unfortunate effect of obscuring their utility. We, therefore, aim to validate whether relevant mutant sets also suffer from such inflation effects. We attempt to observe a difference in size between sets of identified relevant mutants and subsuming-relevant mutants. Plus, we investigate whether there is a correlation between mutants identified as relevant, subsuming-relevant, and mutants on a change (RQ3 and RQ4). In particular, having a strong correlation would indicate that the size of a change itself influences the number of relevant mutants. Otherwise, the relationship does not exist for the reason of the complexity and area of the change.

6.3.3 Analysis Procedure

We focus our empirical study on commits of Java programs as selected subjects. To perform the mutation analysis, we employ PiTest[7] [58], one of the state-of-the-art Java mutation testing tools. We approximate the set of commit-relevant mutants by following the algorithm introduced in Section 6.2.5. Besides the approximated set of commit-relevant mutants located outside of commit-change, we also record and consider as commit-relevant all those mutants residing on the location of commit-change (in our approach, M_Y mutants). This corresponds to work done by [26], whereas the commit-relevant mutants set is made out of mutants located on the commit diff, i.e., statements modified or added by commit.

To make our approximation robust, we follow the steps described in the evaluation process of previous studies [8, 44, 68]. Our approach uses mutant-assertion matrices to identify mutants interactions that constitute, up to our knowledge, the first study conducted on test assertion level for Java programming language (bypassing standard tests passing/failing mutation behaviour for Java programs). Mutant-assertion matrices were computed by running large test pools built by considering developer tests and adding automatically generated tests using EvoSuite[8] [30], a state-of-the-art test case generation tool. From the computed mutant-assertion matrices, we obtain three sets of mutants: *mutants on a change*, *mutants relevant to a change* and *mutants not relevant to a change*.

To answer **RQ1**, we study the prevalence of commit-relevant mutants in every commit by analyzing the average number of relevant/non-relevant mutants and their distribution. We address **RQ2** by studying whether the commit-relevant mutants are located within or outside the committed changes. Moreover, we observe the proportion of those mutants, and we observe the ratio of commit-relevant mutants located within changed methods. To answer **RQ3**, we evaluate whether there is a correlation between the studied categories of mutants using different statistical correlation analysis tests, further described in Section 6.3.5. In the end, we answer **RQ4** by studying the proportion of subsuming commit-relevant mutants among all commit-relevant mutants and all subsuming mutants. This will estimate an extra possible reduction we can achieve if we focus only on subsuming mutants. We consider traditional passing/failing test behaviour to compute the set of subsuming mutants per subject (notice that this information is also captured when mutant-assertion matrices were built).

[7] http://pitest.org/
[8] https://www.evosuite.org/

6.3.4 Subject Programs and Commits

We focus our empirical study on commits of a set of well-known, well-tested, and matured Java open-source projects taken from Apache Commons Proper repository[9]. The process of mining repositories, data analysis, and collection was performed as follows:

1. Our study focuses on the following projects: *commons-collections, commons-lang, commons-net, commons-io, commons-csv*. These projects differ in size while having the most extended history of evolution. We extracted commits from the year 2005 to 2020. To extract commit patches and hunks in our setup, we employ `PyDriller`[10] (`V1.15`) to mine commits from the selected projects[11]. We applied `PyDriller` to query the project's information such as commits hash id, modifications date, modified source code, modification operation, and hunks of the commits and quickly exported such information into a JSON file.

2. We kept only commits that use JUnit4+[12] as a framework to write repeatable tests since it is required by EvoSuite [30], the test generation tool we use for automatically augmenting test suites.

3. We filtered out those commits that do not compile, do not have a green test suite (i.e., some of the tests are failing), or do not affect a program's source code (i.e., commits that only change configuration files). Some commits with failing tests are filtered out since PiTest requires a green test suite to perform mutation testing analysis.

4. Due to the significant execution time for commits containing several files, we set a limit for 72h of execution on a High-Performance Computer to generate and execute mutants per commit. Please note that the test suites contain developer-written and automatically generated tests, where both are used to create mutation matrices. All experiments were conducted on two nodes with 20 physical cores and 256GB of RAM. Specifically on Intel Skylake Xeon Gold 2.6GHz processors, running on Linux Ubuntu OS across four threads.

Overall, we generated 9,368,052 high-order mutants and 260,051 first-order mutants, over 288 commits, that required 68,213 CPUs days of execution. Table 6.2 summarises the details of the mined commits. Column "# Commits" reports the number of commits mined per project, column "# LOC" (Lines Of Code) indicates a subject scope in terms of lines of code, "Maturity" reports on the date of the first commit, column "# FOM" (First-Order Mutant) indicates the total number of First Order Mutants generated for those commits, "#Mutants on Change" indicates the number of First Order Mutants generated on the changed lines, column "#HOM" (High-Order Mutant) indicates the total number of High Order Mutants generated, column "# Dev. Tests" (Developer written Tests) reports on the number of developer-written test cases, and column "# Evosuite Tests" reports on the number of automatically generated tests.

[9]https://commons.apache.org

[10]`https://pydriller.readthedocs.io/en/latest/intro.html`

[11]`PyDriller` is an open-source Python framework that helps developers mine software repositories and extract the information given the GIT URL of the repository of interest.

[12]https://junit.org/junit4/

Table 6.2: Details of Subjects Programs and Studied Commits. Columns: *"# LOC" - Lines Of Code, "# FOM" - First Order Mutants, "# HOM" - Higher Order Mutants, "# Dev. Tests" - Developer written Tests*

Commons Projects	# LOC	# Maturity	# Commits	# FOM	# Mutants on Change	# HOM	# Dev. Tests	# EvoSuite Tests
collections	74,170	14/04/2001	45	27,417	2,026	1,192,188	4,797	1,285
io	29,193	25/01/2002	30	24,970	1,115	668,448	914	286
text	22,933	11/11/2014	46	47,847	4,155	2,073,829	1,084	322
csv	4,844	25/01/2002	101	66,862	3,577	1,968,137	6,144	2,833
lang	85,709	19/07/2002	66	102,072	3,891	3,885,341	7,574	959
Total	216,489	N/A	**288**	**269,168**	**14,764**	**9,787,943**	**20,513**	**5,685**

6.3.5 Metrics and Measurements

Statistical Analysis: To answer our research questions, we performed several statistical analyses to evaluate correlations among several variables. For instance, in **RQ3**, we analyzed whether the number of commit-relevant mutants correlates with the number of mutants residing on a change and whether the number of subsuming commit-relevant mutants correlates with the number of subsuming mutants.

In this study, we employ two correlation metrics, namely *Kendall rank coefficient (τ) (Tau-a)*, and *Spearman's rank correlation coefficient (ρ - (rho))* , with the level of statistical significance set-up to $p - value$ 0.05. The Kendall rank coefficient (τ) measures the similarity in the ordering of studied scores, while Spearman's ρ (rho) measures how well the relationship between two variables can be described using a monotonic function [153]. The correlation metrics calculate values between -1 to 1, where a value close to 1 or -1 indicates strong correlation, while a value close to zero indicates no correlation at all. Additionally, to facilitate comprehension of our figures, we employed *coefficient of determination* (R^2 trendline) as a statistical measure that describes the proportion of the variance in the dependent variable that is predictable from the independent variable(s).

6.3.6 Implementation Details

Our commit-relevant mutant identification approach is implemented in approximately 5 KLOC of Python code, 600 LOC in Shell scripts and 3 KLOC of Java. It employs several external tools and libraries, including `Evosuite`, `git-diff` and `PiTest`. We have also implemented additional infrastructure on `PiTest` to ensure analysis of evolving software and extract assertion information. In the following, we describe each of these tools. For replication and future use, our implementation is publicly available at the following web link: **`https://mutationtesting-user.github.io/evolve-mutation.github.io/`**

6.3.6.1 EvoSuite (V1.1.0)

To obtain a rich test suite for our study, we collected developer-written tests and automatically generated tests. For our mutation testing analysis, we augment developers' test suites with test cases automatically generated with `EvoSuite` [30]. `EvoSuite` is an evolutionary testing tool that generates unit tests for Java software. In our analysis, we run `EvoSuite` against all several coverage criteria (e.g., line, branch,

mutation, method, etc.); we also executed `EvoSuite` with default configurations, especially concerning running time.

6.3.6.2 PiTest (V1.5.1) and git-diff

PiTest does not have built-in functionality to satisfy the requirements of our experiment. Therefore, we extended the framework for High Order Mutants [154] on top of PiTest that takes as an input the `gitdiff` output[13]. Based on the statement difference between the versions, the framework extends the mutants generation functionality by generating, i.e., mapping, mutants on the change, with the mutants around the change. Thus, creating second-order mutants for that particular commit file. Our framework is configured to generate the extended set of mutants available in PiTest, introduced by Laurent *et al.* [139]. Kintis *et al.* [57] has also shown that this extended set of mutants is more powerful than the mutant sets produced by other mutation testing tools.

6.3.6.3 PiTest Assert

`PiTest (V1.5.1)` creates killing matrices and identifies whether a mutant is killed or not based on test case oracle prediction (test fails or passes). These matrices were not suitable for our experimental procedure. Therefore, we built a framework on top of PiTest to extract additional information concerning each test case assertion (from tests that cover mutants). Our framework performs bytecode instrumentation of each test executed on a specific mutant, using ASM[14] as an all-purpose Java bytecode manipulation and analysis framework. By instrumenting each test case assertion, we can obtain execution information. More precisely, each assertion has a unique test name where it locates, an assertion function name, an assertion line number, and an assertion actual execution value. If an assertion triggers an exception, we keep track of the stack-trace execution. However, for this study's purpose, we disregard the assertions that trigger the exception from our relevant mutants calculation (please refer to Algorithm 2) since we only aim at actual mutants' observable behavioural output. Hence, the mutant assertion matrix is a weighted matrix. For each (mutant, test-assertion) pair, the value corresponds to the actual assertion value obtained by running the test on the mutant or the exception stack trace if an assertion throws an exception.

Concretely, we employ the *JUnit4*[15] testing framework, which contains a public class (called `Assert`) that provides a set of assertion methods to specify test conditions. Typically, these methods (e.g., `Assert.assertEquals(expected value, actual value)`) directly evaluate the assertion's conditions, then returns the final assertion's output (e.g., conditions not satisfied, pass, or fail). To obtain the value of parameters within the assert statement, in our framework, we use `PiTest Assert` to instrument each assertion method. Such that we serialize the provided input values in the assert statement before they propagate to conditional checks, i.e., before the conditional check is reached in *org.junit.Assert*[16] and the output values are fed to *org.hamcrest.Matcher*[17] for evaluation. Specifically, we serialize both the expected and actual values after they propagate as input parameters of the assert statement.

[13]https://git-scm.com/docs/git-diff
[14]https://asm.ow2.io/
[15]https://junit.org/junit4/
[16]https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html
[17]http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/Matchers.html

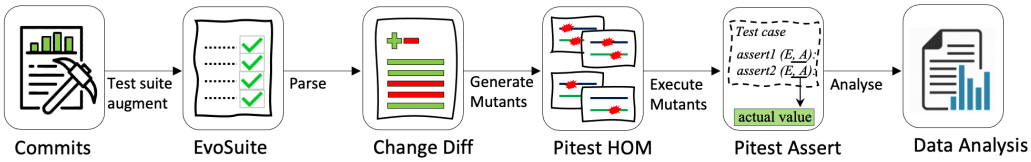


Figure 6.4: Research Protocol

This allows us to assess the input parameters of the `assert` statement (e.g., an expression or a method call (`assertEqual(foo(), bar())`)) for concrete values. Hence, in our setup, we compare the output values of both the expected and actual values present in each assertion. However, our experimental framework does not directly account for the potential dependencies within assertions and test cases; we address this concern in the *threats to validity* (*see* section 6.6). The test assertion framework is built on top of `PiTest` and is publicly available[18].

6.3.7 Research Protocol

Figure 6.4 highlights our experimental protocol, which proceeds as follows: For each project (e.g., *commons-collections*) and each mined commit (e.g., hash: *03543e5f9*, we first *augment the developers' test suite* with automatically generated tests using EvoSuite [30]. Next, we *obtain the commit changes* (a.k.a hunks) of the commit using the `git-diff` tool, in order to identify the changed and unchanged program statements. We then *generate both first-order and second-order mutants* for the program, using PiTest Assert as our extension of PiTest Mutation Testing tool [58]. After mutant generation, we *execute every mutant to obtain the mutant-assertion matrices*, which provides information about test assertion type, position and value. Finally, we execute our relevant mutant detection algorithm 2 to identify commit-relevant mutants.

Our result analysis proceeds after computing mutant assertion matrices and identifying commit-relevant mutants. We then perform the data gathering and analysis required to answer every research question (**RQs**).

Table 6.3: Details of the Prevalence of Commit-relevant Mutants. Columns: *"# C. All R. M." - Number of Commits with all relevant mutants, "# C. No R. M." - Number of Commits with no relevant mutants*

Commons Project	# Commits (C)	# C. All R. M.	# C. No R. M.	# Relevant	# Not Relevant	Ratio	Reduction Ratio
collections	45	2	4	6,833	18,558	32,31%	67,69%
io	30	0	3	6,052	17,803	28,70%	71,30%
text	46	1	4	8,810	34,882	27,10%	72,90%
csv	101	4	0	27,441	35,844	47,39%	53,61%
lang	66	1	2	15,724	82,457	19,22%	80,78%
Total	**288**	**8**	**13**	**64,860**	**189,544**	**N/A**	**N/A**
Average	**58**	**N/A**	**N/A**	**225**	**658**	**29,58%**	**70,42%**

[18]https://github.com/Ojda22/pitest/tree/pit-SOM-RM-AssertCache

6.4 Experimental Results

We start by studying the proportion of *commit-relevant mutants* that affect the commit changes out of *all mutants* by using the pipeline just introduced in Section 6.3. Thus, in the following RQs result analysis, we consider as commit-relevant mutants all mutants identified by our approach, including the set of killable mutants residing on modified statements. We distinguish commit-relevant mutants in the categories of those located on changed and unchanged code to demonstrate and estimate the potential reduction in terms of the number of mutants requiring analysis and the number of test executions required to cover them if the tester focuses testing only on commit-relevant mutants instead of the whole set mutants, or on the mutant set consisting of all mutants residing on the modification.

Additionally, we evaluate the properties of commit-relevant mutants that can inform their selection among all mutants. Thus, we examine the location of commit-relevant mutants, whether they are mostly located within the commit or outside the committed changes. We also assess whether there is a correlation between the number of identified commit-relevant mutants and the number of commit-relevant mutants within the committed change to determine if the number of mutants within a commit can serve as a proxy to determine the number of commit-relevant mutants. Besides providing analysis on commit-relevant mutants, we also cover and analyse subsuming commit-relevant mutants as a minimal set of mutants that characterise all relevant mutants.

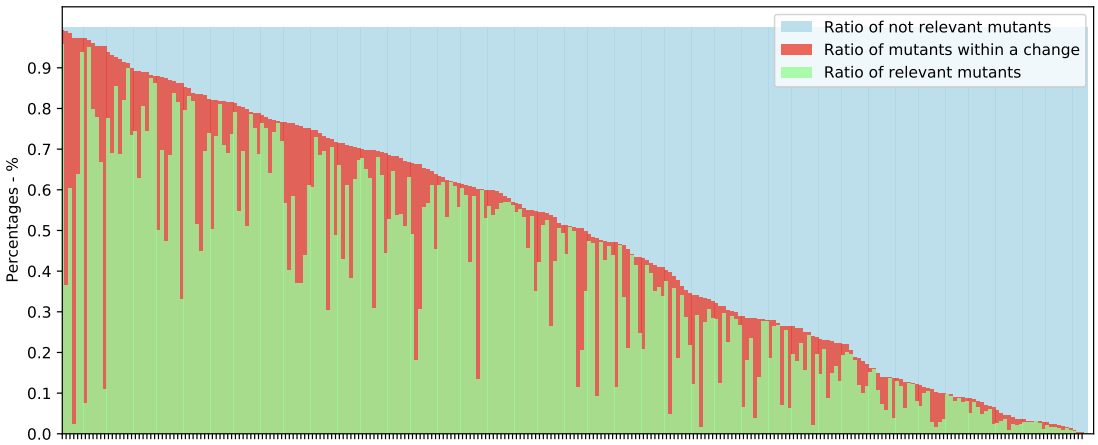


Figure 6.5: Distribution of mutants across all commits showing the proportion of non-relevant mutants (in *blue*) as well as commit-relevant mutants within committed changes (in *red*) and outside committed changes (in *green*)

6.4.1 RQ1: Prevalence

Table 6.3 and Figure 6.5 illustrate the distribution of commit-relevant mutants among all mutants. In our evaluation, we found that *only about one in three (≈30%) mutants are commit-relevant*, on average. In particular, we observed that only about 225 mutants are relevant to a commit out of 833 mutants, on average. This implies that an *effective commit-aware mutation testing technique can reduce significant mutation testing effort, both computational when executing mutants and manual when analysing mutants.* In addition, we found some (21) outliers in our analysis

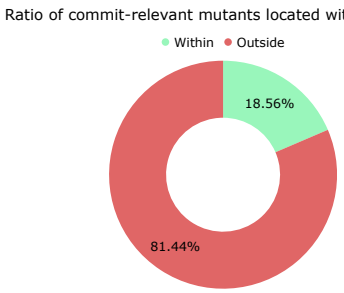


Figure 6.6: Proportion of commit-relevant mutants within the commit (18.56%) and outside the commit (81.44%)

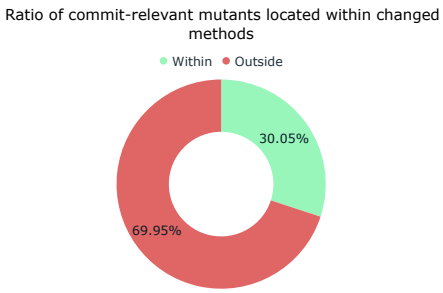


Figure 6.7: Proportion of commit-relevant mutants within the change method (30.05%) and the outside changed methods (69.95%)

of commit-aware mutants, see columns "# C. All R. M." (Number of Commits with all Relevant Mutants) and "# C. No R. M." (Number of Commits with No Relevant Mutants): In particular, we found that only 2.8% of commits (8) had 100% commit-relevant mutants, this portrays the importance of mutant selection for evolving software systems. On the other hand, our evaluation results show that in 4.5% of the commits (13), we found no commit-relevant mutants outside the change; this suggests that it is pertinent to develop commit-aware mutation testing techniques that discern relevant from non-relevant mutants. Overall, these findings demonstrate the importance of developing commit-aware test selection for evolving software systems, in particular, in selecting relevant mutants to reduce testing effort.

> One in three (approximately 30%) mutants are commit-relevant; hence, selecting commit-aware mutants can significantly reduce mutation testing costs.

6.4.2 RQ2: Location

In our evaluation, most (81%) commit-relevant mutants are outside of developers' committed changes (*see* Figure 6.6). Making only about one in five (19%) commit-relevant mutants within the committed changes of developers. For instance, a developer that tests *all commit-relevant mutants within the changed method* will test only 30% of commit-relevant mutants and miss almost 70% of commit-relevant mutants (*see* Figure 6.7). This result suggests that to test the impact of developer changes on the program effectively, it is important to not only test within the committed changes. It is also highly pertinent to test the interaction of committed changes with the rest of the unmodified program.

> Most (81% of) commit-relevant mutants are located outside of the commit, and only a few (19% of) commit-relevant mutants are within the commit.

6.4.3 RQ3: Correlation

Our evaluation results show that *there is a weak trend between the number of commit-relevant mutants and the number of mutants within the commit.* Our statistical correlation analysis shows that there is a *weak correlation* between both variables. In particular, we found Spearman and Kendall correlation coefficients of 0.212 and 0.141, respectively. Indeed, both the Spearman and Kendall correlation

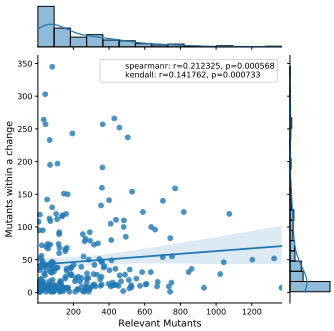


Figure 6.8: Correlation Analysis between the number *mutants within a change* and the number of *commit-relevant mutants*

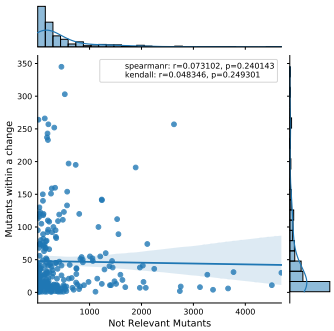


Figure 6.9: Correlation Analysis between the number of *mutants within a change* and the number of *non-relevant mutants*

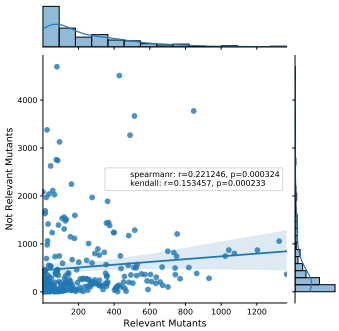


Figure 6.10: Correlation Analysis between the number of *non-relevant mutants* and *commit-relevant mutants*

coefficients are statistically significant (with p-values of 0.0006 and 0.0007, respectively). Figures 6.8, 6.9 and 6.10 summarize the results of the different studied correlations. These correlation results suggest that there is a weak relationship between the number of mutants within a change and the number of commit-relevant mutants, but *no robust and predictable pattern or trend between both variables*. This implies that *the number of mutants within the commit can not reliably predict the number of commit-relevant mutants* (in unmodified code regions), and vice versa.

> There is a statistically significant weak positive correlation between the number of commit-relevant mutants and the number of mutants within the change (Spearman and Kendall correlation coefficients of 0.212 and 0.141, respectively).

6.4.4 RQ4: Subsumption

In this section, we investigate the prevalence of *subsuming commit-relevant mutants* among *commit-relevant mutants*. Estimating the proportion of subsuming commit-relevant mutants is important to demonstrate the further reduction (in the number of mutants to analyse) achieved by "selecting" or "optimizing" for effectively identifying *subsuming commit-relevant mutants*, in comparison to *commit-relevant mutants*, *subsuming mutants* and *all mutants*. The two subsumption relations (i.e., one for the commit-relevant mutants and the other one for all mutants) are computed by following the definition introduced in Section 2.2.5.

Additionally, we examine the correlation between the number of subsuming commit-relevant mutants and the number of commit-relevant mutants within a change and subsuming mutants; this is rather important to determine if these variables hold a relationship and can predict or serve as a proxy for determining subsuming commit-relevant mutants. Thus we ask:

What is the proportion of "subsuming commit-relevant mutants" among commit-relevant mutants, such that a test suite that distinguishes a(ll) subsuming commit-relevant mutant(s) covers (all) other commit-relevant mutants? Figure 6.11 illustrates the proportion of *subsuming commit-relevant mutants* and their intersection with *commit-relevant mutants* as well as all mutants. In our evaluation, we found that *"subsuming commit-relevant mutants" are significantly smaller than commit-relevant mutants and all mutants*. About one in 20 mutants is a *subsuming commit-relevant*

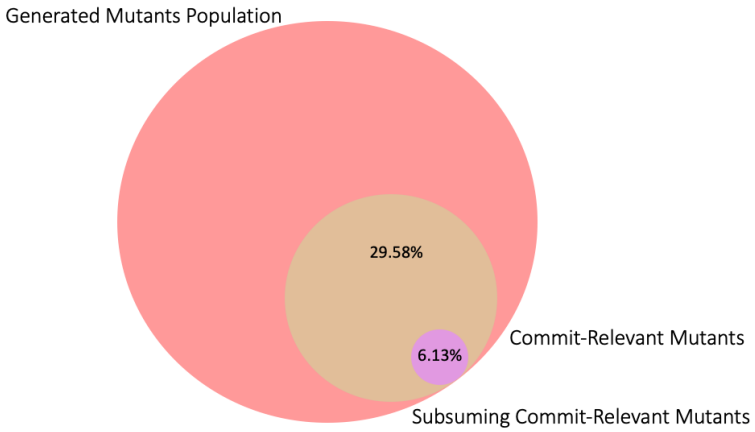


Figure 6.11: Venn diagram showing the proportion of "commit-relevant mutants" (29.58% in *orange*)) and "subsuming commit-relevant mutants" (6.13% in *purple*) among all mutants (in *pink*).

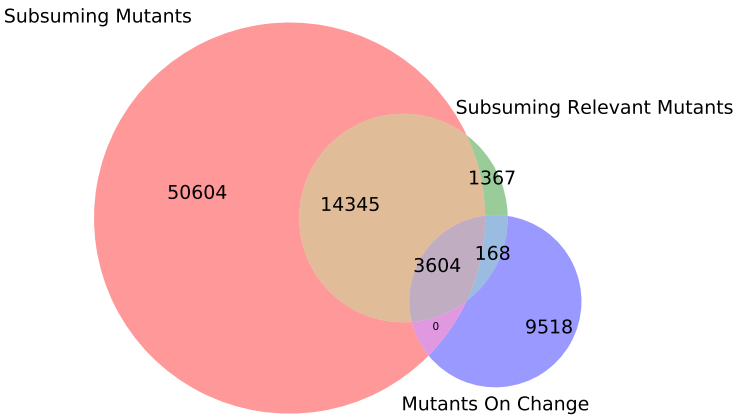


Figure 6.12: Venn diagram showing the number and intersections among "commit-relevant mutants within commit changes" (in *blue*),"subsuming commit-relevant mutants" (in *orange*) and "subsuming mutants" (in *pink*).

mutant, and about one in five (5) commit-relevant mutants is a subsuming commit-relevant mutant. Specifically, "subsuming commit-relevant mutants" represent 20.72% and 6.13% of all commit-relevant mutants and all mutants, respectively. This suggests it is worthwhile to identify and select subsuming relevant mutants from all (commit-relevant) mutants. Invariably, generating only subsuming commit-relevant mutants reduces the number of mutants to analyze by 79% and 93% compared to generating commit-relevant mutants and all mutants, respectively. This result implies that developing automated mutation testing methods that effectively identify, select or generate subsuming commit-relevant mutants can significantly reduce mutation testing costs.

> Selecting "subsuming commit-relevant mutants" can reduce the number of mutants to be considered by about 79% and 93% in comparison to commit-relevant mutants and all mutants, respectively.

What is the proportion of "subsuming commit-relevant mutants" among "subsuming mutants" and "commit-relevant mutants within a change"? Figure 6.12 illustrates

the intersections between all three types of mutants. Notably, *most (92.98% – 18,117 out of 19,484) subsuming commit-relevant mutants are subsuming mutants as well, and they represent 26.42% of all subsuming mutants (68,553).* This implies that searching for subsuming commit-relevant mutants among subsuming mutants (instead of all mutants) is beneficial in reducing the search scope.

We also observed that all subsuming commit-relevant mutants within committed changes are subsuming mutants. Meanwhile, about one in five (19.36% – 3,772 out of 19,484) subsuming commit-relevant mutants are within the developers' committed changes; they represent 28.38% (3,772 out of 13,290) of all mutants within the change. This suggests that less than one in three mutants within the change are subsuming commit-relevant mutants. Hence, it is important to search for subsuming commit-relevant mutants outside of the committed changes since most subsuming commit-relevant mutants (81%, 15,772) are outside the committed changes.

> Most (92.98% of) subsuming commit-relevant mutants are subsuming mutants, while a few (19.36% of) subsuming commit-relevant mutants are located within committed changes.

Is there a correlation between the number of subsuming commit-relevant mutants and the number of mutants within a change? Our correlation analysis shows that *there is a weak positive correlation between the number of commit-relevant mutants within a change and the number of subsuming commit-relevant mutants* (*see* Figure 6.13). Both Spearman and Kendall correlation coefficients report a weak positive correlation, with correlation coefficients 0.222 and 0.148, respectively, (*see* Figure 6.13). In particular, the correlation coefficients are statistically significant with p-values less than 0.05, specifically, 0.0003 and 0.0004 for Spearman and Kendall coefficients, respectively. This result suggests that the number of mutants within a change can not strongly predict the number of subsuming commit-relevant mutants; hence, it is important to

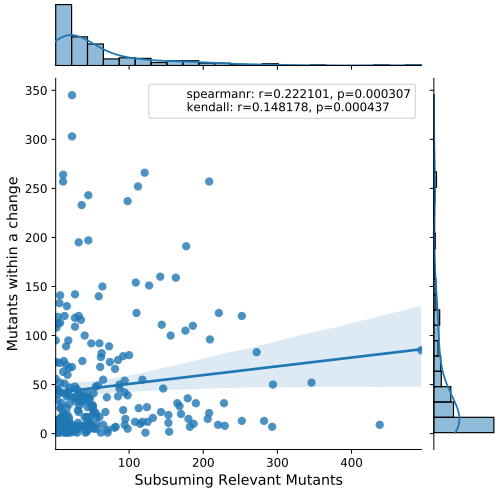


Figure 6.13: Correlation Analysis between the number of mutants within a change and the number of subsuming commit-relevant mutants

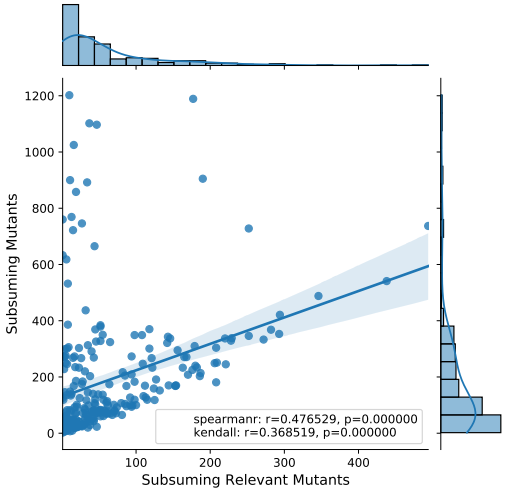


Figure 6.14: Correlation Analysis between the number of subsuming mutants and the number of subsuming commit-relevant mutants

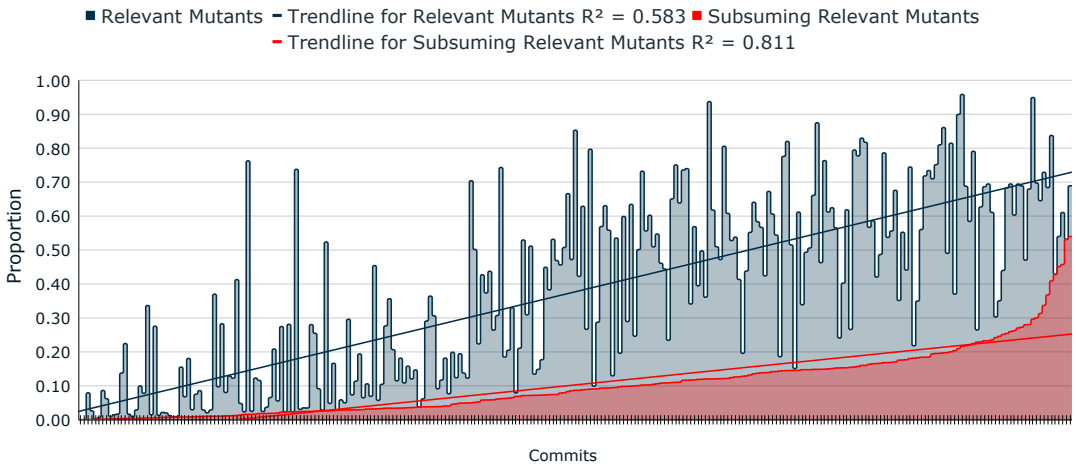


Figure 6.15: Distribution of the proportion of commit-relevant mutants (in *gray*) and subsuming commit-relevant mutants (in *red*); Commits are sorted from left to right in ascending order of the proportion of subsuming relevant mutants

identify all commit-relevant mutants that interact with the committed changes and not only test the change itself.

> The number of mutants within a change can not reliably predict the number of subsuming commit-relevant mutants since there is only a weak positive correlation between both variables.

What is the relationship between the number of subsuming commit-relevant mutants and the number of subsuming mutants? Figure 6.14 illustrates the distribution and correlation between the number of subsuming mutants and the number of subsuming commit-relevant mutants. In this figure, the trending line shows that *there is a moderate positive correlation between both variables.* Indeed, both Spearman and Kendall correlation coefficients report a *moderate positive relationship* between both variables, with correlation coefficients 0.476 and 0.368, respectively, (*see* Figure 6.14). The correlation coefficients also show that the positive relationship is statistically significant (p-value < 0.05). As expected, we observed that the proportion of subsuming relevant mutants per commit increases (trendline R^2=0.881) as the proportion of commit-relevant mutants increases (*see* Figure 6.15). Overall, this result implies that these variables can serve as a proxy for each other, hence predicting one variable could help identify the other. In particular, this implies that selecting subsuming mutants significantly increases the chances of selecting subsuming commit-relevant mutants.

> There is a moderate positive relationship between the number of subsuming commit-relevant mutants and the number of subsuming mutants, such that one can predict the other and vice versa.

6.5 Discussion

6.5.1 Summary of Contributions and Implications

In this study, we propose a novel experimental approach for determining commit-relevant mutants by studying the relevance of an instruction to a program point of interest. To achieve this, we measure the impact of the second-order mutant on the first-order ones, thus approximating its relevance and the explicit relation that exists between a mutant that rests on a changed part of the code and those that rest outside of the changed code. Therefore, we further illustrated that dynamic approaches (like observation slicing) could complement static or machine learning-based approaches in effectively identifying commit-relevant mutants. This formulation of commit-relevant mutants allowed us to create up to date the most extensive dataset of commit-relevant mutants, which counts 10,071,875 mutants and 288 commits extracted from five (5) mature open-source software repositories. Generation of such a dataset took over 68,213 CPU days of computation. After thorough studying of the dataset, we come up with some empirical findings that include the following:

1. Commit-relevant mutants, at the unit level, are *highly prevalent* (30%), and most commit-relevant mutants (81%) are *located outside of program commit changes*. Hence, it is important to conduct a mutation analysis of evolving systems to determine the influence of the program changes on the rest of the unmodified code.

2. Adequate selection of (subsuming) commit-relevant mutants significantly reduces the number of mutants involved (approximately 93%); thus, there is a huge benefit to developing effective and practical techniques for the selection of (subsuming) commit-relevant mutants in evolving systems.

Additionally, our evaluation results show that most commit-relevant mutants are located outside of the commit changes due to the interaction of changes with the unmodified program code. In our evaluation, commit-relevant mutants that capture evolving software behaviour are located all around the program changes. Besides, we observe that the effective selection of commit-relevant mutants would significantly reduce the number of mutants requiring analysis. Thus, we encourage researchers to investigate automated methods for identifying and selecting commit-relevant mutants, for instance, using statistical analysis or program analysis.

Next, one of the main insights of our study is to demonstrate that beyond the committed changes, other program locations are also important for commit-aware mutation testing. Hence, it is important to identify the relevant program locations for commit-aware mutant injection. To achieve this, we encourage the use of program analysis techniques (e.g., slicing) that determines the program dependencies between changes and the rest of the program, such that mutant injection is focused on selecting such dependencies to reduce the search space and cost for mutation testing effectively. It is also pertinent to note that *the subsumption relation of mutants can help considerably reduce the effort during commit-aware mutation testing.* Indeed, it is important to identify and prioritize (subsuming) mutants during mutation testing of evolving systems. Identifying those mutants allows developers to augment their test suite to include new tests that exercise the program change and its dependencies. Besides this use case, the commit-relevant mutants are important for effectively

testing for regression bugs, i.e., if program changes (or commits) introduce new failures or break previous features.

6.6 Threats to Validity

Our empirical study and findings may be limited by the following validity threats.

External Validity: This refers to the generalizability of our findings. We have empirically evaluated the characteristics of commit-relevant mutants on a relatively small set of open-source Java programs, test cases, and mutants. Hence, there is a threat that our experimental protocol and findings do not generalize to other mutants, programs, or programming languages. Additionally, there is the threat that our findings do not generalize to other Java projects since our subject programs are all from the Apache Commons project and may share similar characteristics in terms of architecture, implementation, coding style and contributors. We have mitigated these threats by conducting our experiments on five (5) matured Java programs with a varying number of tests and a considerably large number of mutants. In our experiments, we had 288 commits and 10,071,872 mutants with 25 different groups of mutant types. In addition, our subject programs have 216,489 KLOC and 17 years of maturity, on average. Hence, we are confident that our empirical findings hold for the tested (Java) projects, programs, commits, and mutants. Furthermore, we encourage other researchers to replicate this study using other (Java) programs, projects and mutation tools.

In our experiments, we used PiTest [58] to perform our analysis. However, it is likely that the use of a different mutation tool may impact our findings since it may contain different operators than PiTest. While this is possible, recent empirical evidence [57] has shown that PiTest has one of the most complete sets of mutation operators that subsumes the operators of the most popular grammar-based mutation testing tools in almost all cases. Nevertheless, we are confident in our results since PiTest includes a large sample of mutants; the general results are unlikely to change with different types of simple mutations.

Internal Validity: This threat refers to the *incorrectness* of our implementation and analysis, especially if we have correctly implemented/deployed our experimental tools (e.g., Evosuite, PiTest and PiTest Assert), performed our experiment as described and accounted for randomness in our experiments. We mitigate the threat of incorrectness by (manually) testing our implementation, tools, and experimental protocol on a few programs and commits to ensure our setup works as expected. Specifically, we performed manual testing by examining five (5) representative Apache programs containing about 500 LoC per commit on average. While we inspected in total about 20 commits with over 30 LoC in patch sizes, on average. We also address the threat of randomness in our experiments by repeating our experiments 100 times to mitigate any random or stochastic effects.

Construct Validity: This refers to the *incompleteness* of our experimental approach, in terms of *identifying all commit-relevant mutants*. Despite the soundness of our approach, it only provides an approximation of commit-relevant mutants, such that the set of identified commit-relevant mutants is only a subset of the total number of all commit-relevant mutants. This is due to the finite set of test cases and mutants employed in our experiments. We have mitigated this threat by ensuring we have a reasonably large set of mutants and test cases for our experiments. For instance,

following the standards set up by previous studies [8,44,68], we augmented developers' written tests by automatically generating additional tests (using EvoSuite) to expand the observable input space for commit-relevant mutants. Our experimental findings are also threatened by the potential noise introduced by *equivalent mutants*. First, notice that commit-relevant mutants come either from lines within the change or outside the change. On the one hand, considering our algorithm 2 for identifying commit-relevant mutants outside the change, you can notice that in case mutant X is equivalent, then condition $Yval \neq XYval$ in Line 9 will evaluate to *false* since mutants Y and XY will be equivalent as well, then mutant X will not be considered as commit-relevant. On the other hand, our approach selects by default all the mutants within the change as commit-relevant, so there is a potential threat in selecting some equivalent mutant, even though mutants within the change are a small fraction concerning the total number of mutants. To mitigate this threat, we employ standard methods in mutation testing to reduce the probability of generating equivalent mutants, for instance, by applying `PiTest` to ensure no common language frameworks are mutated.

To determine the interactions between mutants, we employ a coarse-grained assertion check in our experiments. Specifically, our assertion checks are at the assert parameter level. As an example, given a first-order mutant and a second-order mutant, we directly check the equality of the parameter values (i.e., the expected and actual outcomes) for both mutants. This raises the threat of missing more fine-grained assertion properties, especially the effect of dependencies within assertions and test cases. Our approach may mask such dependencies, e.g., if there is a dependency between the expected and actual value within the assertion. Indeed, this assertion checks may limit the number of observed commit-relevant mutants, as a more fine-grained approach (e.g. one that accounts for such dependencies) may reveal more commit-aware mutants. Finally, we encourage other researchers to investigate the effect of these threats on the performance of commit-aware mutation testing.

6.7 Conclusion

We presented a novel approach to identifying commit-relevant mutants by using high-order mutants. More precisely, with this approach, we measure the impact of second-order mutants on the first-order ones, which captures the existence of implicit interactions between the changed and unchanged code parts. This approach allowed us to create up to date the most extensive dataset of commit-relevant mutants, which counts 10,071,875 mutants and 288 commits extracted from five (5) mature open-source software repositories. After thoroughly analysing the mutants, our results show that the commit-relevant mutants are highly prevalent (30%), and most (81 %) are located outside of program commit changes. Additionally, it is pertinent to note that we studied subsuming commit-relevant mutants as a minimal set of mutants sufficient to represent all others, and found that by focusing on this particular category of mutants, we can significantly reduce the number of commit-relevant mutants by around 93%. Obtained results provide a sense of the huge benefit of focusing on mutants that captures change-aware test requirements and open a new direction on opportunities for studying and developing effective selection techniques of commit-relevant mutants. For the link to the code of our approach, scrutiny and future research, we publicly provide our artefacts, data and experimental results: **`https://mutationtesting-user.github.io/evolve-mutation.github.io/`**

7

Mutant Relevance to Code Evolution and Long Standing-Mutants to Keep Mutation Test Suites Consistent

Automatic selection of commit-relevant mutants is an open problem which regards how to most cost-effectively identify commit-relevant mutants. Naturally, the question emerges whether the most extensive dataset, to this end, of commit-relevant mutants can provide scientific insights concerning the selection of mutants in testing evolving systems written in Java language. Precisely, this chapter aims to examine the properties, predictability, and utility of commit-relevant mutants, as well as subsuming commit-relevant mutants such as to quantify further the potential benefits of selecting relevant mutants during project evolutions concerning cost and effectiveness. Furthermore, this chapter argues that there is a remaining barrier to uptake: mutant consistency. Besides tracking test effectiveness after each code modification, a consistent set of mutants for a project is needed so that test effectiveness can be consistently tracked against a common baseline over a series of project releases. In this chapter, we also present a mutation test brittleness metric represented through long-standing mutants, which can be used to assess a mutation suite, and software project, in terms of the rate at which mutation test relevance decays over a series of releases. Our results demonstrate that mutants have diverse life spans across program versions, while our analysis shows that identifying a high-quality suite of long-standing mutants allows us to maintain mutant relevance over a series of releases: a long-standing mutant suite provides test effectiveness relevance for at least 10x longer than a randomly selected suite.

This chapter is based on the work published in the following research papers:

- Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. *"Mutation Testing in Evolving Systems: Studying the Relevance of Mutants to Code Evolution."* 2023, *ACM Transactions on Software Engineering and Methodology.* 32, 1, Article 14 (January 2023), 39 pages. `https://doi.org/10.1145/3530786`
- Milos Ojdanic, Mike Papadakis, and Mark Harman. *"Keeping Mutation Test Suites Consistent and Relevant with Long-Standing Mutants."* Under submission in the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Ideas, Visions and Reflections (IVR) Track, 2023, `arXiv:2212.11762`

Contents

7.1 Introduction

Testing evolving systems requires significant efforts in re-assessing every changed program functionality in every evolution cycle. To reduce these efforts, automatic tests are applied to re-test previously formed functionality, enabling test reuse. While this practice is effective and valuable, it leaves the test adequacy question open, i.e., whether test suites are sufficient for testing the given system version. More often than not, developers add their tests ad hoc, following specific coverage criteria, which often gives the false impression of test adequacy, implying stability for future changes. Ultimately, even if testing was successful on its previous or current version, it might not be the case in the future, given that the program evolves and testing requirements accumulate. Applying traditional mutation testing in CI processes is impractical due to its cost. Meanwhile, Commit-Aware Mutation Testing promises to scale and defines commit-relevant mutants as a set of mutants affected by the changed program behaviour that serve as commit-relevant test requirements to guide test assessment by aiming at the changed program functionality [88, 155]. Commit-relevant mutants capture unforeseen interaction between a changed part of the code and a not changed part of the code and likewise serve as valuable change-aware test assessment metrics. More recently, learning-based approaches [113] emerged capable of learning the commit-relevant mutants defined by a regression change in commit time in C programming language, thus showing potential and opening a direction towards learning mutant's behaviour in evolving context. However, for Java as one of the most popular languages, the task of learning mutant behaviour comes to be a nontrivial task due to its dynamic nature. Therefore, in this chapter, we perform an empirical analysis of commit-relevant mutants in Java programming language to understand the properties of commit-relevant mutants, aiming to identify, select, or predict commit-relevant mutants effectively. Specifically, we examined the location of mutants, their types, effectiveness, predictability, and utility of commit-relevant mutants, as well as subsuming commit-relevant mutants.

However, in this chapter, we also recognise another challenge. Evolving systems require a continuous test assessment to avoid the degradation of testing strengths. In particular, skipping some test requirements will result in debts that will snowball into errors and be costly to localize and resolve. Therefore it is essential to create tests as an investment for the future, also considering parts of the system that do not change often and whose test assurance is pending to converge toward test thoroughness.

We argue that there is a remaining barrier to uptake: mutant consistency. We need a *consistent* set of mutants for a project so that test effectiveness can be consistently tracked against a common baseline over a series of project releases. Unfortunately, almost all existing research on mutation testing assumes that a fresh set of mutants is created for each release of the system [8, 156]. An alternative would be to fix a set of mutants as a baseline and use this to measure ongoing test effectiveness evolution. However, as the results of this study show, such a fixed mutation set will quickly degrade in its relevance.

We introduce a mutation test brittleness metric, which can be used to assess a mutation suite, and software project, in terms of the rate at which mutant relevance decays over a series of releases. Our results demonstrate that mutants have diverse life spans across program versions. We show that identifying a high-quality suite of *long-standing* mutants allows us to maintain mutant relevance over a series of

releases: a long-standing mutant suite provides test effectiveness relevance for at least 10x longer than a randomly selected suite. In order to increase the consistency and relevance of mutation testing, we conclude that the research community should focus on long-standing mutants, their applications, the opportunities they open, and the remaining open questions.

Specifically, this chapter's primary contributions are:

1. Analysis of the properties, predictability, and utility of commit-relevant mutants.

2. Studying effectiveness and test execution performance of (subsuming) commit-relevant mutants in comparison to the selection baselines. State-of-the-art mutant selection approaches miss a large portion of commit- relevant mutants. For example, random mutant selection techniques miss approximately 45% of subsuming commit-relevant mutants when analyzing the scope of 20 mutants. In contrast, commit-relevant mutation testing significantly reduces test executions, specifically reducing the number of test executions by about 16 times compared to random mutant selection.

3. Reveal that several evaluated commit or mutant-related features can not reliably predict (subsuming) commit-relevant mutants. For instance, (the number of) commit-relevant mutants cannot be reliably predicted by features such as the commit size or mutant operator types.

4. The introduction of long-standing mutants as an important category warranting further study.

5. The introduction of metrics for assessing mutant brittleness and visualisations of how this metric varies for a given project over time.

6. An empirical study of long-standing mutants based on four non-trivial systems and 143,500 mutants.

7. The key motivating finding is that long-standing mutant suites enjoy an order of magnitude longer relevance than a randomly selected suite over the four systems studied.

8. An important 'special relationship' between long-standing and subsuming mutants: mutants that are subsuming in one version have a high probability of subsuming in the following versions. This relationship is important because it opens optimistic prospects for mutants' ability to maintain consistency, relevance, effectiveness *and* subsumption over a series of releases.

7.2 Motivation and Problem Formulation

Applying traditional mutation testing in CI processes is impractical due to its cost. Meanwhile, Commit-Aware Mutation Testing scales and defines commit-relevant mutants as a set of mutants affected by the changed program behaviour that serve as commit-relevant test requirements to guide test assessment by aiming at the changed program functionality [88, 155]. More recently, learning-based approaches [113] emerged capable of learning the commit-relevant mutants defined by a regression

change in commit time, thus showing potential and opening a direction towards learning mutant's behaviour in evolving context. Therefore, it is necessary to investigate the properties, predictability, and utility of commit-relevant mutants, as well as subsuming commit-relevant mutants such as to quantify further the potential benefits of selecting relevant mutants during project evolutions concerning cost and effectiveness. However - it is necessary to realise that to improve the testing process continuously and thus quantify overall testing quality - it would require applying the technique after each program change cycle not to indebt and lose test requirements. The merit of reappearing mature mutants is that they preserve test requirements and thus complement commit-relevant mutants. In particular, the long-standing mutants promise to keep overlooked testing requirements from oblivion and provide test assessment for a prolonged time.

In an attempt to further scale and make test assessment affordable, many recent studies (consult the survey by Papadakis et al. [8, 29]) consider subsuming mutants to reduce the number of mutants required to measure test adequacy [8]. Indeed traditional mutation operators introduce many trivial, duplicated, equivalent and redundant mutants [29, 39]. Specifically, a subsumption relationship between mutants emerges from mutant behaviours, thus suggesting that the majority of the mutants fall into the redundancy basket since distinguishing those subsuming mutants will lead to the identification of all other mutants [42]. See the section 2.2.5 for more information about subsumption.

More formally, given a finite set of mutants M and a finite set of tests T, mutant m_i is said to dynamically subsume mutant m_j if some test in T kills m_i and every test in T that kills m_i also kills m_j [44]. Calculating test effectiveness over subsuming mutants offers a much better test effectiveness indicator than the traditional mutation score since subsuming mutants have an almost linear relationship between the number of tests, providing more practicality for determining how much testing work remains over how much has been completed [110].

Although the evidence is strong and the benefits are multi-fold, calculating subsumption in real time requires knowledge of the mutant's behaviour, usually represented through test execution, which is unpractical in real time. Recently few approaches have used learning-based methods to target subsuming mutants with a certain level of confidence, considering their location and properties [63]. In this study, we aim to reuse the guarantee of the quality of test assessment when the mutants are *also* long-standing.

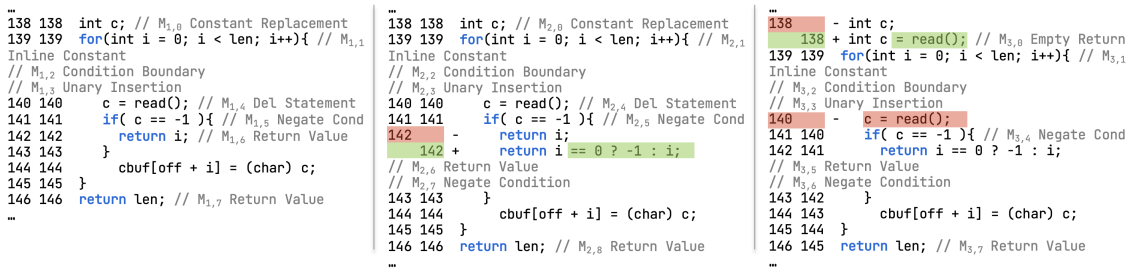


Figure 7.1: Example of mutants standing through 3 chronological sequences of code versions. The example code snippet comes from Apache commons-io project, while method read() is excerpted from the BoundedReader.java (versions around 81210eb). The green and red rectangles represent associated commit changes. While java comments (//) describe the set of mutants $M_{i,j}$, where i is the observed program version, and j is a mutant ID.

7.3 Long-Standing Mutants

7.3.1 Motivating Example

A key challenge in the current state of mutation regression testing is a sequential version-to-version execution. Suppose we keep a record of generated mutants to the same element on which the mutant is generated and version them from one version to another. Figure 7.1, depicts a chronological sequence of 3 different versions of method `read()` extracted from Apache Commons-io project. The figure shows a) code changes, where addition is in a green rectangle and labelled with the "+" sign, and deletion is in a red rectangle labelled with the "-" sign, and b) a set of mutants M for each version containing descriptive mutant operators, written in the format of Java comment "//". Following the evolution of the code, we can observe the evolution of the mutant set. We notice that most mutants reside in the same place across the versions. Moreover, all eight mutants emerging in version 1 are in the same positions in version 2, although a change introduces a new mutant (M_7). When a mutant location is unchanged from version to version, we consider that it stands through time. While if a mutant does not occur in the next version, we consider it to stop standing, e.g., $M_{2,4}$ does not exist as $M_{3,4}$ due to deletion changes. We can observe that instead of $M_{2,0}$ Constant Replacement, there is a new mutant $M_{3,0}$ Empty return, making $M_{2,0}$ cease to exist due to the occurrence of the called method. From the example, we can observe when a system reaches maturity, as in the case of commons-io, the majority of the changes do not touch the core logic, and most mutants $M_{1,n}$ are long-standing (still exist as $M_{3,n}$) precisely six out of eight. In particular, this indicates the potential reuse of past subsumption knowledge concerning selection priority (among other things), avoiding redundancy, addressing technical testing debt and aspiring towards test completeness of mature code components.

7.3.2 Definition

To assess whether mutant M exists (stands) in time or whether the mutant does not exist, the appropriate mapping is required, such as that for a mutant generated on one class statement with a unique mutant operator; there exists the same mutant operator for the exact location on the following program version. Therefore long-standing mutants definition is:

Definition 7. *Given n and m as timepoints of the first and last versions under the study of the program P, where $n > m$. A mutant M is said to be long-standing if it exists on the same code element E throughout consecutive versions P^n, P^{n+1}, ... ,P^m until the point when the mutant M due to a committed program change does not appear on the code element E of a program version P^{m+1}.*

It is worth mentioning that long-standing mutants can exist for several versions or just a few. As long as a mutant stands for at least a version, the knowledge it cares from the previous run has the potential to be reused and provide test assessment for a prolonged time. We envision this property to allow the long-standing measure where mutants can be approached by their maturity.

Given that mutants can 'stand' for several versions or just a few, we argue that the rate at which a mutation suite and mutant relevance decays over a series of releases

suggests a mutation test brittleness. Knowing the degree to which mutants hold high-quality tests for a series of releases helps provide more prolonged test effectiveness. Accordingly, we introduce the mutation brittleness metric that measures mutants' longevity and overflow between versions.

7.4 Experimental Setup

7.4.1 Goals

The main goal of this study is twofold. First, the aim is to investigate the characteristics and predictability of commit-relevant mutants in evolving software systems in terms of their relationship to *commit hunks*, *mutant types* and some other features reportedly used in the related work. We also study the mutants' effectiveness and efficiency in testing evolving systems in comparison to the state-of-the-art. Second, the aim is to investigate the mutation brittleness and long-standing subsuming mutants, such as to demonstrate to what extent subsuming mutants convey their dynamic behaviour and how mutation selection can affect the test assessment capability of mutation testing over time.

Specifically, our empirical goals are the following:

1. study the *properties of commit-relevant mutants*, in terms of their prevalence, mutant types, location and proportions, as well as the *subsumption relation of commit-relevant mutants*;

2. examine the *relationship between commit-relevant mutants and commit properties* (e.g., commit size);

3. investigate the benefit of commit-relevant mutation testing, in terms of their *effectiveness* and *efficiency* in comparison to the baselines.

4. investigate mutants lifetime w.r.t., longevity, over program history.

5. investigates the relationship between long-standing mutants and subsuming mutants such as that the dynamic subsumption relationship can be carried to the next versions.

Overall, our study aims at providing insights into the properties of commit-relevant mutants and demonstrate their importance and effectiveness in testing evolving systems. Plus, to introduce long-standing mutants that evaluate the brittleness of mutation test suites and demonstrate that due to the diverse lifetime of mutants in evolving contexts, some mutants can pour their dynamic subsuming behaviour in the following program versions.

7.4.2 Research Questions

As we aim to evaluate the benefit and assess the potential stronger utility of mutation testing in evolving systems, we investigate the following research questions:

RQ1 Commit Size: *Is there a relationship between the size of the commit (i.e., number of commit hunks) and the number of (subsuming) commit-relevant mutants?*

We answer this question by investigating the distribution of hunks through commits and their association with identified relevant mutants. Answering this question

will help us to understand further how a committed change influences relevant mutants. After analyzing the ratio of sufficient mutants, their location, and their relationship with commit hunks, we want to examine whether it is possible to reduce the number of mutants even further by only focusing on specific types. We study

RQ2 Commit-Relevant Mutant Types: *What is the* distribution of mutant types *in commit-relevant mutants?*

To answer this question, we studied 25 distinct groups of mutant types, among which we were able to identify the most dominant groups across relevant commit mutants. Answering this question will help us to understand patterns when it comes to the identification of relevant mutants and whether a certain type of mutation dominates, making it appropriate for the selection. While the previous questions reported relevant mutants distribution and their characteristics, they do not say much about the usability and properties. Thus, we ask:

RQ3 Comparative Effectiveness: *How effective are (subsuming) commit-relevant mutants, in comparison to the baselines (i.e., random mutation and "commit-only mutation")?*

We answer this question by simulating a scenario where a developer analyses mutants and write tests to kill them. Hence, we are interested to know how much a developer can benefit from relevant mutants by measuring the relative difference between relevant mutation scores. This standard developer simulation aims to quantify the benefit of one mutation-selection technique over another [9, 68]. For different selection techniques, i.e., baselines, we use random mutant selection, mutants on modification, and subsuming-relevant mutants. Answering the above question provides us with insights into the advantages of killing relevant mutants over others. While this is important to demonstrate the usage potential of relevant mutants, there is an open question regarding test efficiency. More precisely, it remains unclear what are the benefits of relevant mutants when it comes to computation. Thus, we ask:

RQ4 Test Executions: *What is the performance of (subsuming) commit-relevant mutants in comparison to the baselines, in terms of the number of required test executions?*

We perform a simulation of the testing scenario, similar to the previous question, in which we measure the number of test executions necessary to reach the same mutation score across baselines. Please note that we are simulating an incremental process where the developer picks mutants, generates a test to kill them, removes killed mutants, and selects the next surviving *(see Section 2.4.2)*. On the opposite side of the commit-relevant mutants, the question emerges how many mutants of a specific file exist on the same code elements through time from their inception without being altered by a code change? The existence of such mutants provides insights into whether there exists a set of consistent mutants being able to track test effectiveness over a series of releases consistently. Therefore we ask:

RQ5 Mutation Brittleness: *What is the longevity (lifetime) of mutants over*

different code evolution timelines?

We answer this question by performing mapping each mutant of a specific program version to its subsequent versions in the program history. As we will see in our results, the longevity of mutants is diverse, and there are mutants with long lifetimes in their project history timelines. Therefore, the next question of interest is whether such mutants can keep their dynamic (subsuming) behaviour even in the future version, thus standing as subsuming bypassing computation of subsuming through test execution. Therefore we ask

RQ6 Long-Standing Subsuming Mutants: *To what extent do subsuming mutants convey their dynamic behaviour, and how can mutation selection affect the test assessment capability of mutation testing over time?*

By answering this question, we can demonstrate the role of mutant selection in the context of evolving systems and emphasise that besides selecting mutants which serve as change-aware test requirements, we should also focus on mutants standing longer in history as they represent "test investment" and be the most optimal set which does make test suite suffer from degradation and obsolete test requirements; moreover quite contrary, those mutants will be capable to continuously track test effectiveness between series of commits (releases).

7.4.3 Analysis Procedure and Implementation Details

To address **RQ1** and **RQ2**, we perform a similar statistical analysis used in the previous chapter Section 6.3.5. In this study, we also look for any correlation between the number of commit-relevant mutants and the size of commit hunks, and the type of mutants. In **RQ3** and **RQ4**, we simulate a mutation testing scenario where the tester starts by picking a mutant for analysis for which a test to kill it is developed. During this simulation, for each analyzed mutant, we randomly picked the test to kill it from the pool and computed which other mutants were collaterally killed by the same test. The process proceeds by picking a survived mutant until every mutant has been killed. We consider a mutant as equivalent if there is no test in the pool that kills it. This kind of simulation has been used in various related works to assess the effectiveness of mutation testing techniques [8, 44, 68, 148]. We consider four different mutant selection techniques when answering these questions. Two of them we use as *baselines*, where one consists of *randomly* selecting from the set of all mutants, and the other one consists of selecting only the mutants on the change. Another selection technique consists of selecting from the pool of commit-relevant mutants, while the last technique consists of selecting subsuming commit-relevant mutants. We aim to obtain the best-effort evaluation by maximizing effectiveness and minimizing the effort. We focus on the first 20 mutants picked by a tester to test commit changes while we measure effectiveness in terms of the *commit-relevant mutation score* reached by the selected mutants that guide the testing process. Simultaneously, we measure the computational effort in terms of the number of *test executions* required to accomplish the same effect over the different baselines (different mutants pools). In this simulation, we are interested in the test executions with the tests derived from the analysed mutants. The dependent variable is the test sets, while the independent variable is the test executions. We iterate the

process (killing all selected mutants) 100 times and compute the relevant mutation score and computation effort. To answer **RQ5** and **RQ6**, we reuse the evolution of program versions and the history of mutants to follow their standing (w.r.t. how long a mutant 'stands' in one location without being altered). For this study, we map mutants considering changed lines and the context of change from git diff tool [157]. Our scripts take two program versions and map code statements from version to version. Figure 7.1 indicates information of line numbers shift from version to version. *Note that for the purpose of this study, the history length of a mutant is computed from the first studied version till the last chronologically observed version.*

We start our analysis by extracting files with the most extended history of change from the open-source mature Apache Commons projects. Then, we use the state-of-the-art PIT mutation testing tool [58] to generate mutants per each changed (committed) file, followed by the execution of tests and generation of the killing matrix. Next to the killing matrix, for each changed file, we keep metadata (info. about hunks, timestamps, mutants bytecode index, location etc.) Using extracted information, we create a regression history for each file, making a file-specific historical timeline. In the timeline of each file, a time-point represents a commit that introduces changes to the file. While for each time-point, we calculate subsuming mutants (*reminder:* the set of mutants, when distinguished, distinguish all other mutants).

Besides the set of mutants, each time point contains information about mapping changes to the consecutive time point. Hence, the long-standing mutation metric is a function $F(M_t, change\_map) = M'_t$. Where M_t is a mutant from time t, and *change_map* is a map containing information about code transition from time $t \rightarrow t'$, while M'_t is the mutant at time t'.[1]

Table 7.1: Observed files through projects evolution

Observed Files	Time points	Mutants	Commons Project
CSVParser	31	8757	csv
CSVRecord	16	2656	csv
Lexer	21	10688	csv
CSVLexer	17	11208	csv
CSVFormat	47	52432	csv
CSVPrinter	21	20382	csv
IterableUtils	10	6441	collections
CharSequenceUtils	10	6802	lang
WordUtils	15	24128	text

7.4.4 Metrics and Measurements

To answer our research questions, we performed several statistical analyses to evaluate correlation among variables similar to Section 6.3.5. However, we also employed mutation-specific metrics such as *the commit-relevant mutation score* and *subsuming commit-relevant mutation score* to measure the effectiveness and efficiency of the selected mutants that guide the testing process. We measure how the test

[1]When mapping mutants, the occurrence of the same operators on the same line is theoretically possible with PIT mutants; however, we didn't witness such a scenario in our experiments. In case of the occurrence, it is possible to distinguish mutants based on bytecode instructions since, theoretically, two mutants with the same mutant operator cannot occur on the same bytecode instruction.

suite effectiveness progresses when we analyze mutants from the different mutant sets (e.g., all mutants, relevant mutants, subsuming relevant, etc.). Similarly, we measure efficiency by counting the number of test executions involved (to identify which mutants are killed by the test suites) when the test suite progresses.

7.4.5 Evaluation Data

For the purpose of the aforementioned empirical goals of this study, we continue to study and use the same dataset we previously defined, described and used for the purposes of the study in the previous Chapter 6.

Yet, in order to explore the longevity of mutants as one of the aims of this study, we observe the same projects and perform the mapping of mutants through different time points (commits). The subject data is described in Table 7.1. From 4 different Apache Commons projects, we extracted nine files with the longest history of change and their corresponding commits. It is important to emphasize that due to technical reasons (e.g., PiTest mutation testing tool requires green test suite to run), the time gap between commits is rather "longer" than one commit. Nevertheless, this didn't stop us from mapping and observing long-standing mutants through the history of evolving systems.

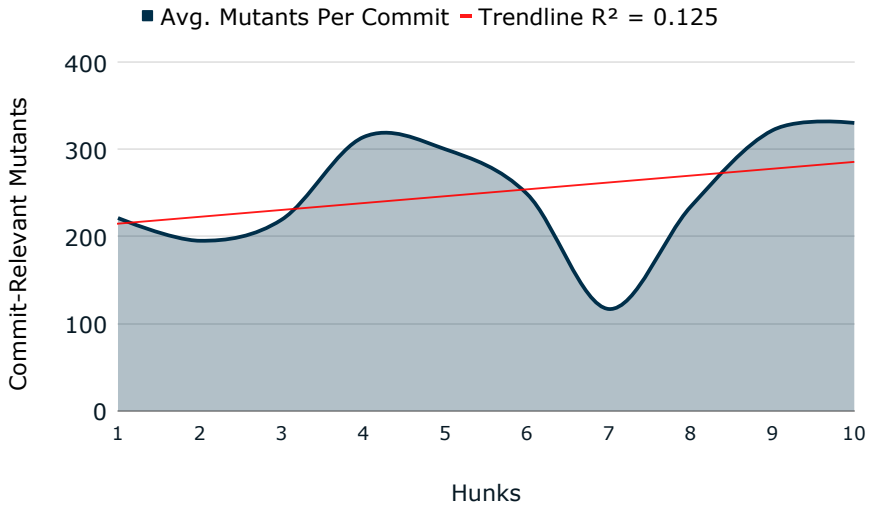


Figure 7.2: Average Number of Commit-relevant mutants per commit

7.5 Experimental Evaluation

7.5.1 RQ1: Commit Size

In this section, we investigate if there is a relationship between the number of (subsuming) commit-relevant mutants and the size of the commit, measured in terms of the number of commit hunks.

In particular, we pose the following question: *Is there a relationship between the number of commit hunks and the number of (subsuming) commit-relevant mutants?*

Figure 7.2 illustrates the relationship between the number of commit-relevant mutants and the number of commit hunks. For commit-relevant mutants, we found that the number of *commit-relevant mutants* (moderately) *increases* (trendline R^2=0.125) as the number of commit-hunks increases. This implies that there

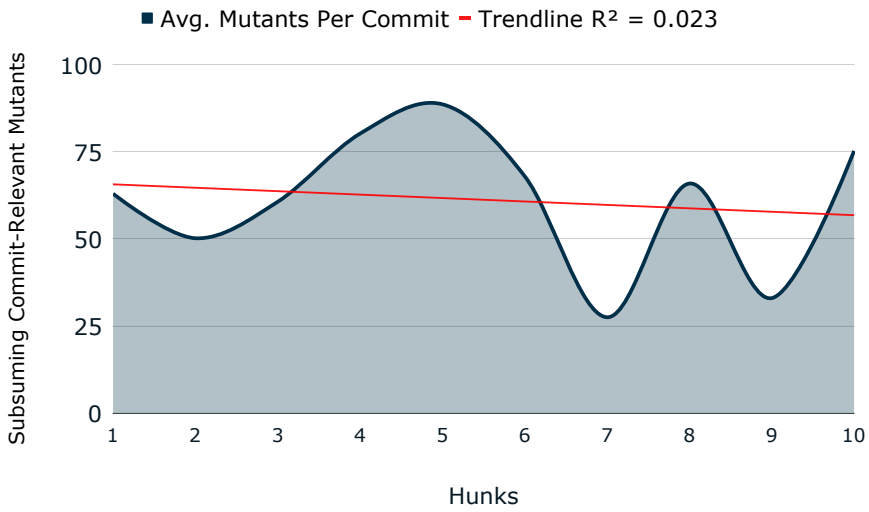


Figure 7.3: Average Number of Subsuming Commit-relevant mutants per commit

is *positive direct relationship* between the size of the commit and the number of *commit-relevant mutants*. However, Figure 7.3 shows that the number of *subsuming commit-relevant mutants* (moderately) *decreases* (trendline R^2=0.023) as the number of commit-hunks increases. These results suggest that there is an *indirect relationship* between the size of the commit and the number of *subsuming commit-relevant mutants*. The size of the commit does not directly predict the number of subsuming commit-relevant mutants. Indeed, the number of *subsuming commit-relevant mutants* decreases as the average size of the commit increases. Overall, this result demonstrates the effectiveness and importance of *subsuming commit-relevant mutants* in reducing testing effort, even for large commit changes.

The number of "commit-relevant mutants" increases as the size of the commit increases; however, the number of "subsuming commit-relevant mutants" decreases as the size of the commit increases.

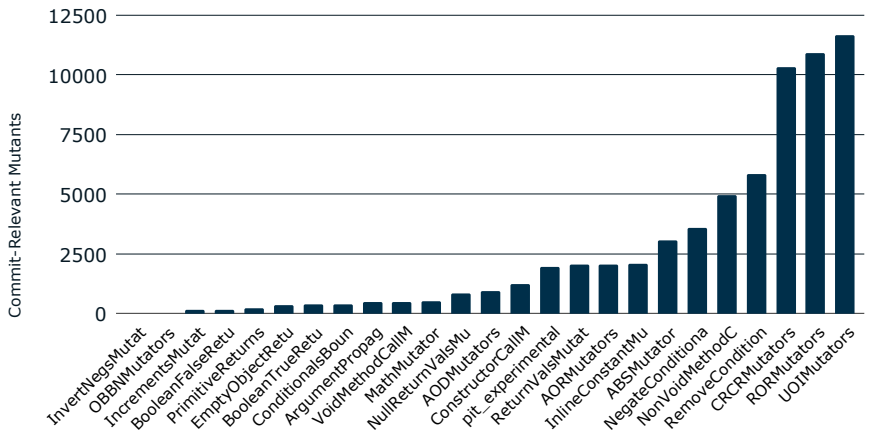


Figure 7.4: Prevalence of Commit-relevant Mutant Types

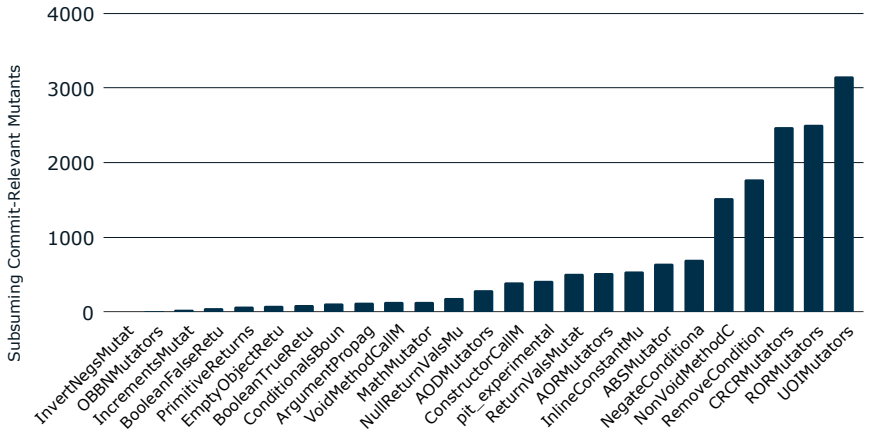


Figure 7.5: Prevalence of Subsuming Commit-relevant Mutant Types

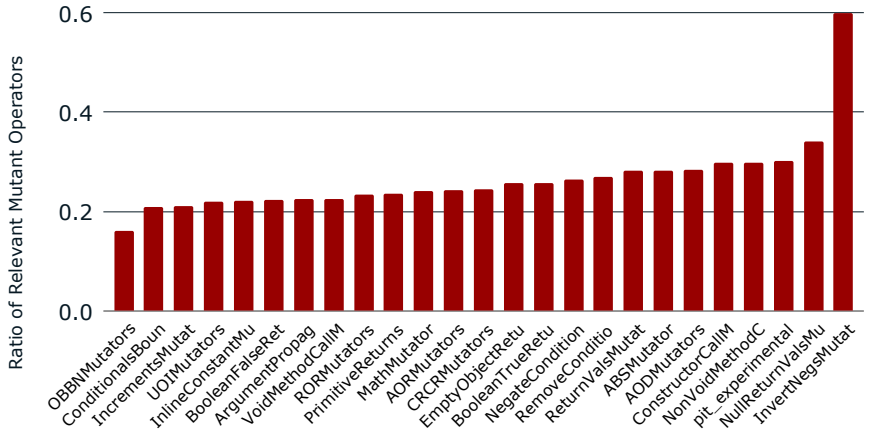


Figure 7.6: Ratio of Commit-relevant Mutants over All Mutants per Mutant Type

7.5.2 RQ2: Commit-relevant Mutant Types

Let us investigate the prevalence of *mutant types* among (subsuming) commit-relevant mutants, using 25 distinct mutant group types from PiTest [58]. This is important to determine whether the generation, selection or identification of commit-relevant mutants can be improved by focusing on specific mutant types.

What is the prevalence of mutant types among (subsuming) commit-relevant mutants? Figure 7.4 illustrates the prevalence of mutant types among commit-relevant mutants. Our evaluation results show that some mutant types are highly prevalent, such as Unary Operator Insertion Mutators (UOIMutators), Relational Operator Replacement Mutators (RORMutators) and Constant Replacement Mutator (CRCRMutators). On the one hand, UOIMutators inject a unary operator (increment or decrement) on a variable; this may affect the values of local variables, arrays, fields, and parameters [58], while RORMutators replace a relational operator with another one, e.g., "<" with ">" or "<=" with "<". On the other hand, CRCRMutators mutates inline constants. For further details about the mutant types, the table of constants and other mutation operators can be found in the official PiTest documentation[2]. Specifically, 50.77% of the commit-relevant mutants are of one of these three mutant types. This is mainly related to the fact these three mutation operators

[2]http://pitest.org/quickstart/mutators/

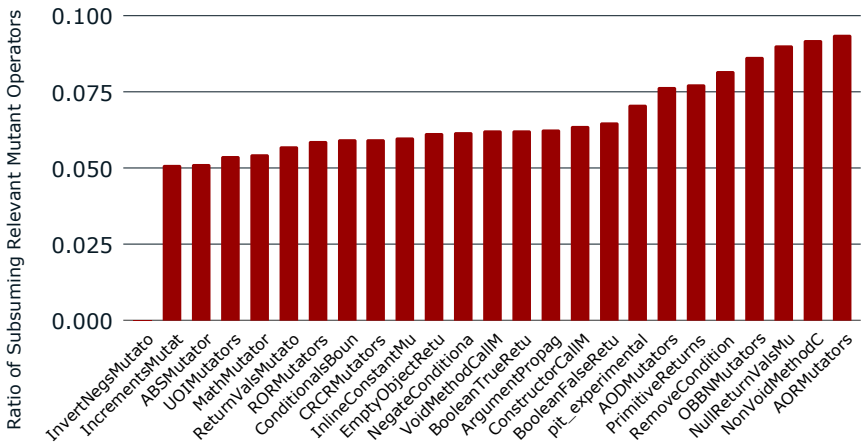


Figure 7.7: Ratio of Subsuming Commit-relevant Mutants over All Mutants per Mutant Type

produced the majority (54.5%) of the mutants considered in our study. Precisely, Figure 7.6 shows that the *distribution* of commit-relevant mutants is clearly *uniform* per mutant type. That is, in general, between 20% and 30% of the mutants for each type result to be commit-relevant. This indicates that mutants type does not increase or reduce the chances for mutants being commit-relevant. The outliers of Figure 7.6, corresponding to mutant types Bitwise Operator Mutator (*OBBNMutators*) and Invert Negatives Mutator (*InvertNegsMutat*), are because of the low number of mutants for these types: 13 out of 81 (16%) mutants are commit-relevant in the case of *OBBNMutators* mutant type, while 3 out of 5 (60%) mutants are commit-relevant for *InvertNegsMutat* mutant type. In particular, OBBNMutators mutates (i.e., reverses) bitwise "AND" (&) and "OR" (|) operators, while InvertNegsMutat operators inverts the negation of integers and floating-point numbers. Similarly, Figures 7.5 and 7.7 show that the ratio of subsuming commit-relevant mutants per mutant type follows a uniform distribution as well. Typically, between 5-7% of the mutants per mutant type turn to be subsuming commit-relevant. The outlier of Figure 7.7 corresponds to *InvertNegsMutat* mutant type, where none of the 3 commit-relevant mutants identified for this mutant type is subsuming (because of mutants of a different mutant type subsume them).

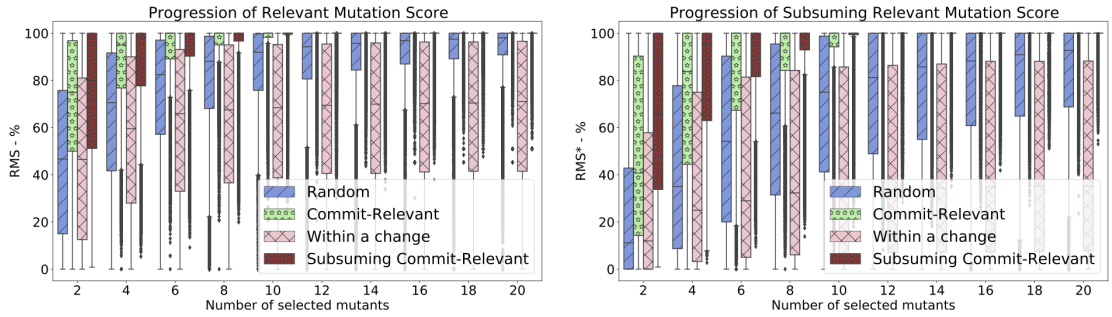
> The distribution of (subsuming) commit-relevant mutants per mutant type is uniform. Typically, between 20-30% (5-7%) of the mutants per mutant type are (subsuming) commit-relevant.

7.5.3 RQ3: Effectiveness of Commit-relevant Mutants Selection

This section simulates a mutation testing scenario where the tester selects a mutant for analysis for which a test to kill it is developed. Note that a test case that is designed to kill a mutant may collaterally kill other mutants. Consequently, opening a space to examine the effectiveness of the test suites developed when guided by different mutant selection strategies. Accordingly, this study compares the following mutant selection strategies: "random mutants selection," "mutants within a

Table 7.2: Comparative Effectiveness of selecting and killing (subsuming) commit-relevant mutants in comparison to "all mutants" and "mutants within a change" by observing RMS (Relevant Mutation Score) and RMS* (Subsuming Relevant Mutation Score)

	RMS										RMS*									
Selection Strategy/Interval	***2***	***4***	***6***	***8***	***10***	***12***	***14***	***16***	***18***	***20***	***2***	***4***	***6***	***8***	***10***	***12***	***14***	***16***	***18***	***20***
Random	46.67	70.59	82.42	88.10	91.95	95.26	95.74	96.85	97.50	98.05	11.11	35.00	54.17	66.13	75.00	81.25	85.71	88.24	90.89	92.76
Within a change	46.48	59.52	65.91	67.48	68.42	69.47	69.96	70.18	70.40	71.09	11.95	25.00	28.95	32.31	33.33	33.67	34.38	34.88	35.23	35.29
Commit-Relevant	75.00	95.05	100	100	100	100	100	100	100	100	40.74	83.72	100	100	100	100	100	100	100	100
Subsuming Commit-Relevant	80.00	98.51	100	100	100	100	100	100	100	100	65.75	95.35	100	100	100	100	100	100	100	100



(a) Relevant Mutants Progression (b) Subsuming Relevant Mutants Progression

Figure 7.8: Comparative Effectiveness of selecting and killing (subsuming) commit-relevant mutants in comparison to "random mutants" and "mutants within a change"

change," and (subsuming) commit-relevant mutants. We measure their effectiveness in terms of the *Relevant Mutation Score* (RMS) and *Minimal-Relevant Mutation Score* (RMS*), which intuitively measures the number of (subsuming) commit-relevant mutants killed by the different test suites. Specifically, we investigate the extent to which selecting and killing each aforementioned mutant type improves the test suite quality in terms of the number of (subsuming) commit-relevant mutants killed by the test suite.

Then we pose the question: *How many (subsuming) commit-relevant mutants are killed if a developer or test generator selects and kills random mutants or only mutants within a change?*

Table 7.2 and Figure 7.8 demonstrates how the effectiveness of the developed test suites progresses when we analyze up to 20 mutants from the different mutant pools. We observed that when the same number of mutants are selected from the different pools, better effectiveness is reached by test suites developed for killing (subsuming) commit-relevant mutants.

For instance, a test suite designed to kill six (6) selected (subsuming) commit-relevant mutants will achieve 100% of RMS and RMS*. However, a test suite designed to kill six randomly selected mutants will achieve 82.42% RMS and 54.17% RMS*, while a test suite that kills six mutants within a change will achieve 65.91% RMS and 28.95% RMS*, respectively. More precisely, even after selecting 20 mutants, neither random selection from all mutants nor within a change selection achieved 100% of RMS and RMS*.

This result demonstrates the significant advantage achieved by selecting (subsuming) commit-relevant mutants.

Moreover, we observed that random selection from all mutants is up to 1.6 times more effective than selecting mutants within a change. For instance, selecting 20 random mutants achieves 98.05% RMS and 92.76% RMS*, while selecting 20

mutants within a change only achieves 71.09% RMS and 35.29% RMS*. This result demonstrates the importance of selecting mutants *outside* developers' committed changes.

> Selecting and killing (subsuming) commit-relevant mutants led to more effective test suites. They significantly reduced the number of mutants requiring analysis compared to random mutant selection and selecting mutants within a change.

7.5.4 RQ4: Test Executions

In this section, we study the *efficiency* of the different mutant sets in terms of the number of *test executions* required to run the tests resulting from the analysis of 2-20 mutants. We, thus, approximate the computational demands involved when using all mutants, relevant mutants, (subsuming) relevant mutants and mutants located within commit changes.

Figure 7.9 illustrates the number of test executions required by the test suites derived by the analysis of 2-20 mutants. We found that the analysis of commit-relevant mutants significantly reduces the number of required test executions by 4.28 times on average over different intervals of analysed mutants (and 16 times when using subsuming commit-relevant mutants) in comparison to test execution required when analysing all mutants. For instance, users will need to perform 601 test executions when deriving tests based on the analysis of 2 mutants, from the set of all mutants, compared to 185 or 52 test executions needed by the use of commit-relevant mutants or subsuming commit-relevant mutants, respectively.

The difference increases with the number of analysed mutants. Thus, for the 2 analysed mutants, the difference in test execution is 2.8 times. For 4 mutants, 3.65, and 6 mutants, the difference in test execution is 4 times comparing all mutants and the commit-relevant mutants. We can also observe an increase in the difference between test executions needed by the use of subsuming commit relevant mutants and all mutants over different intervals. This difference is 11.55 times, 14.68 and 16 for analysed 2,4 and 6 mutants, respectively. Overall, we can compare test execution needed by using commit-relevant and subsuming commit-relevant mutants

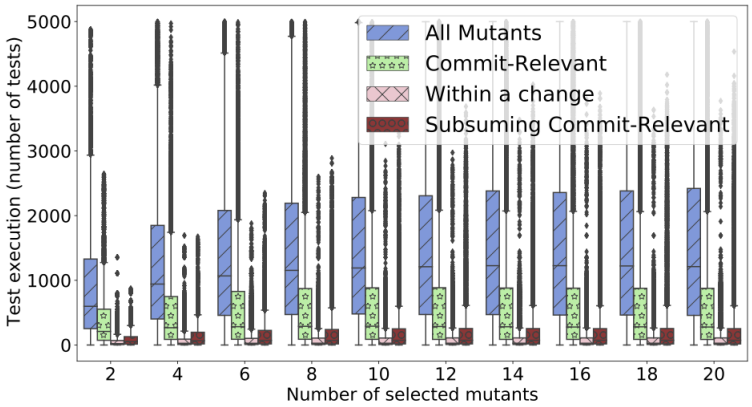


Figure 7.9: Efficiency, number of test executions required when deriving test suite sizes (in the range [2, 20]).

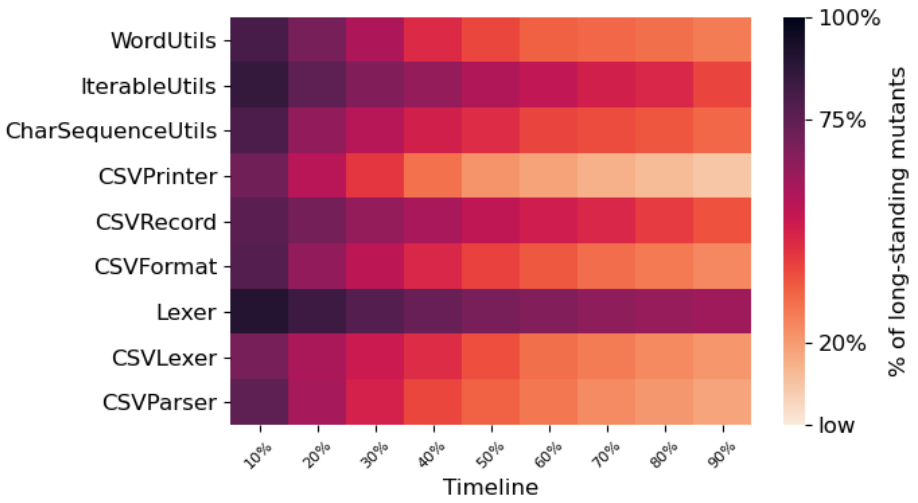


Figure 7.10: Mutant sets for different observed files and their corresponding studied history (timeline). Each cell in the heat map represents a normalised proportion of a subject history length, and the corresponding colour represents a percentage of mutants from the initial set over time through versions. The results show that mutants have diverse longevity, with brittleness, on average, of 52%.

and observe a 4 times difference on average, with no considerable differences between intervals.

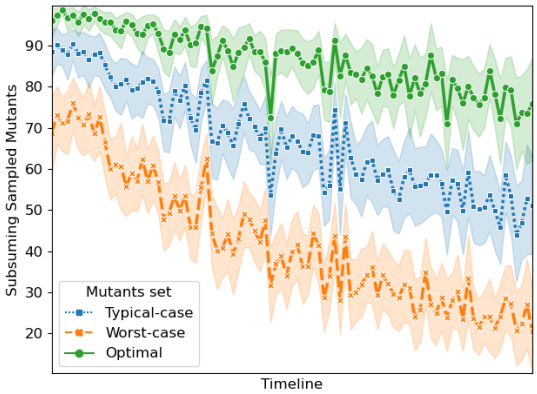
> Selecting subsuming commit-relevant mutants reduces test execution cost (i.e., the number of test executions) by up to 16 times compared to all mutants.

7.5.5 RQ5: Mutation Brittleness

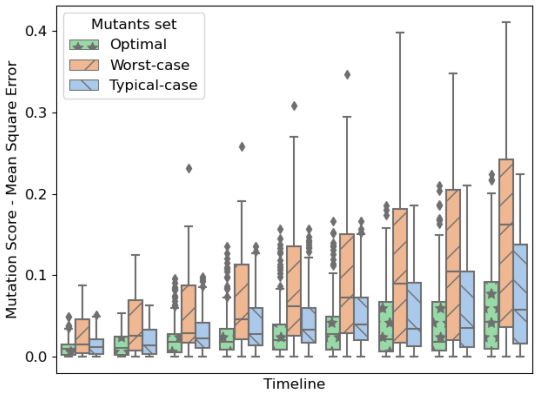
Figure 7.10 depicts the brittleness of mutants over time. In particular, it tells us how many mutants of a specific file exist through time, from their inception, on the same initial code elements, w.r.t., a code change has not altered mutants. From the figure, we can observe the diversity of the longevity distribution. Moreover, for the file Lexer, the ratio of long-standing mutants is significant, over 80% over observed time points. On the contrary, we can see that the CsvPrinter file contains significant changes, and the ratio of the mutants degrade below 50% after the first half of the observed points and below 20% in the second half of the observed history points. For other observed files, we see changes do not impact over 70% of the mutants in the first quartile of the timeline and between 40-60 % for the rest of the time points. These results demonstrate that mutants have a diverse lifetime over different evolution timelines, which suggests further investigation of whether mutants keep subsuming dynamic relationships over time and how mutant selection can affect test assessment.

7.5.6 RQ6: Long-Standing Subsuming Mutants

Figure 7.11 demonstrates to what extent subsuming mutants convey their dynamic behaviour and how mutation selection can affect the test assessment capability of mutation testing over time. In particular, the figure depicts the scenario in which we select subsuming mutants at a certain point in time and observe the capacity in



(a) Long-Standing Subsuming mutants for different observed files throughout their studied *history*. An optimal set of mutants - when selected - shows a long-standing prospect. In contrast, a worst-case set of mutants, when selected, shows brittleness - indicating obsolete test requirements. A 'special relationship' between long-standing and subsuming mutants exists and indicates that subsuming mutants in one version are probably subsuming in the following versions.



(b) Mean Square Error of Mutation Score of initially selected and long-standing subsuming mutants. The optimal set of long-standing subsuming mutants demonstrates a capability to perform test assessments over time - preserving subsumption relationships - unlike the worst-case or typical sets which show higher MSE. An optimal set of long-standing mutant suites enjoy an order of magnitude longer relevance.

Figure 7.11

which they exist over time together with how well they can perform test assessment w.r.t., measuring mutation score. We randomly sample 10-30% of mutants (100 times to remove the threat of randomness; we choose these selection intervals as obviously selecting all mutants leads to traditional mutation testing) from each observed file and consider the file history length - *the figures show aggregated results since each subject has different history length.* Figure 7.11a illustrates the need for intelligent mutant selection as we can significantly distinguish between two sets, a) sets of mutants more optimal as they stand longer throughout observed history; hence longer enjoying mutant suites relevance and b) other sub-optimal sets that suffer relevance degradation, w.r.t., represent obsolete test requirements. Interestingly, optimal mutant selection promises continuous tracking of test effectiveness as the margin of degradation is ≈10% on the ratio of selected mutants. In comparison, we observe worst-case sets of mutants, which indicate obsolete test requirements and typical arbitrary sets as if there was no other way to select, then we would end up with a random. To observe how capable those mutants are of affecting test assessment over time - keeping their subsumption relationships - we calculate the mean square error (MSE) of mutation score (MS) between the initially selected set and those long-standing mutant sets. In Figure 7.11b we assess the difference in the MSE of MS between the optimal and suboptimal sets of long-standing mutants. We observe that the optimal set of long-standing subsuming mutants keeps MS high over time (low MSE ≈ 0.01%-0.04%), indicating a gradual loss in MS as the mutants stand longer in time, thus preserving mutant suite relevance. Accordingly, it is important to realize the potential in conveying knowledge of previously calculated dynamic relationships of mutants for at least 10x longer than a random selection. In particular, by selecting the sub-optimal sets of mutants, the threat of not preserving the knowledge appears, w.r.t., mutants less capable of test assessment over time, suggesting their low priority (higher MSE ≈0.01%-0.20%).

7.6 Discussion and Future Plans

7.6.1 Summary of Findings for Commit-Relevant Mutants

Commit-relevant mutation testing allows developers to identify and select the mutants necessary for testing the program changes to avoid regression bugs and newly introduced failures. This chapter presents an empirical study that examines the prevalence and characteristics of commit-relevant mutants and provides scientific insights concerning the mutation testing of evolving software systems. The main empirical findings include the following:

1. Predicting (subsuming) commit-relevant mutants is not a trivial task. In our evaluation, we studied several candidates *proxy variables* that *do not reliably predict* commit-relevant mutants, including the number of mutants within a change, mutant type, and commit size. Hence, we encourage the development of statistical or machine learning approaches and program analysis techniques to predict or identify commit-relevant mutants automatically.

2. Selecting commit relevant mutants is *significantly more effective and efficient than random mutant selection and the analysis of only mutants within the program change.* Commit-relevant mutation testing can reduce testing effort (i.e., number of test executions) by up to 16 times, and by half, compared to random mutant selection and mutants within a change, respectively.

We also observe that the effective selection of commit-relevant mutants significantly reduces the number of mutants requiring analysis. Thus, we encourage researchers to investigate automated methods for identifying and selecting commit-relevant mutants, for instance, using statistical analysis or program analysis.

In addition, we observed that commit-relevant mutant prediction and selection is a challenging task. For example, many proxy variables could not reliably predict commit-relevant mutants in our analysis (2). To buttress this, we further conducted a correlation analysis of the features of commit-relevant and non-relevant mutants using control and data flow features selected from Chekam *et al.* [112]. The goal is to determine if mutants' features previously used for other prediction tasks, for instance, for selecting fault-revealing mutants [112], can also distinguish commit-relevant mutants. Figure 7.12 presents our findings using a heat map, where each map coordinate represents the Spearman correlation coefficient calculated between two features on the coordinates. These features characterize relevant and not relevant mutants, labelled with the suffix "R" or "N", respectively. Notably, we observe that there are no strong positive or negative correlations among these features. This implies that these features can not directly help distinguish between commit-relevant and non-relevant mutants. However, we can observe two cases of a medium positive correlation between the same class features, in particular, *CfgDepth* and *NumInDataD* between both classes show correlation.[3] This phenomenon is expected since there will be more data-dependent expressions as the depth of a mutant in the control flow graph increases.

[3] *CfgDepth* means the depth of a mutant in the control flow graph, i.e., the number of basic blocks to follow to reach the mutant, and *NumOutDataD* refers to the number of mutants on expressions on which a mutant m is data-dependent.

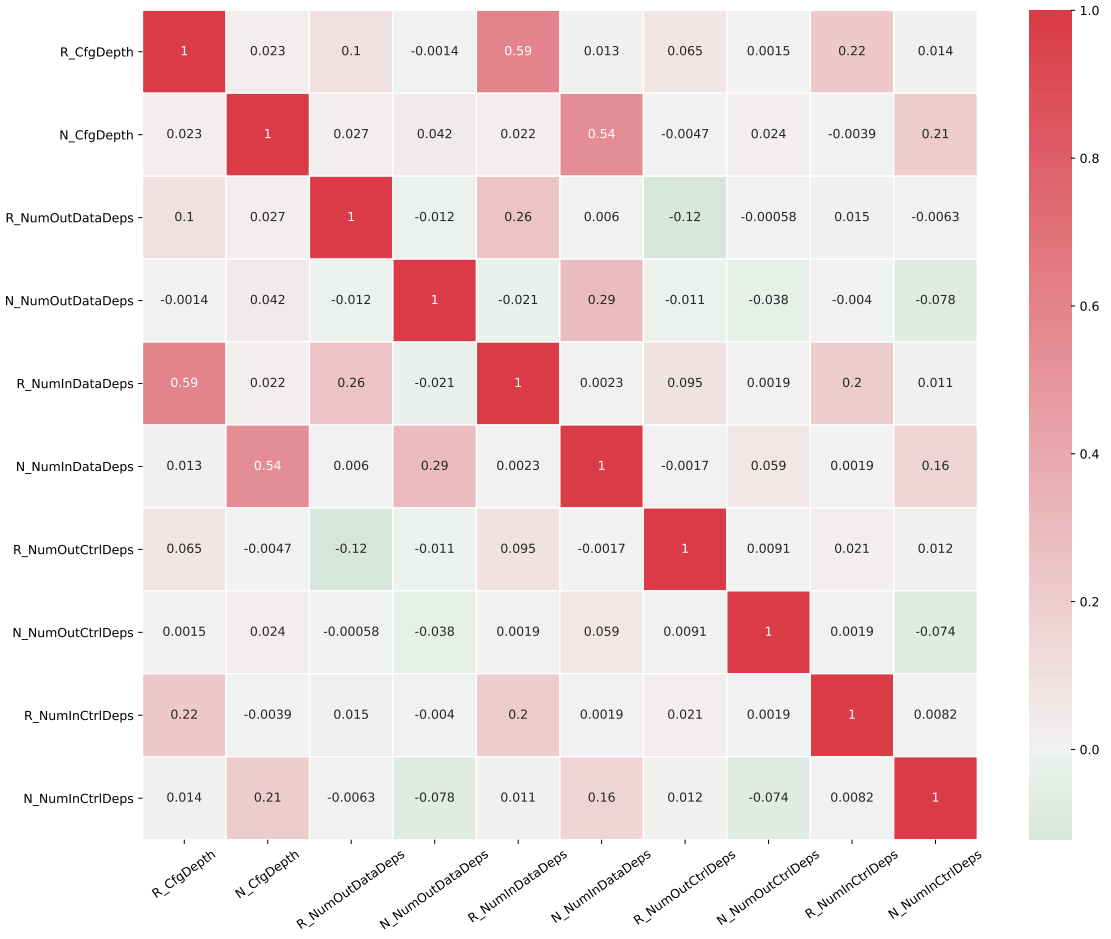


Figure 7.12: Correlation between features of relevant and non-relevant mutants labelled with suffixes “R" and “N", respectively. The features examined include the following: `CfgDepth` - *Depth of a mutant in Control Flow Graph, i.e., the number of basic blocks to follow in order to reach the mutant*; `NumOutDataD` - *Number of mutants on expressions data-dependent on a mutant expression*; `NumInDataD` - *Number of mutants on expressions on which a mutant m is data-dependent*; `NumOutCtrlD` - *Number of mutants on expressions control-dependent on a mutant*; and `NumInCtrlD` - *Number of mutants on expressions on which m is control-dependent.*

Furthermore, we found that commit-relevant mutant selection considerably improves the effectiveness and efficiency of testing evolving systems, especially in comparison to the random mutant selection and using the mutants within the program changes (**RQ3** and **RQ4**). Overall, these empirical findings shed more light on the challenge of mutation testing of evolving systems and provide directions for future research into the selection and prediction of commit-relevant mutants.

7.6.2 Implications, Guidelines and Use-Cases

The main insight of our study is *the need to pay attention to the effective identification, selection or prioritization of commit-relevant mutants.* This is particularly important to reduce the developers' effort required for mutation-based regression testing in continuous integration systems. Random mutant selection or selecting mutants within the change is not effective for identifying commit-relevant mutants since,

as shown in the previous chapters (see chapter 5 and 6), only one in three mutants are commit-relevant, and only one in five commit-relevant mutants are located within the commit (see RQ1). Notably, an effective commit-aware mutant selection method can significantly reduce the number of mutants involved. In particular, the effective selection of commit-relevant mutants significantly reduced the number of mutants requiring analysis and the number of test executions compared to random mutant selection and selecting mutants within a change (see RQ3 and RQ4, respectively).

To achieve the aforementioned goals, i.e., automate the identification and selection of commit-relevant mutants to aid developers, we turn to the research community to develop and investigate the techniques required for effective commit-aware mutation testing. We note that neither the size of the commit nor the type of the mutant reliably predicts (subsuming) commit-aware mutants. Even though the number of commit-relevant mutants increases as the size of the commit increases, the number of subsuming commit-relevant mutants decreases as the size of the commit increases (see RQ1). Additionally, we observed that the distribution of (subsuming) commit-relevant mutants per mutant type is uniform (see RQ2). Thus, the takeaway of this study is the need to develop: a) novel techniques for *selecting, prioritizing and predicting commit-relevant mutants*; and b) *commit-aware test metrics* to determine the adequacy of commit-aware mutation testing. Although the problem of selecting/identifying relevant mutants is active for traditional mutation testing, this is hardly well-studied for commit-aware mutation testing. This is an important problem since several studies [113, 148] (including studies in this dissertation) have demonstrated that traditional (random) mutation testing is significantly costly for evolving software.

The previous chapter has further illustrated that dynamic approaches (like observation slicing) can complement static or machine learning-based approaches in effectively identifying commit-relevant mutants. We have also observed that commit-relevant mutants cannot be predicted using only the committed changes or program dependence properties. This implies that the current state-of-the-art is not generally applicable for commit-aware mutation testing in practice. Thus, for more effective approaches, we believe researchers need to consolidate the knowledge from several sources, including the commit difference, mutant properties, the semantic behaviour of mutants, and the semantic divergence produced by the change.

To this end, we encourage further investigation of the effectiveness of such techniques for commit-aware mutation testing and the development of newer program analysis-based approaches (e.g., symbolic execution or search-based techniques) for identifying commit-aware mutants.

Finally, previous research [113] has shown that commit-aware mutation testing requires different test metrics from traditional mutation testing. Thus, we encourage researchers to define new test metrics targeting the changes and their dependencies and investigate their effectiveness for commit-aware mutation testing. Overall, we expect that addressing these challenges will reduce the performance gap between the state-of-the-art in traditional mutation testing and commit-aware mutation testing.

7.6.3 Implications of Long-Standing Mutants

We believe that long-standing mutants are an interesting category in their own right and worthy of further research. They have implications not only for mutation testing but also beyond mutation testing. In this section, we set out future plans for further evaluation and investigation of the properties of long-standing mutants and

their applications.

Implications regarding subsuming long-standing mutants: Despite showing that subsuming relationships can be preserved from version to version and that mutants' utility can be reused, we do not yet fully understand *why* subsuming mutants tend to last longer than subsumed mutants. A detailed study is needed to understand the subsumption and longevity drivers fully.

Implications for mutation testing tools: Our results also have implications for the development of future mutation testing tools. In particular, our results suggest the development of a robust mutant versioning system. Existing tools [58, 134, 158] focus on the generation of mutants but not sophisticated mutant versioning. In future work, we need to investigate mutation testing tools that allow logging mutants' maturity, execution history, and fluctuation over time, supporting approaches that learn mutant behaviour and relating this to code changes. Previous work on flaky mutant detection [159], predictive modelling [160] and hyper-heuristics [161] (in particular that focused on mutation testing [86]) may form a good starting point for this research agenda.

Maximising long-standing mutant fault revelation: By focusing on long-standing mutants, we favour mutants that reside in relatively unchanging parts of the code. There is a natural concern that this may, in turn, lead to us favouring test suites that do not tend to reveal faults in changing parts of the code. Fortunately, the fact that a mutant lies in code region A does not render it insensitive to bugs that lie in (lexically separate) code region B. If there are transitive *dependencies* between A on B then we can expect high degrees of mutant coupling and even subsumption between the two regions. This suggests future work on identifying mutants that have high 'transitive dependence reach' through their transitive dependencies, using techniques such as slicing [94] and chopping [162].

Implications of long-standing mutants beyond mutation testing research: The findings reported in this paper have implications beyond mutation testing to automated program repair [163, 164] and genetic improvement [165, 166]. It is often been argued that program repair is the inverse of mutation testing. Instead of inserting faults, repair seeks to remove them. Long-standing mutants are, therefore, also likely to find applications and implications in the field of program repair and genetic improvement research. For example, it would be interesting to explore 'long-standing' repairs as a counterpoint to long-standing mutants. One might reasonably conjecture that such repairs would remain relevant for longer than repairs in areas of code subject to high degrees of churn. However, the empirical assessment of this phenomenon remains an open problem for future work.

7.7 Conclussion

We studied the prevalence, location, effectiveness, and efficiency of commit-relevant mutants. We have also examined the comparative advantage of commit-relevant mutants compared to two baseline methods, i.e., random mutant selection and selecting mutants within program changes. In addition, we observed that the effective selection of commit-relevant mutants affords a significant testing advantage. Specifically, it has the potential to significantly reduce the cost of mutation, and it is significantly more effective and efficient than random mutant selection and analysis of only mutants within the program change. We also investigate the predictability of commit-relevant mutants by considering typical proxy variables (such as the number

of mutants within a change, mutant type, and commit size) that may correlate with commit-relevant mutants. However, our empirical findings show that these candidate proxy features do not reliably predict commit-relevant mutants, indicating that more research is required to develop tools that successfully detect this kind of mutants. For replication, scrutiny and future research, we publicly provide our artefacts, data and experimental results: **https://mutationtesting-user.github.io/evolve-mutation.github.io/**. Furthermore, in this chapter, we introduce a mutant brittleness measure and use it to audit software systems and their mutation suites. We also demonstrate how consistent-by-construction long-standing mutant suites can be identified with a 10x improvement in mutant relevance over an arbitrary test suite. Our results indicate that the research community should consider avoiding the re-computation of mutant suites and focus, instead, on long-standing mutants, thereby improving the consistency and relevance of mutation testing.

8

Learning-Based Mutant Selections for Comparison of Mutants Effectiveness

To this end, various fundamentally different mutation approaches have emerged, opening the question of whether the standard grammar-based mutation testing approach is still the most effective in detecting real faults. Plus, with the advances in learning-based mutant selection strategies, it is pertinent to recognise the importance of asking how effective are the mutants of different approaches when using those strategies to remove the noise caused by the trivial mutants, focusing only on the ones of the highest quality. Overall, this chapter analyses the mutant effectiveness of the fundamentally different mutation testing approaches - directed by distinct learning-based selection strategies. Our results lead to different conclusions on fault detection, raising attention to the risk that the suitability of different kinds of mutants can be misinterpreted if not removing the noise of trivial mutants. In particular, the results demonstrate that under learning-based selection strategies, different mutation approaches significantly improved their performance and reveal that when comparing mutation effectiveness, it is imperative to account for a mutant selection suitable for removing the noise, which leads to the drawing of incorrect conclusions.

This chapter is based on the work published in the following research paper:

- Milos Ojdanic, Ahmed Khanfir, Aayush Garg, Renzo Degiovanni, Mike Papadakis and Yves Le Traon, *"On Comparing Mutation Testing Tools through Learning-based Mutant Selection"*, 2023, *The 4th ACM/IEEE International Conference on Automation of Software Test (AST 2023), In press.*

Contents

8.1 Introduction

Mutation testing is considered one of the most powerful testing techniques [1]. It operates by posing the requirement to write tests to reveal artificially injected faults, hence also revealing the real faults coupled with artificial ones [8, 9].

Motivated by its fault-revealing capability and application in various domains [8] such as testing [167], debugging [18,168], maintenance and change-aware dependability analysis [169–171], researchers and practitioners have proposed several mutation testing approaches to automate fault injection and test suite assessment. Most approaches inject faults based on predefined syntactic transformation rules (aka mutation operators) [1, 58], such as replacing an instance of a relational operator with another operator, e.g., replacing > with >=. Other approaches aim at injecting faults by either following fault patterns created or learned from recurrent fault instances [54, 61, 77] or by employing code pre-trained language models [60]. These approaches have been implemented and made openly available as tools, serving the main purposes of mutation testing – tests assessment and guidance criterion.

Interestingly, while several novel mutation testing approaches and their corresponding tools have recently emerged, their fault revelation potential has not been assessed and compared with the traditional grammar-based mutation testing tools. Since these tools rely on fundamentally different underlying techniques such as manually defined patterns [54], deep learning [61], grammar-based rules [58], and code pre-trained language models [60], it is particularly interesting to check for potential complementarities along with their strengths. Previous work [57, 139] have limited their studies to only grammar-based mutation testing tools [58, 172]. Hence, in this chapter, we venture to investigate the effectiveness of the most recent mutation testing tools and contrast their performance with the traditional ones under a new and larger dataset by employing Defect4J *v2.0*.

8.1.1 Rationale behind the comparison

Powerful learning-based mutant selection strategies have been proposed recently [63, 111, 112], intending to reduce the application cost and noise of mutation testing, which has been for long considered as a primary cause that keeps the technique away from broad industrial service.

These strategies aim at discarding redundant mutants and providing testers with mutants that will bring value to the testing process. Typically, these strategies employ user-defined features - code features learned using deep learning - independent of how mutants are introduced. Since these mutant selection strategies give different importance to mutants, this may affect the cost-effectiveness of mutation testing and raises the question of how the different tools compare in terms of fault detection under mutant selection strategy guidance.

To this end, we model the application cost they entail and perform a controlled cost-effectiveness comparison under two different cost models, which reflect the main efforts spent in mutation testing campaigns. These cost models are repeatedly used for work simulation and encompass the number of analysed mutants and the number of written tests required to reveal the injected faults [8, 111].

Precisely, we study the fault detection ability and the related cost-effectiveness of four fundamentally different mutation testing tools that we deemed as representatives of different approaches when guided with and without a mutant selection strategy.

In particular, we consider 1) IBIR [54] – a mutation testing tool that represents manually crafted fault patterns – 2) DeepMutation [61] – a deep learning-based tool that derives patterns from real bug fixes – 3) μBERT [60] – a mutation testing tool that uses a pre-trained NL-PL language model to replace tokens based on large code corpus learning – and 4) two sets of operators from PIT [58] – a popular grammar based mutation testing tool. As a learning-based selection strategy, we use *Cerebro* [63], a deep-learning-based mutant selection technique that has been proven efficient in reducing mutation testing campaign costs.

8.1.2 Contributions

In this study, we hypothesise that different mutation testing approaches - directed by learning-based selection strategies - lead to different conclusions on the fault revelation, raising a risk that their suitability can be misinterpreted. Our results show that IBIR reveals most of the Defects4J faults (approximately 90% of the considered real faults), followed by μBERT (approximately 74%) and PIT (approximately 73%). However, IBIR and PIT introduce significantly more mutants than μBERT; approximately, IBIR produces twice as many mutants as PIT, which produces 3.2 times as many as μBERT. This seems to introduce a size effect on the number of mutants, which influences fault detection. To account for this, we also control the number of mutants (or tests) and perform a cost-effectiveness comparison.

Thus, when cost-effectively comparing the tools, we find that except for DeepMutation, which is the least effective, all tools have similar fault-revealing abilities when controlling cost/effort and applying them out of the box – without any guidance. Perhaps surprisingly, when we combine them with mutant selection strategy, we see a much different picture with μBERT performing significantly better, approximately 12%, than the other tools. Additionally, we find that the other tools subsume DeepMutation by being able to identify more faults and doing it at a much lower cost.

Overall, our work aims to study the fault detection performance of different testing approaches when employing mutant selection strategies. Our key contributions can be summarized by the following points:

1. We perform the first study investigating the fault detection capability of fundamentally different fault seeding approaches (IBIR, DeepMutation, PIT, μBERT) in a newly released bug dataset, i.e., Defect4J *v2.0*.

2. We propose a new way to compare the mutation testing tools using learning-based strategy and show that leads to different conclusions on which is the most cost-effective tool than the ones that could be drawn when not considering it. We investigate the use of transformers, a state-of-the-art deep learning technique *Cerebro*.

3. We show that combining μBERT with learning-based mutant selection yields significantly higher fault detection, approximately 12% higher, than any other tool.

8.2 Mutation Tools and Selection Strategies

8.2.1 Mutation Testing Tools

PIT [58] is one of the state-of-the-art mutation testing tools that seeds faults using syntactic transformation rules (aka mutant operators) at the bytecode level. We selected PIT as a representative of tools for grammar-based transformation since it is considered a state of art tool with a vast community providing continuous support. Besides, recently we have witnessed many empirical proofs and studies distinguishing the tool of its competitors [139]. The tool implements 29 task-specific categories of mutation operators; for instance, the Conditionals Boundary category mutates relational expressions. When considering the 29 categories, PIT has over 120 mutation operators. However, PIT also provides different pre-defined configurations. Thus in this study, we consider two. We will denote by PIT to the setting in which all mutation operators from the 29 categories are considered and by *PIT_Default* to the set of mutants contained in the default configuration of the tool - consisting of 11 categories. The tool default configuration is often used in industry settings, while many existing studies employed all mutants for experimental purposes. We decided to take both configurations for our study to dismiss the threat of biasing the tool. We provide a code snippet demonstrating the mutation induced in the following box.

```
//PIT uses grammar transformations - e.g., relation > to <=
  public boolean contains(final Object object) {
        return indexOf(object) <= 0;
    }
```

μBERT [60] is a mutation testing tool that uses a pre-trained language model (CodeBERT) [117] to generate mutants by masking and replacing tokens. μBERT takes a Java class and extracts tokenized expressions, which mask for token replacement (mutation), e.g., it masks a variable name and invokes CodeBERT to complete the masked sequence (i.e., to predict the missing token). This approach has been proven efficient in increasing the fault detection of test suites [60] and improving the accuracy of learning-based bug-detectors [74]; therefore, we consider it as a representative of pre-trained language-model-based techniques. For instance, please consider the code snippet provided, in sequence `return indexOf(object) > 0;` μBERT mutates the method invocation expression `indexOf` by feeding CodeBERT with the masked sequence `return <mask>(object) > 0;`. CodeBERT predicts the 5 most likely tokens to replace the masked one, e.g., it predicts `contains`, `indexOf`, `lastIndexOf`, `count`, and `size` for the given masked sequence. μBERT takes these predictions and generates mutants by replacing the masked token with the predicted ones (per masked token creates five mutants). μBERT discards non-compilable mutants and those syntactically the same as the original program (cases in which CodeBERT predicts the original masked token).

```
//mBERT uses CodeBERT to alter tokens based on the context
  public boolean contains(final Object object) {
        return lastIndexOf(object) > 0;
    }
```

IBIR [54] is a fault seeding tool that uses automatic program repair inverted fix-patterns to inject faults that are similar to real ones. It takes as input the git repository of the program to mutate and a bug report, written in natural language and seeds (introduces) multiple fault candidates (mutants) that emulate the fault described in the bug report. In particular, IBIR's mutation operators are inverted

fix-patterns crafted from actual bug fixes, and their inverse would induce seeded faults that are similar to actual faults. IBIR, in one of its configurations, applies faulty patterns exhaustively over the system-under-test to generate mutants without any bug-report IRFL guidance. We use this configuration in our study to exclude the advantage brought by the IRFL component to IBIR's performance and, thus, make a fair comparison between the considered approaches mutations. For instance, if you consider the code snippet provided, in sequence `return indexOf(object) > 0;` IBIR can mutate the condition by expanding the expression with an extra one `&& object == null`.

```
//IBIR uses inverted fix-patterns
  public boolean contains(final Object object) {
        return indexOf(object) > 0 && object == null;
    }
```

DeepMutation [61] generates mutants by employing Neural Machine Translation [51], aka NMT. It uses an NMT model trained on a large corpus (∼787k) of existing bug-fixing commits mined from GitHub repositories. It takes a Java method as input and outputs a mutant. Hence, it generates one mutant for every method in a Java class file. In particular, every method is abstracted, in which pre-defined identifiers replace the user-defined variable names and literals to obtain an abstracted code representation. These abstracted code representations are then given as input into the trained NMT model to produce abstracted mutants, which are converted back to source-code mutants by reversing the abstraction.

We use the publicly available trained model of DeepMutation [65] to generate the mutants and `src2abs` [66] tool to perform the abstraction process. This approach is one of its kind until this moment, and we followed its guidelines [61] to generate one mutant per method.

```
//DeepMutation uses Machine Translation for bug-fixing
  public int contains(final Object object) {
        return indexOf(object);
    }
```

8.2.2 Mutant Selection Strategies

The fault seeding techniques generate a very different number of mutants. Thus, to make a fair comparison, we aim to control the number of mutants in answering RQs 1 and 2. Since the order in which mutants are analyzed is relevant due to the existence of equivalent and trivial mutants and can affect the application cost of mutation testing, it can also alter the cost-effectiveness of the tools/techniques used. Thus, we consider two mutant selection strategies that are very different from each other.

Standard Mutant Selection consists of sampling uniformly from the entire set of mutants [57, 68], i.e., every mutant has the same probability of being selected since no prioritization heuristic is considered.

Cerebro [63] is a machine learning approach that has been shown effective in statically selecting *subsuming* mutants. *Subsuming* mutants - a minimal subset of mutants to identify such as to identify the original set reciprocally [29, 42] - are the set of mutants that resides on the top of the subsumption hierarchy and subsume all other mutants [43]. *Cerebro* learns to identify subsuming mutants given their context. In particular, it learns the associations between mutants and their surrounding code by using language-agnostic *Neural Machine Translation* [173], which is also used by

many recent studies [48, 51, 62, 64]. *Cerebro*'s learning scope is a relatively small area around the mutation point that differentiates locally the mutants that are subsuming from those that are not. This procedure allows the selection of the mutants from program elements which fit best to their context rather than using entire codebases with every possible transformation. *Cerebro* demonstrated preserving the mutation testing benefits while limiting application cost, i.e., reducing all cost application factors such as equivalent mutants, mutant executions, and the mutants that require analysis. *Cerebro* outperformed other approaches that concern machine learning models that capture code properties through manually engineered code features [63].

8.3 Experimental Design And Setup

8.3.1 Research Questions

This study aims to compare the cost-effectiveness of the recently proposed mutation testing tools. To do so, we start our analysis by investigating the fault detection ability of the studied tools in a scenario when a developer writes a test that distinguishes a mutant and identifies coupled fault. Thus we ask:

RQ1 *(Tool's effectiveness) What is the fault detection ability of IBIR, DeepMutation, PIT and μBERT mutation testing tools? How do the employed techniques compare in terms of cost-effectiveness?*

The answer to this question allows us to identify the most effective and cost-effective tools in standard comparison settings with random mutant selection, which have merits in deciding on their use and shedding light on their strengths.

Intelligent mutant selection strategies have recently been proposed to prioritize mutants and reduce the mutation testing effort. Hence, our other objective is to investigate whether the cost-efficiency of fault-seeding approaches would take advantage similarly by these strategies. We consider an advanced learning-based mutant selection *Cerebro* aiming to select subsuming mutants utilizing Neural Machine Translation proficiency. We, therefore, investigate the following research question:

RQ2: *(Learning-Based Selection) What is the cost-effectiveness of mutation testing tools when mutants are selected according to a learning-based mutant selection strategy?*

Answering these questions provides evidence of whether and how much the mutation testing tools/approaches benefit from the intelligent mutant selection and whether there is one that benefits more than the others.

Taken all together, by answering the above questions, we study the fault detection ability of fundamentally different mutation testing tools and estimate their cost-effectiveness when guided by mutant selection strategy.

8.3.2 Benchmarks and Ground Truth

We use Defects4J [116] v2.0.0, which contains the build infrastructure to reproduce (over 800) real faults for Java programs. Every bug in the dataset consists of the faulty and fixed versions of the code and a developer's test suite accompanying the project that includes at least one fault-triggering test that fails in the faulty version and passes in the fixed one.

The set of faults spans more than a decade of development history, making it challenging for us to synchronize the execution of faults over different fault-seeding tools, following obsolete dependencies not supported by relatively recent tools and old versions of frameworks or languages, i.e., some of the mutation tools require Java 1.8+. Thus, intending to be as fair as possible with the selected tools, we had not considered those faults that did not satisfy the building requirements — specifically, the 26 faults from the project Jfreechart and 174 from Closure-compiler. Additionally, when conducting this study, we found that 82 faults from the Jsoup project were not compilable due to technical reasons [122]. In total, we analyzed 509 faults from 15 different projects.

It is pertinent to note that when comparing and observing performance between different tools, we strictly use the intersection of faults, i.e., the faults we were able to study for all tools in question, and strictly those faults where every tool generated at least one killable mutant.

8.3.3 Generated Mutants

For each selected faulty project version from Defects4J, we start by identifying the modified classes between the faulty and fixed versions. Then we generate mutants for the fixed version of each modified class by employing the selected mutation testing tools. Table 8.1 records the number of faults analysed and the number of mutants generated by each mutation testing tool. DeepMutation delivers only one mutant per method and produced 5,559 mutants for the 348 analysed faults. μBERT was applied on 499 faults and produced 293,304 mutants. IBIR produced 1,113,113 mutants for the 393 analysed faults. As we previously introduced, we consider two configurations in the case of the PIT mutation testing tool. PIT_Default uses the subset of the mutation operators as specified in the tool's production-ready setup. These categories are considered the most effective ones (11 out of 29) and generate 110,480 mutants for 508 faults analysed. For the sake of thoroughness of the study, as we already mentioned, we also use *all* available mutation operators of the tool, denoted by PIT, and generate 1,212,544 mutants across 29 mutants categories for the 509 faults analysed.

Table 8.1: Number of Faults and mutants used in the study.

Mutation Testing Tool	**# of Analysed Faults**	**# of Mutants**
DeepMutation	348	5,559
PIT_Default	508	110,480
μBERT	499	293,304
IBIR	393	1,113,113
PIT	509	1,212,544

** When comparing different tools, we strictly use the intersection of faults*

8.3.4 Experimental Analysis Procedure

We start by executing all the mutants generated by the different tools on the selected project subjects and recording the failing tests distinguishing those mutations. Next, we use *Cerebro*, the machine learning approach, to obtain the (subsuming) probability associated with each mutant needed for answering RQ2.

The procedure to answer RQ1 studies the cost-effectiveness of the fault seeding techniques when employing standard (random) mutant selection as the strategy of selecting mutants in a developer work simulation. We repeat the procedure for RQ2; however, this time, *Cerebro* guides the selection of mutants by prioritising mutants and assigning the highest probability of being useful to those likely to subsume others, considering their surrounding code context.

In particular, the standard developer workflow simulation emulates a testing scenario where the mutants guide the testing process and serve as test requirements. A tester selects mutants and designs tests to kill them until every (killable) mutant is killed (a standard simulation often reported in the literature [8]). Intuitively, the work simulation starts with an initial empty test set and the set of mutants to be covered. The next step is to select a mutant with high priority given by some strategy and either, select randomly a test (without replacement) that kills it, or judge it as equivalent. Each selected test is added to the test suite, and every mutant killed by that same test is discarded. The simulation is repeated until all mutants are treated.

Precisely, given a list `M` of mutants sorted by a particular mutant selection strategy (i.e., Standard or *Cerebro*) and their predefined test pool `P` (provided within the dataset Defects4J), we incrementally construct and measure the number of tests in test suite `T` required to distinguish every (killable) mutant from `M`, likewise measuring the number of analyzed mutants (killed or judged equivalent) during the process. The simulation starts by picking the top mutant `m`, according to the selection strategy used, among survived mutants (initially considering all mutants from `M`). Next, we check if there exists some test in the test pool `P` that kills `m` (this process simulates a tester picking, analyzing, and designing a test to kill a mutant). If no test kills a mutant `m`, we judge it as equivalent and remove it from `M`. Otherwise, we randomly pick one test `t` from the pool that kills `m`, add `t` to the suite `T`, and remove from `M` every mutant that is killed by `t`. This process continues by taking the next surviving mutant from `M`, finding a test `t` to kill it, and repeating until every mutant in `M` is killed (or judged as equivalent).

In order to perform a more complete and fair comparison between the tools, we measure the *cost* of a mutation testing tool in two ways: The *number of tests* designed/written to kill all (killable) mutants [68], and the *number of analyzed mutants* during the process [57].

Furthermore, it is necessary to note that we consider the *effectiveness* of a mutation testing tool as the ability to devise a test suite `T` to *detect the real fault*. That is, we measure whether, by running forged test suite `T` on the faulty version of the program, we could detect the real fault.

To answer RQ1, we run previously described simulation by *randomly* sorting the list of mutants from the different mutation tools and comparing their effectiveness when applying the same effort, i.e., how many faults we can find when writing the same number of tests or analyzing the same number of mutants. To answer RQ2, we run the same simulation and comparison, but with the mutants prioritized according to *Cerebro*'s importance prediction.

Since our simulation process includes some random effects (e.g., which test `t` is selected to kill a mutant `m`), we repeat this process 100 times for all approaches to reduce the threat of randomness [174].

Overall, this experimental setup promises to investigate the performance and usability of the studied mutation testing tools when applied together with mutant

Table 8.2: Prediction Performance of *Cerebro*.

***Cerebro* trained on:**	**MCC**	**Precision**	**Recall**
μBERT	0.56	0.81	0.52
IBIR	0.40	0.84	0.25
PIT	0.43	0.71	0.35

* Cerebro *maintains similar performance as reported by Garg et al. [63]*

selection strategies.

8.3.5 Cerebro Mutant Selection Prediction Performance

To use the machine-learning approach *Cerebro* [63] and select (subsuming) mutants when addressing RQ2, we need to train it on our data set. We follow the guidelines of Garg et al. [63] to implement *Cerebro*'s approach and perform training. Garg et al. employ a 5-fold cross-validation to evenly split the benchmark into five parts, providing five models to obtain probabilities. To evaluate the performance of *Cerebro* on our dataset, we repetitively use one-fold of our benchmark for testing and 4 for training. Table 8.2 reports the average prediction performance of our implementation of *Cerebro*, which is comparable with the results of Garg et al. [63] when trained on PIT mutants. When we train it on μBERT mutants, we observe better prediction performance indicators (10% in Precision, 17% in Recall, and 13% in MCC) than trained on PIT mutants. When training *Cerebro* on IBIR mutants, we obtain slightly worse prediction performance than when trained on other tools (3% and 16% lower MCC w.r.t to PIT and μBERT) (Note that since DeepMutation produces only one mutant per method, no mutant prioritization is required).

For the sake of clarity, it is pertinent to note that column *Precision* describes the ratio of mutants truly subsuming among all the mutants predicted as subsuming, while column *Recall* is the ratio of mutants correctly predicted as subsuming among all the subsuming mutants. The column MCC (*referring to Matthews Correlation Coefficient*) [175] denotes the coefficient between 1 and -1. An MCC value of 1 indicates a perfect prediction, whereas a value of -1 indicates a perfect inverse prediction, i.e., a total disagreement between prediction and reality. An MCC value equal to 0 indicates that the prediction performance is equivalent to random guessing.

8.3.6 Statistical Analysis

To evaluate whether fault detection under the same invested effort is significantly different between techniques, we use the non-parametric effect size measure Vargha and Delaney A_{12} [71]. Intuitively, A_{12} measure will tell us how frequently one tool obtains better indicators than the others. It returns values between 0 and 1, where $A_{12} = 0.5$, showing that the two measures are completely equivalent; otherwise, they have some differences.

8.4 Empirical Evaluation

8.4.1 RQ1: Tool's effectiveness under Standard Selection

We start our analysis by examining the effectiveness of the mutation testing tools/techniques under the standard mutant selection strategy.

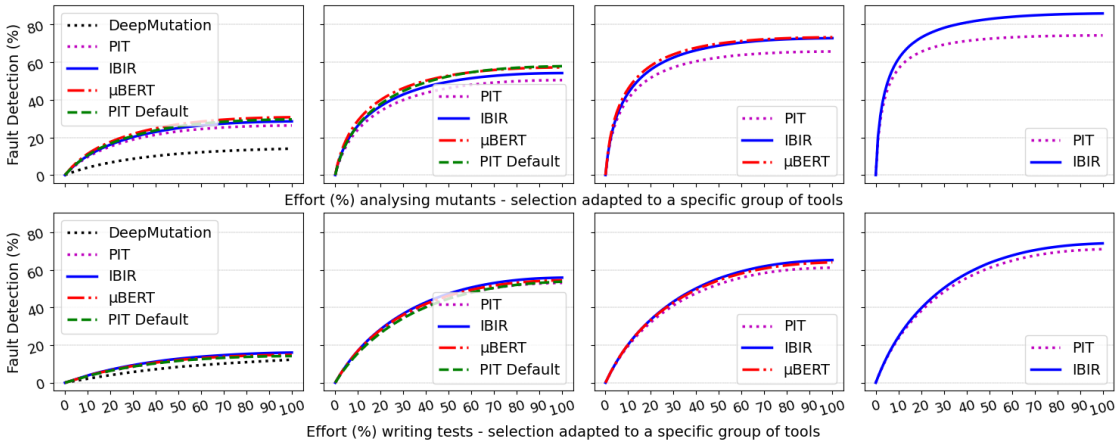


Figure 8.1: RQ1 Standard Selection: tools' **cost-effectiveness** in fault detection over **different effort models** – analysing mutants / writing tests. **Different groups** of tools control the number for selection to address differences in the scope of mutant generation.

Table 8.3: Cost-effectiveness comparison under different mutant selection strategies (RQ1 and RQ2). Each cell represents the absolute difference in fault-detection between the **Observed Tool** and the **Baseline** (-/+ for lower/higher fault detection) when the same effort is invested ($\#M$ stands for the same number of mutants analysed, and $\#T$ stands for the same number of tests written). For instance, using Standard Selection (RQ1) IBIR detects, on average, 14.50% more faults than DeepMutation, when analysing the same number of mutants.

RQ1: Standard Selection

Observed Tools	**Comparison Baseline Tools**							
	DeepMutation		PIT_Default		μBERT		PIT	
	$\#M$	$\#T$	$\#M$	$\#T$	$\#M$	$\#T$	$\#M$	$\#T$
PIT_Default	15.52	1.95	—	—	—	—	—	—
μBERT	16.68	2.57	-0.64	0.72	—	—	—	—
PIT	12.30	2.28	-7.42	-0.65	-7.59	-2.84	—	—
IBIR	14.50	3.79	-3.66	2.12	-0.50	1.15	11.66	3.06

RQ2: Learning-Based Selection (Cerebro)

Observed Tools	**Comparison Baseline Tools**					
	PIT_Default		μBERT		PIT	
	$\#M$	$\#T$	$\#M$	$\#T$	$\#M$	$\#T$
μBERT	12.14%	3.30%	—	—	—	—
PIT	-2.09%	-2.45%	-10.42%	-3.64%	—	—
IBIR	-1.78%	-2.39%	-7.06%	-0.73%	5.77%	-0.70%

** Columns correspond to columns in the grids of Figures 8.1 and 8.2*

Figure 8.1 visualizes fault detection concerning the number of analyzed mutants and the number of written tests. It is pertinent to note that the selection number is controlled since different observed tools generate different numbers of mutants. Hence, when studying the detection of each fault, the maximum cost is directed by the tool that produces the least number of mutants, more precisely, which requires the least effort to analyse all of its mutants. Table 8.3 summarises the differences in fault detection of the tools involving the same effort. Let us consider from Sub-table (RQ1) in Table 8.3, the first column – DeepMutation. This column summarises the fault detection difference between the tools observed in the two left Sub-figures of 8.1, in which DeepMutation is considered the reference tool that limits the maximum cost of the simulation (as its mutants require the least effort to be all analysed, in

the majority of the cases). The sub-columns $\#M$ and $\#T$ values report the fault detection advantage (or disadvantage) of using a tool from the first column instead of using DeepMutation, when spending the same effort in terms of respective mutants analysed and tests written. For instance, the first row indicates that PIT_Default (29.54%) can detect 15.52% more faults than DeepMutation (14.02%) when analyzing the same number of mutants while detecting near 2% more faults when writing the same number of tests. Hence, we use different tools as a baseline to represent our results in Table 8.3 and Figure 8.1, sorted in ascending order according to the number of mutants generated with each tool: DeepMutation, PIT_Default, μBERT and PIT.

We observe that DeepMutation is the least cost-effective technique - other tools detect between 12% and 17% more faults when the same number of mutants is analyzed. Moreover, we notice that the rest of the tools require the analysis of fewer mutants than DeepMutation (around four times less) to reach the same fault detection. The differences are statistically significant, according to the computed p-value. We also compared them with the Vargha-Delaney A measure *($\hat{A}_{12}$) [71], showing that other tools achieve better fault detection on average in 99.6% cases.*

We can also observe that the effectiveness of PIT_Default (58%), μBERT (57%) and IBIR (54%) is similar, outperforming PIT (50%) when analyzing as many mutants as PIT_Default. When writing the same number of tests, we observe that μBERT reaches similar effectiveness (54%) as PIT_Default, PIT 53% and IBIR 56%.

When we focus on μBERT, we can observe that both μBERT (73.2%) and IBIR (72.7%) are more effective than PIT (65.7%) under the same effort. These differences also have statistically significant p-value, and $\hat{A}_{12}$ when compared to their cost-effectiveness, evidencing that μBERT and IBIR in $\approx 99\%$ cases can detect more faults. Moreover, to reach the same effectiveness as PIT, μBERT and IBIR need to analyze 44% and 38% fewer mutants than PIT. When we compare the number of tests, we observe that IBIR (65.2%) and μBERT (64.01%) can detect near 4% more faults than PIT (61.16%) under the same effort.

Finally, we can observe that IBIR (near 90%) is more effective than PIT (74%) when the same number of mutants are analyzed. IBIR needs to analyze 80% fewer mutants (and 25% fewer tests) than PIT to reach the same effectiveness. This difference is also statistically significant *p-value<0.01*.

IBIR is the most effective tool, identifying on average $\approx$ 90% of real faults. It requires the analysis of 80% fewer mutants (and 25% fewer tests) than PIT to reach the same effectiveness. In terms of cost-effectiveness μBERT, IBIR, PIT_Default perform similarly, with PIT performing slightly worse. All other tools subsume DeepMutation concerning fault detection through standard selection.

8.4.2 RQ2: Tool's effectiveness under *Cerebro*

Figure 8.2 visualize cost-effectiveness simulation of the fault seeding techniques when we use a machine learning-based approach, i.e., *Cerebro* for mutant selection. It is important to note that due to findings in the previous research question, we don't find it suitable to consider DeepMutation in this research question since it is subsumed even when using all its mutants.

As can be seen in Figure 8.2, learning-based mutant selection improves the

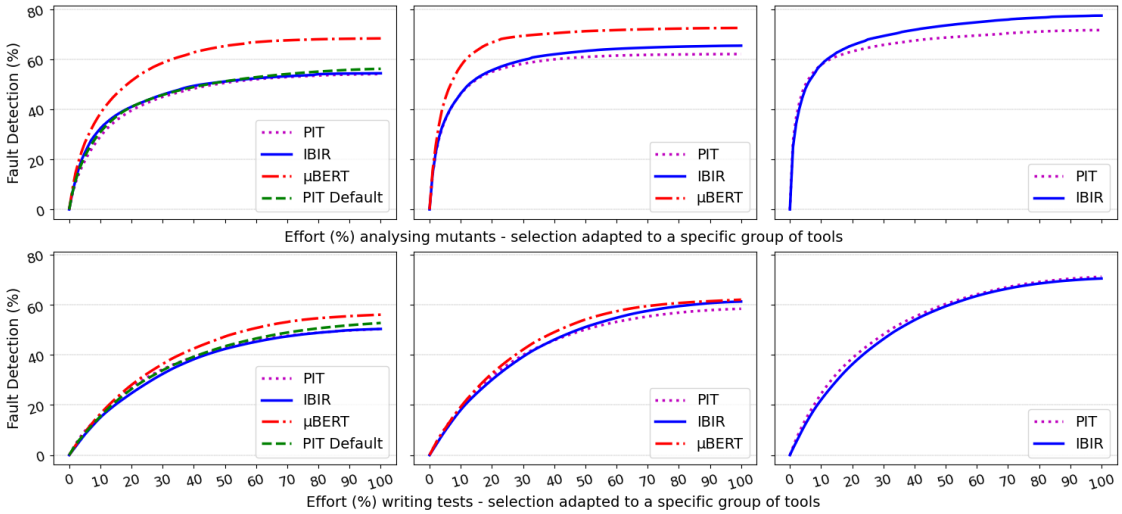


Figure 8.2: RQ2 Learning-Based Selection: tools' **cost-effectiveness** in fault detection over **different effort models** – analysing mutants / writing tests. **Different groups** of tools control the number for selection to address differences in the scope of mutant generation.

remaining fault-seeding techniques' cost-effectiveness. Table 8.3 in RQ2 cells presents the absolute differences in fault detection between the tools.

Interestingly, when using *Cerebro* to select mutants, μBERT achieves ≈12%, ≈14% and ≈13% higher fault detection rate than PIT_Default, PIT and IBIR, respectively, when analyzing the same number of mutants and ≈4%, ≈6% and ≈6% when analysing the same number of tests. The differences are statistically significant, according to the computed p-value (< 0.01). We have also validated these findings by computing the ($\hat{A}_{12}$) measure, showing that it achieves higher fault detection on average in 96.2% cases. Surprisingly, μBERT analyses 50%, 82%, and 78% fewer mutants than PIT_Default, PIT and IBIR, respectively, to reach the same effectiveness. Concerning the number of tests, the difference is also noticeable when compared with PIT, since μBERT obtains ≈5% higher fault detection under the same number of tests written.

Interestingly, in the presence of the learning-based mutant selection strategy, IBIR keeps its significantly high difference in fault detection when analyzing the same number of mutants as PIT, ≈6%. This difference is also statistically significant with *p-value<0.01*.

Overall, our results indicate that using learning-based mutant selection significantly impacts the cost-effectiveness of the fault-seeding techniques. This observation promises to impact how we use fault-seeding techniques and how we compare new fault-seeding techniques' effectiveness.

μBERT has significantly improved its performance under the machine-learning-based selection strategy, i.e., Cerebro*, becoming the most cost-effective fault seeding technique. μBERT needs to analyse 50%, 82%, and 78% fewer mutants than PIT_Default, PIT and IBIR, respectively, to reach the same effectiveness when selecting mutants according to* Cerebro*. IBIR, PIT_Default and PIT all experience improved performance but overall, they all perform similarly.*

8.5 Discussion

8.5.1 Mutant Selection or not: How does the mutant selection impact the mutation testing tools?

Regardless of the fundamental differences between the considered approaches, our results show they are all efficient in guiding testing towards higher fault detection capabilities. In fact, with relatively low efforts, they score comparable fault detection rates. Their difference becomes noticeable only by spending extra efforts or leveraging a mutants selection strategy to spare the efforts lost in analyzing irrelevant mutants - as can be seen from our results in RQ2. While in RQ1, under the standard mutant selection strategy, we do not report any statistically significant difference ($p<0.05$) between the tools, except in the cases where DeepMutation is subsumed and μBERT and IBIR deviate from others. Although we observe that IBIR is the most effective when considering extra effort in analyzing mutants - with statistically significant differences. The reasoning is that a) IBIR provides mutants with high fault detection capabilities but b) produces numerous equivalent and irrelevant ones, which increase the cost to the target.

The impact is revealed when we applied a learning-based mutant selection approach in RQ2, which deflates mutant redundancy. Learning-based selection *Cerebro* makes μBERT the most cost-effective, significantly outperforming the other approaches, which differs from standard selection conclusions. We argue that *Cerebro* boosting μBERT, particularly, unlike the others, lies in the similarity and homogeneity among its mutants, which all replace one token with another considering the code context. In contrast, others may remove or alter multiple tokens or statements, making the learning task harder.

To further investigate this variance in cost-efficiency between the studied techniques, we should check whether the other approaches could also get boosted by classification techniques like μBERT. This question is particularly interesting in the case of IBIR, which introduces significantly more mutants, thus, more subsuming and subsumed mutants than μBERT, which challenges any classifier. To check this hypothesis, we plan in future work to construct and study more classifiers (learning-based or not) together with a perfect classifier (an artificial model that perfectly predicts whether a mutant is subsuming or not) and thus obtain more insights about the cost-effectiveness of the available fault-seeding techniques.

Altogether, we conclude that when comparing mutation testing techniques, the standard mutant selection strategy can lead to incomplete conclusions as the most cost-effective tool/technique is not necessarily the same under learning strategies which we encourage researchers and practitioners to utilize and explore further.

So far, the investigation also implies that some operators of different approaches provide beneficial ingredients that should be considered and further explored as complements to mutation testing tools. In the following, we scratch the surface and pave the way for future researchers towards this direction.

8.5.2 Complementarity of the approaches

So far, we have elaborated on the cost-effectiveness of fundamentally different approaches when guided – or not – by intelligent selection. However, we haven't investigated and given insights into "how?" and "what?" an approach can and cannot reveal, and consequently, 1) what could be the added value of spending more effort

using each approach and 2) whether the approaches could complement each other.

As a first step in this direction – as encouragement for future studies since a thorough breakdown may undermine the intent and scope of this work – we amend our quantitative study with a qualitative one. We investigate the bugs that each approach can find – at least once among our simulation repetitions – and distinguish the ones that could not be found by all approaches. Then, we examine these bugs with their revealing mutants and discuss the particularities and shortages, i.e. the fault injection patterns that make any difference between all considered approaches.

The Venn diagram in Figure 8.3 depicts the distribution of the bugs that are revealed by each tool. Same as per previous results, IBIR outperforms all approaches in terms of fault detection capability, finding 99,57% of the target bugs followed by μBERT (92,7%), PIT (87,55%), PIT_Default (85,83%) and finally DeepMutation (41,20%). In fact, IBIR can discover all the target bugs except one (Mockito 5), which no approach can find, indicating the pseudo-completeness of its mutation operators. Indeed, 2 bugs are only found by IBIR – Math 12 and Mockito 33 – mainly thanks to patterns' power, such as removing or inserting new statements, replacing method invocations, and adding extra conditions.

Concerning μBERT mutations, even if they do not remove or insert new code but only change one token by another, they can reveal 17 bugs which only IBIR could find. Additionally, μBERT finds respectively 26 and 29 bugs that mature and sophisticated operators of PIT_Default and PIT missed. We explain this by the fact that μBERT's pre-trained model CodeBERT and its knowledge of code (the information it retrieves from the code) to mutate perms it to propose real-like code replacements, thus, real-like mistakes and bugs, i.e. changing method calls, access to objects' fields and arrays etc.

PIT and PIT_Default yield comparable results, showing they can amend μBERT capabilities in finding respectively 13 and 14 bugs more, thanks to patterns that involve multiple tokens changing, i.e. the removing mutation operators.

Overall, we believe that future research should investigate the appropriate joint use of IBIR's inverted fix-patterns, well-crafted and mature PIT_Default grammar transformations, and μBERT's code and context-knowledge based mutations.

8.5.3 Implications for practice

Over the last decade, mutation testing (a.k.a. fault seeding, fault injection) has been used exhaustively for testing, debugging, maintenance, change-aware dependability analysis, test assessment, etc. In the context of mutation testing, recent industrial applications less often include the generation of all mutations or coverage-adequate test sets. Cheaper trends concern the effort required, and the risk of unrealistic test requirements is seen through the objective of equivalent mutants. Instead, industrial applications are interested in obtaining a curbed sample of mutants that reliably mimic real faults. Thus, satisfying the famous mutation testing proverb: "Do fewer, do smarter, do faster" [35].

At the same time, with the wave of open possibilities brought by machine learning models, many tools emerged. Their diversity provokes a topic of interest in the development and research circles – together with empirical comparisons – about which approach/tool is more efficient in emulating and revealing actual bugs. However, as we showed in the paper, solely comparing tools based on their mutant generation degree does not lead to solid conclusions. Each technique brings additional value, with

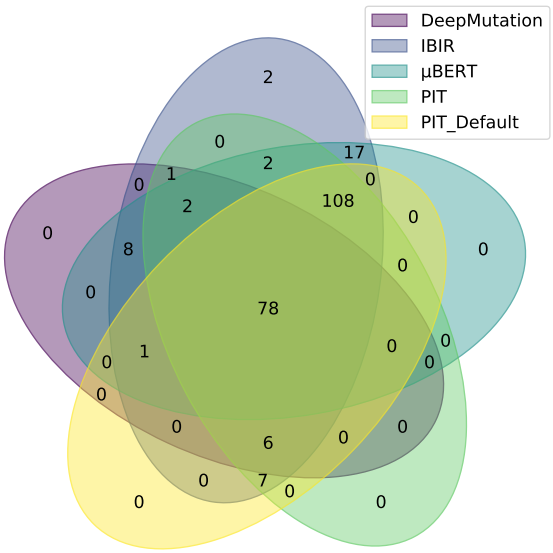


Figure 8.3: Which bugs are revealed by every approach? IBIR is capable of finding almost all (99,57%) bugs, followed by mBERT (92,7%), Pit (87,55%), Pit-default (85,83%) and finally DeepMutation (41,20%).

an inevitable cost in noise. Thus, with this study, we aspire to spread the message to practitioners to consider the actual cost of every technique expressed through some form of intelligent selection and effort analysis, thus discarding the noise. Furthermore, we shed light on different degrees of redundancy that different approaches carry, transformations that make them distinguishable and complimentary, making them less costly. Altogether, the central insight of our study to future researchers and tool developers is to consider appropriate selection strategies when comparing and developing fault-seeding techniques. Moreover, we encourage researchers to explore the combination of models to identify promising locations for mutant generation and joint transformation rules.

> *When comparing mutation testing techniques or tools, it is imperative to account for a mutant selection technique suitable for this purpose. The use of standard mutant selection entails a risk of drawing incorrect conclusions.*

8.6 Threats to Validity

External Validity: To reduce threats that may relate to the subjects we used, we selected 509 faults from 15 mature open-source real-world projects that are well maintained and tested from Defects4J *v2.0*. As we already discussed, while conducting our experiments, we could not compile or run the tests of all the versions available in Defects4J *v2.0*. Although our evaluation expands to many faults and Java projects of different sizes, the results may not generalize to other projects or programming languages.

Another external threat lies in the tools' specificity and running configurations we consider. To reduce this threat, we employ fundamentally different modern mutation tools and run them exhaustively using their corresponding default configurations, generating as many mutants as the tool can generate for each subject.

Another threat can be related to the mutant selection strategies used in the study.

To reduce this threat, we consider two fundamentally different approaches. Nevertheless, we do not remove the threat that results can change when another mutant selection technique is employed or considering another language (e.g. *Cerebro* [63] was also evaluated on C programs, which was not explored in this paper) - yet we plan to follow this line of work in the future.

Internal Validity: Threats to internal validity may arise in how we train the machine learning-based approach *Cerebro* for RQ2. To address this threat, we strictly follow the guidelines reported by Garg et al. [63] and explain the steps in Section 8.3.5. In contrast to Garg et al. [63] that evaluated *Cerebro* only with mutants generated with PIT, in this study, we also train *Cerebro* on mutants generated with μBERT and IBIR. Other threats may relate to how we label mutants as subsuming. To counter this threat, we rely on the developer suites provided by the Defect4J benchmark, as it is regarded as the most detailed dataset of faults from projects with thorough test sets in Java language. Any weakness in the suites may lead to incorrect labelling for mutants, introducing some noise that can affect *Cerebro*'s prediction abilities. Unfortunately, we could not compile and run the master-branch [124] of DeepMutation. Thus, we had to operate the tool from the resources and pre-trained artefacts provided in the repository.

Another threat to internal validity may be that we generate mutants only for the class fixed in the bug-fix pairs provided by Defects4J. Thus, we do not reduce the potential threat that the results do not apply to mutants from other classes interacting with the mutants used in this study.

Construct Validity: Our assessment metrics, number of analyzed mutants, number of written tests and fault detection may not entirely and exhaustively reflect the actual testing cost / effectiveness values. These metrics have been suggested by literature [8, 68, 127] and are intuitive, i.e., the number of analyzed mutants and the number of tests essentially simulate the manual effort involved by testers when mutants guide the testing process [1, 32]. These two cost models illustrate objective comparison as the engineering effort is assumed to be fixed, which would not be the case in a human study, fluctuating based on a participant's experience. While measuring real execution time (computational effort) would be impacted by the environment, e.g., the number of machines, machines' performance, scheduling algorithms and maturity of the tools in a sense if they have built-in support for multi-threading/parallelism.

At the same time, fault detection is the effectiveness metric of interest in this study that can be impacted by randomization. To address this threat, we run and repeat 100 times a simulation scenario where a tester selects mutants and designs tests to kill them. Overall, we mitigate these threats by following suggestions from mutation testing literature [8, 68, 127], using state-of-the-art tools and performing several simulations. We also find consistent and stable results across our subjects.

8.7 Conclusion

We studied the fault detection performance of recently proposed mutation testing tools (DeepMutation, PIT, IBIR and μBERT) on a new and large fault dataset.We also employed two different mutant selection strategies a) standard, b) state-of-the-art deep learning driven one, then we performed a cost-effectiveness comparison using two typically adopted cost models; one associated with the number of mutants requiring analysis and a second one with the number of tests to reveal the injected

faults. Our results showed that IBIR has the highest fault detection capability (≈90% on average) but is not the most cost-efficient. In contrast, μBERT, even less effective, has significantly higher cost-effectiveness when using learning-based selection strategies, approximately 12% higher, than all the other tools. More notably, we found that mutation testing tools perform differently when guided by mutant selection strategies, indicating the need for considering intelligent selection when comparing mutation testing tools.

9

Supported Datasets and Tools

This chapter presents the datasets generated in support of evaluating proposed approaches and the tools used in support of our analysis. We believe the open-sourced datasets and reference on the existence of tools will provide further benefits to practitioners and solid ground to build new techniques and conduct rigorous studies building on the contributions of this dissertation.

Contents

9.1 Mutation datasets

In order to perform the studies described in this dissertation, a substantial amount of time was dedicated to building appropriate datasets. Aiming to point to those datasets for the benefit of future practitioners, this section consolidates and in detail describes the information they carry and provides the links for their straightforward access. For orientation purposes, available datasets refer to Commit-Aware Mutation testing with clean test contract (Chapter 5), Commit-Relevant mutants via HOMs (Chapters 6 and 7), and dataset used for studying the efficiency of mutants of different mutation approaches when using learning-based selection (Chapter 8).

9.1.1 Dataset of mutants with clean test contracts

Table 9.1: Java Mutants Dataset for commits of clean test contracts

Project	# Commits	# Mutants	# Test cases
commons-cli	9	61,419	3,247
commons-collections	5	323,584	55,076
commons-io	3	105,181	3,972
commons-net	6	345,130	1,478
joda-time	5	561,782	20,962
jsoup	8	330,125	4,985

This dataset contains mutants for a set of commits that preserve clean test-contract, w.r.t., a commit does not touch or change test files. Precisely, for each commit, mutants are generated for both the commit in question, a.k.a., post-commit, and the previous commit, a.k.a., pre-commit. Commits are mined from well-known and well-tested Java language programs. In total, this dataset counts 6 different programs and 31 commits. The commits come from projects in the Apache Commons Proper repository[1], precisely, commons-cli[2] counts 9 commits with 61,419 generated mutants executed with 3,247 tests, commons-collections[3] counts 5 commits with 323,584 mutants executed with 55,076 tests, commons-io[4] counts 3 commits and 105,181 mutants executed on 3,972 tests, commons-net[5] counts 6 commits and 345,130 executed mutants on 1,478 test. These projects represent a set of reusable Java component projects. Moreover, the rest of the commits in this dataset come from Joda Time[6], representing Java time and date library, counting 5 commits with 561,782 mutants executed on 20,962 tests, and Jsoup[7], an HTML page render and manipulation library, with 8 commits containing in total 330,125 mutants executed on 4.985 tests.

[1] `https://commons.apache.org/`
[2] `https://github.com/apache/commons-cli`
[3] `https://github.com/apache/commons-collections`
[4] `https://github.com/apache/commons-io`
[5] `https://github.com/apache/commons-net`
[6] `https://github.com/JodaOrg/joda-time/`
[7] `https://github.com/jhy/jsoup`

Worth noting is that for each of the projects, a manual gathering of the most recent commits was conducted at the time of the study in Chapter 5, meeting the following conditions from the project's history: (1) only source code is modified, no modification to configuration files, (2) the commit introduces a significant change, not a trivial one such as a typo fix, (3) test contracts are not modified, in order to compare pre- and post-commit outputs meaningfully and (4) both pre- and post-commit versions of the project build successfully.

As previously noted, this dissertation uses PiT as a state-of-the-art mutation testing tool for mutation generation. For each project commit, the dataset contains two files as the output of the tool, named *v1.xml* and *v2.xml*, describing mutants of pre- and post- commits, respectively. Each element of the corresponding XML files represent a mutant and contains information, a.k.a. attributes, whether the mutant is detected by tests, what is its killing status, and how many tests run on the particular mutant. Besides this information, the mutant element contains sub-elements serving as metadata about the mutant exact location in the code under test, w.r.t., source code class, mutated method, file line number, bytecode index and a block on which the mutant acts together with the information about the mutation operator used as a transformation rule. Furthermore, each mutant element has a list of killing tests, i.e., their names and package location, and a list of tests that successfully run on mutants without distinguishing them.

Overall, this available dataset, in total, offers 36 commits and 1,727,221 mutants executed on complete project source code, with 89,720 JUnit test cases. Table 9.1 summarises information about the available dataset that can be accessed on the following maintained webpage:
https://ojda22.github.io/milos-ojdanic/mutation_datasets.

9.1.2 Dataset of High-Order mutants

This dataset, built for the purposes of studies described in Chapter 6 and 7, contains, to time, the most extensive dataset of project commits and associated mutants in the context of evolving systems. Precisely, the dataset contains chronological commits from the following well-known, well-tested, and matured Java open-source projects taken from Apache Commons Proper repository: *commons-collections - 45 commits, commons-lang - 66 commits, commons-csv - 101 commits, commons-io - 30 commits and commons-text - 46 commit*. The commits are mined from their timestamped year 2005 to 2020 and include the use of JUnit4+[8] testing framework. The test suite for each commit is augmented with EvoSuite [30], the test generation tool. As mentioned, this dataset contains both automatically generated tests, 5,685 test cases in total, and 20,513 developer-written test cases executed on generated mutants. In difference from the previously described dataset, in this dataset, three categories of mutants are generated and clearly distinguished per committed files. In particular, the commons-collection commits count 27,417 first-order mutants, 2,026 mutants on a change, and 1,192,188 high-order mutants generated as a conjunction of mutants from around the change and on a change. From commons-io commits, this dataset contains 24,970 mutants, from which 1,115 is on a change, resulting in 668,448 high-order mutants. The commons-text commits contain 47,847 mutants across all its studied commits, from which 4,155 are on a change, resulting in 2,073,829 high-order mutants. All commons-csv commits result in 66,862 first-order mutants, 3,577 on

[8]https://junit.org/junit4/

Table 9.2: Java Mutants Dataset for commits with high-order mutants

Commons Projects	# LOC	# Maturity	# Commits	# FOM	# Mutants on Change	# HOM	# Dev. Tests	# EvoSuite Tests
collections	74,170	14/04/2001	45	27,417	2,026	1,192,188	4,797	1,285
io	29,193	25/01/2002	30	24,970	1,115	668,448	914	286
text	22,933	11/11/2014	46	47,847	4,155	2,073,829	1,084	322
csv	4,844	25/01/2002	101	66,862	3,577	1,968,137	6,144	2,833
lang	85,709	19/07/2002	66	102,072	3,891	3,885,341	7,574	959
Total	216,489	N/A	**288**	**269,168**	**14,764**	**9,787,943**	**20,513**	**5,685**

a commit-change, and 1,968,137 high-order mutants. The commons-lang contains 102,072 first-order mutants, 3,891 mutants on a change and 3,885,341 high-order mutants. In total, the number of first-order mutants for all 288 commits is 269,168, while the number of mutants generated in a diff-based manner, w.r.t, mutants on a committed change, is 14.674. Combined, as described for the purposes of the study, these mutants generate 9,787,943 high-order mutants in total executed on the associated tests, 20,513 developer-written tests and 5,685 EvoSuite automatically generated tests.

The dataset, besides standard mutants metadata described in the previous section, contains information on every assertion executed as a part of a test. Moreover, each mutant, besides information on test execution, contains information about the test name and the list of its assertions, where each assertion contains an id consisting of the test name, line number and assertion method. Each assertion has its concrete execution value and the output of its oracle. Based on these values and the formalisation of mutant relevance described in Chapter 5, the dataset labelled relevant mutants, strongly relevant mutants, subsuming mutants and subsuming commit-relevant mutants.

Overall the generation of this dataset required 68,213 CPU days of execution on two nodes with 20 physical cores of Intel Skylake Xeon Gold 2.6GHz processors running on Linux Ubuntu OS with 256GB of RAM. Table 9.2 summarises information about the available dataset while the further descriptive information in comma-separated format *(.csv)* can be accessed together with the dataset on the following webpage: *https://ojda22.github.io/milos-ojdanic/mutation_datasets*.

9.1.3 Dataset of mutants from different mutation approaches

This dataset contains mutants of different mutation testing approaches generated on well-known and well-maintained Defects4J [116] (v2.0.0) Java language real faults dataset. Due to technical reasons, which often include challenges in executing relatively recent mutation tools on program versions with relatively old dependencies, this dataset contains mutants only from those faults that satisfy the building requirements. Specifically, this dataset considers the following projects and the commits from the Apache Commons [120] family: commons-cli - 38 commits, commons-codec - 18 commits, commons-compress - 33 commits, commons-csv - 16 commits, commons-math - 100 commits, commons-lang - 63 commits, commons-collections - 4 - commits, commons-jxpath - 21 commits. Also in this dataset can be found projects from the Jackson family, which is a suite of data-processing tools

for Java and includes jackson-core - 26 commits, jackson-databind - 102 commits, and jackson-dataformat-xml - 6 commits. Additionaly, this dataset includes faulty commits from Mockito - 27 commits, one of the most popular mocking frameworks in Java, Jsoup - 11 commits, a Java library for HTML parsing, Gson - 18 commits, a Java library for JSON parsing and generation from and into java objects, and joda-time - 26 commits, for the Java date and time classes. For each selected faulty project version from Defects4J, this dataset contains mutants generated on the modified classes between the faulty and fixed versions. Precisely, mutants are generated for the fixed version of each modified class by employing the following mutation testing tools: DeepMutation fault-seeding tools deliver 5,559 mutants for the 348 analysed faults. μBERT, after being applied to 499 faults, produced 293,304 mutants. IBIR produced 1,113,113 mutants for the 393 analysed faults. While PiT, as the state-of-the-art tool with its *all* available mutation operators, generated 1,212,544 mutants across 29 mutants categories for the 509 faults analysed. For each generated mutant, this dataset offers metadata, such as a unique mutant id, the mutant tool, killing tests, pre-calculated OCHIAI score and FDP score.

Table 9.3 summarises the number of faults analysed and the number of mutants generated by each mutation testing tool, while the available dataset can be fetched on the following webpage:
https://ojda22.github.io/milos-ojdanic/mutation_datasets

Table 9.3: Java Mutants of different mutation approaches from Defects4J Dataset

Mutation Testing Approach & Tool	**# of Analysed Faults**	**# of Mutants**
DeepMutation	348	5,559
PIT_Default	508	110,480
μBERT	499	293,304
IBIR	393	1,113,113
PIT	509	1,212,544

9.2 PiTest Assert

Due to the constraints of the PiTest mutation testing tool to satisfy the requirements of experiments described in this dissertation, we implemented an additional infrastructure on the PiTest fork to ensure the analysis of evolving systems with appropriate mutant generation. The extension was built on top of the PiTest High Order Mutants [154], such as to take as an input parameter the `gitdiff` output[9], a.k.a., a file with statement difference between two commit versions. Based on the difference, the extension augments the mutants' generation function by mapping, i.e., generating mutants on the change, with the mutants around the change. Thus, creating second-order mutants for that particular commit file.

Another reason behind the extension lies in the PiTest testing tool (V1.5.1) evaluation of whether a mutant is killed or not based on test case oracle prediction (test fails or passes), which stood not suitable and too high level for some of the experiments conducted in this dissertation (Chapter 6). To satisfy the requirements of the experiments and focus on lower test output granularity, the extension also extracts additional information concerning each test case assertion - from each test

[9]https://git-scm.com/docs/git-diff

that covers mutants. The PiTest Assert extension extracts test assertions outputs by performing bytecode instrumentation of each test executed on a specific mutant, using ASM[10] as an all-purpose Java bytecode manipulation and analysis framework. By instrumenting each test case assertion, the extension obtains execution information of expressions used for the oracle. More precisely, for each assertion, the extension outputs its unique test name, the line where it locates in a test file, the assertion function name, and the assertion's actual and expected execution value. Indeed, if an assertion triggers an exception, stack-trace execution is logged. The product of PiTest Assert is a weighted mutant-assertion matrix. For each mutant test-assertion pair, the value corresponds to the actual assertion value obtained by running the test on the mutant or the exception stack trace if an assertion throws an exception.

It is worth noting that the extension in the current form employs the *JUnit4*[11] testing framework, which contains a public static class called `Assert`, that provides a set of assertion methods to specify test conditions. Typically, these methods (e.g., `Assert.assertEquals(expected value, actual value)`) directly evaluate the assertion's conditions, then returns the final assertion's output (e.g., conditions not satisfied, pass, or fail).

Therefore, to obtain the value of parameters within the assert statement, `PiTest Assert` instruments each Assert method. Such that it serializes the provided input values in the assert statement before they propagate to conditional checks, i.e., before the conditional check is reached in *org.junit.Assert*[12] and the output values are fed to *org.hamcrest.Matcher*[13] for evaluation.

Specifically, it serializes both the expected and actual values after they propagate as input parameters of the assert statement. This allows assessing the input parameters of the `assert` statement (e.g., an expression or a method call (`assertEqual(foo(), bar())`)) for concrete values. However, it is worth specifying that this experimental framework does not directly account for the potential dependencies within assertions and test cases. This test assertion framework used for experimental purposes and constructed on top of `PiTest` is publicly available on the GitHub[14].

[10]https://asm.ow2.io/
[11]https://junit.org/junit4/
[12]https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html
[13]http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/Matchers.html
[14]https://github.com/Ojda22/pitest/tree/pit-SOM-RM-AssertCache

10

Conclusion and Future Work

This chapter concludes this dissertation with the summary of contributions offered and outlines the future work with the preliminary ideas, results and directions of promise.

Contents

10.1 Summary of Achievements

This dissertation presents a code-change aware mutation-based testing approach that aims at bringing mutation testing closer to practice by considering its application in the context of continuously evolving systems. The studies conducted under this dissertation strive to provide knowledge to software practitioners about the potential of mutation testing to assess the automatic regression testing quality and enable testing of the unexpected effect of the change that often gives rise to regression faults. In particular, the challenges that this dissertation targets relate to the code-mutants' behavioural properties, testing regular code modifications and mutants' fault detection effectiveness. The contributions built around these challenges and offered in this manuscript can be summarised and grouped into *1)* An empirical study to examine the ability of the mutants to capture the behaviour of regression faults and to evaluate the relationship between the mutants and real faults, which is often captured with the semantic and syntactic distance metrics. The outcome of the study shed light on the present misleading assumption indicating to practitioners that tailoring fault patterns, a.k.a, mutants, to be more syntactically close to the real fault makes them also semantically close. The study conducted on the most extensive dataset, to date, of real-faults shows a lack of evidence that syntactic similarity reflects semantic similarity. The various syntactic distance metrics often used to capture this similarity do not show suitability for the task, indicating that the mutants are not the only points in the program but rather follow well established RIPR model often captured by semantic metrics. This dissertation provides experiments that evaluate the suitability of semantic similarity metrics that approximate mutants' behaviour through tests. Next, this dissertation provides *2)* A formalisation of the novel commit-aware mutation testing approach that offers commit-relevant mutants as an efficient and effective change-aware test requirement target. The proposed approach offers modification-focused testing able to capture unforeseen behavioural interaction between changed and unchanged code on which regression faults often occur. In particular, the conducted experiments show that this particular novel commit-relevant mutants are able to capture 30% more regression faults in comparison with the baseline categories, plus it shows that the commit-relevant mutants cannot be approximated or any other studied category used as its proxy due to the substantial noise of traditional mutation when used in continuously evolving systems. Further, this dissertation offered an approach to identify commit-relevant mutants through High-Order mutations, resulting in the most extensive study of this novel category of mutants, eliciting further studies on mutants' properties and scientific insights. Concretely, this dissertation shows that one in three mutants are commit-relevant per committed files, yet when focusing on subsuming commit-relevant mutants, which represent a minimal set of required mutants, a reduction of 93% can be achieved, indicating high redundancy between these mutants. The further findings show the low predictability power of these mutants, highly dependent on the semantics of the change, which is one additional dimension to consider when targeting the nontrivial task of automatically predicting these mutants. The study of the mutants in evolving context led to the discovery of long-standing mutants, which regards a separate category of mutants that ultimately deserve further attention as the experiments demonstrate their ability to assess test suite brittleness and to keep the relevance of mutation test suite an order of magnitude longer than the randomly

selected mutants. *3)* The final contribution of this dissertation comes in the form of an empirical study on the utilisation of learning-based selection strategies when questioning the effectiveness of mutants from different mutation approaches, often questioned due to the noise of impractical mutants. The experiments demonstrate that different mutation approaches significantly improve their performance when guided by learning-based mutant selection solutions, which bring out of them the mutant of the highest quality. More importantly, the results of this dissertation also reveal that it is imperative to account for a mutant selection suitable for removing the noise, which often regards mutation testing as costly, as the noise often leads to the drawing of incorrect conclusions. Finally, this dissertation provides artefacts which are, at the time, the most extensive datasets of mutants available in the context of evolving systems, and together with tools used across all conducted studies, makes the available aid for future practitioners and researchers.

10.2 Future Research Directions

Automatic Prediction of Commit-Relevant Mutants With Graph Neural Networks: Besides the offered contributions in this dissertation and emphasise on the benefits of change-aware mutation testing approaches, in this section, we provide takeaways and some preliminary results and promising directions for future work concerning the automatic prediction of commit-relevant mutants. Manually engineering features for predicting commit-relevant mutants have been shown to be challenging to capture due to the dynamic dimension of change, which impacts different parts of code for each commit [113]. Advances in Graph Neural Networks, in short, machine learning models able to learn on unstructured data, have been shown very effective due to the message passing paradigm, which exchanges graph node features with adjacent nodes and thus places and distinguishes nodes in the latent space based on their neighbourhood and their interaction with the nodes of interest [176]. This mechanism we found worth exploring for the task of capturing change-impacted mutants located in and around changed nodes of a commit code graph. Moreover, as the first step in this direction, we model a suitable knowledge code graph (see the Figure 10.1) that contains information about the mapping of mutants with associated code instruction, data and control dependency between the code nodes, and as the product of our work with High-Order mutants (see Chapter 6), we mapped the relationship as edges between mutants on a change and mutants around the change as it has been shown by the preliminary results that greatly reduces the sparsity in the graph and gives the more context by allowing the nodes of interest to exchange the information. Therefore, a product is a code graph data structure, rich in information for a machine learning model to pick up and learn the change impacted mutants. The following Figure 10.1 describes the process, and future work is necessary to explore the performance and configuration of different kinds of neural network models in capturing the relevance of mutants per each committed change.

Automatic Change-Aware Test Generation guided by Commit-Relevant Mutants: This dissertation offered empirical studies and simulations showing the effectiveness and cost-efficiency of commit-relevant mutants to lead to tests that identify a real regression fault. Due to the ability of this kind of mutant to hold the information about the change-impacted behaviour of the code under test, we believe the promising future direction is to seize those often unexpected execution traces

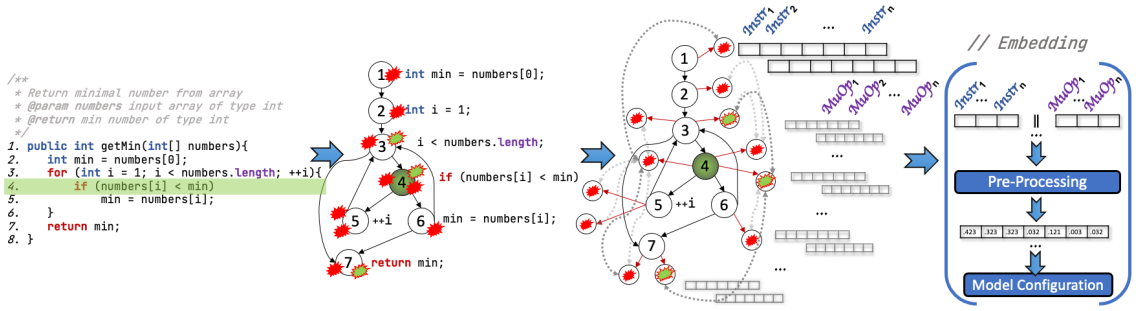


Figure 10.1: Given a code under test and the corresponding code change (line 4, represented in a green rectangle), a program dependency graph can be constructed as such that each code element node holds mutants (represented as explosions) and being defined commit relevant depending on the impact of the change (e.g., node 4). This code graph can be further augmented by isolating each mutant as a separate node and by placing an edge between each mutant around the change and the ones on the change (following similar HOM mapping provided in this dissertation, chapter 6). Each node can be represented with an instruction on which it rests and the mutation operator that defines it. After setting such an environment for each mutant node, the message-passing mechanism can provide further embedding of such mutant vectors in the latent space to position, learn and distinguish mutants based on their node features and related neighbourhood nodes, some of which affected by code-change graph nodes.

caused by commit-relevant mutants and guide automatic test generation. Given a change X and a set of mutants M impacted by X, the promising future step is to explore different search-based solutions, dynamic or static symbolic execution models such as to generate tests T, and evaluate the ability of T to assess the impact of the change X by being able to distinguish and capture observable behavioural output between M and the original program. If such an approach is developed, we believe it would bring further benefits to the field of automatic regression testing.

Implications of Long-Standing Mutants: We believe that long-standing mutants are an interesting category in their own right, worthy of further research, and it will bring promising novel research direction. This category of mutants promises further implications not only for mutation testing but also beyond mutation testing.

Implications regarding subsuming long-standing mutants: Despite showing that subsuming relationships can be preserved from version to version and that mutants' utility can be reused, we do not yet fully understand *why* subsuming mutants tend to last longer than subsumed mutants. A detailed study is needed to fully understand the subsumption and longevity drivers.

Implications for mutation testing tools: Our results also have implications for the development of future mutation testing tools. In particular, our results suggest the development of a robust mutant versioning system. Existing tools [58, 134, 158] focus on the generation of mutants but not sophisticated mutant versioning. In future work, we need to investigate mutation testing tools that allow logging mutants' maturity, execution history, and fluctuation over time, supporting approaches that learn mutant behaviour and relating this to code changes. Previous work on flaky mutant detection [159], predictive modelling [160] and hyper-heuristics [161] (in particular that focused on mutation testing [86]) may form a good starting point for this research agenda.

Maximising long-standing mutant fault revelation: By focusing on long-standing

mutants, we favour mutants that reside in relatively unchanging parts of the code. There is a natural concern that this may, in turn, lead to us favouring test suites that do not tend to reveal faults in changing parts of the code. Fortunately, the fact that a mutant lies in code region A does not render it insensitive to bugs that lie in (lexically separate) code region B. If there are transitive *dependencies* between A on B then we can expect high degrees of mutant coupling and even subsumption between the two regions. This suggests future work on identifying mutants that have high 'transitive dependence reach' through their transitive dependencies, using techniques such as slicing [94] and chopping [162].

Implications of long-standing mutants beyond mutation testing research: The findings reported in this dissertation have implications beyond mutation testing to automated program repair [163, 164] and genetic improvement [165, 166]. It is often been argued that program repair is the inverse of mutation testing. Instead of inserting faults, repair seeks to remove them. Long-standing mutants are, therefore, also likely to find applications and implications in the field of program repair and genetic improvement research. For example, it would be interesting to explore 'long-standing' repairs as a counterpoint to long-standing mutants. One might reasonably conjecture that such repairs would remain relevant for longer than repairs in areas of code subject to high degrees of churn. However, the empirical assessment of this phenomenon remains an open problem for future work.

List of Papers and Services

Papers included in this dissertation:

- Milos Ojdanic, Aayush Garg, Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis and Yves Le Traon, *"Syntactic vs. semantic similarity of artificial and real faults in mutation testing studies."*, 2023, *IEEE Transactions on Software Engineering.*
- Milos Ojdanic, Wei Ma, Thomas Laurent, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. *"On the use of commit-relevant mutants."* 2022, *Empirical Software Engineering Journal.* 27, 5 (Sep 2022).
- Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. *"Mutation Testing in Evolving Systems: Studying the Relevance of Mutants to Code Evolution."* 2023, *ACM Transactions on Software Engineering and Methodology.* 32, 1, Article 14 (January 2023), 39 pages.
- Milos Ojdanic, Mike Papadakis, and Mark Harman. *"Keeping Mutation Test Suites Consistent and Relevant with Long-Standing Mutants."* Under submission in the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Ideas, Visions and Reflections (IVR) Track, 2023.
- Milos Ojdanic, Ahmed Khanfir, Aayush Garg, Renzo Degiovanni, Mike Papadakis and Yves Le Traon, *"On Comparing Mutation Testing Tools through Learning-based Mutant Selection"*, 2023, *The 4th ACM/IEEE International Conference on Automation of Software Test (AST 2023), In press.*

Papers not included in this dissertation - collaborations:

- Garg Aayush, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. *"Cerebro: Static subsuming mutant selection." IEEE Transactions on Software Engineering* 49, no. 1 (2022): 24-43.
- Ma Wei, Thomas Laurent, Miloš Ojdanić, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. *"Commit-aware mutation testing." In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 394-405. IEEE, 2020.

Services:

- Artifact Evaluation Review Committee Member, ISSTA, *The 32nd ACM*

SIGSOFT International Symposium on Software Testing and Analysis, 2023;

- Shadow Reviewer, ECRTS, *The 35th Euromicro Conference on Real-Time Systems*, 2023;
- Review Committee Member, EISEJ, *eInformatica Software Engineering Journal*, 2023;
- Shadow Reviewer, MSR, *The 19th International Conference on Mining Software Repositories*, 2022;
- Student Volunteer Chair, ICSME, *The 37th International Conference on Software Maintenance and Evolution*, 2022;

Bibliography

[1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[2] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

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

[4] M. Ojdanic, "Systematic literature review of safety-related challenges for autonomous systems in safety-critical applications," Master's thesis, Mälardalen University, School of Innovation, Design and Engineering, 2019.

[5] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation analysis.," tech. rep., Georgia Inst of Tech Atlanta School of Information And Computer Science, 1979.

[6] A. Offutt, "The coupling effect: fact or fiction," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131–140, 1989.

[7] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, pp. 121–130, IEEE Computer Society, 2010.

[8] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," *Advances in Computers*, vol. 112, pp. 275–378, 2019.

[9] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pp. 597–608, 2017.

[10] R. Demillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, pp. 34 – 41, 05 1978.

[11] F. Carlos, M. Papadakis, V. Durelli, and E. M. Delamaro, "Test data generation techniques for mutation testing: A systematic mapping," in *Workshop on Experimental Software Engineering (ESELAW'14)*, 2014.

[12] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[13] Y. Lou, D. Hao, and L. Zhang, "Mutation-based test-case prioritization in software evolution," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 46–57, IEEE, 2015.

[14] 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.

[15] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[16] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *2010 Third International Conference on Software Testing, Verification and Validation*, pp. 65–74, IEEE, 2010.

[17] V. Debroy and W. E. Wong, "Combining mutation and fault localization for automated program debugging," *Journal of Systems and Software*, vol. 90, pp. 45–60, 2014.

[18] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[19] A. Panichella and C. C. Liem, "What are we really testing in mutation testing for machine learning? a critical reflection," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 66–70, IEEE, 2021.

[20] W. Ma, M. Papadakis, A. Tsakmalis, M. Cordy, and Y. L. Traon, "Test selection for deep learning systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 13:1–13:22, 2021.

[21] G. Kim, P. Debois, J. Willis, and J. Humble, *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, 2016.

[22] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Mutation testing in the wild: findings from github," *Empirical Software Engineering*, vol. 27, no. 6, p. 132, 2022.

[23] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, *et al.*, "Manifesto for agile software development," 2001.

[24] P. Delgado-Pérez, I. Habli, S. Gregory, R. Alexander, J. Clark, and I. Medina-Bulo, "Evaluation of mutation testing in a nuclear industry case study," *IEEE Transactions on Reliability*, vol. 67, no. 4, pp. 1406–1419, 2018.

[25] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 787–805, 2012.

[26] G. Petrović and M. Ivanković, "State of mutation testing at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, (New York, NY, USA), p. 163–171, Association for Computing Machinery, 2018.

[27] 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.

[28] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer, "What it would take to use mutation testing in industry—a study at facebook," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 268–277, IEEE, 2021.

[29] M. Papadakis, T. T. Chekam, and Y. L. Traon, "Mutant quality indicators," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*, pp. 32–39, IEEE Computer Society, 2018.

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

[31] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test., Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, 2012.

[32] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pp. 354–365, 2016.

[33] 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*, pp. 220–229, IEEE, 2009.

[34] M. Papadakis, M. E. Delamaro, and Y. L. Traon, "Mitigating the effects of equivalent mutants with mutant classification strategies," *Sci. Comput. Program.*, vol. 95, pp. 298–319, 2014.

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

[36] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?," in *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*, ISSRE '14, (USA), p. 189–200, IEEE Computer Society, 2014.

[37] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, p. 5–20, Jan. 1992.

[38] T. A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[39] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1* (A. Bertolino, G. Canfora, and S. G. Elbaum, eds.), pp. 936–946, IEEE Computer Society, 2015.

[40] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Trans. Software Eng.*, vol. 44, no. 4, pp. 308–333, 2018.

[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 (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[42] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010* (J. Han and T. D. Thu, eds.), pp. 300–309, IEEE Computer Society, 2010.

[43] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pp. 176–185, IEEE, 2014.

[44] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 21–30, IEEE, 2014.

[45] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009. Source Code Analysis and Manipulation, SCAM 2008.

[46] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," ACL '02, (USA), p. 311–318, Association for Computational Linguistics, 2002.

[47] A. Islam and D. Inkpen, "Semantic text similarity using corpus-based word similarity and string similarity," *ACM Trans. Knowl. Discov. Data*, vol. 2, July 2008.

[48] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, (Cambridge, MA, USA), p. 3104–3112, MIT Press, 2014.

[49] P. Koehn, F. J. Och, and D. Marcu, "Statistical phrase-based translation," in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, (USA), p. 48–54, Association for Computational Linguistics, 2003.

[50] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[51] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Learning how to mutate source code from bug-fixes," in *2019 IEEE International conference on software maintenance and evolution (ICSME)*, pp. 301–312, IEEE, 2019.

[52] A. OCHIAI, "Zoogeographical studies on the soleoid fishes found in japan and its neighhouring regions-ii," *NIPPON SUISAN GAKKAISHI*, vol. 22, no. 9, pp. 526–530, 1957.

[53] M. Papadakis, D. Shin, S. Yoo, and D. Bae, "Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 537–548, 2018.

[54] A. Khanfir, A. Koyuncu, M. Papadakis, M. Cordy, T. F. Bissyandé, J. Klein, and Y. Le Traon, "ibir: Bug-report-driven fault injection," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–31, 2023.

[55] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[56] N. E. Gold, D. W. Binkley, M. Harman, S. S. Islam, J. Krinke, and S. Yoo, "Generalized observational slicing for tree-represented modelling languages," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017* (E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, eds.), pp. 547–558, ACM, 2017.

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

[58] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016* (A. Zeller and A. Roychoudhury, eds.), pp. 449–452, ACM, 2016.

[59] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (M. Glinz, G. C. Murphy, and M. Pezzè, eds.), pp. 14–24, IEEE Computer Society, 2012.

[60] R. Degiovanni and M. Papadakis, "μBERT: Mutation testing using pre-trained language models," in *Mutation Workshop at ICST*, IEEE, 2022.

[61] M. Tufano, J. Kimko, S. Wang, C. Watson, G. Bavota, M. Di Penta, and D. Poshyvanyk, "Deepmutation: A neural mutation tool," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE '20, (New York, NY, USA), p. 29–32, Association for Computing Machinery, 2020.

[62] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[63] A. Garg, M. Ojdanic, R. Degiovanni, T. T. Chekam, M. Papadakis, and Y. L. Traon, "Cerebro: Static subsuming mutant selection," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[64] A. Garg, R. Degiovanni, M. Jimenez, M. Cordy, M. Papadakis, and Y. L. Traon, "Learning from what we know: How to perform vulnerability prediction using noisy historical data," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 169, 2022.

[65] "Deepmutation." `https://github.com/micheletufano/DeepMutation`.

[66] "src2abs." `https://github.com/micheletufano/src2abs`.

[67] M. Fowler, "Continuous integration." `https://martinfowler.com/articles/continuousIntegration.html`. Online; accessed 31 March 2021.

[68] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the validity of selective mutation with dominator mutants," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pp. 571–582, 2016.

[69] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pp. 90–99, IEEE, 2010.

[70] 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)*, pp. 92–102, IEEE, 2013.

[71] A. Vargha and H. D. Delaney, “A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong,” *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[72] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?,” in *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA* (G. Roman, W. G. Griswold, and B. Nuseibeh, eds.), pp. 402–411, ACM, 2005.

[73] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps, “The care and feeding of wild-caught mutants,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, p. 511–522, Association for Computing Machinery, 2017.

[74] C. Richter and H. Wehrheim, “Learning realistic mutations: Bug creation for neural bug detectors,” in *in the 15th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2022.

[75] P. Bareiß, B. Souza, M. d’Amorim, and M. Pradel, “Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code,” *arXiv preprint arXiv:2206.01335*, 2022.

[76] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, “Self-supervised bug detection and repair,” in *in the 34th pre-proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[77] J. Patra and M. Pradel, “Semantic bug seeding: A learning-based approach for creating realistic bugs,” ESEC/FSE 2021, (New York, NY, USA), p. 906–918, Association for Computing Machinery, 2021.

[78] M. V. Pour, Z. Li, L. Ma, and H. Hemmati, “A search-based testing framework for deep neural networks of source code embedding,” in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 36–46, IEEE, 2021.

[79] R. G. Hamlet, “Testing programs with the aid of a compiler,” *IEEE transactions on software engineering*, no. 4, pp. 279–290, 1977.

[80] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Program mutation: A new approach to program testing,” *Infotech State of the Art Report, Software Testing*, vol. 2, no. 1979, pp. 107–126, 1979.

[81] A. J. Offutt and J. H. Hayes, “A semantic model of program faults,” in *Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996* (S. J. Zeil and W. Tracz, eds.), pp. 195–200, ACM, 1996.

[82] T. T. Chekam, M. Papadakis, M. Cordy, and Y. L. Traon, “Killing stubborn mutants with symbolic execution,” *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 19:1–19:23, 2021.

[83] M. Böhme and A. Roychoudhury, "Corebench: studying complexity of regression errors," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pp. 105–115, 2014.

[84] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris, "Analysing and comparing the effectiveness of mutation testing tools: A manual study," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 147–156, IEEE, 2016.

[85] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012* (M. P. E. Heimdahl and Z. Su, eds.), pp. 331–341, ACM, 2012.

[86] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898–918, 2019.

[87] D. Mao, L. Chen, and L. Zhang, "An extensive study on cross-project predictive mutation testing," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pp. 160–171, 2019.

[88] M. A. Cachia, M. Micallef, and C. Colombo, "Towards incremental mutation testing," *Electronic Notes in Theoretical Computer Science*, vol. 294, pp. 2–11, 2013. Proceedings of the 2013 Validation Strategies for Software Evolution (VSSE) Workshop.

[89] G. Rothermel and M. J. Harrold, "Selecting tests and identifying test coverage requirements for modified software," in *Proceedings of the 1994 International Symposium on Software Testing and Analysis, ISSTA 1994, Seattle, WA, USA, August 17-19, 1994*, pp. 169–184, 1994.

[90] D. W. Binkley, "Semantics guided regression test cost reduction," *IEEE Trans. Software Eng.*, vol. 23, no. 8, pp. 498–516, 1997.

[91] M. E. Delamaro, J. Maidonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE transactions on software engineering*, vol. 27, no. 3, pp. 228–247, 2001.

[92] D. W. Binkley and M. Harman, "Locating dependence clusters and dependence pollution," in *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pp. 177–186, IEEE Computer Society, 2005.

[93] D. W. Binkley, M. Harman, and J. Krinke, "Empirical study of optimization techniques for massive slicing," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, p. 3, 2007.

[94] D. W. Binkley, N. Gold, M. Harman, S. S. Islam, J. Krinke, and S. Yoo, "ORBS: language-independent program slicing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering,*

(FSE-22), Hong Kong, China, November 16 - 22, 2014 (S. Cheung, A. Orso, and M. D. Storey, eds.), pp. 109–120, ACM, 2014.

[95] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants.," *Softw. Test., Verif. Reliab.*, vol. 9, pp. 233–262, 12 1999.

[96] D. W. Binkley and M. Harman, "A survey of empirical results on program slicing.," *Adv. Comput.*, vol. 62, no. 105178, pp. 105–178, 2004.

[97] M. A. Guimarães, L. Fernandes, M. Ribeiro, M. d'Amorim, and R. Gheyi, "Optimizing mutation testing by discovering dynamic mutant subsumption relations," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 198–208, IEEE, 2020.

[98] D. Qi, A. Roychoudhury, and Z. Liang, "Test generation to expose changes in evolving programs," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pp. 397–406, 2010.

[99] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pp. 257–266, 2010.

[100] 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.

[101] B. H. Smith and L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," *Empirical Software Engineering*, vol. 14, no. 3, pp. 341–369, 2009.

[102] T. Apiwattanapong, R. A. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold, "MATRIX: maintenance-oriented testing requirements identifier and examiner," in *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART 2006), 29-31 August 2006, Windsor, United Kingdom*, pp. 137–146, 2006.

[103] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pp. 218–227, 2008.

[104] R. A. Santelices and M. J. Harrold, "Exploiting program dependencies for scalable multiple-path symbolic execution," in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pp. 195–206, 2010.

[105] R. A. Santelices and M. J. Harrold, "Applying aggressive propagation-based strategies for testing changes," in *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pp. 11–20, 2011.

[106] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pp. 226–237, 2008.

[107] P. D. Marinescu and C. Cadar, "KATCH: high-coverage testing of software patches," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pp. 235–245, 2013.

[108] T. Kuchta, H. Palikareva, and C. Cadar, "Shadow symbolic execution for testing software patches," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, pp. 10:1–10:32, 2018.

[109] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce, "Evaluating non-adequate test-case reduction," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 16–26, IEEE, 2016.

[110] B. Kurtz, P. Ammann, J. Offutt, and M. Kurtz, "Are we there yet? how redundant and equivalent mutants affect determination of test completeness," pp. 142–151, Institute of Electrical and Electronics Engineers Inc., 8 2016.

[111] S. J. Kaufman, R. Featherman, J. Alvin, B. Kurtz, P. Ammann, and R. Just, "Prioritizing mutants to guide mutation testing," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 1743–1754, 2022.

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

[113] W. Ma, T. Titcheu Chekam, M. Papadakis, and M. Harman, "Mudelta: Delta-oriented mutation testing at commit time," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 897–909, 2021.

[114] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014* (S. Cheung, A. Orso, and M. D. Storey, eds.), pp. 654–665, ACM, 2014.

[115] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland* (M. Glinz, G. C. Murphy, and M. Pezzè, eds.), pp. 837–847, IEEE Computer Society, 2012.

[116] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, (New York, NY, USA), p. 437–440, Association for Computing Machinery, 2014.

[117] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020* (T. Cohn, Y. He, and Y. Liu, eds.), vol. EMNLP 2020 of *Findings of ACL*, pp. 1536–1547, Association for Computational Linguistics, 2020.

[118] M. Pradel and K. Sen, "Deepbugs: a learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, 2018.

[119] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[120] "Apache commons." `https://github.com/apache`.

[121] "Fasterxml jackson." `https://github.com/FasterXML/jackson`.

[122] "Defects4j issue- 353." `https://github.com/rjust/defects4j/issues/353`.

[123] "Seplemantary data - webpage." `https://mutationtesting-user.github.io/bugs_vs_mutants`.

[124] "Master branch deepmutation." `https://github.com/micheletufano/DeepMutation/commit/a20882d8fbd107762e2d40f5742d838242dbf1e5`.

[125] G. Qian, S. Sural, Y. Gu, and S. Pramanik, "Similarity between euclidean and cosine angle distance for nearest neighbor queries," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, SAC '04, (New York, NY, USA), p. 1232–1237, Association for Computing Machinery, 2004.

[126] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, "Using of jaccard coefficient for keywords similarity," in *Proceedings of the international multiconference of engineers and computer scientists*, vol. 1, pp. 380–384, 2013.

[127] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Software Eng.*, vol. 32, no. 8, pp. 608–624, 2006.

[128] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, (Washington, DC, USA), p. 100–107, IEEE Computer Society Press, 1993.

[129] M. Marcozzi, S. Bardin, N. Kosmatov, M. Papadakis, V. Prevosto, and L. Correnson, "Time to clean your test objectives," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018* (M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, eds.), pp. 456–467, ACM, 2018.

[130] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Regression tests to expose change interaction errors," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pp. 334–344, 2013.

[131] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 192–201, IEEE, 2013.

[132] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

[133] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 523–534, 2016.

[134] T. T. Chekam, M. Papadakis, and Y. L. Traon, "Mart: a mutant generation tool for LLVM," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pp. 1080–1084, 2019.

[135] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (USA), p. 209–224, USENIX Association, 2008.

[136] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pp. 313–324, 2014.

[137] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pp. 549–552, 2007.

[138] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, "Comparing mutation testing at the levels of source code and compiler intermediate representation," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pp. 114–124, 2019.

[139] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. 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 2017, Tokyo, Japan, March 13-17, 2017*, pp. 430–435, IEEE Computer Society, 2017.

[140] R. Gheyi, M. Ribeiro, B. Souza, M. A. Guimarães, L. Fernandes, M. d'Amorim, V. Alves, L. Teixeira, and B. Fonseca, "Identifying method-level mutation subsumption relations using Z3," *Inf. Softw. Technol.*, vol. 132, p. 106496, 2021.

[141] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

[142] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 371–396, 2015.

[143] C. Fang, Z. Chen, and B. Xu, "Comparing logic coverage criteria on test case prioritization," *Science China Information Sciences*, vol. 55, no. 12, pp. 2826–2840, 2012.

[144] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pp. 179–188, IEEE, 1999.

[145] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco, "Assessing transition-based test selection algorithms at google," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, pp. 101–110, 2019.

[146] T. T. Chekam, M. Papadakis, and Y. L. Traon, "Muteria: An extensible and flexible multi-criteria software testing framework," in *In AST '20: International Conference on Automation of Software Test (AST '20), October 7–8, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 4 pages*, 2020.

[147] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," ISSTA 2019, (New York, NY, USA), p. 101–111, Association for Computing Machinery, 2019.

[148] W. Ma, T. Laurent, M. Ojdanić, T. T. Chekam, A. Ventresque, and M. Papadakis, "Commit-aware mutation testing," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 394–405, IEEE, 2020.

[149] M. Kintis, M. Papadakis, and N. Malevris, "Isolating first order equivalent mutants via second order mutation," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012* (G. Antoniol, A. Bertolino, and Y. Labiche, eds.), pp. 701–710, IEEE Computer Society, 2012.

[150] M. Kintis, M. Papadakis, and N. Malevris, "Employing second-order mutation for isolating first-order equivalent mutants," *Softw. Test. Verification Reliab.*, vol. 25, no. 5-7, pp. 508–535, 2015.

[151] "." `https://docs.oracle.com/javase/7/docs/api/java/io/Reader.html`. [Online; accessed 10-December-2021].

[152] 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, pp. 1–31, 2022.

[153] L. Myers and M. J. Sirois, "Spearman correlation coefficients, differences between," *Encyclopedia of statistical sciences*, vol. 12, 2004.

[154] T. Laurent and A. Ventresque, "Pit-hom: an extension of pitest for higher order mutation analysis," in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 83–89, 2019.

[155] M. Ojdanic, E. Soremekun, R. Degiovanni, M. Papadakis, and Y. Le Traon, "Mutation testing in evolving systems: Studying the relevance of mutants to code evolution," *ACM Trans. Softw. Eng. Methodol.*, apr 2022. Just Accepted.

[156] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 649 – 678, September–October 2011.

[157] Git, "Git-diff," Sept. 2022.

[158] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, "Mdroid+: A mutation testing framework for Android," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 33–36, IEEE, 2018.

[159] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019* (D. Zhang and A. Møller, eds.), pp. 112–122, ACM, 2019.

[160] W. Afzal and R. Torkar, "On the application of genetic programming for software engineering predictive modeling: A systematic review," *Expert Systems Applications*, vol. 38, no. 9, pp. 11984–11997, 2011.

[161] M. Harman, E. Burke, J. A. Clark, and X. Yao, "Dynamic adaptive search based software engineering (keynote paper)," in 6^{th} *IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2012)*, (Lund, Sweden), pp. 1–8, September 2012.

[162] D. Jackson and E. J. Rollins, "A new model of program dependences for reverse engineering," in *Symposium on the Foundations of Software Engineering (FSE '94)*, pp. 2–10, Dec. 1994.

[163] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[164] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "SapFix: Automated end-to-end repair at scale," in *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*, (Montreal, Canada), 2019.

[165] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: a comprehensive survey," *IEEE Transactions on Evolutionary Computation*, vol. 22, pp. 415–432, June 2018.

[166] W. B. Langdon and M. Harman, "Evolving a CUDA kernel from an nVidia template," in *2010 IEEE World Congress on Computational Intelligence* (P. Sobrevilla, ed.), (Barcelona), pp. 2376–2383, IEEE, 18-23 July 2010.

[167] M. Papadakis, C. Henard, and Y. L. Traon, "Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing," in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pp. 1–10, IEEE Computer Society, 2014.

[168] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pp. 75–87, ACM, 2020.

[169] R. Natella, D. Cotroneo, J. Durães, and H. Madeira, "On fault representativeness of software fault injection," *IEEE Trans. Software Eng.*, vol. 39, no. 1, pp. 80–96, 2013.

[170] J. Arlat, A. Costes, Y. Crouzet, J. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Trans. Computers*, vol. 42, no. 8, pp. 913–923, 1993.

[171] J. Christmansson and R. Chillarege, "Generation of error set that emulates software faults based on field data," in *Digest of Papers: FTCS-26, The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing, 1996*, pp. 304–313, IEEE Computer Society, 1996.

[172] Y. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Softw. Test. Verification Reliab.*, vol. 15, no. 2, pp. 97–133, 2005.

[173] D. Britz, A. Goldie, M.-T. Luong, and Q. Le, "Massive exploration of neural machine translation architectures," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, (Copenhagen, Denmark), pp. 1442–1451, Association for Computational Linguistics, Sept. 2017.

[174] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), p. 1–10, Association for Computing Machinery, 2011.

[175] B. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure*, vol. 405, no. 2, pp. 442 – 451, 1975.

[176] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI open*, vol. 1, pp. 57–81, 2020.