UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2023-031
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 21/03/2023 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Ahmed KHANFIR
Born on 3rd December 1989 in Tunis, Tunisia

TARGETED, REALISTIC AND NATURAL FAULT INJECTION
(USING BUG REPORTS AND GENERATIVE LANGUAGE MODELS)

To

my mother & my wife

To

my family

Acknowledgements

Firstly, I would like to thank my supervisor, Prof. Yves Le Traon, and my co-supervisor, Prof. Mike Papadakis, for their trust, flexibility and support, throughout this thesis; from the day I was interviewed for the PhD student position, until today. I would like also to thank them for giving me the chance and encouraging me to work on side projects and collaborate with other researchers, including co-supervising master students, which improved my skills and deepened my knowledge in different domains.

I am particularly grateful to my daily supervisor, Prof. Mike Papadakis, for his guidance and assistance. He played a major training role during this thesis period and had a large and valuable contribution in every phase of it, including; proposing the research topic and ideas, asking pertinent research questions, designing the experimental setup, analyzing and interpreting the results, writing papers and presenting research work.

I would like to thank all of my co-authors for their feedback, efforts and support which helped to achieve this thesis. I would like also to thank my colleagues for the great atmosphere and good discussions we had (I am not talking about the COVID-19 pandemic lockdown period, of course).

I am very grateful also to the jury members for their time and effort invested in my dissertation. I am also thankful to the National Research Fund of Luxembourg (FNR - TestFast Project, ref. 12630949) for funding my thesis.

Finally, I would like to thank my wife, Enjie Ghorbel, for her constant encouragement and support and particularly for her patience during this PhD. I thank also my son Habib, who joined us during this thesis, for enlightening our house and our lives. I would like also to thank my parents and my brother, Leila, Jamel and Youssef, who have supported and assisted me, ever since I can remember. Many thanks also to my family-in-law, Molka, Faouzi and Emna for their encouragement. I would like also to thank my friends and family members who made this journey enjoyable.

Index

List of Abbreviations

TAC	Test Adequacy Criterion
TS	Test Suite
PUT	Program Under Test
AST	Abstract Syntax Tree
FD	Fault Detection
BR	Bug Report
FL	Fault Localisation
IRFL	Information Retrieval Fault Localisation
APR	Automated Program Repair
ML	Machine Learning
NMT	Neural Machine Translation
NL	Natural Language
PL	Programming Language
MLM	Masked Language Modeling

JP	Java Parser
UTF8	Universal Character Set Transformation Format - 8 bits
KN	Kneser Ney
MKN	Modified Kneser Ney
SD	Standard Deviation
JVM	Java Virtual Machine
conv	conventional
cos	cosine
conf	confidence
acc	accuracy
min	minimum
max	maximum
SE	Software Engineering
RQ	Research Question
SOA	State Of the Art
w.r.t.	with regards to
aka.	also known as
i.e.	in example
e.g.	example general

List of Figures

List of Tables

Abstract

Artificial faults have been proven useful to ensure software quality, enabling the simulation of its behaviour in erroneous situations, and thereby evaluating its robustness and its impact on the surrounding components in the presence of faults. Similarly, by introducing these faults in the testing phase, they can serve as a proxy to measure the fault revelation and thoroughness of current test suites, and provide developers with testing objectives, as writing tests to detect them helps reveal and prevent eventual similar real ones. This approach – mutation testing – has gained increasing fame and interest among researchers and practitioners since its appearance in the 1970s, and operates typically by introducing small syntactic transformations (using mutation operators) to the target program, aiming at producing multiple faulty versions of it (mutants).These operators are generally created based on the grammar rules of the target programming language and then tuned through empirical studies in order to reduce the redundancy and noise among the induced mutants.

Having limited knowledge of the program context or the relevant locations to mutate, these patterns are applied in a brute-force manner on the full code base of the program, producing numerous mutants and overwhelming the developers with a costly overhead of test executions and mutants analysis efforts. For this reason, although proven useful in multiple software engineering applications, the adoption of mutation testing remains limited in practice.

Another key challenge of mutation testing is the misrepresentation of real bugs by the induced artificial faults. Indeed, this can make the results of any relying application questionable or inaccurate. To tackle this challenge, researchers have proposed new fault-seeding techniques that aim at mimicking real faults. To achieve this, they suggest leveraging the knowledge base of previous faults to inject new ones. Although these techniques produce promising results, they do not solve the high-cost issue or even exacerbate it by generating more mutants with their extended patterns set.

Along the same lines of research, we start addressing the aforementioned challenges – regarding the cost of the injection campaign and the representativeness of the artificial faults – by proposing IBIR; a targeted fault injection which aims at mimicking real faulty behaviours. To do so, IBIR uses information retrieved from bug reports (to select relevant code locations to mutate) and fault patterns created by inverting fix patterns, which have been introduced and tuned based on real bug fixes mined from different repositories. We implemented this approach, and showed that it outperforms the fault injection performed by traditional mutation testing in terms of semantic similarity with the originally targeted fault (described in the bug report), when applied at either project or class levels of granularity, and provides better, statistically significant, estimations of test effectiveness (fault detection). Additionally, when injecting only 10 faults, IBIR couples with more real bugs than mutation testing even when injecting 1000 faults.

Although effective in emulating real faults, IBIR's approach depends strongly on the quality and existence of bug reports, which when absent can reduce its performance to that of traditional mutation testing approaches. In the absence of such prior and with the same objective of injecting few relevant faults, we suggest accounting for the project's context and the actual developer's code distribution to generate more "natural" mutants, in a sense where they are understandable and more likely to occur.

To this end, we propose the usage of code from real programs as a knowledge base to inject faults instead of the language grammar or previous bugs knowledge, such as bug reports and bug fixes. Particularly, we leverage the code knowledge and capability of pre-trained generative language models (i.e. CodeBERT) in capturing the code context and predicting developer-like code alternatives, to produce few faults in diverse locations of the input program. This way the approach development and maintenance does not require any major effort, such as creating or inferring fault patterns or training a model to learn how to inject faults. In fact, to inject relevant faults in a given program, our approach masks tokens (one at a time) from its code base and uses the model to predict them, then considers the inaccurate predictions as probable developer-like mistakes, forming the output mutants set. Our results show that these mutants induce test suites with higher fault detection capability, in terms of effectiveness and cost-efficiency than conventional mutation testing.

Next, we turn our interest to the code comprehension of pre-trained language models, particularly their capability in capturing the naturalness aspect of code. This measure has been proven very useful to distinguish unusual code which can be a symptom of code smell, low readability, bugginess, bug-proneness, etc, thereby indicating relevant locations requiring prior attention from developers. Code naturalness is typically predicted using statistical language models like n-gram, to approximate how surprising a piece of code is, based on the fact that code, in small snippets, is repetitive. Although powerful, training such models on a large code corpus can be tedious, time-consuming and sensitive to code patterns (and practices) encountered during training. Consequently, these models are often trained on a small corpus and thus only estimate the language naturalness relative to a specific style of programming or type of project.

To overcome these issues, we propose the use of pre-trained generative language models to infer code naturalness. Thus, we suggest inferring naturalness by masking (omitting) code tokens, one at a time, of code sequences, and checking the models' ability to predict them. We implement this workflow, named CODEBERT-NT, and evaluate its capability to prioritize buggy lines over non-buggy ones when ranking code based on its naturalness. Our results show that our approach outperforms both, random-uniform and complexity-based ranking techniques, and yields comparable results to the n-gram models, although trained in an intra-project fashion.

Finally, we provide the implementation of tools and libraries enabling the code naturalness measuring and fault injection by the different approaches and provide the required resources to compare their effectiveness in emulating real faults and guiding the testing towards higher fault detection techniques. This includes the source code of our proposed approaches and replication packages of our conducted studies.

Chapter 1

Introduction

We introduce in this chapter the general context and challenges of our work. Particularly, we present the main use cases of fault injection and mutation testing, together with their main application challenges, which we tackle in our dissertation. Finally, we give a summary of our contributions and the dissertation organization.

1.1 Software Testing and Fault injection

Software is playing an essential role in diverse domains, such as finance, health, transportation, etc. This broad adoption made multiple tasks dependent on it, and thus, at risk of outage or malfunctioning whenever the software quality is compromised. Therefore, as in all engineering fields, software engineering practitioners employ mechanisms to ensure software quality and validity.

To assess the reliability and robustness of software as well as to prevent severe failures from happening in real situations, practitioners simulate software execution in faulty or unexpected circumstances [1]. To this end, they inject artificial faults and evaluate how they impact the software and its surrounding environment. Based on how the software handles those faults and how they impact its dependencies and dependents, we can approximate its dependability, robustness and fault tolerance [2], in similar real-world scenarios. This practice is known as Software Fault Injection [2].

Another commonly employed activity to ensure software quality is testing, which aims at validating program behaviour and finding faults early in the development cycle. Testing can be summarized in a nutshell as the activity of executing a program (or part of it) within a given scenario and assessing that it behaves correctly [3], [4], i.e., it returns the expected output when given a specific input. Consequently, as we validate the software through its tested behaviour, its end quality depends on the thoroughness of those tests (test suites). It entails that thoroughly tested software is more trustworthy regarding its behaviour than poorly tested ones. As it is tedious and often impossible to consider every possible scenario, developers tend to focus on testing the most probable use cases or on some key business components of the program. This may lead to the negligence of corner cases, allowing eventual faults to bypass the testing phase, consequently producing a faulty program.

To ensure the quality and thoroughness of test suites, researchers have proposed test adequacy criteria (TAC) [5], [6], such as measuring the portion of code covered (executed) when the tests are being executed. Among these TAC, the usage of artificial faults as a testing objective has been proven effective in guiding testing towards higher fault detection capabilities; guiding the production of test suites that reveal more faults [6]–[8]. This approach is known as mutation testing [9]. It aims at seeding faults in the program under test that would simulate eventual faults and code mistakes, by applying simple syntactic transformations to its code. These mutated versions of the program (mutants) are used to approximate the strength of the test suite based on its ability to detect a difference between the original and the modified version of the program. This way we can attribute a fault permeability score (mutation score) to a test suite which is computed by the ratio between the number of detected mutants and the total generated ones. Intuitively, the not detected (not killed) mutants can also serve as indicators of the not covered scenarios or components of the program, guiding this way the developers in improving their test suites. This is done by analyzing those mutants and designing test cases that reveal them, thereby preventing and revealing real bugs.

Artificial faults have also been proven useful in multiple other adjacent software engineering tasks, like comparing bug detection capabilities, bug fixing, debugging, fault localization, test generation, etc [9]. For instance, a bibliometric analysis performed in 2016 [10] found that fault injection is used in around 19% of all software testing studies published in major software engineering conferences.

In this dissertation, we address challenges that concern fault injection in general, with a particular focus on mutation testing.

1.2 Challenges

Although proven very useful in multiple applications, the adoption of mutation testing in practice remains limited [11], [12]. This is mainly because of its high machine and human costs. In fact, mutation testing tends to produce many mutants, against which tests need to be executed and then manually analysed if they are not killed.

To inject faults into a program, mutation testing operates typically by changing its source code, and creating new faulty versions of it [13]. Considering the infinite number of possible modifications that one could apply to a program, one key challenge is to limit the number of generated mutants, in a manner that enables its practical usability without hindering intolerably its efficiency, i.e. introducing few productive mutants that provoke relevant faulty behaviours. Intuitively, if we consider a simple statement involving the addition of integers $b = a + 100$, just by replacing the integer 100 with another value, we may introduce as many mutants as the number of valid integers allowed by the target language. This large number of mutants would cause a huge consequent computational and manual cost to the practitioners, considering the effort to spend in executing tests and analyzing mutants [9], [14], [15]. When applied to large programs, this would hinder further the usability of the approach, or even make it inconceivable [16].

Another problem with mutation testing is the equivalent mutant problem [17], [18], i.e., the fact that mutants may be functionally equivalent to the original program, despite being syntactically different. Equivalent mutants introduce issues by wasting computational resources, they are executed with tests without these being able to kill them, by obscuring mutation score measurements [18], and by making developers lose time in analysing them [17], [19].

To deal with these problems, since the early days of mutation testing, mutant generation has been limited to simple syntactic transformations in order to keep their number to a relatively small set of mutants [3], [9]. To this end, studies on the mutant coupling [20] have shown that test suites detecting all simple mutants are also capable of detecting almost all complex ones, thereby establishing that simple faults are a good approximation of the test suites' strengths.

The set of mutation operators has been further tuned through empirical studies aiming at reducing the noise and redundancy among the generated mutants. The most prominent results in this direction are the set of five operators proposed by Offut et al. [21] which ended up being incorporated in most mutation testing tools [22]. Although this filtering process limits the number of mutants, it still results in excessive numbers of mutants, the majority of which are of low utility [10], [23]. This is mainly due to the fact that mutation operators are applied in a brute-force manner, i.e., regardless of the target code context or the relevant locations to mutate. Additionally, limiting the mutation operators causes the restriction of the type of faults that can be injected, and thus, the decrease of the overall fault injection efficiency in terms of diversity and real faults representativeness.

To summarize, fault injection challenges mainly originate from the difficulty of automatically generating a few faults, that are relevant and productive in any use case and given any project [13]. In the remainder of this section, we present the challenges that we identified and addressed in this dissertation.

1.2.1 Targeted Fault Injection

To reduce the number of candidate faults, researchers have proposed to restrict the injection to relevant locations, depending on the target applications. For instance, to ameliorate the usability of mutation testing in continuously evolving projects (i.e. Continuous Integration [24] driven projects), researchers have proposed to target the code locations that are impacted by the commit changes [25], [26], instead of reconducting the whole mutation campaign. This, of course, saves large efforts of rerunning the tests or reanalysing the same mutants, on every new version while only a smaller part of the project has been changed.

Similarly, one could target the injection on a specific feature or component of the project. To do so, the code related to this specific feature must be separated from the rest of the program, before being able to apply any fault injection. Considering that some components' implementations and invocations could be spread on multiple places or files, particularly on large programs, the task of selecting the relevant locations to target can become harder, even inconceivable or discouraging by itself, requiring considerable knowledge of the project implementation.

1.2.2 Realistic Faults

One of the challenges of fault injection techniques, i.e. mutation testing, is to inject faults that mimic the behaviour of real bugs [27], [28]. This aspect of similarity with real bugs is very important, especially considering the fact that many software engineering tasks rely on fault injection techniques [9]. Meaning that these artificial-faults-based applications can be misled if they consider faults that do not represent real bugs.

For instance, mutation testing operates typically by making changes in the target code by applying a set of mutation operators (a set of simple syntactic transformations) [29]. These operators definition is based on the language grammar [3] and are applied randomly in a brute-force manner on the entire program code, introducing a numerous amount of mutants, among which some are eventually realistic faults. This means that the approach is agnostic towards the fault types and locations of the injection.

Recent research in the field has proposed enhanced techniques of mining fault patterns [27], [30] to introduce some form of realism in the injected faults with regard to real ones. These results are encouraging and show the impact of the patterns in bringing realism to the injection by answering the ”what” to inject, however, these approaches lack control over the locations ”where” to inject. Considering this limitation, the approaches would inject faults everywhere in the program code-base and lead to comparable results as conventional techniques: 1) a large number of faults inducing a very high cost with 2) probably a small ratio of faults that couple with real ones [7].

Ideally, one would use real faults mined from project repositories or create manually targeted faults to assess or improve its testing technique, i.e. comparing and improving the fault detection capability of fuzzing techniques based on real defects. Although this solution brings some realism, it is tedious and labour-intensive. Particularly, because older buggy versions have often different tests, code and dependencies than the current version, and thus cannot be used as-are in most cases, but instead, the bugs need to be isolated in their occurring versions and then reintroduced in the current one. This explains also why it is difficult to collect and create a real, diverse and reliable bug dataset, as well as, why artificial faults form a convenient alternative to real ones in performing controlled studies [10].

1.2.3 Effective Context-aware Mutation Testing

Among the purposes of fault-driven applications is to enable practitioners to account for and prevent eventual bugs, through artificial ones [2], [9]. For instance, mutation testing analysis consists of introducing faults in a program that developers can target to write and improve their tests. Therefore, these faults should be convenient and understandable by developers to identify their cause and faulty behaviour, and thereby be able to avoid them by strengthening the tests or even changing the program's code.

For instance, in this regard, mutation testing tends to rely on mutation operators that apply small syntactic transformations of code [3], providing small changes to examine by developers. However, such approaches still introduce faults, among which many are perceived as unnatural in the sense that they are unlikely to be introduced by developers, i.e. non-conforming to common coding conventions and practices [16], [31]. This becomes further inevitable when enlarging the mutation possibilities beyond the conventional operator's ones, as more mutants with eventually larger changes will be introduced.

Moreover, considering the large diversity of projects, introducing automatically natural, interesting and engaging faults for developers, i.e. developer-like mistakes, is a difficult task as some mutations can be relevant in some contexts while not in others. Intuitively, a developer can consider a mutant generated by flipping a + to - relevant in some cases or locations of the program, while discarding similar mutants in others.

This is among the reasons that motivated research in proposing fault injection approaches that account for the project context and semantics at hand [32]–[34], instead of just checking whether an operator can be applied or not based on the target code's AST. For instance, to emulate real faults Tufano et al. [32] proposed the use of an NMT-based technique that mutates a given code in a context-preserving manner while applying changes learned from real mined bugs. Precisely, the proposed approach takes as input the code of a target method (function) and outputs a new modified version of it, which is eventually faulty. Although the approach favours the mutants that are the closest syntactically (text distance) to the original code, the diff between the original and the mutated code remains relatively important, as the approach introduces changes in the whole method. This can cause extra overhead for developers to analyse the mutants and distinguish the root cause of the fault from the irrelevant noisy changes introduced by the approach. Moreover, it is not clear whether favouring the syntactically similar mutants over the others can bring any advantage to the mutation testing campaign's effectiveness in terms of emulating or revealing faults [35].

Overall, it is challenging to capture any 1) given code's context, 2) adapt the mutations to it and 3) automatically generate valid (compilable) code but faulty, 4) which is judged relevant by developers and 5) induces the writing (or generation) of thorough test suites.

1.2.4 Generic Code Naturalness Measure

Much of software-engineering research relies on the naturalness of code [36], the fact that code, in small snippets, is repetitive. Considering code as recurrent sequences of words, the naturalness of a given sequence is typically estimated from its likelihood to occur [37]. This notion of naturalness can be very powerful as it helps identify code discrepancies, like detecting unusual and eventually bad programming practices, which are often considered as symptoms of bug-proneness and bugginess [38]. Hence, it can indicate relevant code locations, requiring prior attention from developers.

Typically, language statistical models – like n-gram ones – are used to estimate code naturalness, and thus expressing how surprising a sequence of code is w.r.t the code seen during its training. As training such models on large code corpus can be hard and costly, their training is often limited to smaller ones, which results in introducing models that estimate the naturalness relative to a particular style of programming or type of project [36]. Additionally, their sensitivity to code representation can lead to an accumulation of models, while not being applicable in a cross-project context. For instance, to estimate the naturalness of a given code file, we would typically use an n-gram model that has been trained using the other files from the same project [39], thereby measuring its relative naturalness w.r.t the rest of the project.

1.3 Contributions and Dissertation Organization

In this section, we start by presenting our contributions to address the aforementioned challenges. Next, we explain how the remainder of this dissertation is organized.

1.3.1 Contributions

Leveraging Bug Reports and Automated Program Repair Patterns for a Targeted and Realistic Fault Injection.

We propose introducing realism into the fault injection via targeting reported bugs. More precisely, since bug reports have often relevant information for multiple tasks such as debugging, automatically localising and repairing faults [40]–[42], we leverage this information to guide our injection. Associated with fault patterns collected through the examination of real faults, we can target the injection towards specific features of the program, producing faults that emulate real buggy behaviours. In addition, the use of bug reports makes it possible to inject targeted faults without any deep knowledge of the target system, source code or tests. We implemented this approach, which we name iBiR [43], and assessed its fault injection performance when targeting real bug reports. iBiR outperforms significantly conventional mutation testing in terms of injecting faults that semantically resemble and couple with real ones, in the vast majority of the cases.

This approach allows researchers and practitioners to construct sets of artificial faults that resemble specific bugs or target particular features from the code. However, as the injection approach depends on input bug reports, its performance can be impacted by the quality of these ones and inherits all the information retrieval limitations (because of its IRFL component). Moreover, the approach may also be useless in new projects, i.e. where no bugs have already been reported, or in use cases not targeting particular bugs or bug reports. Intuitively, in such situations, one could use the Mutator component of the approach without the IRFL, by injecting faults in all locations in a brute-force manner or eventually with a limited budget per location or pattern (i.e. maximum 5 mutants per line or method or pattern, etc.). However, this method may lead to similar results as most of the traditional mutation techniques, with an overwhelming number of faults to analyse for the end-user. Indeed, the number of injected faults would be even more important than the one injected by traditional mutation testing techniques due to the newly introduced fault patterns by iBiR.

Therefore, we investigate other possibilities to tackle the challenge of injecting a few relevant mutants, without targeting any particular bug [44]. For instance, with the absence of any specific input i.e. a bug report, we believe that the injection should be focused on the code locations that are the most relevant to review and check by the practitioners, like the code locations that are responsible for the business logic of the program. Additionally, to decrease the injection campaign cost and better serve mutation analysis applications, we should reduce the number of injected faults per code location instead of blindly applying all possible operators on the whole project codebase, i.e. favouring ”natural” or likely-to-occur mutations over others. Therefore, we turned our interest to the usage of generative language models to inject few faults in diverse business-logic-responsible locations from the target project.

Context-aware and Natural Mutation Testing via Pre-Trained Language Models

Aiming at seeding faults that are both “natural” in a sense easily understood by developers and strong (have high chances to reveal faults), we propose using pre-trained generative language models (i.e. CodeBERT) that have the ability to produce developer-like code. This way, when inaccurate, the model would provide developer-like mistakes, introducing probable alterations of code. Hence, the model has the ability to seed natural faults, thereby offering opportunities to perform mutation testing.

Additionally, the usage of actual code from real programs as a knowledge base to inject faults, instead of the language grammar or previous bugs knowledge, makes the approach more flexible and adaptable to the target project. Meaning that it would not produce the same transformations, following predefined patterns regardless of the given project, but rather the fittest mutants to the context of the input code to mutate. Moreover, as the approach uses already trained models, its development and maintenance do not require any major effort, such as creating or inferring fault patterns or training a model to learn how to inject faults.

We realise this idea by implementing μBERT [45], [46], a mutation testing technique that performs mutation testing using CodeBERT and empirically evaluated it using real faulty program versions. Our results show that μBERT is more effective and cost-efficient in guiding the testing towards fault detection than state-of-the-art mutation testing (Pitest), yielding test suites with an average of up to 17% higher fault detection capabilities.

Next, we turned our attention to investigating whether we could distinguish relevant locations to review and check by developers, without any given prior, like a target bug report. Therefore, we investigate whether and how we could measure code naturalness through pre-trained language models, thereby distinguishing locations that are ”unnatural” and thus more likely buggy, bug-prone, etc.

Generic Code Naturalness Measure via Pre-trained Language Models

To overcome the drawbacks of the n-gram-based naturalness metric, we suggest leveraging the knowledge gained by pre-trained language models (i.e. CodeBERT) during their training on large-scale corpora of projects, to infer the code naturalness.

This offers the possibility to obtain a more generic naturalness metric, in the sense that it is not specific to a particular project or any narrow scope while saving the training efforts and bypassing the limitations of n-gram models.

We implemented this approach, named CodeBERT-nt [47], in which we incorporated CodeBERT [48], a large pre-trained language model. To attribute a naturalness score to a code sequence, CodeBERT-nt omits its composing nodes (one at a time), then asks the model to predict them and finally infers the naturalness from the variation in the prediction performance. To this end, it computes three metrics for every prediction, which are the exact match of the prediction with the original code, the embeddings similarity between the original and the predicted code and the confidence of the model when overtaking the task.

We empirically assessed CodeBERT-nt capability in prioritizing buggy lines over non-buggy ones when ranking source code based on its naturalness. We found that the unnaturalness is best reflected by the confidence decrease of the model. Moreover, our results show that it outperforms both, random-uniform and complexity-based ranking techniques, and yields comparable results to n-gram models, although trained in an intra-project fashion.

1.3.2 Dissertation Organization

The remainder of this dissertation is organised as the following: in Chapter 2, we present an overview of the background and concepts used in our work. Next, Chapter 3 discusses previous related works. In Chapter 4, we present and evaluate iBiR, a bug-report-driven fault injection approach. Chapter 5 presents and evaluates μBert, our proposed mutation testing approach that is based on pre-trained language model predictions. In Chapter 6, we present and evaluate CodeBERT-nt, our proposed approach to measure code naturalness through pre-trained language model predictions.

Chapter 2

Background

In this section, we present the general background, definitions and concepts employed in this dissertation.

2.1 Mutation Testing

Mutation testing is an established fault-based testing technique [9]. It operates by introducing artificial faults into a program, thereby creating many different versions of it (named *mutants*). The artificial faults are injected through syntactic changes to all program locations in the original program, based on predefined rules named *mutation operators* [3]. Such operators can, for instance, invert relational operators (e.g., replacing $\geq$ with $<$).

Mutants can be used to indicate the strengths of test suites, based on their ability to distinguish the original program from the mutated versions (mutants). A mutant is said to be *killed* if any of the tests fail (aka. distinguishes the mutant from the original program) when executed on it, otherwise, it is said to be *live* or *survived*. Some mutants cannot be killed as they are functionally *equivalent* to the original program, otherwise, they are said to be *killable*.

The *mutation score* measures the test suite adequacy and is equal to the percentage of killed mutants among all the generated ones. The not killed mutants serve as *test objectives* (criterion) for developers to improve the fault detection capabilities of their test suites. To do so, developers analyse those mutants, extract the not covered faulty behaviours and write tests to reveal them, thereby *killing* these mutants. This way, and based on the principle of ”the more mutants a test suite kills, the more faults it can find”, mutation testing can both: 1) assess and 2) guide testing towards higher fault detection capabilities.

Aiming at improving the efficiency and reducing the cost of mutation testing, researchers defined some mutant categories to help identify the ones that are more relevant than others. Those categories are mainly defined depending on 1) the relation of a mutant with the original code, 2) the relation between a mutant and other generated mutants and 3) the tests that they fail or pass. For instance, automatically generating mutants is deemed to generate almost inevitably *redundant* mutants, which can be *duplicates* [18] – meaning equivalent between each other – or *equivalent* [18], [49] to the original program. Researchers try to reduce and avoid the generation of such mutants as they cause an additional unnecessary effort to the mutation campaign while falsifying the overall tests adequacy measure, i.e. inflating or deflating the number of killed mutants.

In addition, it is preferable to produce *hard-to-kill* [50] mutants than *trivial* (*easy-to-kill*) ones, which are easily killed by many tests. One similar way to categorise whether a mutant is relatively interesting or special among others is by studying the *subsumption* relationship between mutants, based on the results of their execution on the same test suite, particularly the failing tests by each mutant. In this case, the *subsuming* ones are those that when killed by a test suite engender the killing of all remaining (killable) mutants, which are said *subsumed*. We note also that a subsuming mutant can only be subsumed by subsuming mutants. This way, developers can save effort when focusing on killing the subsuming mutants, in order to write test suites that kill all (killable) mutants.

To summarise, mutation testing has two main usages, a) providing guidance for test generation [8], [51], i.e., mutants are the objectives of the test generation process, and b) mutants are used as a test adequacy metric [5], i.e., used to decide when to stop testing. The mutation testing approach is powerful and has been applied for the testing of different software artefacts, such as program specifications [52], [53], program input models [53], [54] and behavioural models [55], [56]. In this thesis, we are focusing on code-based mutation testing and analysis.

2.2 Software Fault Injection (SFI) and Mutation Testing

Software Fault Injection and Mutation Testing (mutation analysis) are two fault-based software quality assurance techniques [2]. Although very similar, as they both involve introducing artificial faults in software, they present few differences due to their different application scopes.

In a nutshell, mutation testing is typically used as a test adequacy criterion, where the faulty induced versions (mutants) are used to evaluate the thoroughness and the fault permeability of a test suite, as well as to guide the testing towards higher fault detection capabilities [9]. In a sense where writing tests to kill those mutants will allow preventing, detecting and discovering similar real bugs. Whereas, Software Fault Injection is typically applied later in the process, meaning after testing, for the purpose of assessing software robustness and fault tolerance [28], risk analysis [57], [58] and dependability evaluation [2], [59].

This application difference explains their slightly different functioning, targets and terminologies. Particularly, mutation testing operates by executing tests on mutated versions of the program under test, while SFI executes the program itself under different workloads (i.e. multiple runs of the software with different inputs) and faultloads (i.e. different faults) that can be initially present in the program or submitted later during its execution [2].

Nevertheless, as both approaches rely on artificially introduced faults, they both face similar challenges, including our main goal, which is producing few relevant faults. For instance, among the desired properties of a *relevant* or *productive* fault (mutant) in the context of mutation testing is the fact that it is *killable* but *hard-to-kill*. Meaning that it is not easily detected by any test, but only by specific ones, and thus, *hard-to-kill*. While in Software Fault Injection, a *relevant* fault is preferably a *residual* one, which bypassed the testing checks. Meaning that it is not detected by the current tests. Similarly, such a fault is considered *relevant* when introduced in a mutation testing campaign, as it would engender the writing of new tests and thus, improve the fault detection of the test suite.

2.3 Fault Localisation

Fault localisation (FL) is the activity of identifying the suspected fault locations, where the code should be fixed. Several automated fault localisation approaches have been proposed [60], among which many consist of dynamically analysing the target program to locate the buggy code via instrumentation of passing and failing test executions. For instance, among the most famous FL techniques are the spectrum-based ones, which typically measure the likelihood of a statement to be buggy or not depending on its coverage (i.e. the statement code coverage [61] or the statement's killed mutants [41]) by failing or passing tests. Therefore, most of those approaches require the existence of at least one failing test by the bug among the given test suite, which will indicate the origin of the target bug. This condition is never met in fault injection applications, which typically function by altering the code of a project that has only passing tests and without any knowledge of existing bugs. Consequently, we consider such approaches inadequate for the purpose of identifying relevant locations to mutate.

Recent research has proposed static-analysis fault localisation approaches which require neither test executions nor failing tests to identify the buggy code, thereby offering possibilities to be used in fault injection campaigns. For instance, fault localisation techniques based on Information Retrieval (IR) [62]–[65] exploit textual bug reports to identify code chunks relevant to the bug, without relying on test cases. Given an input bug report, IR-based fault localisation tools start by parsing it and extracting tokens from it, with which a query is formulated then compared with the collection source code files [40], [66]–[70]. Then, they rank the documents based on their relevance to the query, such that source files ranked higher are more likely to contain the fault. Recently, automated program repair methods have been designed on top of IR-based fault localisation [42]. They achieve comparable performance to methods using spectrum-based fault localisation, yet without relying on the assumption that test cases are available, particularly failing ones.

In Chapter 4, we leverage IR-based fault localisation to achieve a different goal; instead of localising the reported bug, we aim at *injecting faults* at code locations that implement functionality similar to the one described by the bug report.

2.4 Fix Patterns

In automated program repair [71], a common way to generate patches is to apply fix patterns [72] (also named fix templates [73] or program transformation schemes [74]) in suspicious program locations (detected by fault localisation). Patterns used in the literature [72]–[79] have been defined manually or automatically (mined from bug fix datasets).

Instead of fix patterns, iBiR (Chapter 4) uses *fault patterns* that are fix patterns inverted. Since fix patterns were designed using recurrent faults, their related fault patterns introduce them. This helps injecting faults that are similar to those described in the bug reports. iBiR inverts and uses the patterns implemented by *TBar* [80] as we detail in Chapter 4.

2.5 Generative Language Models

Advances in deep learning approaches gave birth to new language models for code generation [81]–[84]. These models are trained on large corpora counting multiple projects, thereby acquiring a decent knowledge of code, enabling them to predict accurately source code to developers. Among these pre-trained models, CodeBERT [84], a language model that has been recently introduced and made openly accessible for researchers by Microsoft.

CodeBERT is an NL-PL bimodal pre-trained language model (Natural Language Programming Language) that supports multiple applications such as code search, code documentation generation, etc. Same as most large pre-trained models, i.e. BERT [85], CodeBERT's developing adopts a Multilayer Transformer [86] architecture. Its training involved a large code and natural language corpus collected from over six million projects available on GitHub, counting 6 different programming languages, including Java. The model was trained in a cross-modal fashion, through bimodal NL-PL data, where the input data is formed by pairs of source code and its related documentation, as well-as unimodal data, including either natural language or programming language sequences per input. This way, it enables the model to offer both – PL and NL-PL – functionalities. The training targets a hybrid objective function, that is based on replaced token detection.

μBERT (Chapter 5) incorporates the Masked Language Modeling (MLM) functionality [48] of CodeBERT in its workflow, to generate “natural” mutants. The CodeBERT MLM pipeline takes as input a code sequence of maximum 512 tokens, including among them one masked as `<mask>`, whose value will be predicted by the model based on the context captured from the remaining tokens. CodeBERT provides by default 5 predictions per token, among which we use the inaccurate and compilable predicted codes as mutants.

Similarely, in Chapter 6, we incorporate the Masked Language Modeling (MLM) functionality [48] of CodeBERT in our experiments pipeline, in order to study the possibility of inferring code naturalness from the CodeBERT prediction results.

Chapter 3

Related Work

In this section, we present the related works to our contributions. Additionally, we position and discuss our choices and decisions based on the results and findings of these related works.

3.1 Mutation Operators (Fault Patterns)

Mutation testing [87] has been widely studied since the 1970s and has been proven useful in multiple software engineering and testing applications [9], [10].

Despite this long history of research, the generation of relevant mutants remains an open question. Most of the related research has focused on the design of fault patterns (mutation operators) which are usually defined based on the target language grammar [3], [9] then refined through empirical studies [21], [22], [88] aiming at reducing the redundancy and noise among their generated mutants.

The most prominent mutant selection approach is that of Offutt et al. [21], which proposed a set of 5 mutation operators:

- *ABS:* makes every arithmetic expression take a zero, a positive and a negative value.

- *AOR:* replaces every arithmetic operator with every other possible operator that is syntactically valid.
- *LCR:* replaces every logical connector (AND and OR) with every other possible logical connector.
- *ROR:* replaces relational operators with other relational operators.
- *UOI:* inserts unary operators in front of expressions.

This set has been incorporated in most of the modern mutation testing tools [22] and is similar to the one that we use in our baseline [89] in chapter 4. The continuous advances in this sense were followed by a constant emergence of pattern-based mutation testing tools and releases [89], [90]. Some of these tools are becoming popular and widely adopted by researchers and practitioners, such as PiTest [90], [91], from which we consider three configurations as our comparison baseline in Chapter 5.

Recent research has focused their interest on improving the representativeness of artificial faults aiming at reducing the mutation space to real-like faults. For instance, instead of basing the mutation operators' design on the programming language grammar, Brown et al. [30] proposed inferring them from real bug fixes. Similarly, Tufano et al. [27], [32] and Tian et al. [34] proposed a neural machine translation technique that learns how to inject faults from real bug fixes. Along the same line, Patra et al. [33] proposed a semantic-aware learning approach, that learns and then adapts fault patterns to the project of interest. Their results are promising, however, the fact that these techniques depend on the availability of numerous, diverse, comprehensive and untangled fix commits [92] of uncoupled faults [20], which is often hard to fulfil in practice, may hinder their performance. Nevertheless, IBIR (Chapter 4) goal is complementary to the above studies as it aims at mimicking real bugs by applying fault patterns that have been constructed by inverting fix-patterns, collected and designed via the mining and the study of multiple developer bug fixes in various projects.

3.2 Fault Injection via Pre-trained Language Models

Overall, designing the mutation operators based on the known faults space yields more diverse mutants, representing more fault types. However, these extended operator sets tend to increase the number of generated mutants and consequently the general cost of the mutation campaign i.e. the fault patterns proposed by Brown et al. [30] as well as the ones we propose in IBIR counted also most of the conventional mutators in addition to new ones. Unlike these techniques, we propose in Chapter 5, μBERT which leverages pre-trained models to introduce mutants based on code knowledge instead of the faults one. As code is more available than faults, it offers a more flexible and complete knowledge base than faults, i.e. it perms to overcome the limitations and efforts required 1) to collect clean bug-fixing commits, 2) to capture the faulty behaviour and 3) design fault patterns, be it manually or via machine learning techniques.

The closest related work to our approach presented in Chapter 5 is a preliminary implementation of μBERT that was recently presented in the 2022 mutation workshop [45]. This implementation, denoted as μBERT$_{conv}$ in our evaluation, includes the conventional mutations (to mask and replace tokens by the model predictions), but it does not include the condition-seeding additive mutations that provide major benefits for fault detection. Moreover, μBERT$_{conv}$ was evaluated only on 40 bugs from Defects4J, and compared only to an early version of Pitest (similar to Pit-rv-all). In this work, we perform an extensive experimental evaluation including 689 bugs from Defects4J and compare μBERT effectiveness with three different configurations from Pitest. Moreover, we show that μBERT finds on average more bugs than μBERT$_{conv}$ without requiring more effort.

Recently, Richter et al. [93] employed pre-trained language models to produce context-dependent artificial faults. Their results showed that their approach can inject natural and realistic faults that can be used as artificial bug datasets, in order to train and improve the accuracy of learning-based bug detectors. Their approach is very similar to the one of μBERT$_{conv}$, thus should yield same results in terms of fault detection effectiveness and cost-efficiency, when evaluated in a mutation testing context.

3.3 Injection-Relevant Locations

To improve the effectiveness and cost-efficiency of fault injection, researchers have proposed different strategies to reduce the effort lost in treating redundant and trivial (easily-killed) mutants [9].

For instance, the studies of Andrews et al. [94] Natella et al. [28] and Chekam et al. [95] showed that the pair of mutant location and type is what makes mutants powerful and not the type or the applied operator itself. These findings motivated the importance of selecting relevant locations to mutate.

In this regard, Mirshokraie et al. [96] propose an approach that computes complexity metrics from program executions to extract locations with good observability to mutate. Sun et al. [97] suggest mutating multiple places within diverse program execution paths. Gong et al. [98] also propose the mutation in diverse locations of the program extracted from graph analysis. Along the same lines of research, in Chapter 5 we do not target any specific feature or narrow use case, but instead perform fault injection in a brute-force way similar to mutation testing, aiming at covering diverse locations of the program.

Other approaches restrict the fault injection on specific locations of the program, such as the code impacted by the last commits [26], [99] for better usability in continuous integration. Similarly, in our current approach IBIR (Chapter 4) we target locations related to a given bug report to target a specific feature or behaviour.

3.4 Mutant Selection

Random mutant sampling forms a natural cost-reduction method proposed since the early days of mutation testing [29]. Despite that, most of the mutant selection methods fail to perform better than it. Recently, Kurtz et al. [100] and Chekam et al. [95] demonstrated that selective mutation and random mutant sampling perform similarly. This implies that despite the efforts put into selective mutation, random mutant sampling remains among the most effective fault injection techniques. That is the reason why we adopt it as a baseline in our experiments.

More recent advances have resulted in powerful techniques for cost-effectively selecting mutants, i.e., by avoiding the analysis of irrelevant mutants (basically, duplicate, equivalent and subsumed ones) [101]–[103]. For instance, Natella et al. [28] used complexity metrics as machine learning features and applied them on a set of examples in order to identify (predict) which injected faults have the potential to emulate well the behaviour of real ones. Chekam et al. [95] also used machine learning, with many static mutant-related features to select and rank mutants that are likely fault revealing (have high chance to couple with a fault). Garg et al. [101] utilises the knowledge of mutants' surrounding context, using a Seq2Seq-based learning approach, to predict whether a mutant is likely to be subsuming (as defined in Section 2.1) or not. These studies assume the availability of historical faults to train on and do not aim at injecting specific faults as done by iBiR in Chapter 4.

More generally, these approaches address the problem of mutant or (fault) selection (prioritisation), once the mutants are generated and do not aim at injecting faults like iBiR and μBert in Chapters 4 and 5. Nevertheless, we believe that these approaches can complement and be associated with our proposed techniques to improve their cost efficiency [104].

3.5 Test Execution

Among the issues with mutation testing stands its high computational cost. The problem stems from the vast number of faults that are injected, which need to be executed with large test suites, thereby requiring expensive computational resources [9], taking several minutes if not hours per every mutant execution. Intuitively, reducing the number of injected faults reduces the tests execution cost, however, this cost remains important when tests have to be executed on every mutant separately, with a large number of mutants. Moreover, the mutant execution problem becomes intractable when test execution is expensive or the test suites involve system-level tests, thereby often limiting mutation testing application to the unit level.

Aiming to reduce this cost, several approaches have been proposed to limit the test executions [14]. For instance, Kim et al. [105] proposed to execute only the tests that trigger the infection of the program state by the considered mutation. Vercammen et al. [106] suggested executing only the tests that are related to the mutated method (function) instead of running all tests for all mutants. Wang et al. [107] suggested leveraging the mutant schemata concept [108] to enable shared execution states between different mutants' test execution, and thus reducing their computational costs.

Overall, these methods offer promising results in terms of mutant execution reduction but cannot be applied in fault tolerance assessment and do not aim at injecting faults as done by iBiR (Chapter 4) and μBERT (Chapter 5). Although we acknowledge the importance of reducing this cost, we consider it as out of our dissertation's scope and focus on the faults generation phase challenges and not on the test execution ones. Precisely, we executed mutants in parallel separately, achieving a considerable cost gain in conducting our experiments without compromising the effectiveness and precision of our results.

Recent studies aim at reducing the computational demands of the mutant execution through a combination of static and dynamic metrics [15], i.e. by predicting whether a mutant will be killed or not without executing any test. While this approach offers the possibility of obtaining quick estimates of the adequacy of test suites, it does not offer accurate and precise indications on which mutants to target and thereby which tests to write.

Nevertheless, we believe that these test execution cost-reduction strategies can be incorporated or combined with both of our proposed approaches. Together with parallel mutant executions, they may offer considerable improvements in the usability of mutation testing and hence, encourage and facilitate its adoption in practice.

3.6 Relation between Mutants and Faults

The relationship between injected and real faults has also received some attention [9]. The studies of Ojdanic et al. [35], Papadakis et al. [7], Just et al. [109] and Andrews et al. [110] investigated whether mutant kills and fault detection ratios follow similar trends. The results show the existence of a correlation and, thus, that mutants can be used in controlled experiments as alternatives to real faults.

In the context of testing, i.e., using mutants to guide testing, injected faults can help identify corner cases and reveal existing faults. The studies of Frankl et al. [111], Li et al. [112] and Chekam et al. [6] demonstrated that guidance from mutants leads to significantly higher fault revelation than that of other test techniques (test criteria).

Based on these findings, we assess the proposed approaches – IBIR and μBERT (in Chapters 4 and 5) – based on the relation between the injected and real faults, in terms of failing tests.

3.7 Code-naturalness

Estimating the naturalness of source code and thus its latent predictability has been widely investigated. Research on this area started from the observation of Hindle et al. [37] that code alike natural language is repetitive and thus techniques designed for the latter could be applied to the former. This led to the development of code naturalness which quantifies how surprising a piece of code is, given a reference corpus and amidst the neighbouring pieces.

The most notable way to evaluate naturalness is through the use of Language models and particularly n-gram ones. N-gram models approximate the naturalness of a sequence of tokens based on its occurrence likelihood, estimated relatively to the sequences observed in the training set. This probability follows a Markov chain conditional probability series, where the probability $P(t)$ of a token t to occur depends on the $n-1$ preceding tokens. To highlight irregular sequences (low probabilities of occurring), the naturalness of a sequence is usually expressed through cross-entropy [113], [114] which is computed by aggregating the logarithm of the token probabilities as follows:

$$\mathsf{H}(S) = -\frac{\sum_{i=1}^{m} \log(P(t_i|t_{i-n+1}...t_{i-1}))}{n}, \tag{3.1}$$

where n denotes the order of the n-gram model, $\{t_1, ..., t_m\}$ the set of m tokens forming the sequence S and $P(t_i|t_{i-n+1}...t_{i-1})$ the probability $P(t_i)$ knowing $t_{i-n+1}...t_{i-1}$. Consequently, n-gram models attribute high entropy values to unusual (unnatural) code relatively to regular (natural) code.

Still, n-gram models fail to assign a probability to every token and sequence of tokens. Indeed, it is often the case in programming like in natural language to observe sequences and tokens that are unseen in the training corpus, like variable names. To avoid assigning a zero probability in these scenarios, n-gram models usually replace every token occurring less than k times by a placeholder i.e. `<UNK>` and attribute a non-zero probability to it, where k and `<UNK>` are usually called the *unknown threshold* and *unknown word* [39], [115], [116]. Similarly, to deal with unseen sequences of tokens in the training set, smoothing techniques can be applied. Several have been proposed and evaluated over the past decades, among which Kneser Ney (KN) [117] and Modified Kneser Ney (MKN) [116] are the best performing *smoothers* [37], [39], [116].

Although code naturalness has broad use, its related n-gram-based metric is suffering from various drawbacks. Indeed, the n-gram models tend to overspecialize and become specific to the project and programming practices at hand [36]. To obtain a more generic naturalness measure and overcome the training limitations, we propose inferring code naturalness through Large Language Models predictions. We introduce this approach (CODEBERT-NT) in Chapter 6.

Naturalness and Buggyness

Since its appearance, many applications of the naturalness of code have been developped [36]. Baishakhi et al. [38] have shown empirical evidence that buggy lines are on average less natural than not-buggy ones and that n-gram entropy can be useful in guiding bug-finding tasks at both file- and line-level. Jimenez et al. [39] evaluated the sensitivity of n-gram w.r.t. its parameters and code tokenization techniques, via a file-level naturalness study. Their results confirmed Baishakhi et al. [38] findings and provided recommendations on the best n-gram configurations for naturalness-based applications, including the differentiation between buggy and fixed code.

Based on these findings, we conduct a study to investigate whether and how we can approximate code naturalness using pre-trained language models and compare the measured metrics to that computed by n-gram models, trained following Jimenez et al. [39] recommendations. We present this study and the underlying approach (CODEBERT-NT) in Chapter 6 and provide an empirical evaluation of its capability in capturing code naturalness, based on its accuracy in distinguishing buggy from non-buggy code.

Chapter 4

Leveraging Bug Reports and Automated Program Repair Patterns for a Targeted and Realistic Fault Injection

Much research on software engineering relies on experimental studies based on fault injection. Fault injection, however, is not often relevant to emulate real-world software faults since it "blindly" injects large numbers of faults. It remains indeed challenging to inject few but realistic faults that target a particular functionality in a program. In this work, we introduce iBiR, a fault injection approach that addresses this challenge by exploring change patterns associated to user-reported faults. To inject realistic faults, we create mutants by re-targeting a bug report driven automated program repair system, i.e., reversing its code transformation templates. iBiR is further appealing in practice since it requires deep knowledge of neither code nor tests, but just of the program's relevant bug reports. Thus, our approach focuses the fault injection on the feature targeted by the bug report. We assess iBiR by considering the Defects4J dataset.

Experimental results show that our approach outperforms the fault injection performed by traditional mutation testing in terms of semantic similarity with the original bug, when applied at either system or class levels of granularity, and provides better, statistically significant, estimations of test effectiveness (fault detection). Additionally, when injecting 100 faults, iBiR injects faults that couple with the real ones in around 36% of the cases, while mutation testing achieves less than 4%.

This chapter is based on the following article:

Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawende F. Bissyandé, Jacques Klein, and Yves Le Traon. IBiR: Bug Report driven Fault Injection. In : *ACM Transactions on Software Engineering and Methodology (TOSEM 2022). Accepted on May 2022.* https://doi.org/10.1145/3542946

4.1 Introduction

A key challenge of fault injection techniques (such as mutation analysis) is to emulate the effects of real faults. This property of representativeness of the injected faults is of particular importance since fault injection techniques are widely used by researchers when evaluating and comparing bug finding, testing and debugging techniques, e.g., test generation, bug fixing, fault localisation, etc, [9]. This means that there is a high risk of mistakenly asserting test effectiveness in case the injected faults are non-representative.

Typically, fault injection techniques introduce faults by making syntactic changes in the target programs' code using a set of simple syntactic transformations [28], [29], [118], usually called mutation operators. These transformations have been defined based on the language syntax [3] and are "blindly" mutating the entire codebase of the projects, injecting large numbers of mutants, with the hope to inject some realistic faults. This means that there is a limited control on the fault types and the locations where to inject faults. In other words, the appropriate “what” and “where” to inject faults in order to make representative fault injection has been largely ignored by existing research.

Fault injection techniques may also draw on recent research that mines fault patterns [27], [30] and demonstrate some form of realism w.r.t. real faults. These results indicate that the injected faults may carry over the realism of the patterns, fact that removes a potential validity threat. However, at the same time, they are limited as they do not provide any control on the locations and target functionality, thus impacting fault representativeness [7], [28], [95].

This is an important limitation especially for large real-world systems because of the following two reasons: a) injecting faults everywhere escalates the application cost due to the large number of mutants introduced and b) the results could be misleading since a tiny ratio of the injected faults are coupled to the real ones [7] and the injected set of faults does not represent the likelihood of faults appearing in the field [28]. Therefore, representativeness of the injected faults in terms of fault types and locations is of utmost importance w.r.t. both application cost and accuracy of the method.

To bypass these issues, one could use real faults (mined from the projects' repositories) or directly apply the testing approach to a set of programs and manually identify potential faults. While such a solution brings realism into the evaluations, it is often limited to few fault instances (of limited diversity), requires an expensive manual effort in identifying the faults and fails to offer the experimental control required by many evaluation scenarios.

We advance in this research direction by bringing realism in the fault injection via leveraging information from bug reports. Bug reports often include sufficient information for debugging techniques in order to localise [40], debug [41] and repair faults [42] that happened in the field. Therefore, together with specially crafted defect patterns (mined through systematic examination of real faults) such information can guide fault injection to target critical functionality, mimic real faulty behaviour and make realistic fault injection. Perhaps more importantly, the use of bug reports removes the need for knowledge of the targeted system or code.

Our method starts from the target project and a bug report (BR) written in natural language. It then applies Information Retrieval (IR)-based fault localisation [40] in order to identify the relevant places where to inject faults. It then injects recurrent fault instances (fault patterns) that were manually crafted using a systematic analysis of frequent bug fixes, prioritized according to their position and type. This way our method performs fault injection, using realistic fault patterns, by targeting the features described by the bug reports. Moreover, by applying our method on many programs and BRs (injecting few bugs per BR), one gets fault pools to be used for test and fault tolerance assessment.

We implemented our approach in a system called IBIR and evaluated its ability to imitate 280 real faults. In particular we evaluated a) the semantic similarity of real and injected faults, b) the coupling[1] relation between injected and real faults, and c) the ability of the injected faults to indicate test effectiveness (fault detection) when tested with different test suites. Our results show that IBIR manages to imitate the targeted faults, with a median semantic similarity value of 0.577, which is significantly higher than the 0.134 achieved by using traditional mutation testing, when injecting the same number of faults.

Interestingly, we found that IBIR injects faults that couple with the real ones in around 36% of the targeted cases. This is achieved by injecting 100 faults per target (real) fault and it is approximately 9 times higher than the coupled mutants produced by mutation testing. Fault coupling is one of the most important testing properties [22], [50], here indicating that one can use the injected faults instead of the real ones.

[1]Injected faults couple with the real ones when injected faults are detected only by test cases that detect the real faults. This implies that the injected faults provide good indications on whether tests are capable of detecting the coupled faults.

Another key finding of our study is that the injected faults provide much better indication on test effectiveness (fault detection) than mutation testing as their detection ratios discriminate between actual failing and passing test suites, while mutant detection rates cannot. This implies that the use of iBiR yields more accurate results than the use of traditional mutation testing.

4.2 Scope & Motivation

iBiR aims at injecting realistic faults, i.e., faults imitating the behaviour of previously reported ones, to be used for test and fault tolerance assessment. As such, it injects faults in a current stable (fixed) version of the same system where test techniques are assessed with respect to a) fault revelation potential, in the case of test assessment, and b) the reaction of the system under unexpected (faulty) behaviour to support controlled studies. This means, that we assume the existence of relatively stable projects with Fixed/Closed bug reports. In principle, iBiR could be use to guide testing towards open bug reports or to support the discovery of bugs that are similar to those reported. However, these two use cases regard the fault revelation ability of the fault injection campaigns (the test guidance provided by fault injection) and not the realistic fault injection problem (the ability of injecting faults to imitate the behaviour of real ones) that we are aiming at. Therefore, we have left them open for future research.

4.2.1 Assessment of testing techniques

Fault injection is used extensively by researchers as a tool to evaluate the fault-revealing capability of automated test techniques such as automated test generation techniques. This approach was found to be used by approximately 19% of all software testing studies published in major SE conference by a bibliometric analysis performed in 2016 [10].

This is because real and diverse bug-datasets are hard to collect and make it hard to perform controlled studies as they usually result in faulty versions including single faults. Fault injection is thus a fast and convenient way to perform control studies since it avoids the costly and tedious work of creating fault-datasets. In such cases, the realism of the injected faults is a major validity question that may impact the results of the experiments. Recent studies [7] have shown that conventional mutation testing doesn't perform well in this regard as it introduces many faults that are unrealistic. To deal with such cases, we develop IBIR and show that it injects more semantically similar faults than traditional mutation testing.

4.2.2 Fault tolerance assessment

Fault injection is also frequently used to evaluate the system's performance under faulty test executions. In such a case, the injected faults simulate the effects of real ones by performing arbitrary code changes everywhere. To this end, IBIR guides the injection towards specific error-prune targets/features and fault types. This is particularly important in order to improve the realism of the analysis. Interestingly, previous research on fault tolerance assessment [28] has shown that fault injection realism can be improved by appropriately controlling the locations and types of the injected faults. We therefore, propose a way to do so by leveraging information from bug reports.

4.3 Approach

We propose IBIR, the first fault injection approach that utilizes information extracted from bug reports to emulate real faults. A high level view of the way IBIR works is shown in Figure 4.1 and a step by step overview of IBIR's approach is illustrated in the Algorithm 1. Our approach takes as input (1) the source code of the program of interest

Algorithm 1 IBIR approach algorithm

```
Require: $bugReport, projectRepository, numberOfFaults$
1: $patterns[] \leftarrow$ loadListOfPatterns()
2: $patches[] \leftarrow []$
3: $result[] \leftarrow []$
4: $rankedSuspFiles[] \leftarrow$ fileLevelIRFL($bugReport, projectRepository$)
5: $first20RankedSuspFiles[] \leftarrow$ head($rankedSuspFiles, 20$)
6: $rankedStatements[] \leftarrow$ statementLevelIRFL($bugReport, first20RankedSuspFiles[]$)
7: for $statement$ in $rankedStatements[]$ do
8:     $fileAstTree \leftarrow$ loadAstTree($statement.containingFile$)
9:     $statementNodes[] \leftarrow$ parseTree($tree, statement$)
10:    for $astNode$ in $statementNodes[]$ do
11:        for $pattern$ in $patterns[]$ do
12:            if patternIsAppliableOnNode($pattern, astNode$) then
13:                $patch \leftarrow$ createPatch($pattern, astNode, fileAstTree$)
14:                add($patch, patches[]$)
15:            end if
16:        end for
17:    end for
18: end for
19: for $patch$ in $patches[]$ do
20:    $faultyVersion \leftarrow$ apply($patch, projectRepository$)
21:    if isCompilable($faultyVersion$) then
22:        add($patch, result[]$)
23:    end if
24:    if $numberOfFaults ==$ length($result[]$) then
25:        return $result[]$
26:    end if
27: end for
28: return $result[]$
```

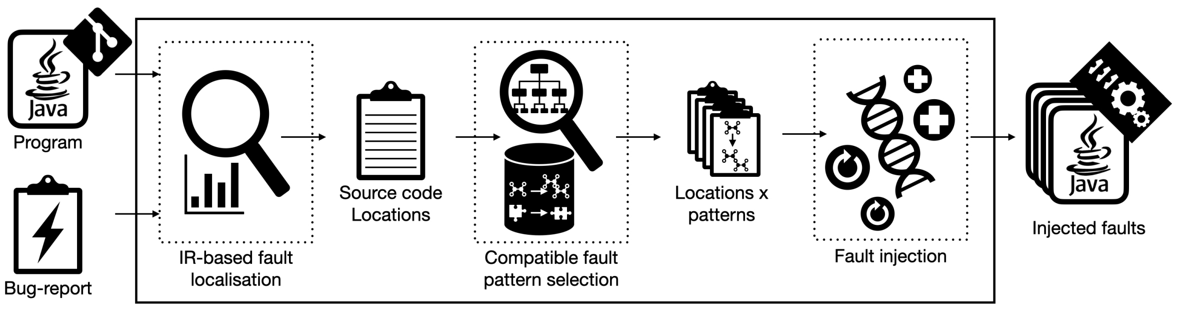


Figure 4.1: The iBiR fault injection workflow.

and (2) a bug report of that program. The objective is to inject artificial faults in the program (one by one, creating multiple faulty versions of it) which imitate the original bug. To do so, iBiR proceeds in three steps.

First step: iBiR identifies relevant locations to inject the faults. It applies IR-based fault localisation to determine, from the bug report, the code locations (statements) that are likely to be relevant to the target fault. These locations are ranked according to their likelihood to be the feature described by the bug report, hence are relevant to inject faults.

Second step: iBiR mutates the identified code locations by applying fault patterns. We build our patterns by inverting fix ones, that have been proposed and used in automated program repair (APR) approaches [80]. Our intuition is that, since fix patterns are used to fix bugs, inverted patterns may introduce a fault similar to the original bug. For each location, we apply only patterns that are syntactically compatible with the code location. This step yields a set of faults to inject, i.e., pairs composed of a location and a pattern.

Third step: our method ranks the location-pattern pairs w.r.t. the location likelihood and priority order of the patterns. Then iBiR takes each pair (in order) and applies the pattern to the location, injecting a fault in the program. We repeat the process until the desired number of injected faults has been produced or until all location-pattern pairs have been considered.

4.3.1 Bug Report driven Fault Localisation

IR-based fault localisation (IRFL) [119], [120] leverages eventual similarity between the description and content of a bug report and the source code of the target program to identify eventual relevant buggy code locations. To do so, these approaches operate typically by formulating a *query* using tokens extracted from an input bug report, then computing its similarity with the *documents*' composing tokens. For instance, when applied on the file level of granularity, each source code file is considered as a document, and hence, part of the target search space [40], [66]–[69], [121]. Next, IRFL techniques rank these files based on their measured similarities with the given bug report, providing insights on which files are more likely to be related to this latter. Meaning that the files that are more likely to be buggy are ranked first.

We follow the same principle to identify promising locations where to inject realistic faults, relying on information contained in bug report to find the code location with the highest similarity score. Most IRFL approaches focus on file-level localisation, which can be coarse-grained for our purpose of fault injecting. Thus, we rather use a statement-level IRFL approach that has been successfully applied to support program repair [42].

It is to be noted that, contrary to program repair, we do not aim to identify the exact bug location. We are rather interested in locations that allow injecting realistic faults (similar to the bug). This means that IRFL may pinpoint multiple locations of interest for fault injection even if those were not buggy code locations.

To identify fault injection locations that are related to the targeted bug-report, we leverage an existing IRFL tool that was originally developed as part of the iFixR [42] tool. The IRFL works by matching words of a bug report with source code file(s) using 17

Table 4.1: Information retrieval features collected from bug reports and source code files by iFixR [42].

Bug Report Features	
Feature	**Description**
summary	The title and the summary of the bug report
description	The description part of the bug report
rawBugReport	The whole bug report as in textual form
stackTraces	The stack traces in the bug report
codeElements	Code snippets in the bug reports
summaryHints	Code-related terms in summary
descriptionHints	Code-related terms extracted when parsing the description
Source Code Features	
Feature	**Description**
packageNames	The parsed package names of the source code files
classNames	The parsed class names of the source code files
methodNames	The parsed method names of the source code files
methodInvocations	The parsed method invocation of the source code files
formalParameters	The parsed formal parameters of the source code files
memberReferences	The parsed member references of the source code files
documentation	The parsed class names of the source code files
rawSource	Source file as a text
hunks	The hunks from the commits on the file
commitLogs	The commit logs of the file

features. These features are extracted from the bug report (7 features) and the source code git repository (10 features) and are listed in Table 4.1.

For every feature, the tokenizer applies a lexical analysis where (1) it extracts tokens from the retrieved text, (2) then drops stopwords to reduce the noise, i.e., caused by the programming language keywords, and (3) applies stemming on all the tokens to obtain homogeneous tokens based on their roots. The tokens are extracted by considering both white space and source code specific separators, such as punctuation and camel case splitting, i.e., $calculateMaximum$ is split to $calculate$ and $Maximum$. Next, the tokens are filtered by checking them with the WordNet [122] dictionary and discarding the unknown ones. An additional sanity check is then applied in order to extract the stack-traces and source code elements, using regular expressions.

The IRFL calculates then the similarity coefficient ($Cosine$ [123]) between the bug report and a source code file using a revised Vector Space Model (rVSM) [40] based on the occurrences-frequency of the extracted tokens in the preprocessing tokenization step (the vectors are calculated using $tf - idf$ [124]).

Next, an ensemble of classification models provided by D&C [125] was used in order to rank the source code files according to their suspiciousness. This ensemble takes as input the calculated 7x10 weights of all pairs $<$ bug report, source code file $>$ and outputs their averaged prediction results. This ensemble was used as it has been shown to work well on a diverse set of bug reports [125] since every classifier of the ensemble model was trained on a different set of data.

In a last step, as iFixR [42], the IRFL localises suspicious statements from the 20 most suspicious files based on their rVSM cosine-similarity [123] with the given bug report (the vectors are calculated using $tf - idf$ [124]) and outputs these statements in a list of statements ranked according to their suspiciousness. Further details on the IRFL can be found in the D&C work [125] and our implementation [126].

4.3.2 Fault patterns

We start from the fix patterns developed in TBar [80], a state of the art pattern-based program repair tool. Any pattern is described by a context, i.e., an AST node type to which the pattern applies, and a recipe, a syntactical modification to be performed similar to program repair techniques [127]. For each pattern, we define a related fault injection pattern that represents the inverse of that pattern. For instance, inverting the fix pattern that consists of adding an arbitrary statement yields a *remove statement* fault pattern. Interestingly, some fix patterns are symmetric in the sense that their inverse pattern is also a fix pattern, e.g., inverting a Boolean connector. These patterns can

Table 4.2: iBIr fault injection patterns.

Pattern category	**Bug injection pattern**	**example input**	**example output**
Insert Statement	Insert a method call, before or after the localised statement.	*someMethod(expression);*	*someMethod(expression); method(expression);*
	Insert a return statement, before or after the localised statement.	*statement;*	**statement; return VALUE;**
	Wrap a statement with a try-catch.	*statement;*	*try{ statement; } catch (Exception e){ ... }*
	Insert an if checker: wrap a statement with an if block.	*statement;*	*if (conditional_exp) { statement; }*
Mutate Class Instance Creation	Replace an instance creation call by a cast of the super.clone() method call.	*... new T();*	*... (T) super.clone();*
Mutate Conditional Expression	Remove a conditional expression.	*condExp1 && condExp2*	*condExp1*
	Insert a conditional expression.	*condExp1*	*condExp1 && condExp2*
	Change the conditional operator.	*condExp1 && condExp2*	*condExp1 || condExp2*
Mutate Data Type	Change the declaration type of a variable.	*T1 var ...;*	*T2 var ...;*
	Change the casting type of an expression.	*... (T1) expression ...;*	*... (T2) expression ...;*
Mutate float or double Division	Remove a float or a double cast from the divisor.	*... dividend / (float) divisor ...;*	*... dividend / divisor ...;*
		... intVarExp / 10d ...;	*... intVarExp / 10 ...;*
	Remove a float or a double cast from the dividend.	*... (float) dividend / divisor ...;*	*... dividend / divisor ...;*
		... 1.0 / var ...;	*... 1 / var ...;*
	Replace float or double multiplication by an int division.	*... (1.0 / divisor) * dividend ...*	*... dividend / divisor ...;*
		*... 0.5 * intVarExp ...;*	*... intVarExp / 2 ...;*
Mutate Literal Expression	Change boolean, number or string literals in a statement by another literal or expression of the same type.	*... string_literal1 ...*	*... string_literal2 ...*
		... int_literal ...	*... int_expression ...*
Mutate Method Invocation	Replace a method call by another one.	*... method1(args) ...*	*... method(args) ...*
	Replace a method call argument.	*... method(arg1, arg2) ...*	*... method(arg1, arg3) ...*
	Remove a method call argument.	*... method(arg1, arg2) ...*	*... method(arg1) ...*
	Add an argument to a method call	*... method(arg1) ...*	*... method(arg1, arg2) ...*
Mutate Return Statement	Replace a return experession.	*return expr1;*	*return exp2;*
Mutate Variable	Replace a variable by another variable or an expression of the same type.	*... var1 ...*	*... var2 ...*
		... var1 ...	*... exp ...*
Move Statement	Move a statement to another position.	*statement; ...*	*... statement;*
Remove Statement	Remove a statement.	*statement; ...*	*...*
	Remove a method.	*method(args){ statement; }*	*...*
Mutate Operators	Replace an Arithmetic operator.	*... a + b ...*	*... a - b ...*
	Replace an Assignment operator.	*... c += b ...*	*... c -= b ...*
	Replace a Relational operator.	*... a <b ...*	*... a >b ...*
	Replace a Conditional operator.	*... a && b ...*	*... a || b ...*
	Replace a Bitwise or a Bit Shift operator.	*... a & b ...*	*... a | b ...*
	Replace an Unary operator.	*a++*	*a--*
	Change arethmetic operations order.	*a + b * c*	*c + b * a*

thus be used for both bug fixing and fault injection. Table 4.2 enumerates the resulting set of fault injection patterns used by our approach.

Given a location (code statement) to inject a fault into, we identify the patterns that can be applied to the statement. To do so, our method starts from the AST node of the statement and visits it exhaustively, in a breadth-first manner. Each time it meets an AST node that matches the context of a fault pattern, it memorizes the node and the pattern for later application. Then the method continues until it has visited all AST nodes under the statement node. This way, we enumerate all possible applications of all fault patterns onto the location.

Since more than one pattern may apply to a given location, we prioritize them by leveraging heuristic priority rules previously defined in automated program repair methods (these were inferred from real-world bug occurrences [80]). This means that every fault injection pattern gets the priority order of its inverse fix pattern.

4.3.3 Fault injection

The last step consists of applying, one by one, the fault patterns to inject faults at the program locations identified by IRFL. Locations of higher ranking are considered first. Within a location, pattern applications are ordered based on the pattern priority. By applying a pattern to a corresponding AST node of the location, we inject a fault within the program before recompiling it. If the program does not compile, we discard the fault and restart with the next one. We continue the process until it reaches the desired number of (compilable) injected faults or all locations and patterns have been considered.

4.3.4 Demonstration Example

Figures 4.2 and 4.3 illustrate the execution steps of iBiR when injecting faults in commons-math project, based on the content of the bug report MATH-329[2].

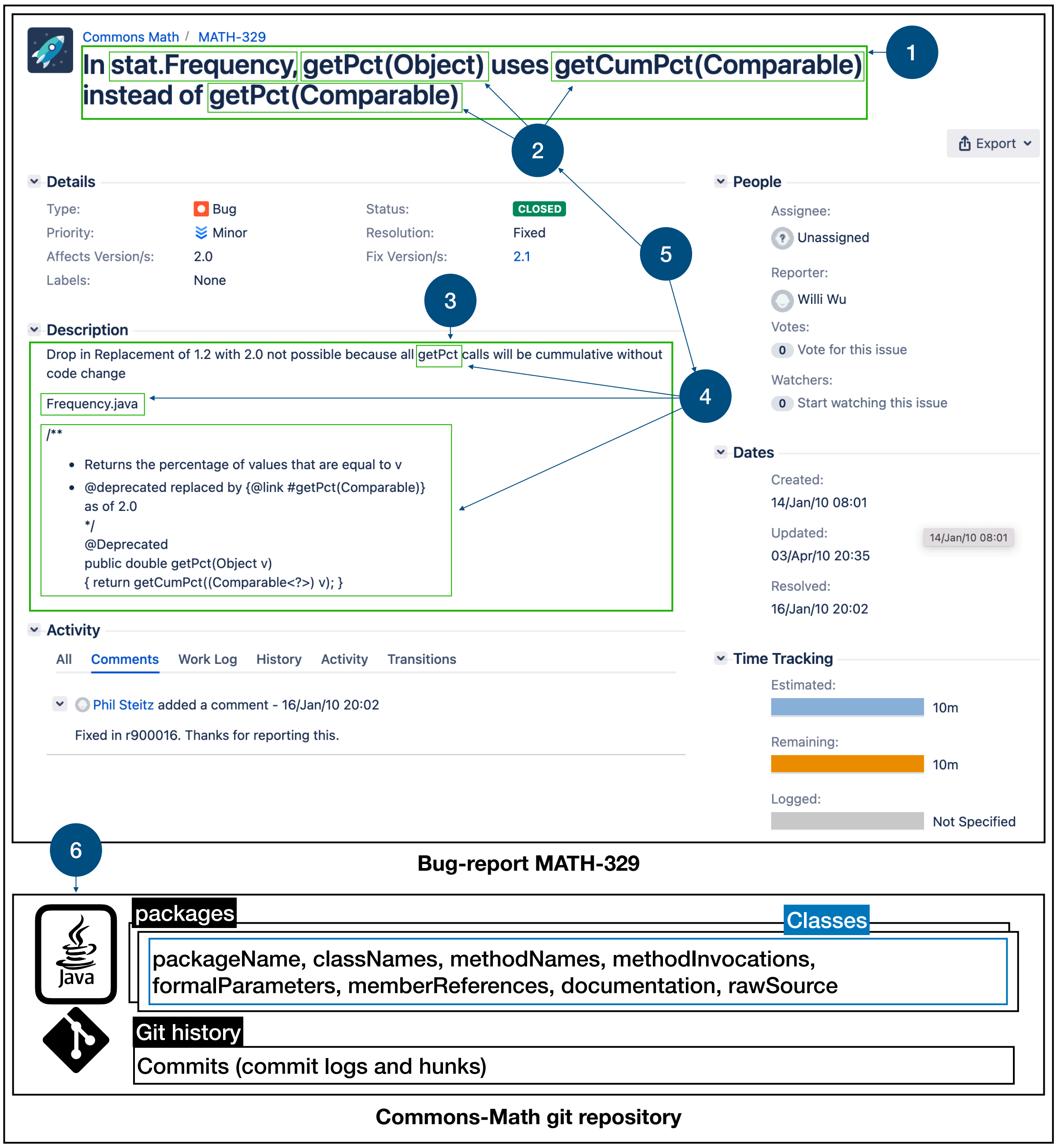
iBiR starts by parsing the bug report and extracting its relevant information: the summary (1), the summary hints (2), the description (3), the description hints (4), code elements (5) and the raw bug-report. This example bug report does not contain any stack-trace as the corresponding bug causes a misbehavior but does not trigger any crash or throw any exception.

iBiR loads also all the required information from the projects repository (6) then uses all of these features to find the code locations that are the most likely related to the input bug-report. This search happens in two steps - file-level then statement-level localisation - and ends by the output of a sorted list of source-code lines (7), as detailed in subsection 4.3.1.

iBiR parses these lines one by one starting with the highest rank. In this example, the 1st rank is attributed to the line number `303` of the file `src/main/java/org/apache/commons/math/stat/Frequency.java` (8), which corresponds to a `return` statement that invokes the method `getPct` with a variable `v` which is cast to the type `Comparable`. iBiR selects all compatible fault patterns with this statement's AST and applies them one by one on the source-code, inducing multiple faults. In Figure 4.3 we illustrate the modified source-code corresponding to 5 faults injected in the line `303` of the `Frequency.java` file (9): Faults 1 and 2 are injected by invoking respectively the methods `getCumPct` and `getCumFreq` instead of `getPct`. In fault 3, the method `getPct` is invoked with the field `this.freqTable` as variable instead of `v`. Faults 4 and 5 are injected by inserting additional method calls before the `return` statement, respectively `addValue(v);` and `clear();`.

[2]Bug report link: https://issues.apache.org/jira/browse/MATH-329

ıBıR continues parsing the sorted source code locations by the IRFL until all of them are treated or the requested number of faults has been injected.



Commons Math / MATH-329

In stat.Frequency, getPct(Object) uses getCumPct(Comparable) instead of getPct(Comparable)

Details

Type:	Bug	Status:	CLOSED
Priority:	Minor	Resolution:	Fixed
Affects Version/s:	2.0	Fix Version/s:	2.1
Labels:	None		

Description

Drop in Replacement of 1.2 with 2.0 not possible because all getPct calls will be cummulative without code change

Frequency.java

```
/**
 • Returns the percentage of values that are equal to v
 • @deprecated replaced by {@link #getPct(Comparable)}
   as of 2.0
   */
   @Deprecated
   public double getPct(Object v)
   { return getCumPct((Comparable<?>) v); }
```

Activity

All | Comments | Work Log | History | Activity | Transitions

Phil Steitz added a comment - 16/Jan/10 20:02

Fixed in r900016. Thanks for reporting this.

People

Assignee:
Unassigned

Reporter:
Willi Wu

Votes:
0 Vote for this issue

Watchers:
0 Start watching this issue

Dates

Created:
14/Jan/10 08:01

Updated:
03/Apr/10 20:35 14/Jan/10 08:01

Resolved:
16/Jan/10 20:02

Time Tracking

Estimated:
10m

Remaining:
10m

Logged:
Not Specified

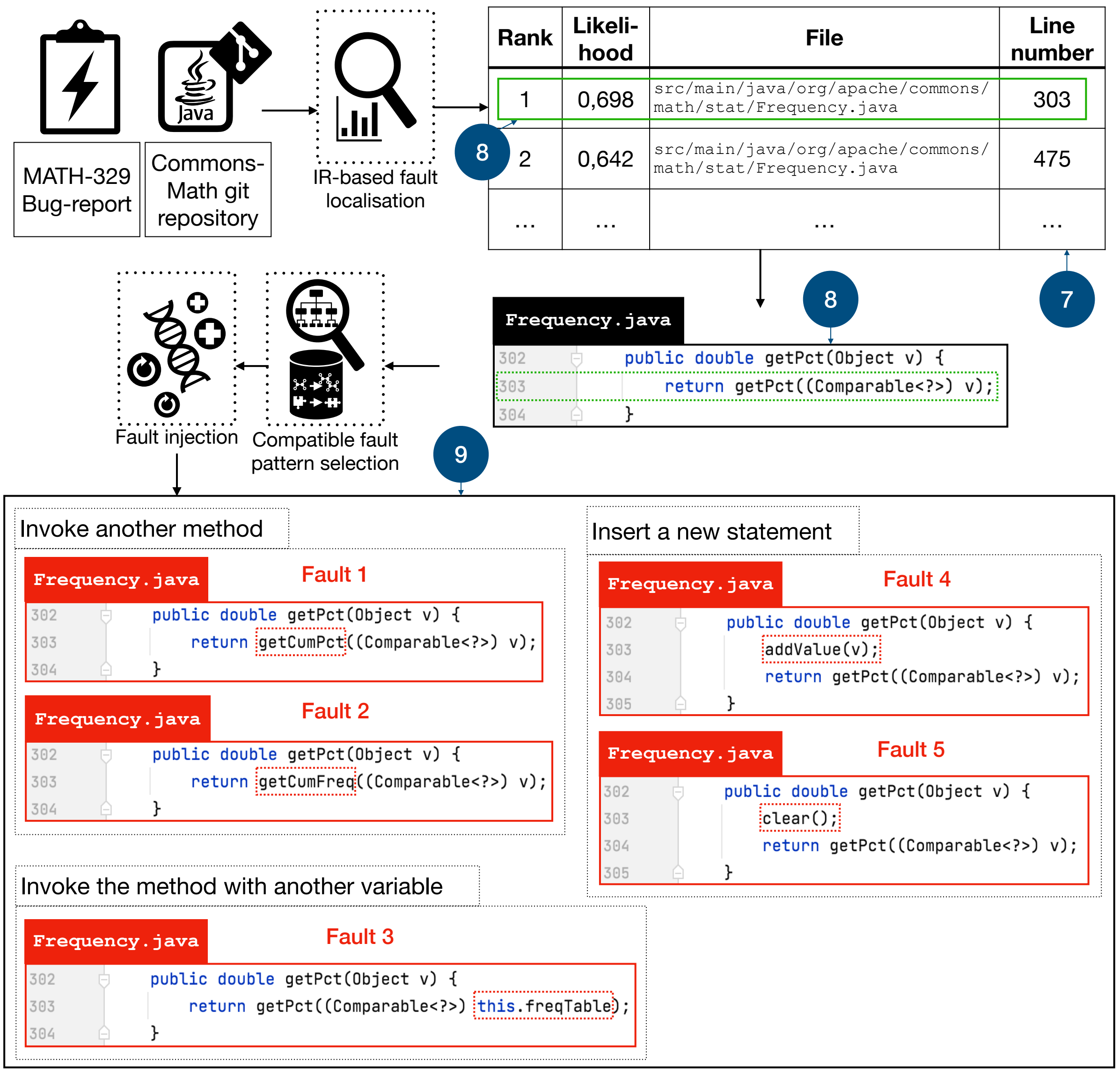
Bug-report MATH-329

packageName, classNames, methodNames, methodInvocations, formalParameters, memberReferences, documentation, rawSource

Commits (commit logs and hunks)

Commons-Math git repository

Figure 4.2: Example of ıBıR's input: the bug report `MATH-329` (1- the summary, 2- the summary hints, 3- the description, 4- the description hints, 5- code elements) and the Commons-Math git repository (6-).


MATH-329 Bug-report
Commons-Math git repository
IR-based fault localisation
8
7
8
9
Fault injection
Compatible fault pattern selection
Invoke another method
Invoke the method with another variable
Insert a new statement
Fault 1
Fault 2
Fault 3
Fault 4
Fault 5

Rank	Likeli-hood	File	Line number
1	0,698	src/main/java/org/apache/commons/math/stat/Frequency.java	303
2	0,642	src/main/java/org/apache/commons/math/stat/Frequency.java	475
…	…	…	…

Frequency.java
```
302     public double getPct(Object v) {
303         return getPct((Comparable<?>) v);
304     }
```

Frequency.java
```
302     public double getPct(Object v) {
303         return getCumPct((Comparable<?>) v);
304     }
```

Frequency.java
```
302     public double getPct(Object v) {
303         return getCumFreq((Comparable<?>) v);
304     }
```

Frequency.java
```
302     public double getPct(Object v) {
303         return getPct((Comparable<?>) this.freqTable);
304     }
```

Frequency.java
```
302     public double getPct(Object v) {
303         addValue(v);
304         return getPct((Comparable<?>) v);
305     }
```

Frequency.java
```
302     public double getPct(Object v) {
303         clear();
304         return getPct((Comparable<?>) v);
305     }
```

Figure 4.3: Example of IBIR's execution on the bug report MATH-329: the IRFL extracts tokens from the bug-report and the projects repository. Then, it outputs a list of statements ranked by their suspicioussness (7- the 2 first ranked statements by IBIR). The mutator loads every statement in this list, parses its AST, selects the applicable patterns and apply them one by one to inject faults (8- the statement with the highest suspicioussness, 9- faults injected when processing the first statement).

4.4 Research Questions

Our approach aims at injecting faults that imitate real ones by leveraging the information included in bug reports. Therefore, a natural question to ask is how well iBiR's faults imitate the targeted (real) ones. Thus, we ask:

RQ1 *(Imitating bugs):* Are the iBiR faults capable of emulating, in terms of semantic similarity, the targeted (real) ones? How they compare with mutation testing?

To answer this question, we check whether any of the injected faults imitate well the targeted ones. Following the recommendations from the mutation testing literature [7] we approximate the program behaviour through the project test suites and compare the behaviour similarity of the test cases w.r.t. their pass and failing status using the Ochiai similarity coefficient. This is a typical way of computing the semantic similarity of mutants and faults in mutation-based fault localisation [41], [128].

We then compare these results with the mutation testing ones by injecting mutants using the standard operators employed by mutation testing tools [22] and measuring their semantic similarity with the targeted faults. To make a fair comparison, we inject the same number of faults per target. For iBiR we selected the top-ranked mutants while for mutation testing we randomly sampled mutants across the entire project code-base. Random mutant sampling forms our baseline since it performs comparably to the alternative mutant selection methods [95], [100]. Also, since we are interested in the relative differences between the injected fault sets, we repeat our experiments multiple times using the same number of faults (mutants).

Our approach identifies the locations where bugs should be injected through an IR-based fault localisation method. This may give significant advantages when applied at the project level, but these may not carry on individual classes. Such class level granularity may be well suited for some test evaluation tasks, such as automatic test generation [129]. To account for this, we performed mutation testing (using the traditional mutation operators) at the targeted classes (classes where the faults were fixed). To make a fair comparison we also restricted IBIR to the same classes and compared the same number of mutants. This leads us to the following question:

RQ2 *(Comparison at the target class):* How does IBIR compare with mutation testing, in terms of semantic similarity, when restricted to particular classes?

We answer this question by injecting faults in only the target classes using the IBIR bug patterns and the traditional mutation operators. Then we compare the two approaches the same way as we did in RQ1.

Up to this point, the answers to the posed questions provide evidence that using our approach yields mutants that are semantically similar to the targeted bugs. Although, this is important and demonstrates the potential of our approach, it does not necessarily mean that the injected faults are strongly coupled with the real ones[3]. Mutant and fault coupling is an important property for mutants that significantly helps testing [109]. Therefore, we seek to investigate:

RQ3 *(Mutant and fault coupling):* How does IBIR compare with mutation testing with respect to mutant and fault coupling?

To answer this question we check whether the faults that we inject are detected only by the failing tests, i.e., only by the tests that also reveal the target fault. Compared to similarity metrics, this coupling relation is stricter and stronger.

After answering the above questions we turn our attention to the actual use of mutants in test effectiveness evaluations. Therefore, we are interested in checking the correlations between the failure rates of the sets of the injected faults we introduce and the real ones. To this end, we ask:

[3]Mutants are coupled with real faults if they are killed only by test cases that also reveal the real faults

RQ4 *(Failure estimates):* Are the injected faults leading to failure estimates that are representative of the real ones? How do these estimates compare with mutation testing?

The difference of RQ4 from the other RQs is that in RQ4, a set of injected faults is evaluated while, in the previous RQs only isolated mutant instances.

4.5 Experimental Setup

4.5.1 Dataset & Benchmark

To evaluate iBiR we needed a set of benchmark programs, faults and bug reports. We decided to use Defects4J [130] since it is a benchmark that includes real-world bugs and it is quite popular in software engineering literature.

Linking the bugs with their related reports

We used the bug-report to revision-id (commit) mapping provided by the Defects4J dataset. Unfortunately, none of the provided revisions-ids for the projects Lang and Math were pointing to the actual git repositories. As the projects have been migrated into GitHub but the revision-ids didn't get updated in the dataset. So for these two projects, we mapped the bug reports with their corresponding bugs in Defects4J, by following the same process as in the study of Koyuncu et al. [42]. We used the bug-linking strategies that are available in Jira and used the approach of Fischer et al. [131] and Thomas et al. [132] to map the sought bugs with the corresponding reports. Precisely, we crawled the relevant bug reports and checked their links. We selected bug reports that were tagged as "BUG" and marked as "RESOLVED" or "FIXED" and have a "CLOSED" status. Then we searched the commit logs to identify related identifiers (IDs) that link the commits with the corresponding bug.

Additionally, because of limitations in our current IRFL implementation, we included only the projects that are using Jira as issue tracking software.

Our resulting bug dataset included the 316 faults of Defect4J related to the Cli (39), Codec (18), Collections (4), Compress (47), Csv (16), JxPath (22), Lang (64) and Math (106) projects. We discarded 36 defects because they were sharing the same bug report and we could not map the correct one with its related issue, or issues with the buggy program versions such as missing files from the repository, or execution issues, at the reporting time. This leaves us with a total of 280 faults.

4.5.2 Experimental Procedure

To compare the fault injection techniques we need to set a common basis for comparison. We set this basis as the number of injected faults since it forms a standard cost metric [21] that puts the studied methods under the same cost level. We used sets of 5, 10, 30, and 100 injected faults since our aim is to equip researchers with few representative faults, per targeted fault, in order to reach reasonable execution demands. To reduce the arbitrariness due to the stochastic nature of mutation testing, we reproduced the injection 15 times, then we sorted the executions by their average Ochiai coefficient (for every bug separately) and we reported the mean execution. In the other hand, we run iBiR only once as its approach does not depend on random decisions.

To measure how well the injected faults imitate the real ones (answer RQ1 and RQ2) we use a semantic similarity metric (Ochiai coefficient) between the test failures on the injected and real (targeted) faults. Precisely, let fTS_M and fTS_B be the sets of failing tests when executing a test suite TS correspondingly on a mutant M and a buggy project B, the Ochiai coefficient is 0 if any of fTS_M or fTS_B is empty, else is calculated as

$$\text{Ochiai}(M, B) = \frac{|fTS_M \cap fTS_B|}{\sqrt{|fTS_M|.|fTS_B|}}, \tag{4.1}$$

where $|set|$ denotes the set size. In our study, as we're executing the fixed-version test-suites provided by defects4j, every targeted bug breaks at least one test, thus, fTS_B is never empty. This coefficient quantifies the similarity level of the program behaviours exercised by the test suites and is often used in mutation testing literature [7]. The metric takes values in the range [0, 1] with 0 indicating complete difference and 1 exact match. We treated the injected faults that were not detected by any of the test suites as equivalent mutants [17], [49]. This choice does not affect our results since we approximate the program behaviours through the projects test suites, i.e., they are never killed.

To measure whether the injected faults couple with the existing ones (answer RQ3), we followed the process suggested by Just et al. [109] and identified whether there were any injected faults that were killed by at least one failing test (test that detects the real fault) and not by any passing test (test that does not detect the real fault). In RQ4 we randomly sampled 50 test suites, random subsets of the accompanied test suites, that included between 10% to 30% test cases of the original test suite (provided by defects4j). Thus, we ensure that the selected samples (1) are smaller than the original test suite, (2) have different sizes and (3) different ratios of killing the mutants and detecting the targeted bug. Then we recorded the ratios of the injected faults that are detected when injecting 5, 10, 30 and 100 faults. We also recorded binary variables indicating whether or not each test suite detects the targeted fault. This process simulates cases where test suites of different strengths are compared. Based on these data, we computed two statistical correlation coefficients, the Kendall and Pearson.

To further validate whether the two approaches provide sufficient indicators on the effectiveness of the test suites, we check whether the detection ratios of the injected faults are statistically higher when test suites detect the targeted faults than when they do not.

To reduce the influence of stochastic effects we used the Wilcoxon test with a significance level of 0.05. This helped deciding whether the differences we observe can be characterised as statistically significant. Statistical significance does not imply sizable differences and thus, we also used the Vargha Delaney effect size $\hat{A}_{12}$ [133]. In essence, the $\hat{A}_{12}$ values quantify the level of the differences. For instance, a value $\hat{A}_{12} = 0.5$ can be interpreted as a tendency of equal value of the two samples. $\hat{A}_{12} > 0.5$ suggest that the first set has higher values, while $\hat{A}_{12} < 0.5$ suggest the opposite.

4.5.3 Implementation

To perform our experiments, we implemented IBIR's approach as described in Section 4.3: we have used the IRFL implementation proposed in iFixR [42] and implemented the mutator component which is responsible of injecting faults in specific locations, as a java standalone application. Second, for the mutation testing, denoted as "Mutation" in our experiments, we used randomly sampled mutants from those produced by typical mutation operators, coming from mutation testing literature. In particular we implemented the muJava intra-method mutation operators [89], which are the most frequently used [22]. Third to reduce the noise from stillborn mutants, i.e., mutants that do not compile, we discarded without taking into any consideration, i.e., prior to our experiment, every mutant that did not compile or its execution with the test suite exceeded a timeout of 5 minutes. Fourth, when answering the RQ1, we found out that there were many cases where IBIR injected less than 100 faults. To perform a fair comparison, we discarded these cases (for both approaches). This means that we always report results where both studied approaches manage to inject the same number of faults.

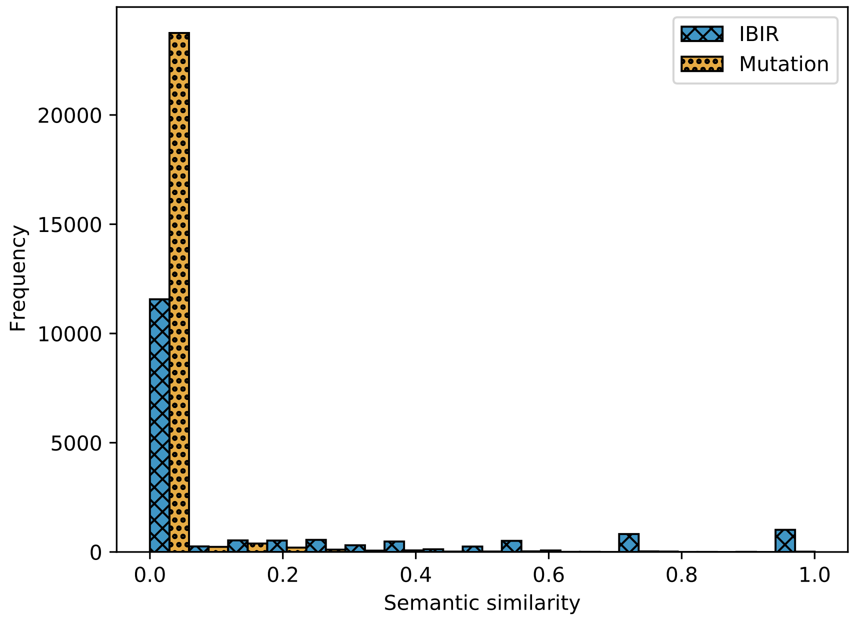
4.6 Results

4.6.1 RQ1: Semantic similarity between iBiR injections and the targeted real faults

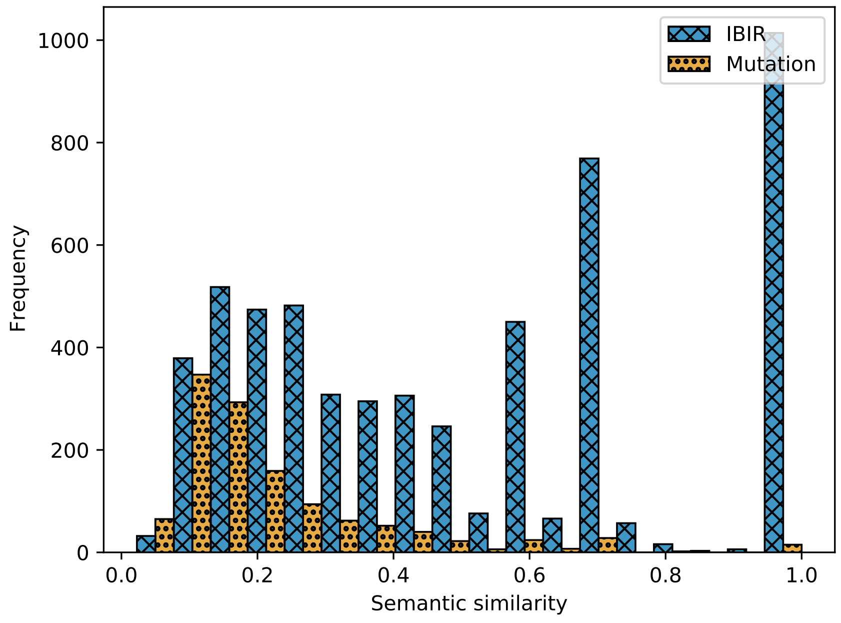
To check whether the injected faults imitate well the targeted ones, we measured their behaviour (semantic) similarity w.r.t. the project test suites (please refer to Section 4.5 for details). Figure 4.4 shows the distribution of the similarity coefficient values that were recorded in our study. As can be seen, iBiR injects hundreds of faults that are similar to real ones, whereas mutation testing (denoted as Mutation in Figure 4.4) did not manage to generate any. At the same time, as typically happens in mutation testing [7], a large number of injected faults have low similarity. This is evident in our data, where mutations have 0 similarity.

To investigate whether iBiR successfully injects any fault that is similar (semantically) to the targeted ones, we collected the best similarity coefficients, per targeted fault, when injecting 5, 10, 30 and 100 faults. Figure 4.5 shows the distribution of these results. For more than half of the targeted faults, iBiR yields a best similarity value higher than 0.5, when injecting 100 faults, indicating that iBiR's faults imitate relatively well the targeted ones. We also observe that in many faults the best similarity values are above 0 by injecting just 10 faults. This is important since it indicates that iBiR successfully identifies relevant locations for fault injection.

To establish a baseline and better understand the value of iBiR, we need to contrast iBiR's performance with that of mutation testing when injecting the same number of faults. Mutation testing forms the current SOA of fault injection and thus a related baseline. As can be seen from Figure 4.5, the similarity values of mutation testing are significantly lower than those of iBiR.



(a) All injected faults.



(b) Faults with an Ochiai coefficient higher than zero.

Figure 4.4: Distribution of semantic similarities of 100 injected faults per targeted (real) fault.

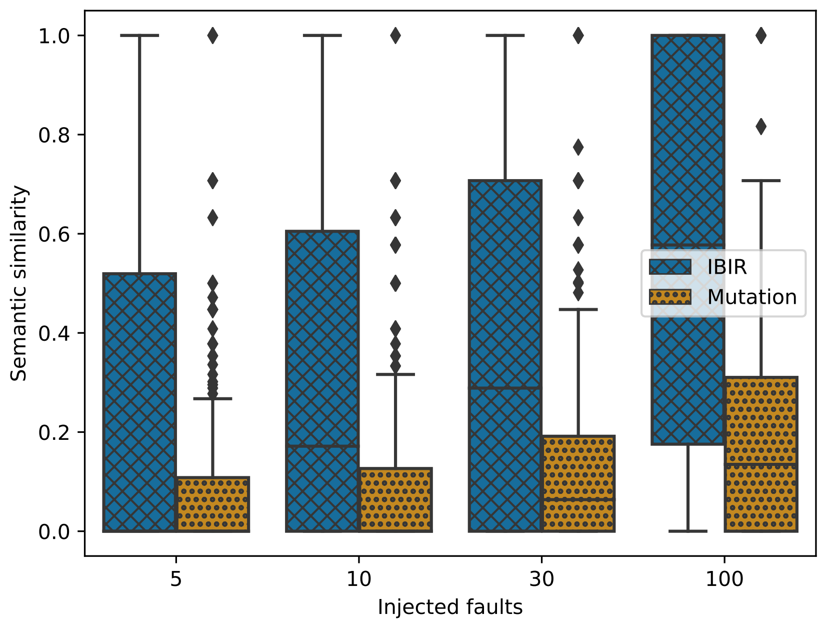


Figure 4.5: Semantic similarity per targeted (real) fault, top values. IBIR injects faults with higher similarity coefficients than mutation testing.

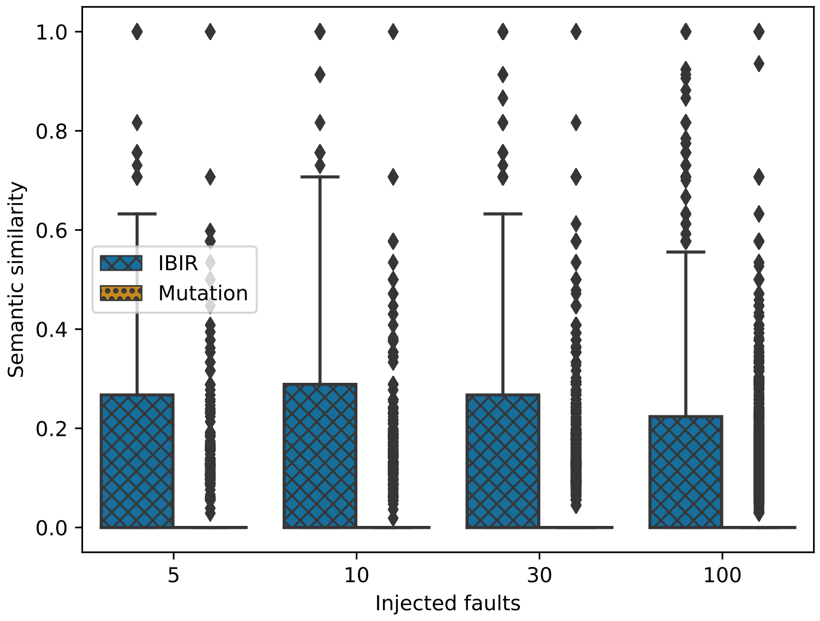


Figure 4.6: Semantic similarity of all injected faults. iBiR injects faults with higher similarity coefficients than mutation testing.

iBiR injects faults that resemble those described in Bug Reports. iBiR injects a fault that imitates the real targeted one, significantly better than traditional mutation testing.

Figure 4.6 shows the distribution of the semantic similarities, between real and injected faults, when injecting 5, 10, 30 and 100 faults. As can be seen from the boxplots, the trend is that a large portion of faults injected by iBiR have positive similarity scores with the targeted ones.

Interestingly, in mutation testing, only outliers have their similarity above 0. In particular, mutation testing injected faults with similarity values higher than 0 in 87, 112, 145, 189 of the targeted faults (when injecting 5, 10, 30, 100 faults), while iBiR injected in 130, 156, 190, 226 of the targeted faults, respectively.

To validate this finding, we performed a statistical test (Wilcoxon paired test) on the data of both figures 4.5 and 4.6 to check for significant differences. Our results showed that the differences are significant, indicating the low probability of this effect to be happening by chance. The size of the difference is also big, with IBIR yielding $\hat{A}_{12}$ values between 0.64 and 0.68 indicating that IBIR injects faults with higher semantic similarity to real ones in the great majority of the cases. Due to the many cases with 0 similarity values, the average similarity values of IBIR's faults is 0.163, while for mutation it is 0.010, indicating the superiority of IBIR.

IBIR injects faults that better resemble real faults, than traditional mutation testing, in 64%-68% of the cases.

4.6.2 RQ2: IBIR Vs Mutation Testing at particular classes

To check the performance of IBIR at the class level of granularity we repeated our analysis by discarding, from our priority lists, every mutant that is not located on the targeted classes, i.e., classes where the targeted faults have been fixed. Figure 4.7 shows the distribution of the semantic similarities when injecting 5, 10, 30 and 100 faults at a particular class. As expected, mutation testing scores are higher than those presented before, but still mutation testing falls behind.

To validate this finding, we performed a statistical test and found that the differences are significant. The size of the difference is between 0.62 and 0.65, meaning that IBIR scores more than 60% times higher than mutation testing. The average similarity values of the IBIR faults is 0.217, while for mutation is 0.066, indicating that IBIR is better.

IBIR outperforms traditional mutation testing, imitating real faults, even when restricted to a particular (target) class. The difference is significant with IBIR scoring more than 60% of the time higher than mutation testing.

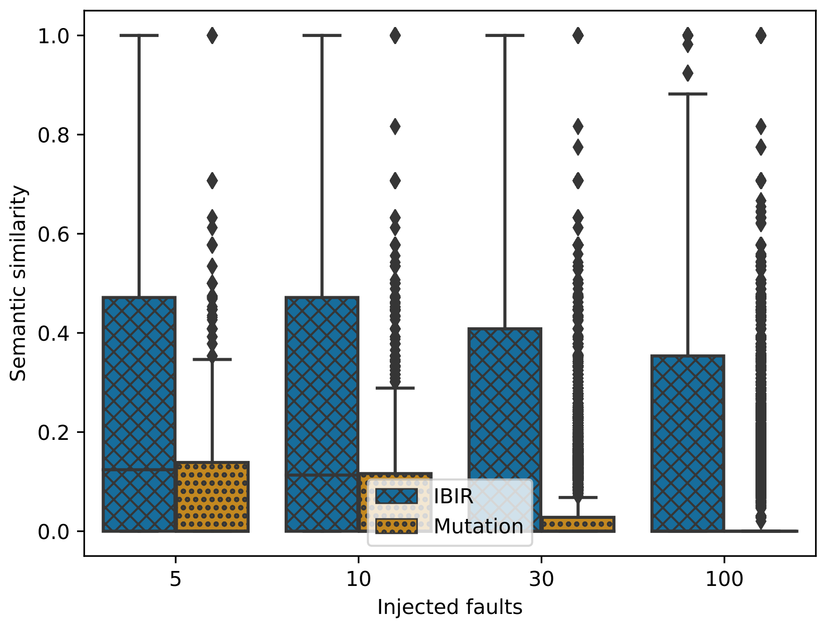


Figure 4.7: Semantic similarity of injected faults at particular classes. IBIR injects faults with higher similarity coefficients than mutation testing (at class level granularity).

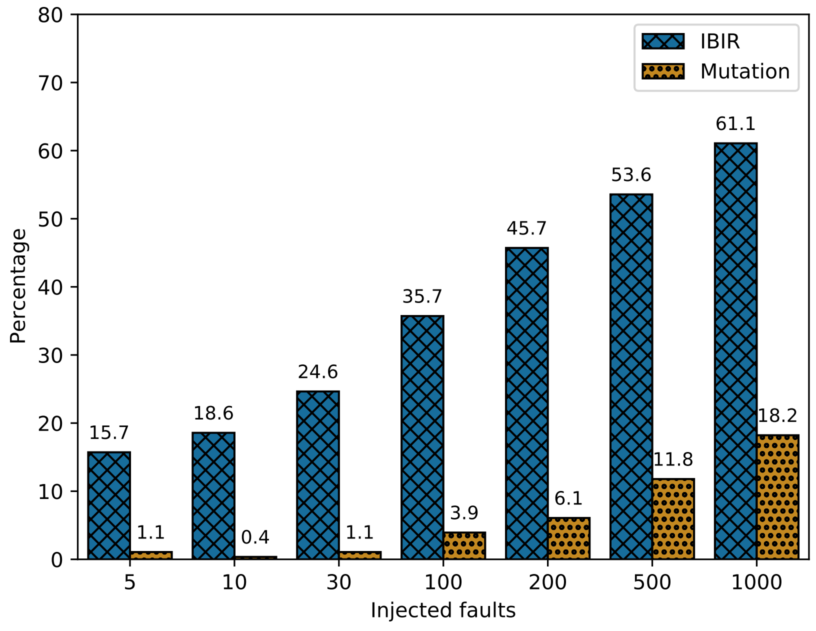


Figure 4.8: Percentage of real faults that are coupled to injected ones when injecting 5 to 1000 faults.

4.6.3 RQ3: Fault Coupling

The coupling between the injected and the real faults forms a fundamental assumption of the fault-based testing approaches [130]. An injected fault is coupled to a real one when a test case that reveals the injected fault also reveals the real fault [130]. This implies that revealing these coupled injected faults results in revealing potential real ones. We therefore, check this property in the faults we inject and contrast it with the baseline mutation testing approach.

Figure 4.8 shows the percentage of targeted faults where there is at least one injected fault that is coupled to a real one. This is shown for the scenarios where 5, 10, 30 and 100 faults, per target, are injected. As we can see from these data, IBIR injects coupled faults for approximately 16% of the target faults when it aims at injecting 5 faults.

Table 4.3: Vargha and Deianey $\hat{A}_{12}$ (iBiR VS Mutation) of Kendall and Pearson correlation coefficients.

Number of injected faults	**5**	**10**	**30**	**100**
Kendall	0.605	0.620	0.681	0.655
Pearson	0.580	0.612	0.627	0.652

This percentage increases to 36% when the number of injected faults is increased to 100.

Perhaps surprisingly, mutation testing did not perform well (it injected coupled faults for around 4% of the targeted, when injecting 100 faults per target). These results differ from those reported by previous research [7], [109], because a) previous research only injected faults at the faulty classes and not the entire project and b) previous research injected all possible mutant instances and not 100 as we do.

iBiR injects coupled faults for approximately 16%-36% of the cases, while mutation testing does it in around 4%. This is achieved by injecting 5-100 faults.

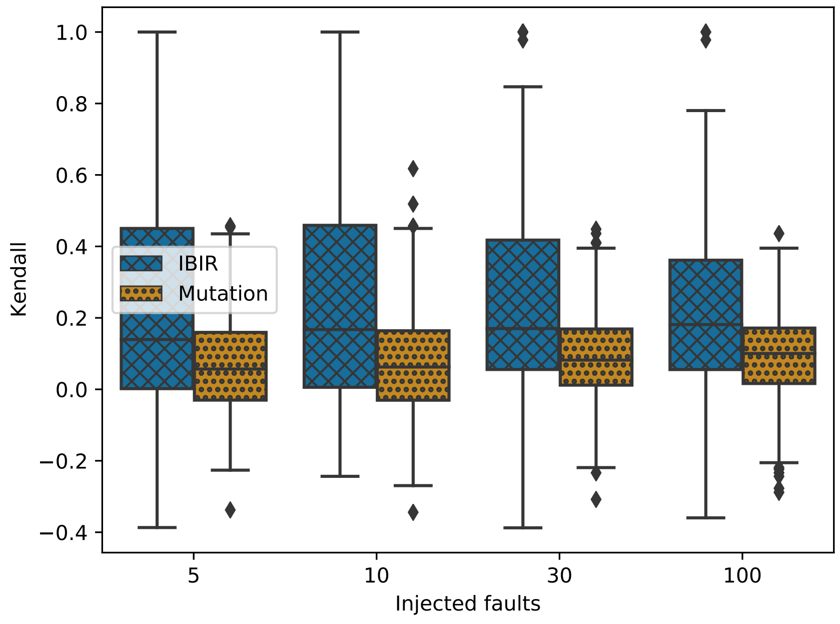
4.6.4 RQ4: Fault detection estimates

The results presented so far provide evidence that some of the injected faults imitate well the targeted ones. Though, the question of whether the injections provide representative results of real faults remains, especially since we observe a large number of faults with low similarity values. Therefore, we check the correlations between the failure rates of the sets of injected faults and the real faults when executed with different test suites, (please refer to section 4.5 for details).

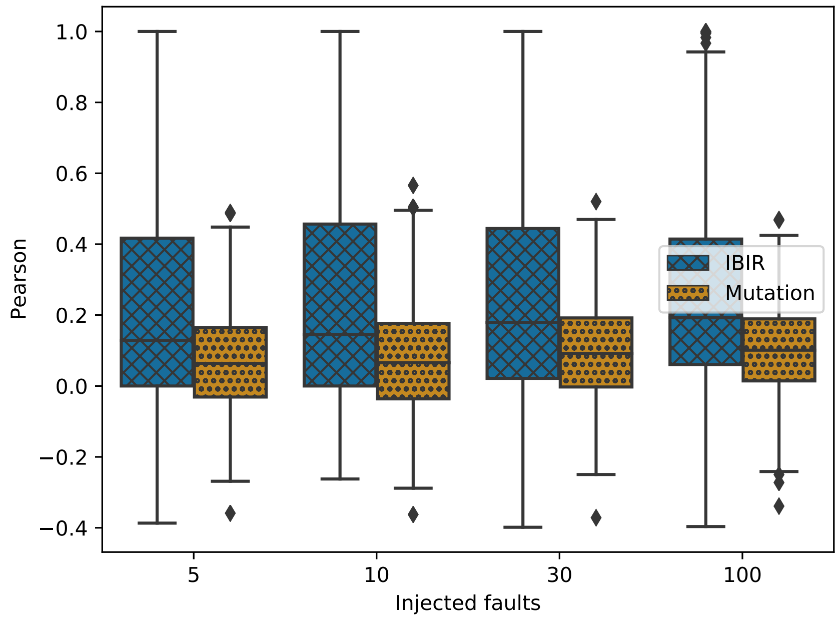
Figure 4.9 shows the distribution of the correlation coefficients, when injecting different numbers of faults. Interestingly, the results on both figures show a trend in favour of iBiR. This difference is statistically significant, shown by a Wilcoxon test, with an effect size of approximately 0.6. Table 4.3 records the effect size values, $\hat{A}_{12}$, for the examined strategies. In essence, these effect sizes mean that iBiR outperforms the mutant injection in 60% of the cases, suggesting that iBiR could be a much better choice than mutation testing, especially in cases of large test suites with expensive test executions.

To further validate whether iBiR's faults provide good indicators (estimates) of test effectiveness (fault detection) we split our test suites between those that detect the targeted faults and those that do not. We then tested whether detection ratios of the injected faults in the test suite group that detects the real faults are significantly (statistically) higher than those in the group that does not detect it. In case this happens, we have evidence that our injected faults favour test suites capable of detecting real faults. This is important when comparing test generation techniques, where the aim is to identify the most effective (at detecting faults) technique.

Figure 4.10 records the number of faults where (real) fault detecting test suites detect a statistically higher number of injected faults than those test suites that do not detect them. As can be seen by these results, iBiR differs greatly from mutation, i.e., it distinguishes between passing and failing test suites in 126 faults, while Mutation in 55 faults. We also measured the Vargha and Delaney $\hat{A}_{12}$ effect size values on the same data, recorded in Figure 4.11. Of course it does not make sense to contrast insignificant cases, so we only performed that on the results where iBiR has statistically significant difference. Interestingly big differences are recorded (in approximately 80% of the cases) in favour of our approach.



(a) Kendall



(b) Pearson

Figure 4.9: Correlation coefficients of test suites (samples from the original project test suite). The two related variables are a) the percentage of injected faults that were detected by the sampled test suites and b) whether the targeted fault was detected or not by the same test suites.

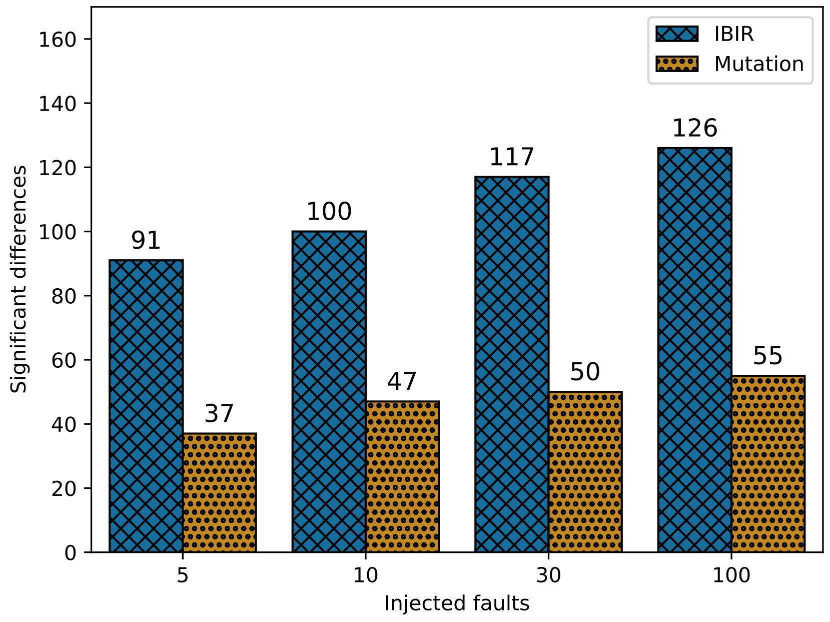


Figure 4.10: Number of (real) faults where injected faults provided good indications of fault detection. Particularly, number of cases with statistically significant difference, in terms of ratios of injected faults detected, between failing and passing test suites (w.r.t. real faults).

IBIR injects faults that provided better fault detection estimates than traditional mutation testing in approximately 80% of the cases.

4.7 Discussion

The effectiveness of IBIR in generating faults that are similar to real ones is endorsed by its two main components: the IRFL and the mutator. The IRFL indicates where the faults need to be injected and the mutator decides what changes should be made, depending on the AST tree of each location.

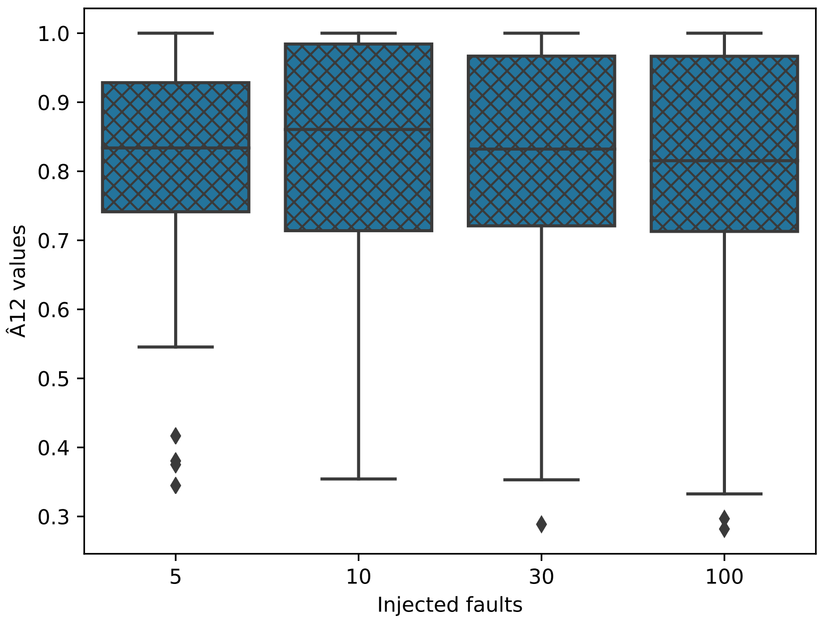


Figure 4.11: Vargha and Delaney values for IBIR. $\hat{A}_{12}$ values computed on the detection ratios of injected faults of the test suites that detect and do not detect the (real) faults.

Table 4.4: Percentage of injected faults that are coupled to real ones when injecting 5 to 1000 faults.

Number of injected faults	5	10	30	100	200	500	1000
IBIR	5.93%	5.61%	5.78%	5.23%	4.57%	3.43%	2.57%
Mutation	0.29%	0.04%	0.05%	0.06%	0.06%	0.07%	0.07%

Particularly, compared to conventional mutation testing, we can see that the IRFL is narrowing down the area of injection to the source-code features described by the bug report, while the patterns-set of IBIR extends the injection possibilities in that area. In the other hand, conventional mutation testing targets all the source code and injects faults only in statements where their operators are applicable. For instance, applying the typical mutation operators - the `Mutate Operators` and `Remove Statement` ones - on a specific area of code would not induce any fault, if no statement can be removed without breaking the compilation, or there is no operator to mutate. While in such case, IBIR may inject faults by applying other patterns like mutating the method invocation or the used parameters or inserting a statement, etc.

4.7.1 Injecting large number of faults

The Figure 4.8 shows that IBIR injects much more faults that couple with the real ones than conventional mutation testing. In fact, it achieves a higher coupling percentage when injecting only 10 faults than the percentage achieved by conventional mutation testing when injecting 1000 faults. We can see also that when injecting 1000 faults we achieve the coupling percentages of 61.1% and 18.2% for respectively IBIR and mutation testing. This is obviously because the more faults we inject, the more chances we have to inject faults that couple with the real ones. Considering that injecting more faults comes with a considerable consequent cost-increase, as the practitioners will need more time to analyse the produced mutants, this option is often not favoured in practice, where it is better to have few relevant faults than many.

To have a better understanding of the impact of injecting multiple faults, we illustrate in Table 4.4 the averaged faults-coupling success-rates when injecting 5, 10, 30, 100, 200, 500 and 1000 faults with IBIR and mutation testing. We define the success rate as the percentage of coupled faults among all the injected ones. As an example, a coupling success-rate of 5% corresponds to 5 coupled faults when injecting 100 ones. In our study, IBIR achieves a much higher success rate than mutation testing: 20, 87, 49 and 36.7 times higher when injecting 5, 100, 500 and 1000 faults. Even if the coupling percentage increases by injecting more faults (Figure 4.8), we can see that the more we inject faults, the more the success rate decreases for IBIR. This is a direct consequence of the decrease of the injection-locations likelihood to be related to the targeted bug-report. As we explain further in Section 4.3, IBIR starts by injecting faults in the highly ranked code locations found by its IRFL then iterates further until all locations are treated or the requested number of faults has been injected. So the higher the requested number of injected faults is, the more faults in lower ranked locations are injected. In the other hand, we see that the success rate of conventional mutation testing remains relatively low and far behind the one of IBIR. For instance, it remains at 0.07% even when doubling the number of injected faults from 500 to 1000. In Table 4.4, we notice that injecting 5 faults with mutation testing achieves a success rate of 0.29% which is much higher than the ratios achieved when injecting more faults, by the same technique. This is caused by the randomness in the conventional mutation testing results.

4.7.2 Distribution of the patterns inducing most effective injections

To understand better the impact of the used patterns in injecting faults that are similar to real ones, we grouped the faults by their creating patterns and compared the sizes of

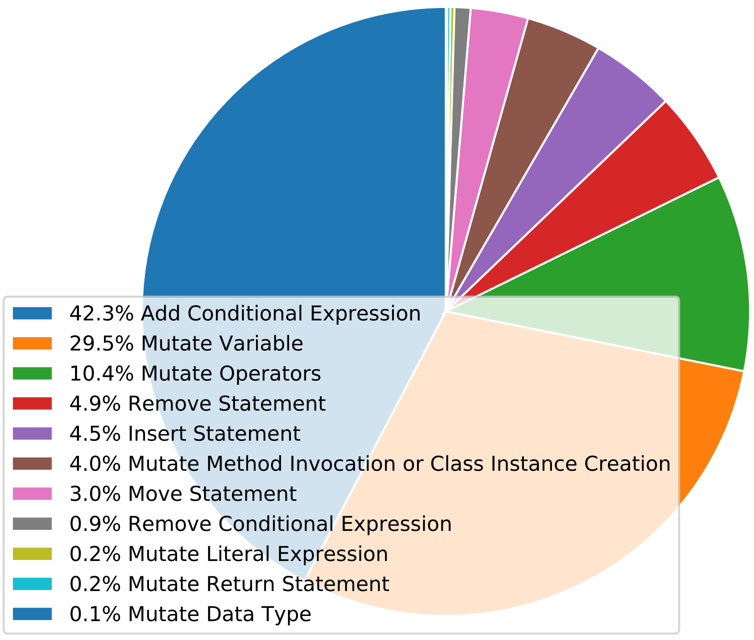


Figure 4.12: Distribution of the patterns inducing mutants with an Ochiai coefficient higher than 0.8 for IBIR when injecting 1000 faults.

each group. Figure 4.12 illustrates the proportion of every pattern' induced faults that have high Ochiai Coefficients (more than 0.8), when injecting 1000 faults by IBIR in the current dataset. Clearly, more than 70% of the faults with high similarity coefficients have been generated by patterns that are not commonly used in conventional mutation testing techniques: mainly by adding conditional expressions (42.3%) or by mutating variables (29.5%). This is significantly higher than the 15.3% generated with the commonly used conventional mutation operators (10.4% by mutating operators and 4.9% by removing statements). This highlights the fact that IBIR's patterns are bringing a clear advantage over mutation-testing.

These percentages and the general performance of every pattern depends on the targeted bug-report and the project nature. For instance, the low percentages of multiple patterns in Figure 4.12 can be the consequence of multiple factors, such as: 1) the fact that some faults are occurring less frequently in the current dataset or 2) the fact that some patterns are only applicable on few specific statement-ASTs or 3) that some patterns produce relatively more mutants in the same location, thus have higher percentages (i.e. the "Mutate Method Invocation" which induced Fault 1 and Fault 2 in the same statement in Figure 4.3 in Section 4.3.4).

4.7.3 IBIR Vs typical mutation operators

Early research on mutation testing defined mutation operators based on all possible simple removals or replacements of programming language elements [134], [135]. This practice was then adopted when defining mutation operators for other languages, such as Java, and in defining object oriented related mutants [89], [136]. To reduce the number of mutants involved, many tool developers applied a restrictive set of mutation operators, usually referred to as the 5-operator set, based on the selective mutation testing studies performed by Offutt et al. [21], [137] with the result that the majority of modern mutation testing tools implementing a version of this 5-operator set together with some deletion operators [9], [138].

In view of the above all the IBIR injections that involve addition of code elements, i.e., “Insert Statement” and “Mutate Return Statement” categories of Table 4.2, are fundamentally different from what has been used in mutation testing studies over the years. The “Mutation Literal Expression” category is also something that has not been used by mutation testing studies. The rest of the operators have some similarities with operators used in some studies overall differ significantly from the operators used by any single tool or study. In the following we provide a detailed list of IBIR operators and their related similarities (or novelties) with respect to other studies.

Operators that have not been used by other studies:

- Insert Statement : *Insert a method call, Insert a return statement, Wrap a statement with a try-catch, Insert an if checker*.
- Mutate Conditional Expression : *Insert a conditional expression*.
- Mutate float or double Division : *Remove a float or a double cast from the divisor, Remove a float or a double cast from the dividend, Replace float or double multiplication by an int division*.
- Mutate Literal Expression : *Change boolean, number or string literals in a statement by another literal or expression of the same type*.
- Mutate Return Statement : *Replace a return expression by an other one*.

Operators that have similarities with those used by other studies:

- Mutate Class Instance Creation : *Replace an instance creation call by a cast of the super.clone() method call*. Similar to the class mutation operators of MuJava [136].

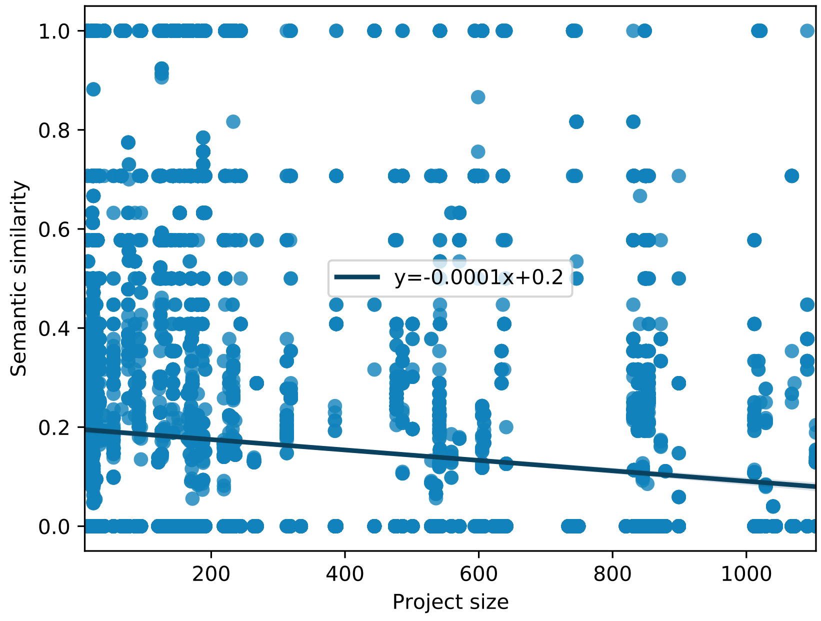
- Mutate Data Type : *Change the declaration type of a variable, Change the casting type of an expression*. Similar to the interface mutation in C [135], [139].
- Mutate Method Invocation : *Replace a method call by another one, Replace a method call argument by another one, Remove a method call argument, Add an argument to a method call*. Similar to the interface mutation [139].
- Mutate Variable : *Replace a variable by another variable or an expression of the same type*. Similar to the variable mutations in C [135].
- Move Statement: *Move a statement to another position*. Similar to the move out of a loop operators in C [135], [139].

Operators that are frequently used by other studies:

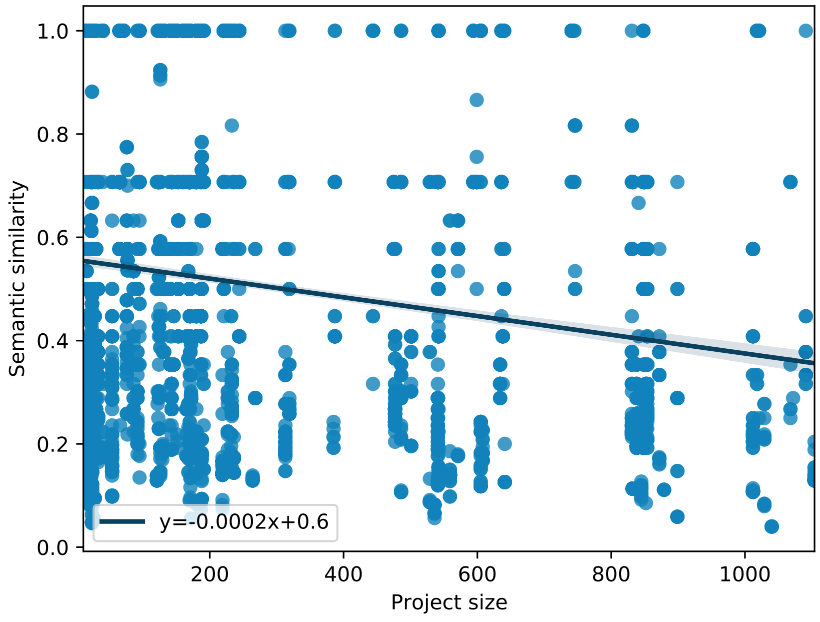
- Mutate Conditional Expression : *Remove a conditional expression, Change the conditional operator* [134], [135].
- Remove Statement : *Remove a statement, Remove a method* [94], [134], [138].
- Mutate Operators : *Replace an Arithmetic operator, Replace an Assignment operator, Replace a Relational operator, Replace a Conditional operator, Replace a Bitwise or a Bit Shift operator, Replace an Unary operator, Change arithmetic operations order* [94], [134], [138].

4.7.4 Project size and IBIR's effectiveness

Considering the fault injection as a search task where the target is injecting faults similar to real ones and the search space is the combination of the source-code locations and mutation possibilities, we were interested in assessing IBIR's performance for different project sizes. Figure 4.13a and Figure 4.13b show the scatter plots of the semantic similarity by the project size in terms of number of classes. Figure 4.13a and



(a) All injected faults.



(b) Faults with an Ochiai coefficient higher than zero.

Figure 4.13: Correlation between the semantic similarities and the project size (100 injected faults per targeted (real) fault).

Figure 4.13b consider respectively all the injected faults and the faults having an Ochiai coefficient higher than zero. We can see that the project size has no impact on the effectiveness of iBiR.

4.8 Threats to Validity and Limitations

The question of whether our findings generalise, forms a typical threat to validity of empirical studies. To reduce this threat, we used real-world projects, developer test suites, real faults and their associated bug reports, from an established and independently built benchmark. Still though, we have to acknowledge that these may not be representative of projects from other domains. In addition, as the approach's injection depends on the input bug reports, its effectiveness may be impacted by the content of the reports, such as partial/incomplete or vague descriptions. To reduce this threat, we have run our experiments with all available bug reports in the studied dataset without any particular selection and got encouraging results. We acknowledge though that the results may vary depending on the information provided in the reports. In practice, one should make a careful selection of bug reports based on which iBiR could be applied to avoid such cases. Nevertheless, the appropriate selection of bug reports falls outside the scope of this work and has been left open for future research.

Other threats may also arise from the way we handled the injected faults and mutants that were not killed by any test case. We believe that this validation process is sufficient since the test suites are relatively strong and somehow form the current state of practice, i.e., developers tend to use this particular level of testing. Though, in case the approach is put into practice things might be different. We also applied our analysis on the fixed program version provided by Defects4J. This was important in order to show that we actually inject the actual targeted faults. Though, our results might not hold on the cases that the code has drastically changed since the time of the bug report. We believe that this threat is not of actual importance as we are concerned with fault injection at interesting program locations, which should be pinpointed by the fault localisation technique we use. Still future research should shed some light on how useful these locations and faults are.

Furthermore, some implementation changes of iBiR may improve its usability. For instance, adding an advanced integrity check before applying the patterns would shorten the execution time of the tool. As currently, the generated faulty programs are mainly validated via the compilation, only 52% of the mutants are compilable and thus outputted, while the rest are discarded. Also, one can consider using the same approach with different IRFL techniques. This would eliminate the training cost and reduce the eventual risk of threats that may be induced by the machine-learning module currently used to rank the suspicious files. In fact, some of the projects in our evaluation set has been used during the training phase of this latter. Although, we did not notice any bias or bad impact on our results, we are aware that this can be considered as an additional threat to validity. However, these threats concern only the file-level localisation of the IRFL and not the statement-level one, thus, they would not impact its results. This is because the IRFL is performing a VSM cosine-similarity to rank the suspicious statements without involving any machine learning technique in this step, as explained in Section 4.3.1.

Finally, our evaluation metrics may induce some additional threats. Our comparison basis measurement, i.e., number of injected faults, approximates the execution cost of the techniques and their chances to provide misleading guidance [7], while the fault couplings and semantic similarity metrics approximate the effectiveness of the approaches. These are intuitive metrics, used by previous research [95], [100] and aim at providing a common ground for comparison.

4.9 Conclusion

We presented iBiR; a bug-report driven fault injection tool. iBiR (1) equips researchers with faults (to inject) targeting the critical functionality of the target systems, (2) mimics real faulty behaviour and (3) makes relevant fault injection.

iBiR's use case is simple; given a program and some bug reports, it injects faults emulating the related bugs, i.e., iBiR generates few faults per target bug report. This allows constructing realistic fault pools to be used for test or fault tolerance assessment.

This means that iBiR's faults can be used as substitutes of real faults, in controlled studies. In a sense, iBiR can bring the missing realism into fault injection and therefore support empirical research and controlled experiments. This is important since a large number of empirical studies rely on artificially-injected faults [10], the validity of which is always in question.

While the use case of iBiR is in research studies, the use of the tool can have applications in a wide range of software engineering tasks. It can, for instance, be used for asserting that future software releases do not introduce the same (or similar) kind of faults. Such a situation occurs in large software projects [140], where iBiR could help by checking for some of the most severe faults experienced. Testers could also use iBiR for testing all system areas that could lead to similar symptoms than the ones observed and resolved. This will bring benefits when testing software clones [141] and similar functionality implementations.

Another potential application of iBiR is fault tolerance assessment, by injecting faults similar to previously experienced ones and analysing the system responses and overall dependability. We hope that we will address these points in the near future.

To support this research and enable reproducibility, we have made our data and code available [126].

Chapter 5

Context-aware and Natural Mutation Testing via Pre-Trained Language Models

Mutation testing operates by seeding faults into a program under test and asking developers to write tests that reveal them. To this end, mutation testing should seed faults that are both “natural” in a sense easily understood by developers and strong (have high chances to couple with real bugs). To achieve this we propose using pre-trained generative language models (i.e. CodeBERT) that have the ability to produce developer-like code that operates similarly, but not exactly, as the target code. This means that the models have the ability to seed natural faults, thereby offering opportunities to perform mutation testing. We realise this idea by implementing μBERT, a mutation testing technique that performs mutation testing using CodeBERT and empirically evaluated it using 689 faulty program versions. Our results show that the fault revelation ability of μBERT is higher than that of a state-of-the-art mutation testing (Pitest), yielding tests that have up to 17% higher fault detection potential than that of Pitest. Moreover, we observe that μBERT can complement Pitest, being able to detect 47 bugs missed by Pitest, while at the same time, Pitest can find 13 bugs missed by μBERT.

This chapter is based on the following article:

Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Efficient Mutation Testing via Pre-Trained Language Models. Submitted to : *IEEE Transactions on Software Engineering (TSE 2023)*. 2023. https://doi.org/10.48550/arXiv.2301.03543

5.1 Introduction

Mutation testing aims at seeding faults using simple syntactic transformations [29]. These transformations, also known as mutation operators are typically constructed based on syntactic rules crafted based on the grammar of the target programming language [3], i.e. replacing an arithmetic operator with another such as a `+` by a `-`. Unfortunately, such techniques generate mutants (seeded faults), many of which are "unatural", i.e., non-conforming to the way developers code, thereby perceived as unrealistic by developers [142]. At the same time, the syntactic-based fault seeding fails to capture the semantics of the code snippets that they apply, leading to numerous trivial or low utility faults [50].

To deal with the above issue we propose forming natural mutations by using big code. Thus, we aim at introducing modifications that follow the implicit rules, norms and coding conventions followed by programmers, by leveraging the capabilities of pre-trained language models to capture the underlying distribution of code and its writing, as learned by the pre-training process on big code.

To this end, we rely on CodeBERT [84], an NL-PL bimodal language model that has been trained on over 6.4 million programs. More precisely, we use its Masking Modelling Language (MLM) functionality, which given a code sequence with a masked token, predicts alternative replacements to that token, that is best matching the sequence context. This is important, since the predictions do not follow fixed predefined patterns as is the case of conventional mutation testing, but are instead adapted to fit best the target code. For instance, given a sequence `int a = 1;`, we pass a masked version of it as `int a = <mask>;`, then CodeBERT by default proposes 5 predictions sorted by likelihood score: `0`, `1`, `b`, `2`, and `10`. Being the most likely fitting tokens to the code context, our intuition is that replacing the masked token with these predictions would induce "natural" mutants.

Precisely, we introduce μBERT, a context-aware mutation testing approach that leverages a pre-trained language model (CodeBERT) to restrict the mutant space to the most probable code alterations, thereby, producing more natural mutants. μBERT iterates through the program statements and modifies their tokens by the model predictions.

Our results of the previous Chapter 4, particularly the distribution of patterns inducing faults with high semantic similarity to real ones in Figure 4.12, shows that some real bugs are best resembled by using complex patterns, i.e. patterns that require more than one token mutation. To account for such cases, μBERT is equipped with additive mutations, i.e., mutations that add code (instead of deleting or altering). For example, consider a boolean expression e_1 (typically present in `if`, `do`, `while` and `return` statements), which is mutated by μBERT by adding a new condition e_2, thereby generating a new condition $e_1 || e_2$ (or $e_1 \&\& e_2$), which is then masked and completed by CodeBERT. For instance, given a condition `if(a == b)`, μBERT produces a new condition `if(a == b || a > 0)` that is masked and produces `if(a == b || b > 0)`.

We implement μBERT, and evaluate its ability to serve the main purposes of mutation testing, i.e. guiding the testing towards finding faults. We thus, evaluate μBERT effectiveness and cost-efficiency in revealing[1] 689 bugs from Defects4J. Our results show that μBERT is very effective in terms of fault revelation, finding on average 84% of the faults. This implies that μBERT mutants cover efficiently faulty behaviours caused by real bugs.

[1]Tests are written/generated to kill (reveal) the mutants. A bug is revealed by a mutation testing approach, if the written tests to kill its mutants also reveal the bug.

More importantly, the approach is noticeably more effective and cost-efficient than a traditional mutation testing technique, namely Pitest [90], that we use as a baseline in our evaluation. Precisely, we consider three different configurations for Pitest that uses different sets of mutation operators (DEFAULT, ALL and RV[2]). In fact, test suites that kill all mutants of μBERT find on average between 5.5% to 33% more faults than those generated to kill all mutants introduced by Pitest. Moreover, even when analysing the same number of mutants, μBERT induces test suites that find on average 6% to 16% more faults than Pitest. These results are promising and endorse the usage of μBERT over the considered mutation testing technique, as a test generation and assessment criterion.

We also study the impact of the condition-seeding-based mutations in the fault detection capability of μBERT. We observe that test-suites designed to kill both kinds of μBERT mutants – induced by 1) direct CodeBERT predictions and 2) a combination of conditions-seeding with CodeBERT predictions – find on average over 9% more bugs than the ones designed to kill direct CodeBERT prediction mutants only (1).

Overall, our main contributions are:

- We introduce the first mutation testing approach that uses pre-trained language models, μBERT. It leverages the model's code knowledge captured during its pretraining on large code corpora and its ability to capture the program context, to produce "natural" mutants.
- We propose new additive mutations which operate by seeding new conditions in the existing conditional expressions of the target code, then masking and replacing their tokens with the model predictions.
- We provide empirical evidence that μBERT mutants can guide testing towards higher fault detection capabilities, outperforming those achieved by SOA tech-

[2]Research Version of Pit: it enables the usage of *ALL* Pit mutation operators in addition to experimental ones, that have been made available for researchers.

niques (i.e. Pitest), in terms of effectiveness and cost-efficiency. In our empirical study, we validate also the advantage of employing the new additive mutation patterns, w.r.t improving the effectiveness and cost-efficiency in writing test suites with higher fault revelation capability.

5.2 Approach

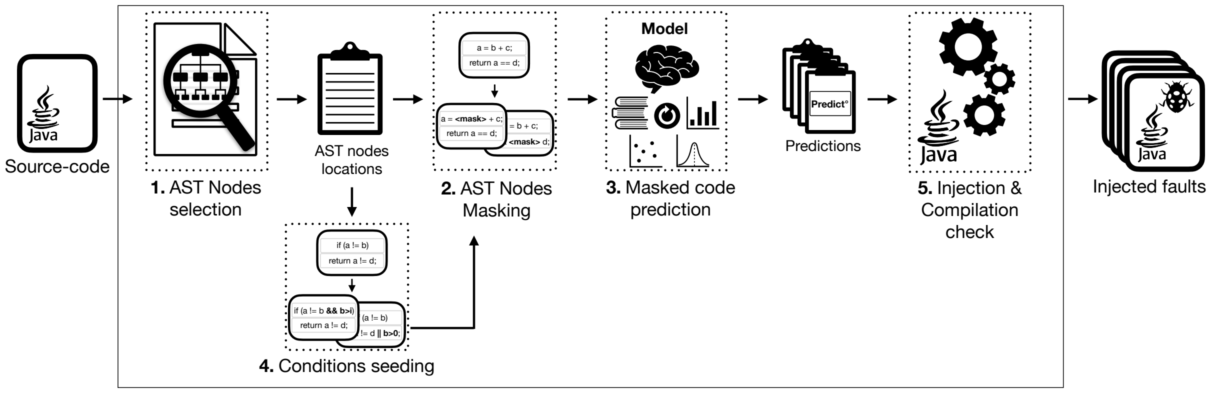


Figure 5.1: The Workflow of μBERT: (1) input Java code parsing, and extraction of the target expressions to mutate; (2) creation of simple-replacement mutants by masking the selected tokens and invoking the model; (3) mutants generation via masked tokens replacements with the corresponding model predictions; (4) complex mutants generation via a) conditions-seeding, b) tokens masking then c) replacement by CodeBERT predictions; and finally, (5) discarding of not compiling and syntactically identical mutants.

We propose μBERT, a generative language-model-based mutation testing approach, which is described step by step in Figure 5.1. Given an input source code, μBERT leverages CodeBERT's knowledge of code and its capability in capturing the program's context to produce "natural" mutations, i.e. that are similar to eventual developer mistakes.To do so, μBERT proceeds as follows in six steps:

1. First, it extracts relevant locations (AST [3] nodes) where to mutate

[3]AST: Abstract Syntax Tree.

2. Second, it masks the identified node-tokens, creating one masked version per selected token.

3. Then, it invokes CodeBERT to predict replacements for these masked tokens.

4. In addition to the mutants produced in Step (3), μBERT also implements some condition-seeding additive mutations that modify more than one token. Precisely, it modifies the conditional expressions in the control flow (typically present in `if`, `do`, `while` and `return` statements) by extending the original condition with a new one, combined with the logical operator `&&` or `||`. Then, the new conditional expression is mutated by following the same steps (2) and (3) – masking and replacing the masked tokens by the CodeBERT predictions.

5. Finally, the approach discards duplicate predictions or those inducing similar code to the original one, or not compiling, and outputs the remaining ones as mutants, from diverse locations of the target code. More precisely, it iterates through the statements in random order and outputs in every iteration one mutant per line, until achieving the desired number of mutants or all mutants are outputted.

5.2.1 AST Nodes Selection

μBERT parses the AST of the input source code and selects the lines that are more likely to carry the program's specification implementation, excluding the import statements and the declaration ones, e.g. the statements declaring a class, a method, an attribute, etc. This way, the approach focuses the mutation on the business-logic portion of the program and excludes the lines that are probably of lower impact on the program behaviour. It proceeds then, by selecting from each of these statements, the relevant nodes to mutate, i.e. the operators, the operands, the method calls and variables, etc., and excluding the language-specific ones, like the separators and the flow controls, i.e.

semicolons, brackets, `if`, `else`, etc. Table 5.1 summarises the type of targeted AST nodes by μBERT, with corresponding example expressions and induced mutants. We refer to these as the conventional mutations provided by μBERT, denoted by μBERT$_{conv}$ in our evaluation, previously introduced in the preliminary version of the approach [45].

5.2.2 Token Masking

In this step, we mask the selected nodes one by one, producing a masked version from the original source code for each node of interest. This means that every masked version contains the original code with one missing node, replaced by the placeholder `<mask>`.

This way, μBERT can generate several mutants in the same program location. For instance, for an assignment expression like `res = a + b`, μBERT can create up-to 25 mutants using the following masked sequences:

- `<mask> = a + b`
- `res <mask>= a + b`
- `res = <mask> + b`
- `res = a <mask> b`
- `res = a + <mask>`

5.2.3 CodeBERT-MLM prediction

μBERT invokes CodeBERT to predict replacements for the masked nodes. To do so, it tokenizes every masked version into a tokens vector then crops it to a subset one that fits the maximum size allowed by the model (512) and counts the masked token with the surrounding code-tokens. Next, our approach feeds these vectors to CodeBERT MLM to predict the most probable replacements of the masked token. Our intuition is that

Table 5.1: Example of μBERT conventional mutations, available in the preliminary version of the approach [45], denoted by μBERT_{conv}.

Ast node	**Expression**	**Masked Expression**	**Mutant Example**
literals	`res + 10`	`res + <mask>`	`res + 0`
identifiers	`res + 10`	`<mask> + 10`	`a + 10`
binary expressions	`a && b`	`a <mask> b`	`a || b`
unary expressions	`--a`	`<mask>a`	`++a`
assignments	`sum += current`	`sum <mask>= current`	`sum -= current`
object fields	`node.next`	`node.<mask>`	`node.prev`
method calls	`list.add(node)`	`list.<mask>(node)`	`list.push(node)`
array access	`arr[index + 1]`	`arr[<mask>]`	`arr[index]`
static type references	`Math.random() * 10`	`<mask>.random() * 10`	`Random.random() * 10`

the larger the code portion accompanying the mask placeholder, the better CodeBERT would be able to capture the code context, and consequently, the more meaningful its predictions would be. This step ends with the generation of five predictions per masked token.

5.2.4 Condition seeding

μBERT generates second-order mutants by combining condition seeding with CodeBERT prediction capabilities. To do so, our approach modifies the conditions in control flow and return statements, including `if`, `do`, `while` and `return` conditional expressions. For every one of these statements, it starts by extending the original condition by a new one, separated with the logical operator `&&` or ||, in both orders (original condition first or the other way around) and with or without negation (`!`).

Next, all substitute conditions are put one by one in place in the original code, forming multiple condition-seeded code versions, that we pass as input to Step (2), in which their tokens are masked and then (3) passed each to CodeBERT to predict the best substitute of their corresponding masked tokens.

The seeded conditions are created in two ways:

Using existing conditions in the same class

To mutate a given condition – `if`, `do`, `while` and `return` conditional expressions –, we collect all other conditions existing in the same class, then combine each one of them with the target condition, using logical operators.

Precisely, let Exp_t a conditional expression to mutate and $S_E = \{Exp_0, ..., Exp_n\}$ the set of other conditional expressions appearing in the same class, excluding the null-check ones (i.e. `var == null`). The alternative replacement conditions generated for Exp_t are the combinations of:

- Exp_t `op neg` Exp_i and
- Exp_i `op neg` Exp_t,

where `op` is a binary logical operator taking the values in {`&&`,`||`}, `neg` is either the negation operator `!` or nothing and Exp_i is a condition from S_E.

Using existing variables in the same class

When the target `if` conditional expression to mutate contains variables (including fields), we create new additional conditions by combining these variables with others of the same type from the same class. Then we combine each one of the newly created conditions with the original one, using logical operators.

Precisely, let Exp_t be a conditional expression to mutate containing a set of variables S_{vt}. For every variable var_t in S_{vt}, we load $S_v = \{var_0, ..., var_n\}$ the set of other variables appearing in the same class and of the same type T as var_t, then we generate the following new conditions:

- Exp_t `op` $(var_t\ rel_{op}\ var_i)$ and
- $(var_t\ rel_{op}\ var_i)$ `op` Exp_t,

where $\mathtt{op}$ is a binary logical operator taking the values in $\{\texttt{\&\&}, \texttt{||}\}$, rel_{op} is a relational operator applicable on the type T and var_i is a variable from S_v.

5.2.5 Mutant filtering

In this step, our approach starts by discarding accurate and duplicate predictions; the redundant predictions and the ones that are exactly the same as the original code. Then, it iterates through the statements and selects in every iteration one compilable prediction by line, while discarding not compilable ones. Once all first-order mutants are selected (issued by one single token replacement), our approach proceeds by selecting second-order ones (issued by the combination of condition seeding and one token replacement) in the same iterative manner. μBERT continues iterating until achieving the desired number of mutants or all mutants are outputted.

5.3 Research Questions

We start our evaluation by assessing the advantage brought by the additive mutations (a.k.a. conditions seeding ones) on the fault detection capability of our proposed approach. Thus we start by asking:

RQ1 *(μBERT Additive mutations)* What is the added value of the additive mutations on the fault detection capabilities of μBERT?

To answer this question, we generate two sets of mutants using μBERT: 1) the first set using all possible mutations that we denote as μBERT and 2) a second one using only the conventional μBERT' mutations – part of our preliminary implementation [45], excluding the additive ones – that we denote as μBERT_{conv}. Then we compare the fault detection ability of test suites designed to kill the mutants from each set.

The answer of this question provides evidence that the additive mutations increase the fault detection capability of μBERT. Yet, to assess its general performance we compare it to state-of-the-art (SOA) mutation testing, particularly Pitest [90], and thus, we ask:

RQ2 *(Fault detection)* How does μBERT compare with state-of-the-art mutation testing, in terms of fault detection?

To answer this question, we generate mutants using the latest version of Pitest [90], on the same target projects as for RQ1. As we are interested in comparing the approaches and not the implementations of the tools, we exclude the subjects on which Pitest did not run correctly or did not generate any mutant. This way we ensure having a fair base of comparison by counting exactly the same study subjects for both approaches (further details are given in Section 5.4). Then, we compare the fault detection capability of test suites selected to kill the same number of mutants produced by each approach.

Finally, we turn our attention to the specificity of the generated mutants by μBERT and particularly how our design intuitions are reflected in the outcome mutants. Therefore, we analyse manually some mutants that helped μBERT find bugs in the previous RQs to answer:

RQ3 *(Qualitative analysis)* How different are μBERT mutants from those generated by conventional mutation testing, in terms of diversity, readability and program context-fitting?

To answer this question, we selected some mutants that have been generated by μBERT and helped in detecting bugs that Pitest missed. Additionally, we discuss the program-context-capturing importance in μBERT's functioning, by rerunning it with a reduced size of the masked codes passed to the model, and comparing examples of yielded mutants with those obtained in our original setup.

5.4 Experimental Setup

5.4.1 Dataset & Benchmark

To evaluate μBERT's fault detection, we use real bugs from a popular dataset in the software engineering research area – Defects4J [143] v2.0.0. In this benchmark, every subject bug is provided with a buggy version of the source code, its corresponding fixed version, and equipped with a test suite that passes on the fixed version and fails with at least one test on the buggy one. The dataset includes over 800 bugs from which, we exclude the ones presenting issues, i.e. with wrong revision ids, not compiling or with execution issues, or having failing tests on the fixed version, at the reporting time. Next, we run μBERT and Pitest on the corresponding classes impacted by the bug from the fixed versions of the remaining bugs and exclude the ones where no tool generated any mutant, ending up with 689 bugs covered by μBERT and 457 covered by Pitest. As we're interested in comparing the approaches and not the tools' implementations, and to exclude eventual threats related to the environment (i.e. supported java and juint versions by each technique, etc.) or the limitations and shortages of the dataset, we establish every comparison study on a dataset counting only bugs covered by all considered approaches: 689 bugs to answer RQ1 and 457 to answer RQ2 and RQ3.

5.4.2 Experimental Procedure

To assess the complementary and added value in terms of fault revelation of the condition-seeding-based mutations (answer to RQ1), we run our approach with and without those additional mutations – that we name respectively μBERT and μBERT$_{conv}$–, and thus, generating all possible mutants on our dataset programs' fixed versions. Next, we compare the average effectiveness of the test suites generated to kill the mutants of each set; induced by μBERT and μBERT$_{conv}$.

Once the added value of the proposed condition-seeding-based mutations is validated, we compare its performance to S.O.A. mutation testing (answer to RQ2 and RQ3). We use Pitest [90], a stable and mature Java mutation testing tool, because it has been more effective at finding faults than other tools [22] and it is among the most commonly used by researchers and practitioners [9], [144], as of today. The tool proposes different configurations to adapt the produced mutations and their general cost to the target users, by excluding or including mutators. Among these configurations we used the three following:

- Pit-all (`ALL`) which counts all available mutation operators available in the current version[4].
- Pit-default (`DEFAULTS`) whose mutators are selected to form a stable and cost-efficient subset of operators by producing less but more relevant mutants.
- Pit-rv-all (`ALL`) which is a version[5] that includes the mutators of Pit-all and extra experimental [146] ones that are made available for research studies.

To compare the different approaches, we evaluate their effectiveness and cost-efficiency in achieving one of the main purposes of mutation testing, i.e., to guide the testing towards higher fault detection capabilities. For this reason, we simulate a mutation testing use-case scenario, where a developer/tester selects mutants and writes tests to kill them [95], [100].

We run every approach on the fixed versions and test suites provided by Defects4J, then collect the mutants and their test execution results; whether the mutant is killed (breaks at least one test of the test suite) and if yes by which tests. Next, we suppose that the not killed mutants are equivalent or irrelevant, explaining why no tests have been

[4]Version 1.9.4 available in Pitest's [145] GitHub repository (branch=master, repo=`https://github.com/hcoles/pitest.git`, rev-id=17e1eecf)

[5]Version 1.7.4 available in Pitest's [145] GitHub repository (branch=master, repo=`https://github.com/hcoles/pitest.git`, rev-id=2ec1178a)

written to kill them by the developers. Then, we simulate the scenario of a developer testing the fixed version, in a state where 1) it did not have any test 2) thus all mutants did not have killing tests and 3) the developer had no knowledge of which mutants are equivalent or not. This way, we can reproduce the developer flow of

1. selecting and analysing one mutant,
2. to either (a) discard it from the mutant set if it is equivalent (not killed in the actual test suite) or (b) write a test to kill it (by selecting one of the actual killing tests of the mutant),
3. then discarding all killed mutants by that test and
4. iterating similarly over the remaining mutants until all of them are analysed.

We say that a bug is found by a mutation testing technique if the resulting test suite – formed by the written (selected) tests by the developer – contains at least one test that reveals it; a test that breaks when executed on the buggy version.

We express the testing cost in terms of mutants analysed, and hence, we consider the effort required to find a bug as the number of mutants analysed until the first bug-revealing test is written. To set a common basis of comparison between the approaches, accounting for the different number of generated mutants, we run the simulations until the same maximum effort is reached (maximum number of mutants to analyse), which we set to the least cost required to kill all the mutants by one of the compared approaches. During our evaluation study, we use the same mutation selection strategy for all compared approaches, iterating through the lines in random order and selecting 1 arbitrary mutant per line per iteration. To reduce the process randomness impact on our results (in the selection of mutants and tests), we run every simulation 100 times, then average their results for every target-bug and considered approach. Finally, we aggregate these averages computed on all target bugs and normalise them as global percentages of achieved fault detection by spent effort, in terms of mutants analysed.

Finally, to answer RQ3, we select example mutants that enabled μBERT to find bugs exclusively (not found by any of Pitest versions), from the results of RQ2. Then we discuss the added value of μBERT mutations through the analysis of the mutants' behavioural difference from the fixed version and similarity with the buggy one.

5.4.3 Implementation

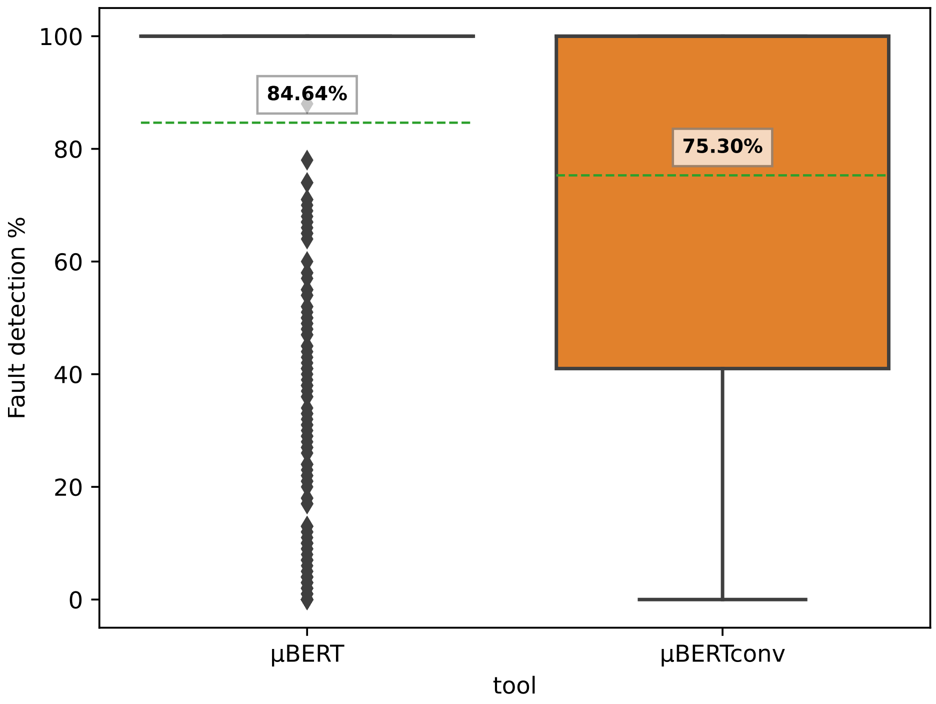
We implemented μBERT's approach as described in Section 5.2: we have used Spoon [147] and Jdt [148] libraries to parse and extract the business logic related AST nodes and apply condition-seeding mutators. To predict the masked tokens we have used our implementation proposed in our next Chapter 6 (CodeBERT-nt), using CodeBERT Masked Language Modeling (MLM) functionality [48], [84].

We provide the implementation of our approach and the reproduction package of its evaluation at `https://github.com/Ahmedfir/mBERTa`.

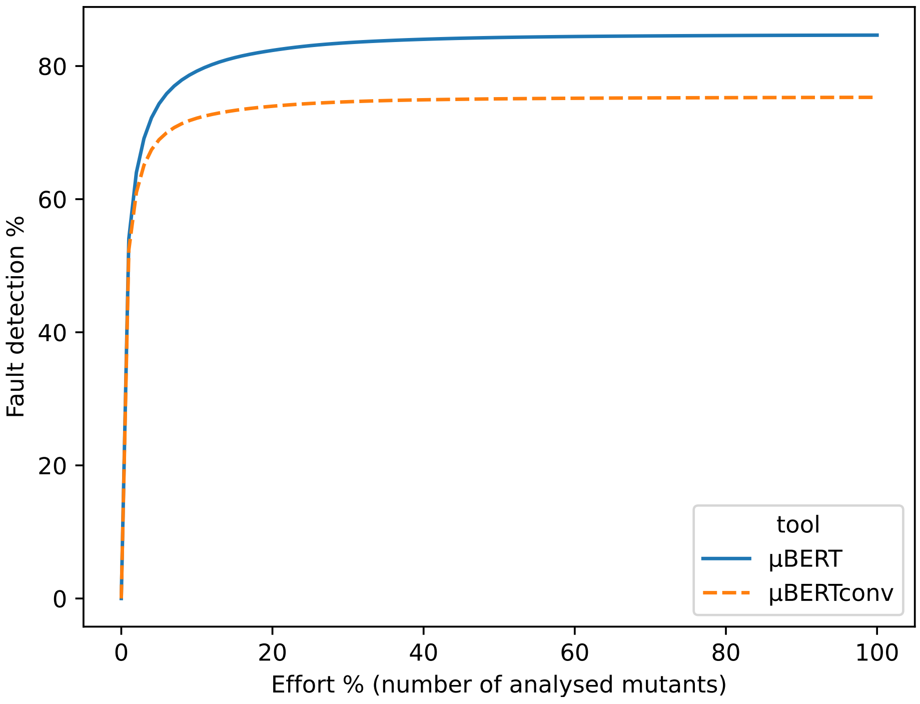
5.5 Experimental Results

5.5.1 RQ1: μBERT Additive mutations

To answer this question we compare the fault detection effectiveness of test suites written to kill mutants generated by μBERT with and without additive mutations, noted respectively μBERT and μBERT$_{conv}$. Figure 5.2 depicts the fault detection improvement when extending μBERT mutations by the additive ones. In fact, μBERT fault detection increased on average by over 9% compared to the one achieved by μBERT$_{conv}$, achieving 84.64% on average. We can also see that besides outliers, the majority of bugs are found in 100% of the times. Moreover, when examining the bugs separately, we find that μBERT finds 20 more bugs than μBERT$_{conv}$ (with fault detection $> 0\%$), and 70 more when considering bugs found with fault detection percentages above 90%. This



(a) Effectiveness: mean fault-detection per subject.



(b) Cost-efficiency: fault detection by the number of mutants analysed.

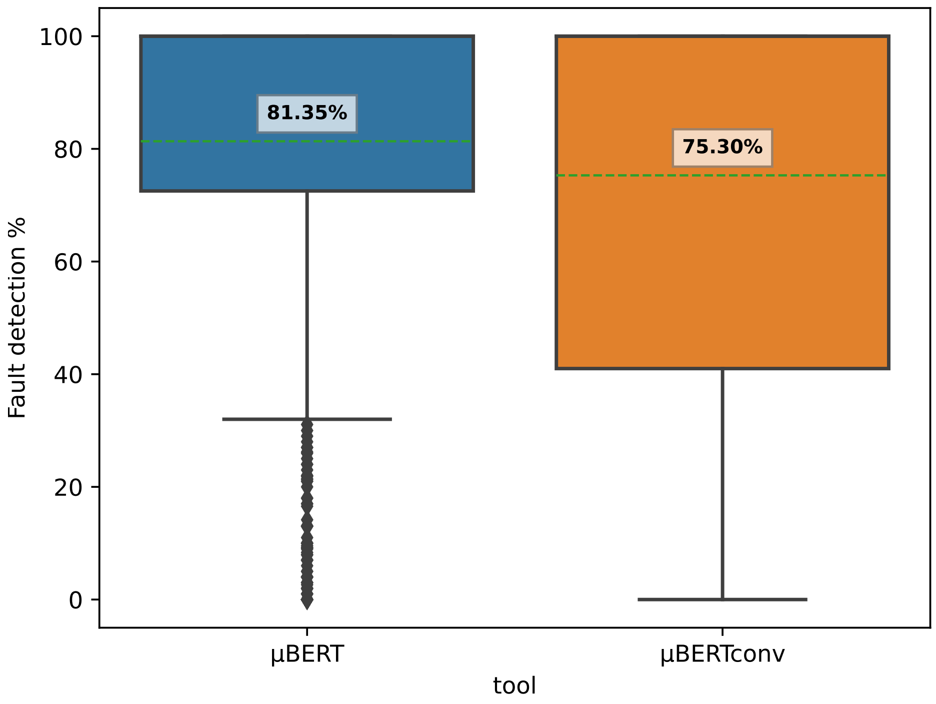
Figure 5.2: Fault-detection performance improvement when using additive patterns. Comparison between μBERT and μBERT$_{conv}$, w.r.t. the fault-detection of test suites written to kill **all** generated mutants.

confirms that the additive patterns induce relevant mutants ensuring the detection of some bugs always or in most of the cases, as well as representing better new types of faults, which were not detectable otherwise.

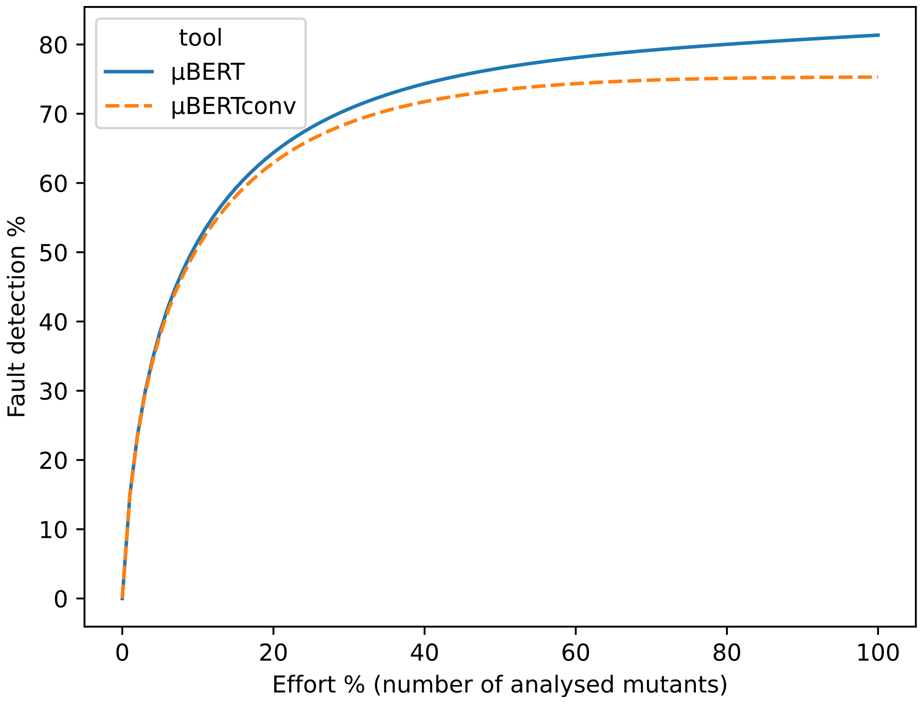
To check the significance of the fault detection advantage brought by the additive patterns, we performed a statistical test (Wilcoxon paired test) on the data of Figure 5.2a to validate the hypothesis "the fault detection yielded by μBERT is **greater** than the one by μBERT$_{conv}$". The very small obtained p-values of 5.92e-21 ($\ll$ 0.05) showed that the differences are significant, indicating the low probability of this fault detection amelioration to be happening by chance. The difference size confirms also the same advantage, with $\hat{A}_{12}$ values of 0.5827 ($>$ 0.5), indicating that μBERT induces test-suites with higher fault detection capability in the majority of the cases.

Next, we compare the fault detection performance of μBERT and μBERT$_{conv}$ when analysing the same number of mutants, and illustrate in Figure 5.3 their average fault detection effectiveness and cost-efficiency in terms of analysed mutants. The box-plots of the Subfigure 5.3a show that even when spending the same effort as μBERT$_{conv}$, μBERT keeps a similar advantage of on average 6.05% higher fault detection, achieving a maximum of 81.35%. From the line-plots of the Subfigure 5.3b, we can see that both approaches achieve a comparable fault detection ($\approx 70\%$) at ($\leq\approx 40\%$) of the maximum costs. At higher costs, μBERT$_{conv}$'s curve increases slowly until achieving a plateau at $\approx 60\%$ of the effort, whereas μBERT's curve keeps increasing towards higher fault detection ratios even when achieving the $\approx 100\%$ of the fixed maximum effort.

To validate these findings we re-conducted the same statistical tests on the data of Subfigure 5.3a and found that μBERT outperforms significantly μBERT$_{conv}$ with negligible p-values of 1.15e-19 and $\hat{A}_{12}$ values of 0.5711.



(a) Effectiveness: mean fault-detection per subject.



(b) Cost-efficiency: fault detection by the number of mutants analysed.

Figure 5.3: Fault-detection comparison between μBERT and μBERT$_{conv}$, with the **same effort**: where the maximum effort is limited to the minimum effort required to analyse all mutants of any of them, which is μBERT$_{conv}$ in most of the cases.

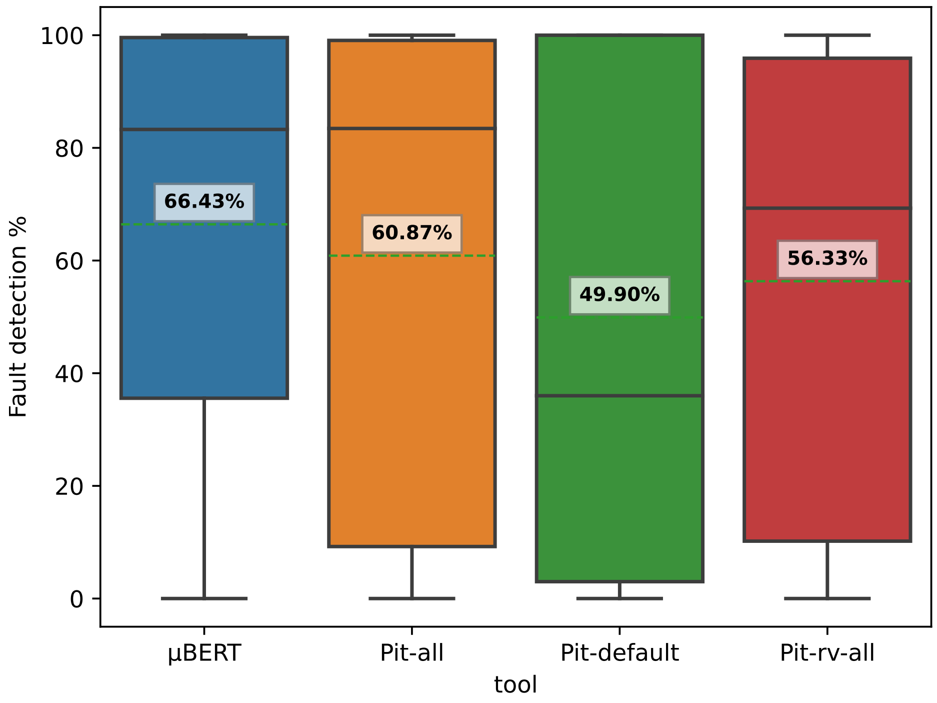
The additive patterns increased the fault detection capability of μBERT by over 6% on average, when analyzing the same number of mutants and over 9% on average when all mutants are analyzed, yielding respectively an average FD of 81.35% and 84.64%.

5.5.2 RQ2: Fault Detection comparison with Pitest

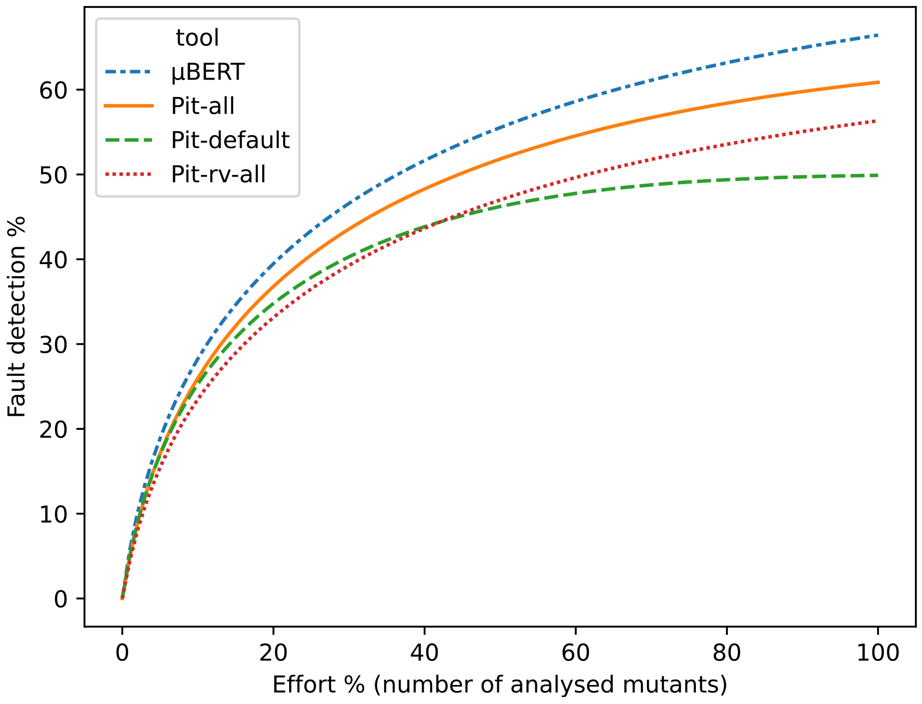
To answer this research question we reduce our dataset to the bugs covered by μBERT and the 3 considered versions of Pitest approaches: ”Pit-default” which contains the default mutation operators of Pitest, ”Pit-all” containing all Pitest operators including the default ones and ”Pit-rv-all” which contains *experimental* operators [146] in addition to the ”Pit-all” ones. Then, we perform the same study as in RQ1, where we compare the considered approaches’ effectiveness and cost-efficiency based on the fault detection capability of test suites written to kill their generated mutants. To have a fair base of comparison, we compare the approaches under the same effort in analysing mutants, which is equal to the least average effort required to kill all mutants of one of the approaches (which is the one of Pit-default in the majority of the cases). As we are interested in comparing the mutation testing approaches and not mutant selection strategies, we run the simulation with the same one-mutant-per-line random sampling of mutants for all techniques (see Subsection 5.4.2).

Figure 5.4b shows that with small effort ($\leq\approx 5\%$) all approaches yield comparable fault detection scores ($\approx 10\%$). However, the difference becomes more noticeable when spending more effort, with μBERT outperforming all versions of Pitest; achieving on average 16.53% higher fault detection scores than Pit-default, 10.10% higher than Pit-rv-all and 5.56% higher than Pit-all (see Figure 5.4a).

To validate these results, we performed the same statistical tests as in RQ1, checking the hypothesis that ”μBERT yields better fault detection capabilities than the other



(a) Effectiveness: mean fault-detection per subject.



(b) Cost-efficiency: fault detection by the number of mutants analysed.

Figure 5.4: Fault-detection comparison between μBERT and PiTest, with the **same effort**: where the maximum effort is limited to the minimum effort required to analyse all mutants of any of them, which is Pit-default in most of the cases.

Table 5.2: Paired (per subject bug) statistical tests of the average fault detection of test suites written to kill the **same number of mutants** generated by each approach (data of Figure 5.4a).

(a) Wilcoxon paired test p-values computed on every dataset subject, comparing each approach (A1) from the first column to the other approaches (A2). p-values smaller than 0.05 indicate that (A1) yields an average fault detection significantly higher than that of (A2).

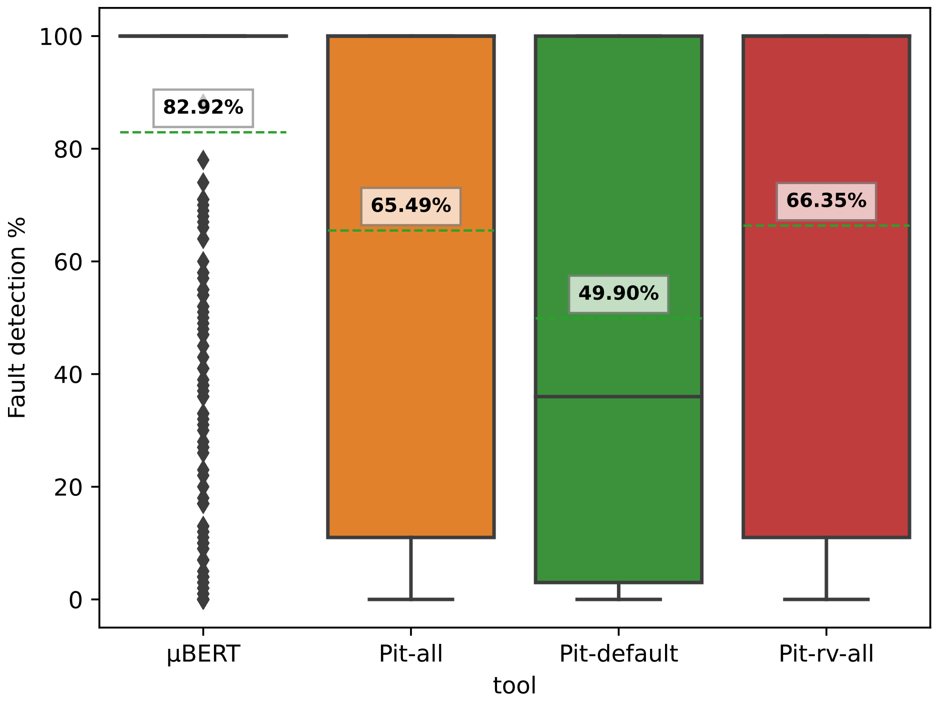
p-values	Pit-rv-all	Pit-default	Pit-all
μBERT	7.78e-11	1.18e-12	3.32e-02
Pit-all	1.54e-22	8.87e-06	–
Pit-default	9.55e-01	–	–

(b) Vargha and Delaney $\hat{A}_{12}$ values computed on every dataset subject, comparing each approach (A1) from the first column to the other approaches (A2). $\hat{A}_{12}$ values higher than 0.5 indicate that (A1) yields an average fault detection higher than that of (A2) in the majority of the cases.

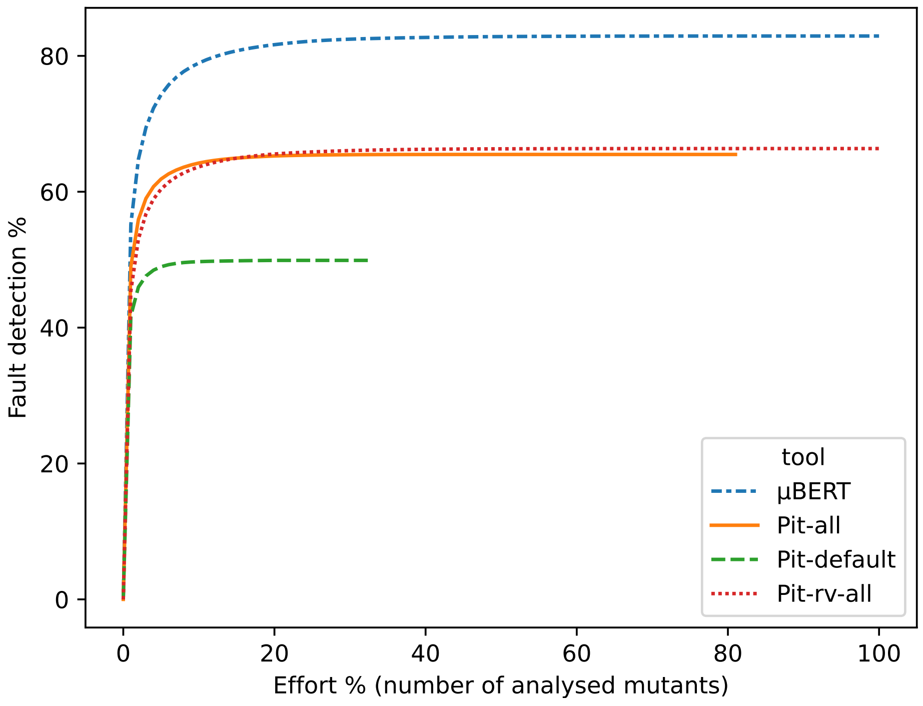
$\hat{A}_{12}$	Pit-rv-all	Pit-default	Pit-all
μBERT	0.6488	0.5514	0.5066
Pit-all	0.7210	0.4956	–
Pit-default	0.5449	–	–

approaches". We illustrate in the first row of Tables 5.2a and 5.2b the corresponding computed Wilcoxon paired test p-values and Vargha and Delaney $\hat{A}_{12}$ values. Our results show that μBERT has a significant advantage over the considered SOA approaches with p-values under 0.05. Additionally, μBERT scores $\hat{A}_{12}$ values above 0.5 which confirms that guiding the testing by μBERT mutants instead of those generated by SOA techniques yields comparable or higher fault detection ratios, in the majority of the cases. Indeed, the $\hat{A}_{12}$ difference between Pit-all and μBERT is small (0.5066), indicating that both approaches perform similarly or better on some studied subjects and worst on others.

We notice also from the sub-figure 5.4b that Pit-default achieves a plateau at around 60% of the effort while the other tools keep increasing, showing that they are able to



(a) Effectiveness: mean fault-detection per subject.



(b) Cost-efficiency: fault detection by the number of mutants analysed.

Figure 5.5: Comparison between μBERT and Pitest, relative to the fault-detection of test suites written to kill **all** generated mutants.

achieve higher fault detection capabilities, at a higher cost. This is very noticeable when we compare the sub-figures (a) and (b) of Figure 5.4 with the figure 5.2, where the average fault detection of μBERT is way lower than what it achieves in RQ1 – around 66% instead of 84%. This is a direct consequence of the fact that Pit default produces fewer mutants than the other approaches, limiting considerably the maximum effort of the mutation campaigns and thus the fault detection ratios, in the majority of the cases. Indeed, as illustrated in Figure 5.5, all approaches score higher fault detection percentages when spending more effort, achieving on average $\approx$65% for Pit-all, $\approx$66% for Pit-rv-all and $\approx$83% for μBERT. We explain the small decrease of 1.72% in the mean fault detection achieved by μBERT in comparison with RQ1 (82,92% in RQ2 instead of 84.64% in RQ1) by the difference in the considered dataset for each RQ.

Table 5.3: Paired (per subject bug) statistical tests of the average fault detection of test suites written to kill **all the mutants** generated by each approach (data of Figure 5.5a).

(a) Wilcoxon paired test p-values computed on every dataset subject, comparing each approach (A1) from the first column to the other approaches (A2). p-values smaller than 0.05 indicate that (A1) yields an average fault detection significantly higher than that of (A2).

p-values	Pit-rv-all	Pit-default	Pit-all
μBERT	2.49e-13	2.14e-33	1.47e-14
Pit-all	4.71e-01	2.76e-23	–
Pit-default	1.00e+00	–	–

(b) Vargha and Delaney $\hat{A}_{12}$ values computed on every dataset subject, comparing each approach (A1) from the first column to the other approaches (A2). $\hat{A}_{12}$ values higher than 0.5 indicate that (A1) yields an average fault detection higher than that of (A2) in the majority of the cases.

$\hat{A}_{12}$	Pit-rv-all	Pit-default	Pit-all
μBERT	0.6028	0.7123	0.6061
Pit-all	0.5077	0.6400	–
Pit-default	0.3676	–	–

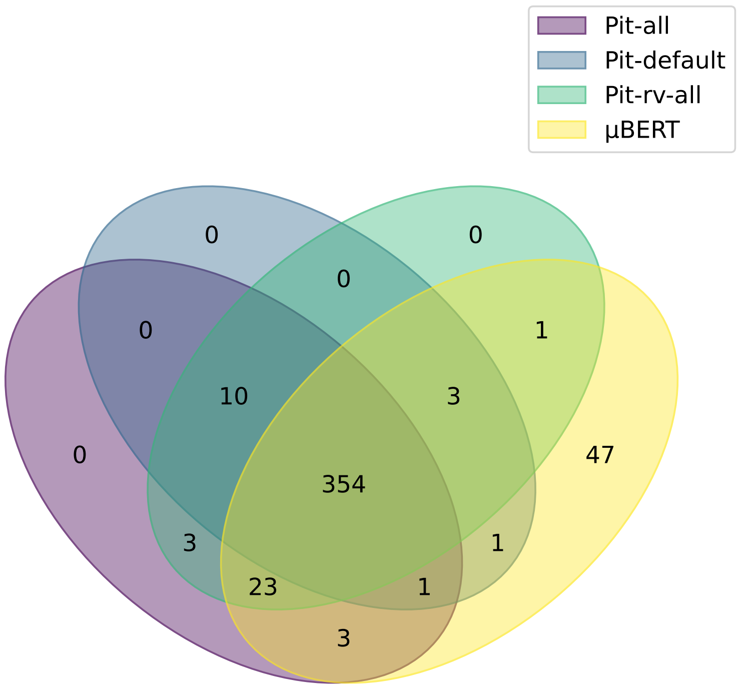
In Table 5.3, we illustrate the $\hat{A}_{12}$ and p-values computed on data of the boxplots in Sub-figure 5.5a. The results confirm that μBERT outperforms significantly SOA mutation testing w.r.t the fault detection capability of test suites written to all kill mutants generated by each approach.

μBERT outperforms significantly Pitest in inducing test suites with higher fault detection. With the same number of mutants, it scores an average FD of 16.53% , 10.10% and 5.56% higher than that of Pit-default, Pit-rv-all and Pit-all.

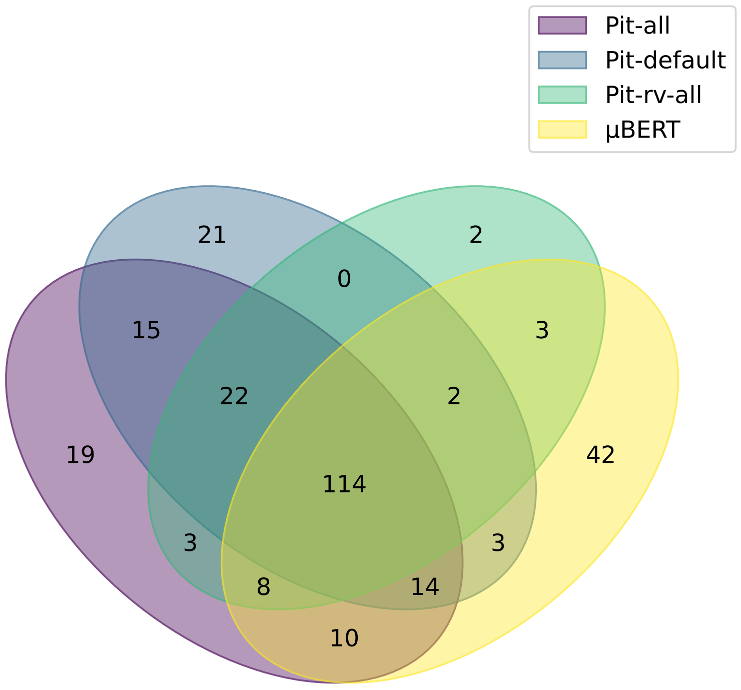
Next, we turned our interest to the set of particular bugs that every approach can and cannot reveal when spending the same effort. Hence, we map each bug with its revealing tool, from the simulation results of Figure 5.4a and illustrate their corresponding Venn diagrams in Figure 5.6.

From the disjoint sets in Sub-figure 5.6a, we notice a clear advantage in using μBERT over the considered SOA baselines, as it finds most of the bugs they find in addition to finding exclusively 47 bugs when spending the same effort. More precisely, μBERT finds 52, 77 and 52 more bugs than Pit-all, Pit-default and Pit-rv-all, respectively, whereas they find each 13, 10 and 13 bugs that μBERT missed.

This endorses the fact that μBERT introduces mutants that represent more real bugs than SOA mutation techniques, and encourages the investigation of the eventual complementary between the approaches. This observation is more noticeable when considering the overlapping between bugs found by each approach in at least 90% of the simulations (Sub-figure 5.6b). We notice that the approaches perform comparably, with a particular distinction of Pit-all and Pit-default results which find exclusively 19 and 21 bugs with these high fault detection percentages instead of 0, as observed in Sub-figure 5.6a. Nevertheless, μBERT conserves the same advantage over the considered baselines in this regard, finding exclusively 42 bugs more. It finds also 50,



(a) Faults discovered at least once per 100 runs (Fault detection $> 0\%$).



(b) Faults discovered in over 90% of the runs (Fault detection$\geq 90\%$).

Figure 5.6: Number of faults discovered by test-suites written to kill mutants generated by μBERT and Pitest versions when analysing the **same number of mutants** (same effort).

63 and 69 more bugs than respectively Pit-all, Pit-default and Pit-rv-all, whereas they find each 59, 58 and 27 bugs that μBERT missed.

μBERT can complement the fault detection capabilities of Pitest, as it is able to reveal exclusively 52, 77 and 52 more bugs than Pit-all, Pit-default and Pit-rv-all, respectively, with the same effort in terms of mutants analyzed. μBERT can also find all bugs revealed by Pitest versions, except for 13, 10 and 13 bugs found respectively by each of Pit-all, Pit-default and Pit-rv-all, which μBERT missed.

5.5.3 RQ3: Qualitative Analysis

To answer this research question we investigate the mutants generated by μBERT, which induced test suites able to find bugs that were not detected otherwise, i.e. by the considered SOA approaches (see Figure 5.6). Meaning that the mutants break similar tests as the target real buggy version.

As a simple bug example (requiring only one change to fix it), we consider Lang-49 from Defects4J and we investigate mutants that have been generated by μBERT and helped in generating tests that reveal it. This bug impacts the results of the method `reduce()` from the class `org.apache.commons.lang.math.Fraction`, which returns a new reduced fraction instance, if possible, or the same instance, otherwise. The bug is caused by a miss-implementation of a specific corner case, which consists of calling the method on a fraction instance that has `0` as numerator. In Table 5.4, we illustrate example mutants generated by μBERT that helped in revealing this bug. Every mutant is represented by a diff between the fixed and the mutated version by μBERT.

As can be seen, μBERT can generate mutants that can be induced by applying conventional pattern-based mutations, i.e., Mutant 1 replaces a relational operator (==) with another (>) and Mutant 2 replaces an integer operand (`0`) with another one (`1`).

Table 5.4: Example of μBERT mutants that helped find the bug Lang-49 from Defects4J.

Mutant 1: **replacing binary operator**

```
@@ org.apache.commons.lang.math.Fraction : 466 @@
- if (numerator == 0) {
+ if (numerator > 0) {
```

Mutant 2: **replacing literal implementation**

```
@@ org.apache.commons.lang.math.Fraction : 466 @@
- if (numerator == 0) {
+ if (numerator == 1) {
```

Mutant 3: **adding a condition to an if statement**

```
@@ org.apache.commons.lang.math.Fraction : 466 @@
- if (numerator == 0) {
+ if ((numerator == 0)
+    || !(numerator==Integer.MIN_VALUE)) {
```

Mutant 4: **replacing a condition**

```
@@ org.apache.commons.lang.math.Fraction : 467 @@
- return equals(ZERO) ? this: ZERO;
+ return this;
```

Mutant 5: **replacing `this` access by another object**

```
@@ org.apache.commons.lang.math.Fraction : 467 @@
- return equals(ZERO) ? this: ZERO;
+ return equals(ZERO) ? ONE: ZERO;
```

Mutant 6: **replacing method argument**

```
@@ org.apache.commons.lang.math.Fraction : 469 @@
int gcd = greatestCommonDivisor(
- Math.abs(numerator), denominator);
+ Math.abs(numerator), 1);
```

Mutant 7: **replacing a variable**

```
@@ org.apache.commons.lang.math.Fraction : 473 @@
- return Fraction.getFraction(numerator / gcd,
+ return Fraction.getFraction(numerator / 3,
    denominator / gcd);
```

Mutant 8: **adding a condition to a return statement**

```
@@ org.apache.commons.lang.math.Fraction : 840 @@
return (getNumerator() == other.getNumerator()
-  && getDenominator() == other.getDenominator());
+  && getDenominator() == other.getDenominator()))
+   || (numerator == other.numerator);
```

In addition, it proposes more complex mutations that are unlikely achievable without any knowledge of either the AST or the context of the considered program. For instance, it can generate Mutant 4 by changing a conditional return statement with (`this`) the current instance of `Fraction`, which matches the return type of the method. Similarly, to generate Mutant 5, it replaces (`this`) the current instance of the class `Fraction` by an existent instance of the same type (`ONE`), making the statement returning either the object `ONE` or the object `ZERO`.

To produce more complex mutants, μBERT applies a condition seeding followed by token-masking and CodeBERT prediction, such as adding `|| (numerator == other.numerator)` to the original condition of a `return` statement, inducing Mutant 8, or adding `|| !(numerator == Integer.MIN_VALUE)` to the original condition of an `if` statement, inducing Mutant 3.

To investigate further the impact of the code context captured by the model on the generated mutants, we have rerun μBERT on 5 subjects from our dataset, with a maximum number of surrounding tokens equal to 10 (instead of 512). Then, we compared manually the induced mutants with those generated by our default setup, in the same locations. From our results, we observed a noticeable decrease in the number of compilable predictions, indicating the general performance decrease of the model when it lacks information about the code context. Particularly, we notice that it is not able to produce program-specific mutants, i.e. by changing an object by another or a method call with another. In Table 5.5, we illustrate some example mutants that helped find each of the studied subjects (breaking same tests as the original bug), which μBERT failed to generate when the maximum number of surrounding tokens is limited to 10.

Table 5.5: Example of mutants generated by μBERT that helped in finding bugs from Defects4J and could not be generated when limiting the maximum number of surrounding tokens to 10.

Mutant 1 (JacksonCore-4) : **replacing a method call**

`@@ com.fasterxml.jackson.core.util.TextBuffer : 515 @@`
`- unshare(1);`
`+ expand(1);`

Mutant 2 (Closure-26) : **replacing an object**

`@@ com.google.javascript.jscomp.ProcessCommonJSModules : 89 @@`
`- .replaceAll(Pattern.quote(File.separator), MODULE_NAME_SEPARATOR)`
`+ .replaceAll(Pattern.quote(filename), MODULE_NAME_SEPARATOR)`

Mutant 3 (Closure-35) : **replacing a method call**

`@@ com.google.javascript.jscomp.TypeInference : 1092 @@`
`- scope = traverseChildren(n, scope);`
`+ scope = traverse(n, scope);`

Mutant 4 (Lang-27) : **replacing a method call**

`@@ org.apache.commons.lang3.math.NumberUtils : 526 @@`
`- if (!(f.isInfinite() || (f.floatValue() == 0.0F && !allZeros))) {`
`+ if (!(f.isInfinite() || (f.round() == 0.0F && !allZeros))) {`
`// also "f.floatValue()" to "f.scale()"`

Mutant 5 (Math-64) : **replacing an object**

`@@ org.apache.commons.lang.math.Fraction : 852 @@`
`- for (int j = k; j < jacobian.length; ++j) {`
`+ for (int j = k; j < beta.length; ++j) {`

Mutant 6 (Lang-27) : **replacing an object**

`@@ org.apache.commons.lang3.math.NumberUtils : 526 @@`
`- if (!(f.isInfinite() || (f.floatValue() == 0.0F && !allZeros))) {`
`+ if (!(f.isInfinite() || (f.round() == 0.0F && !zero))) {`

μBERT can introduce similar mutants as those generated by conventional pattern-based mutation testing techniques, i.e. changing a + operator by a -, in addition to more complex ones, i.e. via condition seeding or by changing variables and method calls. The performance of the approach depends on the model's gained ability to capture the context of the input code, which is decisive when it comes to proposing meaningful transformations, such as proposing relevant variables or methods to use instead of the original ones.

5.6 Threats to Validity

One external threat to validity concerns the generalisation of our findings and results in the empirical evaluation. To reduce this threat, we used a large number of real bugs from popular open-source projects with their associated developer test-suites, provided by an established and independently built benchmark (i.e. Defects4J [143]). Nevertheless, we acknowledge that the results may be different considering projects in different domains.

Other threats may arise from our way of assessing the fault detection capability of mutation testing approaches, based on their capability of guiding the testing via a developer/tester simulation in which we assume that the current test suites are complete and the not killed mutants are equivalent. Although we acknowledge that this may not be the case in real-world scenarios, we believe that this process is sufficient to evaluate our approach, particularly considering the fact the test suites provided by Defects4J are relatively strong. Additionally, to mitigate any comparison threat between the considered approaches, we use consistently and similarly the same test-suites, setups and simulation assumptions in all our study.

The choice of our comparison baseline may form other threats to the validity of our findings. While different fault-seeding approaches have been proposed recently, Pitest remains among the most mature and stable mutation testing tools for Java programs, thus, forming an appropriate comparison baseline to evaluate our work. Furthermore, we compared our results with those obtained by mutants from different configurations proposed by Pitest, enlarging our study to the different audiences targeted by this latter. We acknowledge however that the results may change when considering other techniques and consider the evaluation of the effectiveness and cost-efficiency of different mutation testing techniques as out of the scope of this paper.

Other construct threats may arise from considering the number of mutants analysed as metric to measure the effort required to find a fault. In addition to the fact that this metric has been widely used by the literature [9], [100], [110], we believe that it is intuitive and representative of the global manual effort of the tester in analysing the mutants, discarding them or writing tests to kill them. While being the standard in the literature, we acknowledge that this measure does not account for the cost difference between mutants, attributing the same cost to all mutants. This is simply because we do not know the specific effort required to analyse every specific mutant or to write every specific test. Additionally, our cost-efficiency results may be impacted by costs that are not captured with this metric, such as the execution or the developing effort of either CodeBERT, the key component of μBERT, or the set of patterns and execution enhancements over the different releases of Pitest. Nevertheless, we tried to mitigate any major threats that can be induced by the implementation of the different tools, i.e. we reduce the dataset subjects to those on which every approach generated at least one mutant and consider any implementation difference between the approaches as out of the current scope.

5.7 Conclusion

We presented μBERT; a pre-trained language model-based fault injection approach. μBERT provides researchers and practitioners with easy-to-understand "natural" mutants to help them in writing tests of higher fault revelation capabilities.

Unlike state-of-the-art approaches, it does neither require nor depend on any kind of faults knowledge or language grammar but instead on the actual code definition and distribution, as written by developers in numerous projects. This facilitates its developing, maintainability, integration and extension to different programming languages. In fact, it reduces the overhead of learning how to mutate, be it via creating and selecting patterns or collecting good bug-fixes and learning from their patches.

In a nutshell, μBERT takes as input a given program and replaces different pieces of its code base with predictions made by a pretrained generative language model, producing multiple likely-to-occur mutations. The approach targets diverse business code locations and injects either simple one-token replacement mutants or more complex ones by extending the control-flow conditions. This provides probable developer-like faults impacting different functionalities of the program with higher relevance and lower cost to developers. This is further endorsed by our results where μBERT induces high fault detection test suites at low effort, outperforming state-of-the-art techniques (Pitest), in this regard.

We have made our implementation and results available [149] to enable reproducibility and support future research.

Chapter 6

Generic Code Naturalness Measure via Pre-trained Language Models

Much of recent software-engineering research has investigated the naturalness of code, the fact that code, in small code snippets, is repetitive and can be predicted using statistical language models like n-gram. Although powerful, training such models on large code corpus can be tedious, time-consuming and sensitive to code patterns (and practices) encountered during training. Consequently, these models are often trained on a small corpus and thus only estimate the language naturalness relative to a specific style of programming or type of project. To overcome these issues, we investigate the use of pre-trained generative language models to infer code naturalness. Pre-trained models are often built on big data, are easy to use in an out-of-the-box way and include powerful learning associations mechanisms. Our key idea is to quantify code naturalness through its predictability, by using state-of-the-art generative pre-trained language models. Thus, we suggest to infer naturalness by masking (omitting) code tokens, one at a time, of code-sequences, and checking the models' ability to predict them. We explore three different predictability metrics; a) measuring the number of exact matches of the predictions, b) computing the embedding similarity between the original and predicted code, i.e., similarity at the vector space, and c) computing the confidence of the model when doing the token completion task regardless of the outcome. We implement this workflow, named CodeBERT-nt, and evaluate its capability to prioritize buggy lines over non-buggy ones when ranking code based on its naturalness. Our results, on 2,510 buggy versions of 40 projects from the SmartShark dataset, show that CodeBERT-nt outperforms both, random-uniform and complexity-based ranking techniques, and yields comparable results to the n-gram models.

This chapter is based on the following article:

Ahmed Khanfir, Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. CODEBERT-NT: code naturalness via CodeBERT. In : *22nd IEEE International Conference on Software Quality, Reliability and Security (QRS'22)*. 2022.

6.1 Introduction

There is a large body of research demonstrating that code alike natural language, in small snippets, is repetitive and thus predictable [37]. A typical way to capture and leverage this repetitiveness is through the use of statistical language models, such as n-grams [115]. Indeed, those models can determine the appearance likelihood of a sequence amidst the ones preceding (or succeeding) it, given a reference corpus, usually composed of the project's other code components. Through this likelihood, practitioners can quantify how surprising a code sequence is with respect to other sequences, i.e. within a project. This quantification, commonly referred to as naturalness, has been proven useful in identifying unusual code sequences, that may reflect code that is smelly [150], [151], of low readability [152], [153], or simply a rare specific implementation instance [154].

Although powerful, training such models on a large code corpus can be tedious, time-consuming and is sensitive to code patterns (and specificities) of the used projects.

Additionally, the resulting models are sensitive to numerous meta-parameters such as tokenizers, smoothing techniques, unknown thresholds and n values [39]. Consequently, this often leads to an accumulation of models trained on smaller and more contextual corpora to which their naturalness will relate [37], with questionable, even poor, generalization ability [36].

In this paper, we intend to address this shortcoming of naturalness-based metrics, i.e. the accumulation of models due to their poor generalization, by leveraging generative pre-trained language models, such as CodeBert. Pre-trained models have been shown to provide strong results on several cross-project code-related tasks such as code generation and translation. They are built upon large corpora of code, are easy to use in an out-of-the-box way and include powerful learning associations mechanisms that allow them to generalize well to unseen code and projects, making them interesting candidates. However, extracting naturalness metrics out of such models is not a straightforward process due to their generative nature, i.e. they are not designed to output naturalness-like metrics.

To bypass this, we hypothesize that code naturalness can be seen as a derived form of predictability and can thus be inferred by measuring how well code tokens generated by the models correspond to actual ones. This means that by masking tokens in a sequence and evaluating the ability of a model to find them back, we could evaluate code naturalness.

To this end, we implement a code-prediction-based approach, which we call CODEBERT-NT [1] on top of the CodeBERT pre-trained generative model, and derive the following metrics from it:

- counting the number of exact matches of the predictions.
- computing the embedding similarity between the original and predicted code, i.e., similarity at the vector space.

[1] https://github.com/Ahmedfir/CodeBERT-nt

- computing the confidence of the model when doing the token completion task, regardless of the outcome.

To evaluate those metrics' suitability for code naturalness estimation, we need to compare them with traditional naturalness metrics based on n-grams models. Yet, as the naturalness of code is a relative measure, i.e. strongly related to the model's data and parameters, we cannot directly contrast the value of one metric over another. Thus, we compare the metrics relative performance on an end task, for which we have solid ground truth. In particular, we evaluate the performance of the metrics w.r.t. their ability to rank buggy lines of code.

Indeed, previous work has shown that unusual code is often linked with bug proneness and bugginess, thereby making n-gram-based naturalness a tool capable of identifying likely buggy code [38], [154]. This means that the tendency of buggy lines to be unnatural, or at least more unnatural compared to the clean ones should be found in our metrics.

We, therefore, investigate the ability of CODEBERT-NT to distinguish natural (clean) from unnatural (buggy) code. To do so, we need a solid ground truth of buggy and clean code. Hence we select the SmartShark dataset [155], which contains manually untangled buggy and fixed code versions. This allows us to know the buggy code lines for which we can perform our experiment. We thus, used 2,510 buggy versions from 40 projects and investigated the performance of CODEBERT-NT and contrast it with that of typical baselines such as uniform-random and complexity-based rankings [95]. Overall, we find that buggy lines are indeed less natural in our metrics, especially when basing it on the decrease in prediction confidence.

To further validate this, we also compared our results with that of n-gram models when trained on an intra-project fashion (as typically performed in the literature [38], [39]). Our results suggest that our (inter-project) CODEBERT-NT yields comparable results (slightly better) than the (intra-project) naturalness predictions of the n-gram models.

Overall, our primary contributions are:

- We demonstrate that pre-trained generative models like CodeBERT capture the language naturalness notion. We can infer the naturalness aspect of a source-code from the CodeBERT prediction results.
- We introduce a novel approach to compute source-code naturalness that works in cross-project context and does not rely on the aspects of naturalness that are tied to a specific project. For instance, It does not require any further training or specific knowledge of the target project, to rank its lines by bugginess likelihood, thus can be easily used in the future as a baseline.
- We provide a tool for computing code naturalness that is applicable in a cross-project fashion, allowing both researchers and practitioners to integrate naturalness computations (as features) in their ML-based approaches and perform for instance bug detection [38], code smell detection [150], [151], code readability analysis [152], [153], support mutation testing [156], automated bug fixing [157] and many other applications, as surveyed in the work of Allamanis et al. [36].

6.2 Naturalness through CodeBERT

6.2.1 Naturalness Metrics

In this study, we explore the possibility to rely on pretrained generative models and more precisely CodeBERT to evaluate the naturalness of code.

While these models are neither designed to output metrics nor related to naturalness at first sight, they can be easily used on existing code by simply masking tokens and requesting suggestions. This means that by evaluating the ability of models to find back masked tokens, it is possible to estimate how predictable the code is according to the models.

From this, we hypothesize that code naturalness can be seen as a derived form of predictability and thus this predictability could be considered as a form of naturalness computation. Yet, to reach naturalness metrics, we still need to quantify this predictability through the suggestions of the model.

One possibility, perhaps the simplest one, is to select a sequence, successively mask the tokens and report the number of tokens of accurate guess of the model. While this type of metric can work with any available model, it reflects neither the confidence of the model in its suggestions nor the ranking of suggestions. Thus, we suggest two additional metrics based on additional information provided by CodeBERT, i.e., the prediction confidence score. Indeed, this score provides along with the suggestions the confidence in the said suggestion, which brings us closer to the behaviors of n-gram models and their probabilities.

From there, we investigate the following metrics:

- *CBnt_conf:* the prediction confidence score of CodeBERT. This metric represents a probability, thus is a floating number varying between 0 and 1, where the closer to 1 its value is, the more CodeBERT is confident about the prediction.

  We believe that this metric may mirror directly the naturalness of code as it reflects how predictable and usual is the code, relatively to the code knowledge learned by the model through its training phase on a large-scale code dataset. Thus, low confidence scores may imply low naturalness.

- *CBnt_cos:* the cosine similarity between the CodeBERT embeddings of the predicted and the original source-code. This metric has also a float value varying between 0 and 1, where 1 implies an exact similarity between the two embeddings and 0 the absence of similarity. This metric is often used in NLP and has shown some interesting results in filtering unnatural sentences [158]. CodeBERT embedding is the encoded representation of the code in the latent space, where every token is represented by 1 vector. To calculate the cosine similarity between the embeddings, we start by concatenating their token-vectors into two vectors, one for each embedding, then we compute the cosine as follows:

$$\text{Cosine}(\mathbf{V}_o, \mathbf{V}_p) = \frac{\mathbf{V}_o.\mathbf{V}_p}{\|\mathbf{V}_o\|.\|\mathbf{V}_p\|}, \tag{6.1}$$

where $\mathbf{V}_o$ and $\mathbf{V}_p$ are the concatenated embedding vectors of respectively the original and predicted code.

Our intuition is that the less natural the code is, the more CodeBERT will have difficulties noticing small changes in it, i.e., changing a single token in unnatural code would not impact much its resulting embeddings. Consequently, a high similarity between both embeddings – of the original and predicted code – may be a symptom of unnaturalness in the code.

- *CBnt_acc:* the accuracy of prediction (whether the predicted code matches the original one or not). This is a Boolean metric where 1 is attributed to a matching prediction and 0 otherwise. Intuitively, we believe that the more the code is natural, the more CodeBERT predictions are accurate.

6.2.2 CodeBERT-nt

We implemented a tool named CodeBERT-nt able to compute those metrics, which process is described in Figure 6.1. More precisely, CodeBERT-nt proceeds as follows:

For every selected file, the tool starts by parsing the Abstract Syntax Tree (AST) and extracting the interesting nodes to mask, excluding the language-specific tokens such

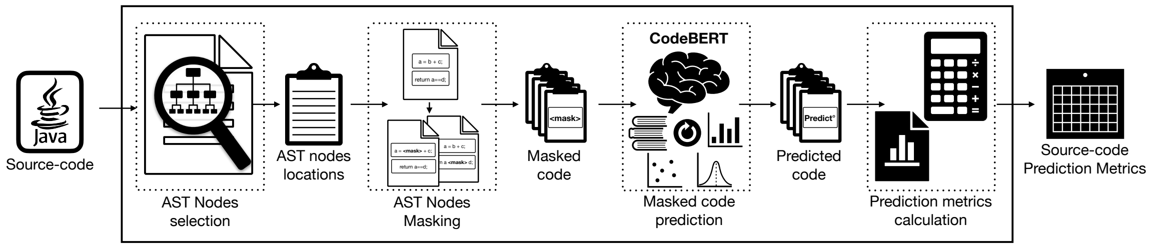


Figure 6.1: The CODEBERT-NT source-code metrics calculation workflow.

as control flow and import-related tokens, e.g., `if`, `else`, `for`, `while`, `import`, etc. Then, for every line, the tool iterates over the selected set of nodes and replaces each node's content by the placeholder `<mask>`, thus, generating one masked version of the input code per selected node.

Next, every masked version is tokenized into a vector of tokens using the CodeBERT tokenizer and crops it to only encompass the masked token and its surroundings, as the model encoder can only take up to 512 tokens. Once the shrinking is done, the sequences of masked code are fed to CodeBERT to determine the best fitting substitute for the mask placeholder. By default, the model provides 5 propositions ranked by likelihood (also called confidence) to match the masked node's original value. In our setting, we only consider the first proposition as we believe it to be the most naturalness-revealing one (we discuss this choice further in Section 6.6.3) and compute the 3 suggested metrics.

Finally, each line is mapped with its prediction scores forming a matrix $\mathbf{M} \in \mathbb{R}^{3\times n}$ where n is the number of collected propositions in that source-code line.

The AST parsing and node-location extraction part has been implemented in Java and uses Spoon [147], a Java code-source analysis library. Whereas, the rest of the process has been implemented in python using PyTorch [159] and the CodeBERT Masked Language Modeling (MLM) task to predict the masked codes and compute the metrics.

6.3 Research Questions

In this paper, we explore the possibility to rely on pre-trained generative models to evaluate the naturalness of code.

Naturalness of code is by essence a relative measure, i.e., it strongly depends on the models and training data used, which means that we cannot directly establish that metrics actually reflect naturalness. Therefore, we contrast them based on their performance on an end task, for which we have a solid ground truth. In particular, we evaluate the performance of our metrics w.r.t. the naturalness of bugs hypothesis that states: ”*unnatural code is more likely to be buggy than natural code*” [38]. This means that a good naturalness metric should be capable of distinguishing natural (clean) from unnatural (buggy) code.

Hence, we ask:

RQ1 *(*CodeBERT-nt *metrics):* Which metric leads to the best segregation between buggy and not buggy lines?

To answer this question, we check if any metric discriminates better the buggy from the clean lines. In particular, we investigate each metric performance with several aggregation methods as well as ordering directions. This results in a best discriminating combination of aggregation and ordering per metric which can then be compared against.

However, observing better results doesn't necessarily indicate that the metric is of actual use. We thus, turn our attention to the significance of these results and contrast them with some obvious baselines, such as the random order and source-code complexity order (ranking the most complex lines first). Random order offers a sanity check for coincidental results, and complexity offers an unsupervised baseline [160], i.e., shows that our metrics do not simply measure complexity instead of naturalness. Therefore, we ask:

RQ2 *(Comparison with baseline metrics):* How does CodeBERT-nt metrics compare with random and source-code complexity, in terms of buggy lines ranking?

To check whether the naturalness captured by CodeBERT-nt is useful, we compare the values observed in RQ1 to the ones from the 2 aforementioned baselines on the same subject lines and buggy versions. A clear advantage of the CodeBERT-nt metrics would validate that the metrics capture more than complexity. Yet, as those baselines are relatively weak, we also want to compare our metrics with n-gram models that were originally used to show the naturalness of buggy code hypothesis. This leads us to the following question:

RQ3 *(Comparison with existing naturalness metrics):* How does CodeBERT-nt compare with n-gram based naturalness metrics in terms of lines ranking by bugginess?

We answer this question by ranking the subject lines based on their naturalness, i.e., cross-entropies, measured by n-gram models trained on the source-code of the same version of the considered project and comparing it to the results of our metrics, in a similar way as of RQ2. More precisely, for every considered bug we train two n-gram models (using each of UTF8 and Java Parser tokenizers) on the source code of the files that have not been changed by the fix patch. Then, we use these models to calculate the cross-entropies of the subject lines.

This obviously caused an accumulation of models – precisely 2 per target bug in this case – which highlights a major drawback of this approach. Nonetheless, the RQ3 comparison does not aim at answering the question of which model is best but to show a relative performance of the models. Indeed, as the training corpora of the models are significantly different, the associated learning differs with each one having significant associated advantages. In particular, CodeBERT would benefit from being trained on a much larger corpus, while n-grams would have the advantage of operating on an intra-project fashion, which provides more information on the context at hand [36].

6.4 Experimental Setup

6.4.1 Dataset & Benchmark

To perform our investigation, we use a recently crafted dataset of commits; SmartSHARK [155]. This dataset is distributed as a MongoDB database describing commit-details extracted from multiple open-source python and java projects. A specificity of Smartshark is that its data are manually refined, i.e., the commits are untangled and the fix validated, hence reducing the threat of noisy data.

In our study, we use the issue-fixing commits of the 40 available java projects the dataset includes.

Buggy versions selection

For most of the issues, SmartSHARK provides one or more related commits, among which the fixing-commits are labeled as validated fix-commits. Thus, to build our bugs dataset, we exclude all issues having no corresponding fix-commit, or having multiple fix-commits. While the first case is straight forward, the latter is applied to further reduce noise in the data as pinpointing bugs origin from multiple fixing commits is harder and more error prone. This way, we obtain one fix-commit per issue where the changed lines form the complete set of buggy lines. Then, as we focus on Java source-code naturalness, we exclude the issues whose fixes are not involving java files, e.g., configuration files. Finally, we map the resulting issues to their related buggy version, i.e., the project version preceding the fix commit.

Lines subject to study

For every considered issue, we retrieve the files changed by the patching commit, among which we mark as buggy any line changed by the fixed commit and define as neutral ones the remaining lines in these files.

Then to ensure that the observed ranking performances are related to code naturalness extraction and not to any non-uniform distribution of the buggy-lines over the code, we exclude all lines that are not part of the business-logic of the program like the imports, the fields declaration, the brackets, etc. We do so to focus on logic types of faults, which are unlikely to be caused by non-business-logic statements as also suggested by the work of Rahman et al. [161]. Moreover, the naturalness of these lines is unlikely to be insightful. (We discuss this further in Section 6.6.1)

Additionally, this allows us to exclude any interference between comparing the ranking-performance by naturalness and other forms of rankings related to the buggy lines unequal distribution between business-logic and not business-logic related source-code portions. In fact, close to half of the considered bugs see all of their buggy lines located within the business-logic code, while only 10% of the bugs see all of their buggy-lines outside of it. Unaccounting for those 10%, we observe a median of 90% of buggy lines located within the business-logic code per bug, while this type of line only amounts to 60% of the lines overall.

To sum-up, by excluding these lines we exclude 257 bugs out of 2510 to end up with a final dataset counting 2253 buggy versions, represented by a set of business-logic buggy and not buggy lines.

6.4.2 n-gram ranking

To compare CodeBERT-nt with n-gram models in terms of code naturalness aspect capturing, we proceed as follows:

For every considered issue, we train two n-gram models specific to that version of the project using two distinct tokenization techniques[2]:

- Java Parser tokenizer (noted JP) which tokenizes the code according to the Java grammar, and thus, discarding empty lines as well as java-doc and code-comment lines.
- UTF8 tokenizer (noted UTF8), which operates on the full raw representation of the source-code.

We name the created models based on their underlying tokenization technique, JP and UTF8 n-gram. In the training phase, we use all the lines from the files that have not been changed by the fix commit, then we use each of these models to attribute a cross-entropy score to the subject lines of the buggy files, counting buggy and neutral lines as detailed in Sub-section 6.4.1. Finally, we rank the lines according to their cross-entropy score in a descendant order such-as high values of cross-entropy are associated with less code naturalness and higher likelihood of bugginess.

To run this experiment, we use the current version of the n-gram utilities library Tuna [162] with one of the recommended configurations by Jimenez et al. [39] for distinguishing buggy and fixed lines: 4 as n-order, 1 as unknown threshold and Kneser Ney smoothing (KN). Note that we use KN instead of the Modified Kneser Ney smoothing (MKN) because it is not suited for short sequences.

6.4.3 Lines ranking

To assess the relevance of the information inferred from the CODEBERT-NT predictions, we rely on its ability to rank the buggy lines before the supposed neutral ones. As we

[2]We use `UTFLineTokenizer` and `JavaLemmeLineTokenizer` available in Tuna [162] GitHub repository under `tokenizer/line/` (branch=master,repo=`https://github.com/electricalwind/tuna`, rev-id=44188e1)

do not have any labelling of ”natural” and ”unnatural” source-code dataset and based on the buggy-code naturalness hypothesis, we believe that the prediction variation of CodeBERT-nt under naturalness variation of the input source-code can be observed through its ranking of buggy and not buggy lines. (Please see Section 6.3 for more details)

For every considered approach, metric and aggregation method considered in our experiments, we rank all the lines by bug first, then normalise the ranks by the total number of studied lines for that bug. We report two ranking results per bug per approach:

- the first hit: corresponds to the rank of the first-ranked buggy line and
- mean: corresponds to the mean of the ranks of all buggy lines.

To cut ties when multiple lines share the same score, we attribute the estimated rank by a uniform random selection. For instance, if we have 100 lines sharing the same rank, among which 3 lines are buggy, the random first hit rank will be equal to 25, while the mean rank of the 3 buggy lines will be 50.

To check whether any of the 3 aforementioned metrics are impacted by the naturalness variance of the source-code (answer to RQ1), we generate one CodeBERT prediction only by masked token. Then we aggregate the scores of each line's predictions by applying one of the following aggregation metrics: minimum, maximum, mean, median and entropy. Where the entropy is calculated as the following:

$$\mathsf{H}(l,m) = -\frac{\sum_{i=1}^{n}\log(s_i)}{n} \tag{6.2}$$

where $\{s_1, s_2, ..., s_n\}$ denotes the set of n scores attributed to the line l for the metric m.

We then rank the subject lines by each of the metrics using the different aggregation methods and following both sorting directions: ascending and descending. As most of the results are close to each other, we calculate the paired Vargha and Delaney $\hat{A}_{12}$ ratios, to conclude which combinations are the best in terms of buggy lines ranking.

To check whether the extracted information from CodeBERT predictions (answer to RQ2) reflect actually code naturalness, we compare the ranking results of the 3 metrics using their best performing aggregation method and sorting order, with the rankings results of random and complexity-based rankings. Where, the complexity of a line corresponds to its number of Java Parser tokens. This is inspired from the study of Leszak et al. [163], where complex source code has been proven to be more likely to be buggy. For the random ranking, instead of rerunning the ranking multiple times, we simply used a basic probability calculation of the rankings.

Finally, to compare the effectiveness of CodeBERT-nt with similar techniques in capturing code naturalness we compare its buggy-line rankings with the n-gram cross-entropy, measured as described in Sub-section 6.4.2 (answer to RQ3). For this comparison, we consider the CodeBERT-nt rankings effectuated by its best performing metric, selected from the previous research questions.

To have a better understanding on the differences significance we run statistical tests of Wilkixon and Vargha and Delaney $\hat{A}_{12}$ on all of our comparison results.

6.4.4 Threats to Validity

The question of whether our findings generalise, forms a typical threat to validity of empirical studies. To reduce this threat, we used real-world projects, real faults and their associated commits, from an established and independently built benchmark. Still, we have to acknowledge that these may not be representative of projects from other programming languages, domains or industrial systems.

Other threats may also arise from the assumption that all changed lines by the fix commits are buggy lines. While our heuristics are the standard in the literature, we believe that this selection process is sufficient given that we have used a dataset where

Table 6.1: Paired Vargha and Delaney $\hat{A}_{12}$ effect size values of the buggy lines ranking by different pairs of metrics-aggregation methods in ascendant and descendant order. cos refers to CBnt_cos, conf refers to CBnt_conf and the acc metric refers to CBnt_acc.

Metric agg	**conf min**	**conf max**	**conf mean**	**conf median**	**conf entropy**	**cos min**	**cos max**	**cos mean**	**cos median**	**cos entropy**	**acc min**	**acc max**	**acc mean**	**acc median**
1st hit	0.6196	0.4993	0.5528	0.5542	0.4387	0.5084	0.4763	0.5016	0.4989	0.4980	0.5004	0.4996	0.5118	0.4996
Mean	0.6325	0.5009	0.5613	0.5530	0.4350	0.5138	0.4818	0.5027	0.5058	0.4980	0.5004	0.4996	0.5122	0.4996

the fix commits have been manually untangled. Additionally, we focus our study on the sole business logic lines, thus reducing further the risk of considering as buggy, lines irrelevant to the bug at hand.

Finally, our evaluation metrics may induce some additional threats. Our comparison basis measurement, i.e., comparing the ranking of source-code lines that has been trained on the same source-code's project with approaches that are agnostic and has been trained on multiple projects. It is hard to compare the advantage gained by training CodeBERT on a large number of projects, including eventually the projects in our dataset, against the advantage gained by the n-gram models when trained on a specific project source-code and predicting the naturalness of lines of that same project.

6.5 Results

6.5.1 RQ1: Metrics and aggregation methods

To evaluate the CODEBERT-NT metrics, we rank the subject lines according to the aforementioned metrics. We start by calculating one score per line for every metric, by aggregating the scores of the predictions in that line. Then, we sort the lines according to their score in both orders - ascendant and descendant - and calculate the paired (by bug) Vargha and Delaney $\hat{A}_{12}$ difference between both orders effectiveness in terms of attributing the lowest ranks to the buggy lines; more precisely the average ranking of

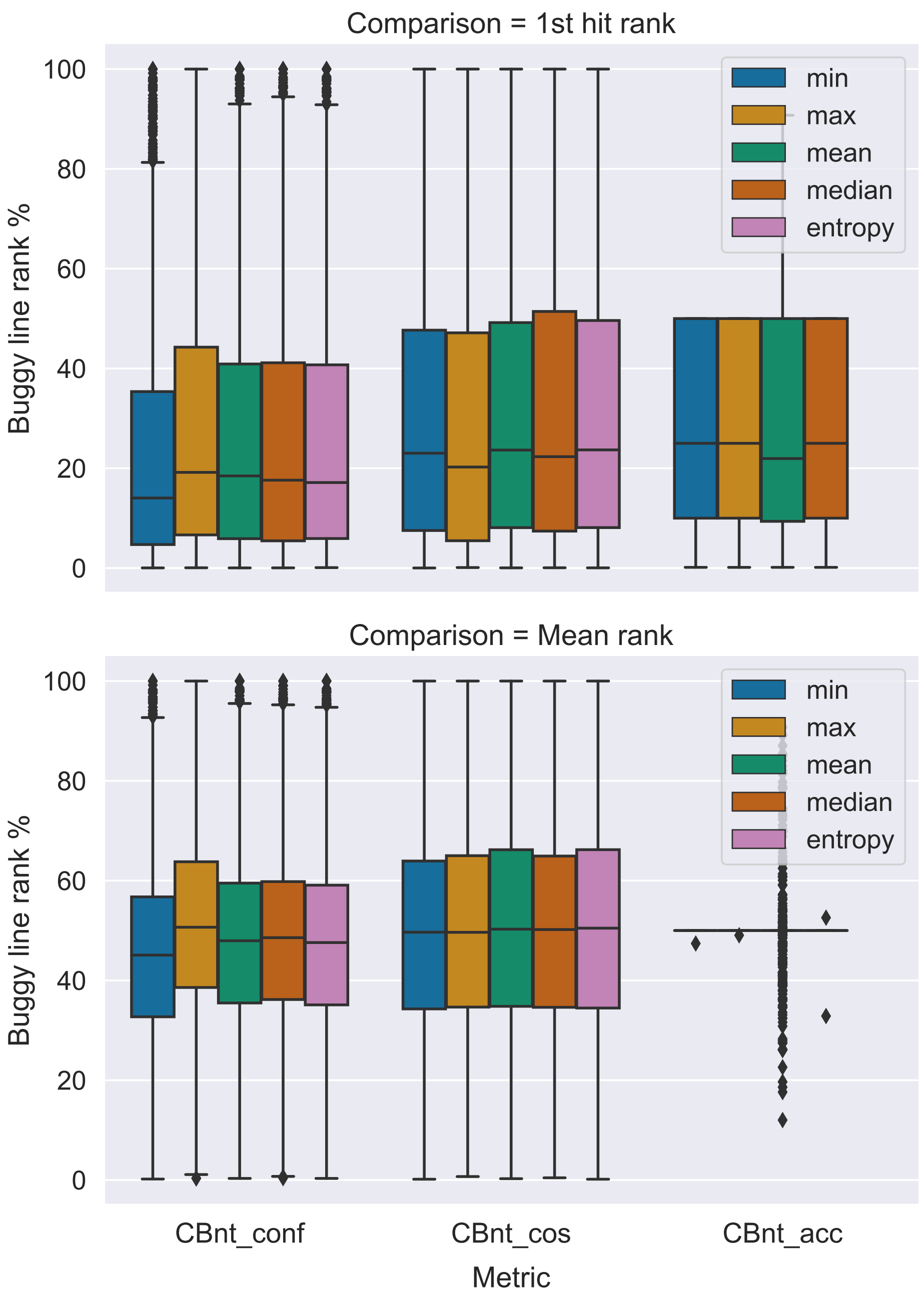


Figure 6.2: Buggy lines ranking using the three available metrics with different aggregation functions. A line is more likely to be buggy when it has a *low min CBnt_conf*, a *high max CBnt_cos* and *low mean CBnt_acc*.

the buggy lines and the smallest rank attributed to a buggy line, per bug. (Please refer to section 6.4 for details)

These results are depicted in Table 6.1 where values around 0,5 depict that the considered pair of metric-aggregation method does not bring any advantage in ranking buggy lines first, while values above 0,5 confirm an advantage for the ascendant sorting order and below 0,5 an advantage for the descendant one. From these results, we select the most suited sorting order for each pair of metric-aggregation and use it in the following experiments.

Figure 6.2 depicts the box-plots of the normalised rankings by number of lines of each bug. Interestingly, the prediction confidence score (CBnt_conf) seems to be a good indicator of naturalness and thus of bugginess likelihood. Noticeably, low confidence scores are more often attributed to buggy lines than neutral ones. This explains why aggregating this metric by using the maximum value by line gives the worst guidance to our target, while sorting by the minimum score gives the best ranking over all considered pairs metric-aggregation.

At the same time, we do not observe any relevant differences when ranking the lines by either the cosine of the embedding (CBnt_cos) or the correctness of predictions (CBnt_acc). This is also confirmed by the Table 6.1 where most of the $\hat{A}_{12}$ values are around 0,5 when using each of the aggregation methods on these two metrics. Nevertheless, we notice that the best method to rank the buggy lines by cosine similarity is via ranking the lines by their highest scores of similarity, where the high (max) values are considered as symptoms of unnaturalness. We also notice a trend that confirms the correlation between naturalness and code predictability, such as the lower the mean of correct predictions in a code is, the less natural is the code.

By checking Table 6.1 we also see that the difference of ranking the lines by low confidence, from the other metric-aggregation methods is significant, Vargha and Delaney $\hat{A}_{12}$ effect size values are more than 0.6, which are significantly higher to the rest of confidence aggregation rankings. Less noticeable, the ranking by increase of the line's maximum embeddings cosine-simalirity and mean prediction accuracy yield respectively $\hat{A}_{12}$ values of around 0,52 and 0,51 when compared to the other studied aggregation methods when applied on the same metrics.

CODEBERT-NT can infer the naturalness of code through masked token predictions. The unnatural information can be inferred the best from the decrease of prediction confidence (considering the minimum value per line), the increase of cosine similarity between the embedded original and predicted code (considering the maximum value per line) and the decrease of average prediction accuracy per line.

6.5.2 RQ2: Comparison with random and complexity based rankings

Figure 6.3 shows the distribution of the normalised rankings of the first ranked buggy line and the average rank of buggy lines by bug when using CODEBERT-NT metric-aggregation pairs selected from the results of RQ1 (ascendant minimum CBnt_conf, descendent maximum CBnt_cos and ascendant mean CBnt_acc), uniform random ranking and token-number-complexity descendant ranking.

Surprisingly, random and complexity lead to similar rankings with a small advantage for random. This observation implies that tokens-number-complexity does not capture well the code naturalness in the studied setup: naturalness on the line-level-granularity of the business-logic code.

As can be seen from the boxplots, CODEBERT-NT outperforms the baseline techniques in ranking the buggy lines first, using any of its three metrics. In fact, except for a small portion of our dataset bugs, CODEBERT-NT low-confidence performs the best in estimating the bugginess of the target lines, with respectively a 1st hit and mean

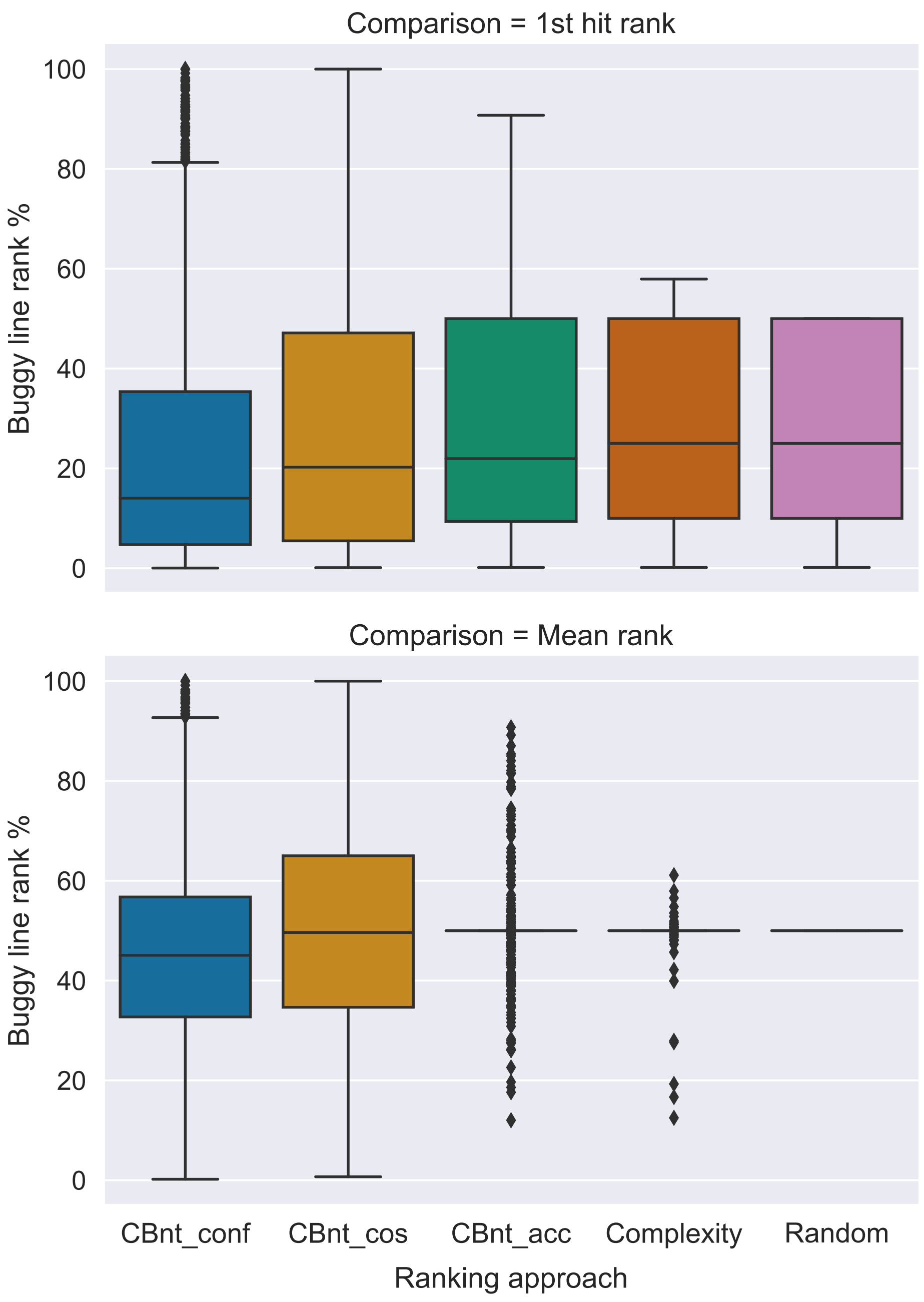


Figure 6.3: Comparison of the buggy lines rankings by CODEBERT-NT (CBnt_conf), Random and Complexity (number of tokens by line). CODEBERT-NT outperforms Random and Complexity in ranking buggy lines.

Table 6.2: Vargha and Delaney $\hat{A}_{12}$ of CodeBERT-nt low-confidence-metric (CBnt_conf) rankings compared to the other ones.

CBnt_conf Vs	**CBnt_cos**	**CBnt_acc**	**Complexity**	**Random**
1st hit	0.578	0.609	0.605	0.607
mean	0.565	0.619	0.620	0.622

buggy-line ranks lower in average by around 6% and 4.7% than the following ranks – attributed by the cosine similarity – and around 11% and 5% lower than the least performing ranking effectuated by uniform random. This noticeable difference between the CodeBERT-nt results with the two considered techniques, more specifically the low confidence CBnt_conf ranking, endorses the fact that CodeBERT-nt can be considered as a comparison baseline and a new method for naturalness based tasks.

Although, the two remaining CodeBERT-nt metrics, average accuracy CBnt_acc and high embeddings similarity CBnt_cos, outperform both baselines and yield lower rankings than uniform random by respectively 3% and 5% in ranking the first hit buggy line, they perform however similarly to the baselines when comparing their mean ranking of all the buggy lines, with a small advantage of only 0,4% for the ranking by embeddings similarity.

To validate this finding, we perform a statistical test (Vargha and Delaney $\hat{A}_{12}$ and Wilcoxon paired test) on the data of Figure 6.3 to check for significant differences. Our results showed that the differences are significant, indicating the low probability of this effect to be happening by chance. As illustrated in Table 6.2, the size of the difference is also big, with CodeBERT-nt low-confidence yielding Vargha and Delaney $\hat{A}_{12}$ values between 0.58 and 0.6 indicating that CBnt_conf ranks the buggy lines the best in the great majority of the cases. However, the CodeBERT-nt accuracy and embeddings-similarity metrics outperform random respectively, in only 51% and 48% of the cases.

CodeBERT-nt metrics describe source-code naturalness more accurately than the baselines uniform-random-selection and tokens-count-complexity based rankings. CodeBERT-nt low confidence (CBnt_conf) is the most effective metric and outperforms the uniform-random-ranking by 11% in ranking the first hit buggy line and 5% in ranking all the buggy lines, in average.

6.5.3 RQ3: Comparison with n-gram

To answer this question, we train two n-gram models per buggy version of our dataset that we then use to compute the cross-entropies of the subject lines from the corresponding bug, then we rank these lines according to the resulting values and reproduce the same analysis as in RQ2. We illustrate in Figure 6.4 the distribution of the normalised rankings of the first ranked buggy line and the average rank of buggy lines by bug when using CodeBERT-nt low confidences ranking – CBnt_conf –, the descendant cross-entropy ranking from a UTF8-tokenizer-based n-gram model and a JavaParser-tokenizer-based one. Additionally, we illustrate the random ranking in the boxplots as the simplest baseline for this task.

As expected, the three approaches outperform the uniform-random-ranking in most of the cases and yield very comparable results with a small advantage to CodeBERT-nt on ranking the first buggy line over the n-gram techniques and a small advantage to JP n-gram in regards of the average rank of buggy lines. In both comparisons, UTF8 n-gram falls slightly behind these two latter techniques.

To validate this finding, we performed a similar statistical test as in RQ2 on the data of Figure 6.4 and found that the differences with random are significant, while the differences between n-gram cross-entropy and CodeBERT-nt low-confidence rankings are negligible. As illustrated in Table 6.3, the size of the $\hat{A}_{12}$ differences are equal to 0.518 and 0.468 between CodeBERT-nt and JP n-gram models and 0.536 and 0.502

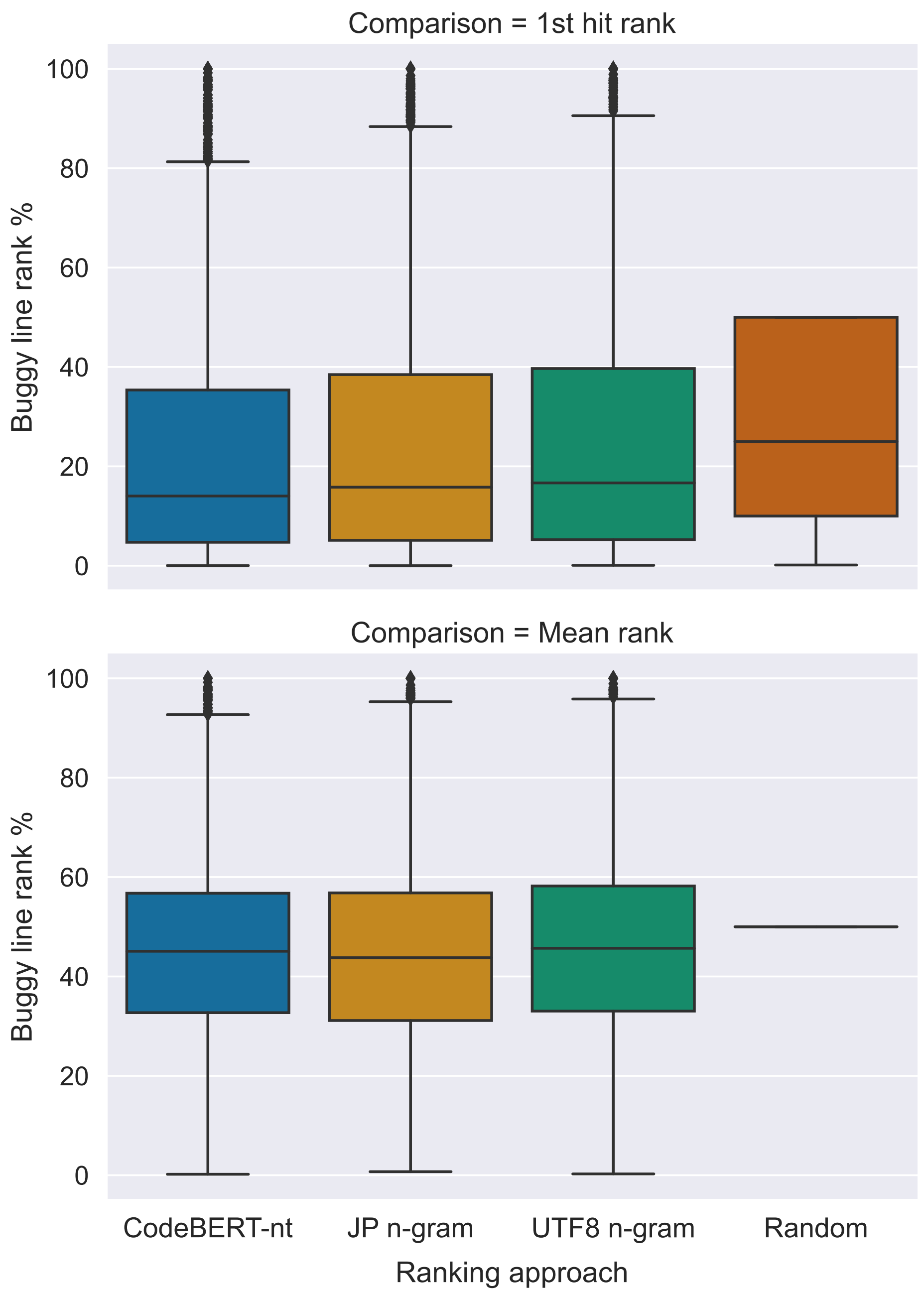


Figure 6.4: Comparison of the buggy lines rankings by CodeBERT-nt (CBnt_conf), UTF8 n-gram and JP n-gram models (created respectively using UTF8 and Java Parser tokenizers). CodeBERT-nt ranking is comparable to the n-gram models one.

Table 6.3: Vargha and Delaney $\hat{A}_{12}$ of CODEBERT-NT low-confidence-metric (CBnt_conf) rankings compared to n-gram and Random ones.

Approaches $\hat{A}_{12}$	**Random**		**UTF8**		**JP**	
	1st hit	mean	1st hit	mean	1st hit	mean
CBnt_conf	0.606	0.621	0.536	0.502	0.518	0.468
JP	0.575	0.630	0.559	0.588		
UTF8	0.560	0.602				

between CODEBERT-NT and UTF8 n-gram models for both reported rankings, meaning that both approaches yield comparable results, which confirms that CODEBERT-NT can carry out code naturalness related applications with similar effectiveness as the statistical language n-gram models.

CODEBERT-NT masked token prediction confidence indicates naturalness of bugs as accurately (slightly better) as program-specific n-gram models.

6.6 Discussion

6.6.1 Impact of interesting lines selection

Our empirical results show evidence that CODEBERT-NT can infer source-code naturalness yielding the same results as n-gram models and outperforming the uniform-random and code-complexity based techniques in attributing higher ranks to buggy lines when sorting source-code by naturalness. These experiments have been driven on the same buggy versions source-code whose lines count at least one buggy line. Precisely, we have excluded all the lines outside the business-logic source-code and kept only the bugs that counted at least one buggy line within their remaining lines.

To better understand the impact of this line selection step, we reintroduce all the bugs with their full source-code in our dataset and reproduce the same study as in

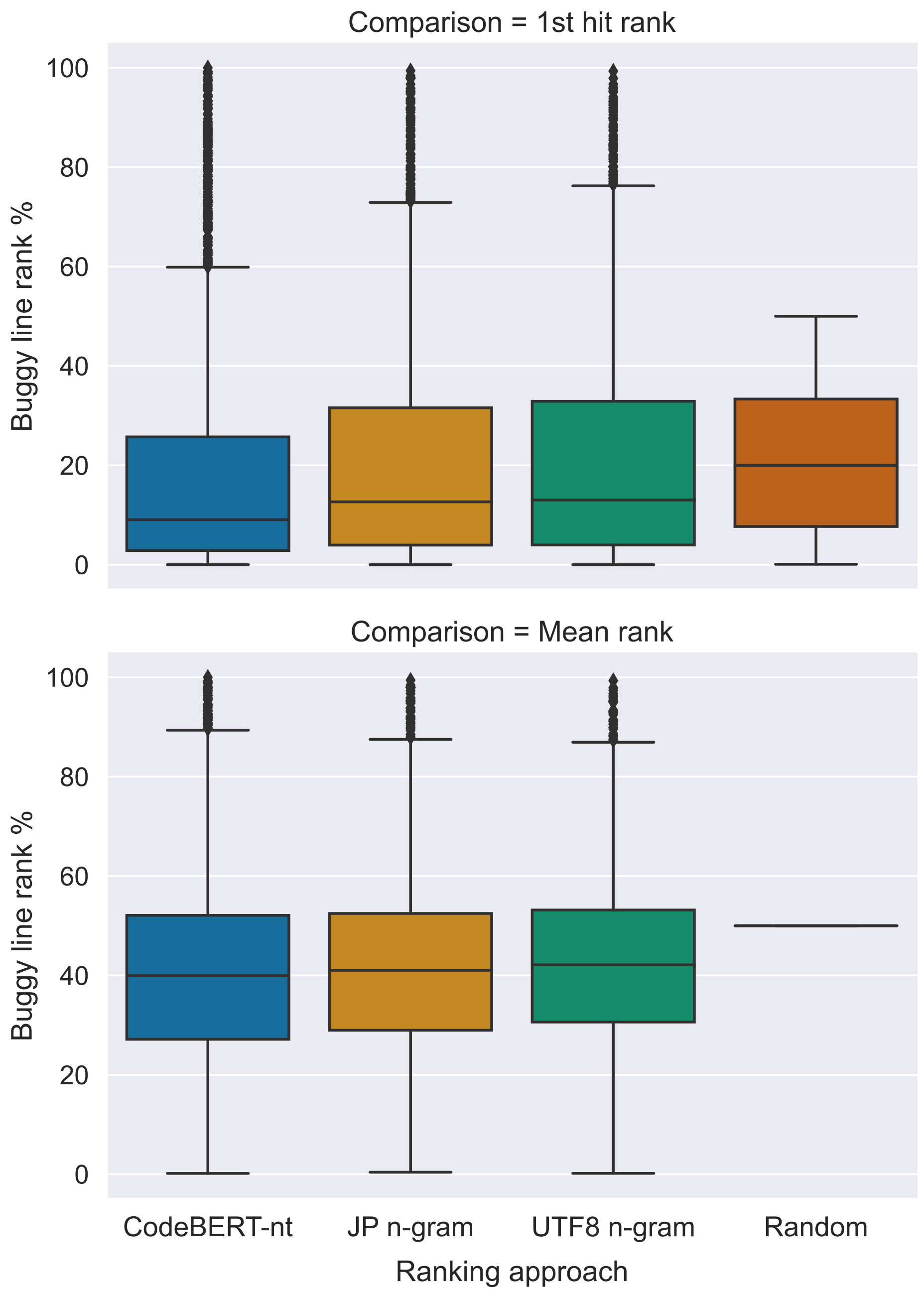


Figure 6.5: Comparison of the buggy lines rankings by CODEBERT-NT (CBnt_conf), UTF8 n-gram and JP n-gram models (created respectively using UTF8 and Java Parser tokenizers) when ranking all lines. CODEBERT-NT ranking is comparable to the n-gram models one and ranking the business-logic source-code lines first give it an advantage over the n-gram ranking.

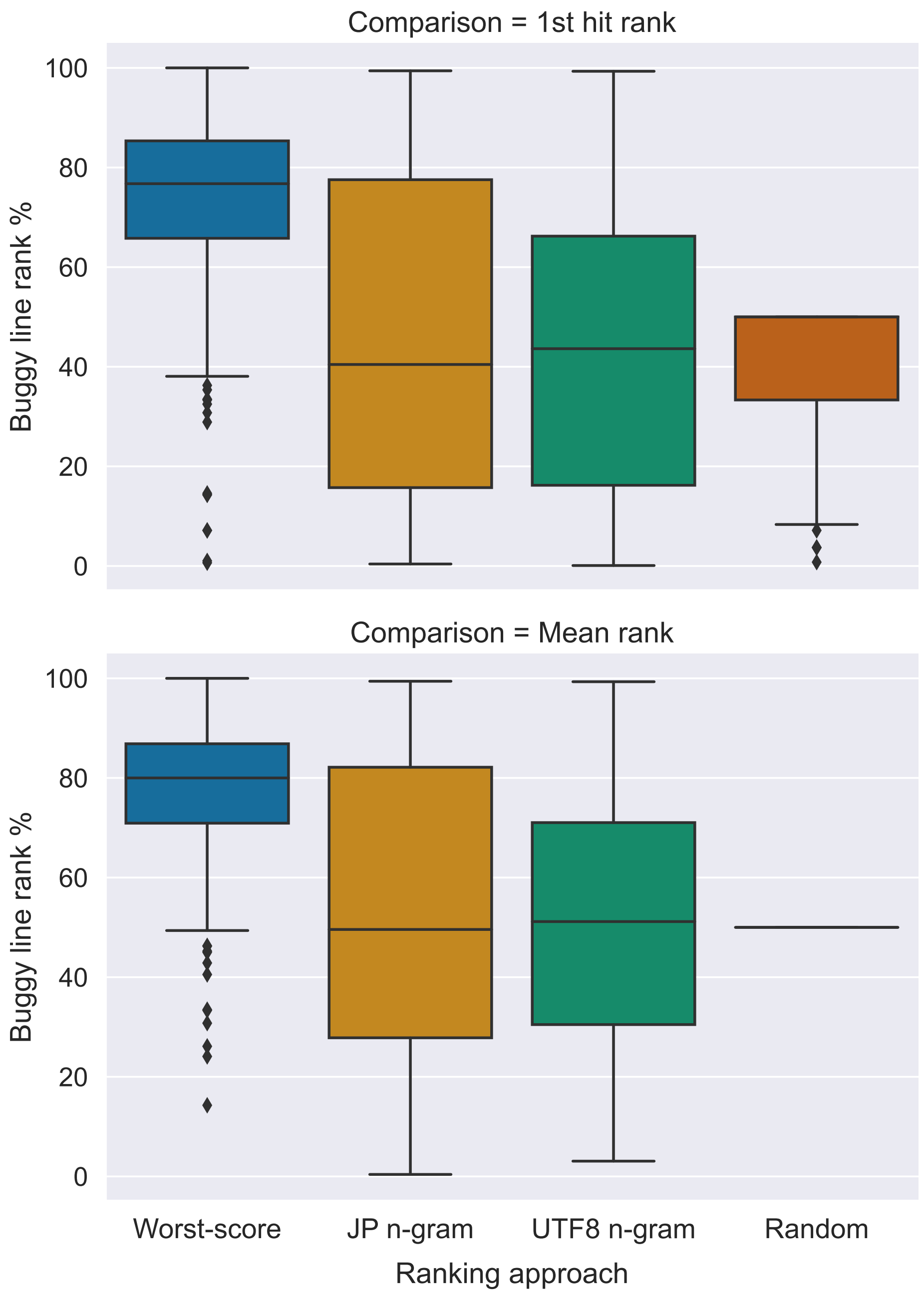


Figure 6.6: Comparison of the buggy lines rankings by worst possible scores, UTF8 n-gram and JP n-gram models (created respectively using UTF8 and Java Parser tokenizers) when targeting the buggy versions not exposing any business-logic-related buggy line. n-gram techniques perform similarly to random on these subject buggy versions.

RQ3. We have then attributed the worst rank to all unranked lines by CodeBERT-nt, i.e., outside of the business-logic. This implies that for CodeBERT-nt, the business-logic lines are ranked first by their min-confidence and the remaining lines are ranked after with a random uniform selection logic. The n-gram approaches ranking is applied as described previously, the same way on all lines – business- and non-business-logic related ones – having each two cross-entropy values from every corresponding n-gram model. The ranking distributions are illustrated in Figure 6.5.

Although the results of the three approaches remain comparable, the trend is that for a noticeable portion of studied bugs, CodeBERT-nt remains able to rank buggy lines better than the n-gram models. The difference is wider and more visible between the ranks of the first buggy line which can be seen in the left boxplot of Figure 6.5. Interestingly, we observe that ranking the business-logic lines with CodeBERT-nt and the remaining lines with a uniform random ranking outperforms ranking all the lines (business- and not business-logic related ones) by their n-gram calculated cross-entropies. These results lead to the conclusion that the naturalness analysis of the non-business-logic lines do not contribute with useful information to the considered ranking tasks, but instead alters its results when attributing a higher rank to the targeted lines.

To check whether this observed decrease of performance for n-gram is indeed caused by the additional lines or because they performed worst on the previously excluded bugs from our dataset, we reproduce the same comparison on the subset of our dataset where all bugs are located outside of the business-logic code, implying that CodeBERT-nt will attribute the worst score to every buggy line. We illustrate the rankings distribution of the ”worst-score” strategy (ranking all buggy-lines last), JP and UTF8 n-gram models and uniform-random in Figure 6.6.

Although small, the n-gram models kept some advantage over random ranking as in the Figures 6.4 and 6.5, in contrast to CodeBERT-nt’s ”worst-score” results.

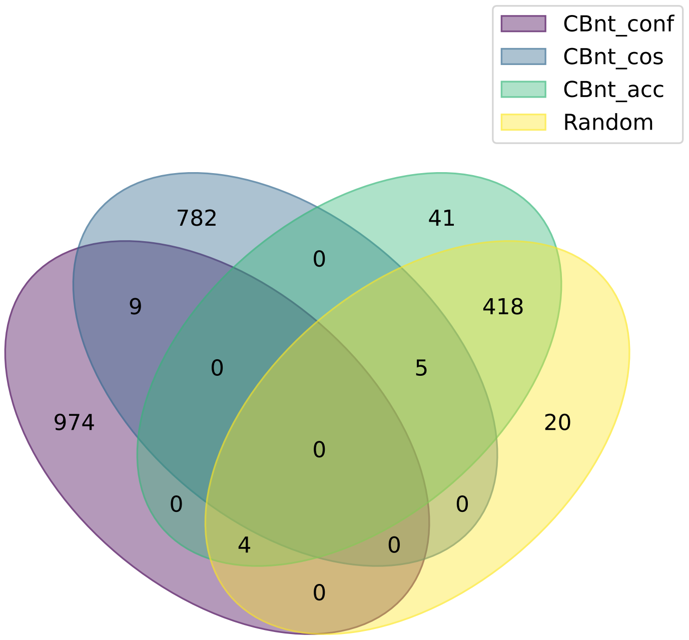
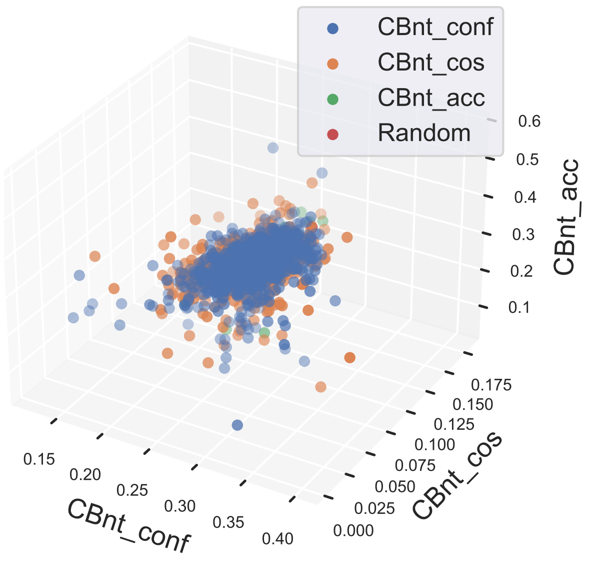


Figure 6.7: Which metric ranks the best the buggy lines, in most of the cases? CodeBERT confidence CBnt_conf performs the best for around 50% of the cases, followed by CBnt_cos then CBnt_acc which perms almost similarly to random. Therefore, There's no big benefit in using CBnt_acc while it could be interesting to complement CBnt_conf capabilities, using CBnt_cos.

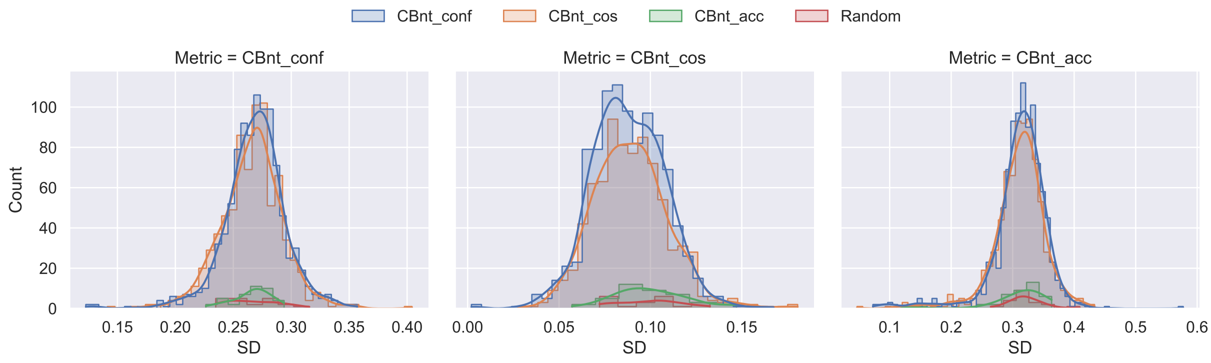
The contrasting results between the Figures 6.6 and 6.5 highlight the negative impact of ranking the not business-logic lines by naturalness as they compensated CODEBERT-NT's disadvantage of attributing the worst ranks to buggy lines, in 10% of the studied cases. Consequently, these results reinforce our conclusion that including the non-business-logic lines in the analysis adds noise to the search-space [161], and consequently hinders the ranking accuracy.

6.6.2 Which metric to use for which bug?

The empirical results driven on our large set of buggy versions highlight the effectiveness of CODEBERT-NT in capturing the naturalness of source-code, especially via its low-



(a) Scatterplot of bugs best-ranking-metric by standard deviation.



(b) Histogram of bugs best-ranking-metric by standard deviation.

Figure 6.8: Distribution of Bugs best-ranking-metric by standard deviation of the metrics measured on their corresponding subject lines. Except for few scores, CBnt_conf ranks the majority of the bugs the best independently from the measured SDs.

.

confidence metric. However, from the outliers in the Figures 6.2, 6.3, 6.4 and 6.5, we notice that CODEBERT-NT does not perform equally on all considered bugs. Which means for instance that it outperforms uniform-random ranking in the majority of the cases, but yields worst rankings for a small portion of the dataset. Therefore, we turn our attention towards investigating the possibility of better handling those bugs, leveraging one of the other CODEBERT-NT metrics.

We start by mapping every metric with the bugs on which it attributed the best mean ranking to the buggy lines. In figure 6.7, we illustrate a Venn diagram of this distribution, including uniform-random-ranking as baseline. As shown in our results, the CODEBERT-NT confidence is the best naturalness indicator for the majority of bugs, followed by the cosine similarity and the prediction accuracy. We also notice that except for a minority of 20 bugs, at least one of the CODEBERT-NT metrics achieves better scores or similar to random-uniform ranking. Additionally, except the large intersection set of bugs that are best-ranked by the prediction accuracy and uniform-random-ranking, the metrics rarely achieve their best rankings of the buggy lines for the same bugs. This observation introduces the hypothesis that the metrics are complementary and eventually, one could rely on different metrics for different bugs.

Aiming at distinguishing between our dataset bugs, we measure the metrics standard deviation (SD) per studied lines. In this setup, we exclude from our clusters the bugs that are intersecting with random and plot the SD distributions of the remaining ones, in Figure 6.8. The plots illustrate that the bugs from different clusters share similar ranges of SD with mean values around 0.27, 0.09 and 0.31 for respectively CBnt_conf, CBnt_cos and CBnt_acc. Also in the majority of the cases, the SD of the bugs best treated by other metrics than CBnt_conf fall in the same range of this latter, thus, cannot be distinguished from each other. However, we notice that for some SD values, CBnt_cos ranks better more bugs than CBnt_conf. This difference is noticeable for roughly: $SD(CBnt\_conf)$ values between 0.2 and 0.24 or above 0.35, $SD(CBnt\_cos)$ values above 0.14 and

$SD(CBnt\_acc)$ values below 0.05 or between 0.19 and 0.28 or between 0.395 and 0.42. These SD ranges represent a small fraction of our dataset, exactly 357 bugs, among which 193 where CBnt_cos performed the best and 200 where it outperformed CBnt_conf, which correspond respectively to 15.8%, 8.5% and 8.8% of the studied bugs. Nevertheless, these results may motivate future investigation on the use of CBnt_cos over CBnt_conf in similar cases, on different setups.

6.6.3 Impact of generating more predictions per token?

To have a better understanding on whether generating more predictions from the model could improve the bugginess information retrieved by CODEBERT-NT, we extend our experiment of RQ1 by comparing the ranking results using the best proven pairs of metric-aggregation from our results, when generating 1, 2, 3, 4 and 5 predictions per token. We illustrate in Figure 6.9 the box-plots of the normalised rankings by number of lines of each bug. Although comparable, the results depict a clear dissipation of the bugginess indicators retrieved from the prediction confidence and the cosine similarity metric, when we aggregate the values of more than one prediction by token. In the other hand, we can see that the ranking performance effectuated by the prediction accuracy raises when we consider 2 and 3 predictions then converges to a stable value. Besides the fact that this increase confirms further the correlation between code-naturalness and predictability, it remains negligible and keeps this metric-ranking far below the low-confidence one. Therefore, we believe that it would be more cost-efficient and appealing for similar studies, to generate only 1 – eventually up to 3 – predictions instead of 5, as the default setting of CodeBERT.

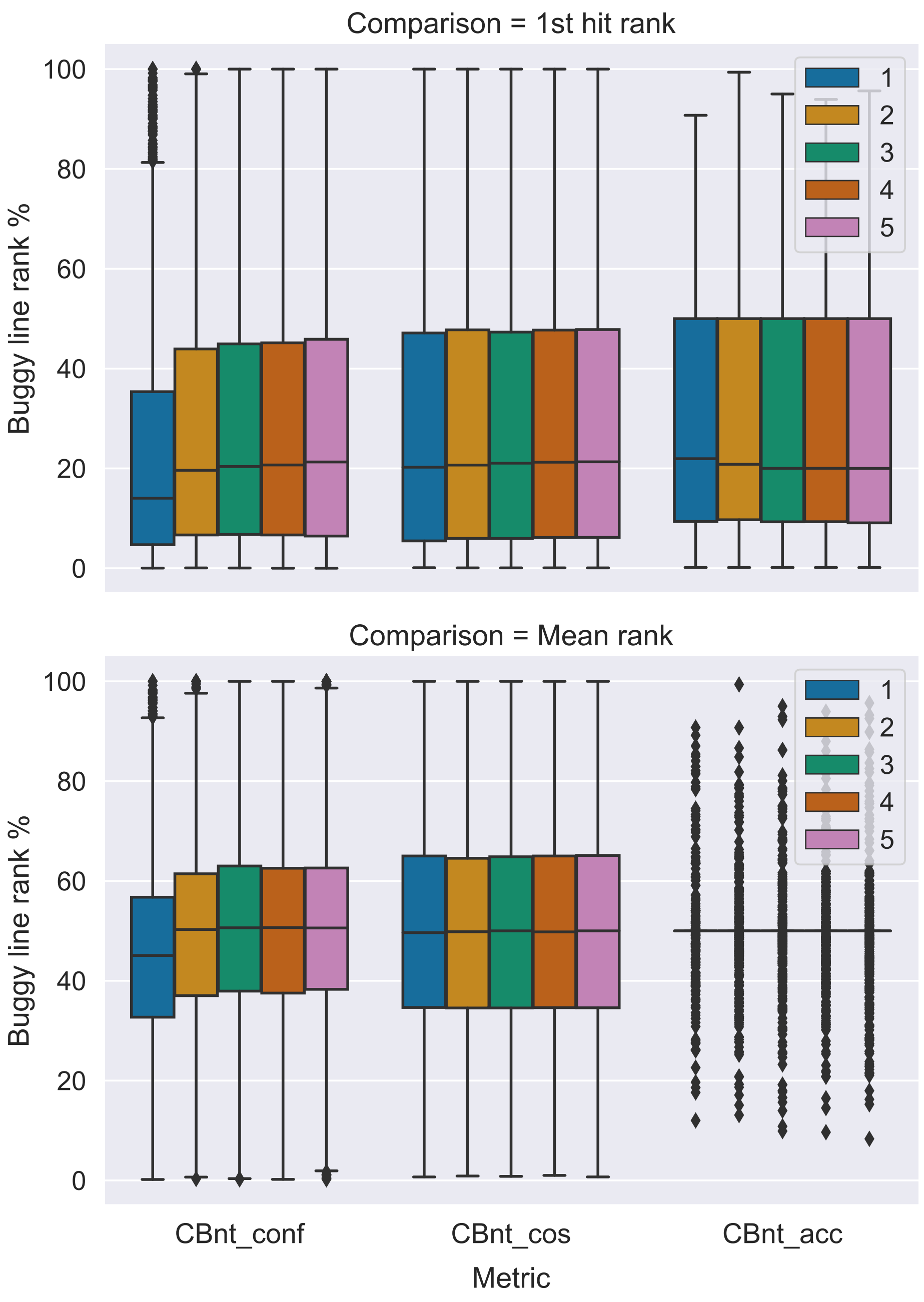


Figure 6.9: Buggy lines ranking using 1, 2, 3, 4 and 5 predictions per token. The more predictions we use, the more the information about the confidence gets dissipated, thus the more the ranking performance decreases, except for CBnt_acc.

6.7 Conclusion

Naturalness of code forms an important attribute often needed by researchers when building automated code analysis techniques. However, computing the naturalness of code using n-gram requires significant amount of work and a salable infrastructure that is not often available. An alternative solution is to use other readily available language models, perhaps more powerful than n-grams, such as transformer-based generative models (CodeBERT-like). Unfortunately, these models do not offer any token-based appearance estimations since they aim at generating tokens rather than computing their likelihood. To this end, we investigate the use of predictability metrics, of code tokens using the CodeBERT model and check their appropriateness in bug detection. Our results show that computing the confidence of the model when masking and generating a token, irrespective of whether the predicted token is the one that was actually predicted by the model, offers the best results, which are comparable (slightly better) to that of n-gram models trained on the code of the same project (intra-project predictions).

Chapter 7

Conclusions

Dissertation Summary

This dissertation presents studies, approaches and tools that tackle important challenges of fault injection, with the aim of improving the usability of artificial faults and mutation testing in practice. The main contributions are: 1) an approach to introduce faults that mimic the behaviour of real bugs using information retrieved from bug reports and inverted automated-program-repair fix-patterns, 2) a mutation testing approach that seeds “natural” faults based on big code knowledge by leveraging pre-trained language models and 3) an approach that measures code naturalness through pre-trained language models prediction. We made the implementation of these approaches as well as the reproduction packages of our studies publicly available, to support further research in this direction.

In the first part, we propose a targeted bug-report driven fault injection approach, that we untitled IBIR. Its key goal is to enable researchers and practitioners to inject “realistic” faults in a specific targeted feature or component of the software or system under test. This way IBIR faults can be used as substitutes of real ones in targeted test assessment and fault tolerance campaigns. To do so, it takes as input the target project and bug report. It starts by finding the locations that are the most likely to be related to the input bug report description by applying an information retrieval fault localisation technique (IRFL). Then it applies on these locations, fault-patterns that has been created by inverting fix-patterns collected by automated-program-repair researchers from multiple real bug fixes. This way, our approach brings realism to the injection by targeting the features described in the given bug report and applying patterns crafted from real bug fixes.

In the second part, we approach the fault injection challenges from a different perspective, i.e. in a mutation testing context, in which we target the whole scope of the project under test and not a specific bug or feature. We propose an approach, that we named μBERT, which aims at generating "natural" mutants, in the sense that they are easy to understand and more likely to happen, in order to guide practitioners in writing tests of higher fault detection capabilities. To do so, μBERT replaces different pieces of the input program with pre-trained language model predictions, producing numerous likely-to-occur mutations. The approach targets diverse business code locations and injects either simple one-token replacement mutants or more complex ones by extending the control-flow conditions. This provides probable developer-like faults impacting different functionalities of the program with higher relevance and lower cost to developers.

In the third part, we turn our interest to the possibility of identifying relevant locations for developers, without any given prior such as a target bug report or commit changes. Particularly, we investigated whether we could infer code naturalness through pre-trained language model predictions, thereby distinguishing unnatural code locations from natural ones. This unnaturalness can be an indicator of smelly code or of low-readability which may be a symptom of bugginess and bug-proneness, and thus requiring prior attention from developers and testers. Although powerful, computing this measure can be tedious, involving training language models, typically n-gram ones, which suffer from scalability issues and tend to overspecialize. For this purpose, we propose the usage of already trained generative language models to estimate code naturalness through their prediction performance. This way, we overcome the training issues and efforts and enable the computing of a more generic code naturalness measure. We implemented this approach which we untitled CODEBERT-NT and showed empirically that it can capture naturalness efficiently, particularly obtaining comparable results as intra-project n-gram models and outperforming random and complexity-based approaches in distinguishing buggy lines.

Perspectives

In the following, we discuss some potential future research that follows the contributions and ideas presented in this dissertation:

- *Complementarity of mutation approaches*: from Figure 5.6, we can see that some faults are found by one mutation testing approach and not by the others. This indicates that the tools complement each other and it would be interesting if we could find an efficient way to take advantage of each approach's capabilities. This is further highlighted by the distribution of IBIR most semantically similar faults with respect to their inducing patterns in Figure 4.12. Although with different proportions, we can see that all patterns succeed to introduce strong mutants. Hence, it is interesting to envisage the subsumption study of the mutation operators from different approaches and eventually propose a more "complete" fault injection approach, able to represent and reveal a larger space of faults, while producing fewer duplicate and equivalent mutants.

- *Mutant selection and prioritization*: Our experiments provide empirical evidence that IBIR is efficient in injecting faults that emulate target bugs. However, as the injection approach depends on input bug reports, it may be useless in new projects, i.e. where no bugs have already been reported, or in use cases not requiring particular bugs or bug reports to target. Intuitively, in such situations, one could use the mutator component of IBIR without the IRFL, by injecting faults in all locations in a brute-force manner. However, this method may lead to similar results as most of the traditional mutation techniques, with an overwhelming number of faults to analyse for the end-user. Similarly, μBERT injects currently faults in diverse locations of the code in an arbitrary order. It would be interesting to investigate whether we could improve the usability of the proposed approaches by employing a smart mutant selection strategy, i.e. prioritizing subsuming ones or those with higher fault detection probability over the others, as explained in the Related Works in Section 3.4. One could also investigate whether we could amend such selection approaches with new code features, such as the naturalness of the mutant or the mutated location, as they may give hints on whether a location is likely to be buggy or not, and thus may favour fault revealing mutants over others.

- *Generative language model based code naturalness*: From Figure 6.7, we can see that the low confidence of the model is not always the best indicator of code unnaturalness, or of probable bugginess. Indeed, for a large proportion of our dataset, the cosine of the embeddings distinguished better the buggy lines from the others. Although it is hard to clearly define these cases, we could determine some of them based on the standard deviation of the target code measured by CODEBERT-NT metrics. This remains a shallow first step in this direction and it would be interesting to investigate whether other techniques or measures could better determine which naturalness metric to use depending on the project at hand.

Chapter 8

List of Papers and Tools

8.1 Papers included in the dissertation

1. Ahmed Khanfir, Anil Koyuncu, Mike Papadakis, Maxime Cordy, Tegawende F. Bissyandé, Jacques Klein, and Yves Le Traon.
   IBiR: Bug Report driven Fault Injection.
   In: *ACM Transactions on Software Engineering and Methodology (TOSEM 2022). Accepted on May 2022.*
   https://doi.org/10.1145/3542946

2. Ahmed Khanfir, Matthieu Jimenez, Mike Papadakis, and Yves Le Traon.
   CODEBERT-NT: code naturalness via CodeBERT.
   In: *22nd IEEE International Conference on Software Quality, Reliability and Security (QRS'22)*. 2022.
   http://hdl.handle.net/10993/53506

3. Ahmed Khanfir.
   Effective and scalable fault injection using bug reports and generative language models.

In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. 2022. Association for Computing Machinery, New York, NY, USA, 1790–1794. https://doi-org.proxy.bnl.lu/10.1145/3540250.3558907

4. Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Efficient Mutation Testing via Pre-Trained Language Models. Submitted to: *IEEE Transactions on Software Engineering (TSE 2023)*. 2023. https://doi.org/10.48550/arXiv.2301.03543

8.2 Papers not included in the dissertation

During my journey at the Interdisciplinary Centre of Security Reliability and Trust (University of Luxembourg) as a PhD student, I had the opportunity to collaborate on side projects related to the fields of Fault Injection and Software Security. These research activities involved collaboration with researchers at the national and international levels, precisely from:

- the *Interdisciplinary Centre of Security Reliability and Trust, University of Luxembourg, Luxembourg*,
- the *University of Namur, Namur, Belgium*,
- the *Sabanci University, Üniversite Caddesi No:27 34956 Istanbul, Turkey* and
- the *University of Copenhagen, Umeå University, Sweden*

8.2.1 Fault Injection

In these collaborative works, we show that:

1. The syntactic similarity (code textual similarity) of injected faults w.r.t. real ones does not imply their semantic similarity to real bugs [35]. In addition, we show that real bugs can be semantically resembled by artificially injected faults. For instance, IBIR [43] is able to resemble 76.44% of the considered real bugs in this study, followed by Pitest [164] and μBERT$_{conv}$ [45], resembling respectively 65.11% 61.39% and 9.76% of the considered real bugs.

   More details on this work can be found on the following 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. Submitted revision to : *IEEE Transactions on Software Engineering (TSE 2023).* 2023. https://doi.org/10.48550/arXiv.2112.14508

2. Mutation Testing techniques do not benefit equally from the cost reduction offered by Mutant Selection strategies, which give an advantage to certain approaches over others [165]. This leads to observing different results when comparing the cost efficiency of mutation testing techniques (in inducing test suites that reveal real faults), under different mutant selection strategies. For instance, among other results, we observed that IBIR is the most effective in inducing test suites that reveal real bugs (an average of about 90% of the studied bugs). However, its cost-efficiency is comparable to other considered approaches (μBERT$_{conv}$, Pitest), under a uniform random mutant selection strategy. In contrast, when the selection is guided by CEREBRO [101] (an NMT-based mutant selection strategy), we observed an improvement in the cost-efficiency of all the approaches, with a significant advantage for μBERT$_{conv}$. In fact, test suites written to kill μBERT$_{conv}$ mutants reveal more bugs than other studied approaches, when spending the same effort.

   More details on this work can be found in the following article:

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

8.2.2 Software Security

In these side projects, we address challenges related to Software Security, particularly to the fields of vulnerability detection, vulnerability injection and Android malware detection.

1. We proposed Confuzzion [166], a novel java virtual machine (JVM) fuzzer that aims at detecting java type confusion vulnerabilities. The fuzzer generates input java programs through mutations (i.e. adding methods and statements), which it executes on a JVM under test aiming at revealing type confusion vulnerabilities. A contract checker is added after each statement to verify whether the variable types at run-time correspond to their declared ones or otherwise, a type confusion has been triggered. The fuzzer decisions (i.e. which mutation operator to apply or which method to target) are pseudo-random. In fact, they are guided by the execution results from the previous iterations, to help in generating valid input programs that enable exploring new/diverse execution paths. More precisely, It keeps evolving valid (successfully executed) inputs and favours invoking methods that are harder to execute successfully; those that tend to throw exceptions, as they may require specific arguments that are hard to construct randomly.

   More details on this work can be found in the following article:

   William Bonnaventure, Ahmed Khanfir, Alexandre Bartel, Mike Papadakis and Yves Le Traon.
   Confuzzion: A Java Virtual Machine Fuzzer for Type Confusion Vulnerabilities. In : *21st IEEE International Conference on Software Quality, Reliability and Security (QRS'21)*. 2021. https://doi.org/10.1109/QRS54544.2021.00069

2. We proposed IntJect [167], a novel vulnerability injection technique that combines semantic-preserving mutations with an NMT learning approach. Given a dataset of vulnerabilities (presented as $<$ benign, vulnerable $>$ code pairs), our approach

creates a new semantically similar dataset by employing semantic-preserving program mutations on the pairs' code. Then, it learns how to inject the vulnerabilities via an NMT approach (Seq2Seq). The idea behind using Seq2Seq is to learn the intent (context) of the vulnerable code in a manner that is agnostic of the specific program instance.

More details on this work can be found in the following article:

Benjamin PETIT, Ahmed Khanfir, Ezekiel Soremekun, Gilles Perrouin, Michail Papadakis.
IntJect: Vulnerability Intent Bug Seeding. In : *22nd IEEE International Conference on Software Quality, Reliability and Security (QRS'22)*. 2022.
http://hdl.handle.net/10993/53858

3. We have investigated the possibility of detecting Android Malware based on the Manifest data of the applications, by tuning BERT (a pre-trained language model) for this task [168]. We have also investigated whether we could train BERT to classify malware into families. To this end, we used a large-scale dataset of Android applications (containing malware and goodware) to train multiple models on top of BERT, using different features from the applications' manifests. Our study showed that BERT can accurately (97%) distinguish between malware and goodware. In addition, BERT can also classify accurately (93%) malware into their corresponding families. Moreover, we observed that the Android permissions are not the key data responsible for classifying whether an application is malware or goodware.

   More details on this work can be found in the following article:

   Badr Souani, Ahmed Khanfir, Alexandre Bartel, Kevin Allix and Yves Le Traon.
   Android Malware Detection Using BERT. In: *Security in Machine Learning and its Applications (SiMLA 2022)*. 2022. https://doi.org/10.1007/978-3-031-16815-4_31

8.3 Software developed during PhD

- IBIR*:* a java fault injection tool using bug reports and automated program repair (APR) patterns. https://github.com/serval-uni-lu/IBIR.git
- CODEBERT-NT*:* a tool to measure code-naturlaness using CodeBERT predictions. https://github.com/Ahmedfir/CodeBERT-nt.git
- μBERT*:* a tool to inject faults using CodeBERT predictions.

  https://github.com/Ahmedfir/mBERTa.git
- Mutation testing developer simulation for effectiveness and cost-efficiency comparison. https://github.com/Ahmedfir/mu-FD-simulation.git

References

[1] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, “Fault injection techniques and tools,” *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[2] R. Natella, D. Cotroneo, and H. S. Madeira, “Assessing dependability with software fault injection: A survey,” *ACM Comput. Surv.*, vol. 48, no. 3, Feb. 2016, ISSN: 0360-0300. DOI: 10.1145/2841425. [Online]. Available: https://doi.org/10.1145/2841425.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008, ISBN: 978-0-521-88038-1. DOI: 10.1017/CBO9780511809163. [Online]. Available: https://doi.org/10.1017/CBO9780511809163.

[4] G. Fraser and J. M. Rojas, “Software testing,” *Handbook of Software Engineering*, pp. 123–192, 2019.

[5] H. Zhu, P. A. V. Hall, and J. H. R. May, “Software unit test coverage and adequacy,” *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997. DOI: 10.1145/267580.267590. [Online]. Available: https://doi.org/10.1145/267580.267590.

[6] 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*, 2017, pp. 597–608. DOI:

10.1109/ICSE.2017.61. [Online]. Available: https://doi.org/10.1109/ICSE.2017.61.

[7] 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*, 2018, pp. 537–548. DOI: 10.1145/3180155.3180183. [Online]. Available: https://doi.org/10.1145/3180155.3180183.

[8] 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, 19:1–19:23, 2021. DOI: 10.1145/3425497. [Online]. Available: https://doi.org/10.1145/3425497.

[9] 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. DOI: 10.1016/bs.adcom.2018.03.015. [Online]. Available: https://doi.org/10.1016/bs.adcom.2018.03.015.

[10] 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*, 2016, pp. 354–365. DOI: 10.1145/2931037.2931040. [Online]. Available: https://doi.org/10.1145/2931037.2931040.

[11] 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. DOI: 10.1016/j.jss.2009.06.031. [Online]. Available: https://doi.org/10.1016/j.jss.2009.06.031.

[12] J. Možucha and B. Rossi, “Is mutation testing ready to be adopted industry-wide?,” Nov. 2016, pp. 217–232, ISBN: 978-3-319-49093-9. DOI: 10.1007/978-3-319-49094-6_14.

[13] A. J. Offutt and R. H. Untch, “Mutation 2000: Uniting the orthogonal,” English, in *Mutation Testing for the New Century*, vol. 24, 2001, pp. 34–44, ISBN: 978-1-4419-4888-5. DOI: 10.1007/978-1-4757-5939-6_7.

[14] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, “A systematic literature review of techniques and metrics to reduce the cost of mutation testing,” *Journal of Systems and Software*, vol. 157, p. 110 388, 2019, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.07.100. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219301554.

[15] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, “Predictive mutation testing,” *IEEE Trans. Software Eng.*, vol. 45, no. 9, pp. 898–918, 2019. DOI: 10.1109/TSE.2018.2809496. [Online]. Available: https://doi.org/10.1109/TSE.2018.2809496.

[16] M. Beller, C.-P. Wong, J. Bader, *et al.*, *What it would take to use mutation testing in industry–a study at facebook*, 2021. arXiv: 2010.13464 [cs.SE].

[17] 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. DOI: 10.1016/j.scico.2014.05.012. [Online]. Available: https://doi.org/10.1016/j.scico.2014.05.012.

[18] 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., IEEE Computer Society, 2015, pp. 936–946.

DOI: 10.1109/ICSE.2015.103. [Online]. Available: https://doi.org/10.1109/ICSE.2015.103.

[19] D. Schuler and A. Zeller, “Covering and uncovering equivalent mutants,” *Softw. Test. Verification Reliab.*, vol. 23, no. 5, pp. 353–374, 2013. DOI: 10.1002/stvr.1473. [Online]. Available: https://doi.org/10.1002/stvr.1473.

[20] A. J. Offutt, “Investigations of the software testing coupling effect,” *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 1, pp. 5–20, 1992. DOI: 10.1145/125489.125473. [Online]. Available: https://doi.org/10.1145/125489.125473.

[21] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, “An experimental determination of sufficient mutant operators,” *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, 1996. DOI: 10.1145/227607.227610. [Online]. Available: https://doi.org/10.1145/227607.227610.

[22] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. L. Traon, “How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults,” *Empir. Softw. Eng.*, vol. 23, no. 4, pp. 2426–2463, 2018. DOI: 10.1007/s10664-017-9582-5. [Online]. Available: https://doi.org/10.1007/s10664-017-9582-5.

[23] 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., IEEE Computer Society, 2010, pp. 300–309. DOI: 10.1109/APSEC.2010.42. [Online]. Available: https://doi.org/10.1109/APSEC.2010.42.

[24] M. Fowler, *Continuous integration*, https://martinfowler.com/articles/continuousIntegration.html, Online; accessed 10 February 2020.

[25] W. Ma, T. T. Chekam, M. Papadakis, and M. Harman, "Mudelta: Delta-oriented mutation testing at commit time," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, IEEE, 2021, pp. 897–909. DOI: 10.1109/ICSE43902.2021.00086. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00086.

[26] W. Ma, T. Laurent, M. Ojdanic, T. T. Chekam, A. Ventresque, and M. Papadakis, "Commit-aware mutation testing," in *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution, ICSME*, 2020.

[27] M. Tufano, C. Watson, G. Bavota, M. D. 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 2019*, IEEE, 2019, pp. 301–312. DOI: 10.1109/ICSME.2019.00046. [Online]. Available: https://doi.org/10.1109/ICSME.2019.00046.

[28] 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. DOI: 10.1109/TSE.2011.124. [Online]. Available: https://doi.org/10.1109/TSE.2011.124.

[29] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978. DOI: 10.1109/C-M.1978.218136. [Online]. Available: https://doi.org/10.1109/C-M.1978.218136.

[30] D. B. Brown, M. Vaughn, B. Liblit, and T. W. 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*, ACM, 2017, pp. 511–522. DOI: 10.1145/3106237.3106280. [Online]. Available: https://doi.org/10.1145/3106237.3106280.

[31] M. Jimenez, T. T. Chekam, M. Cordy, *et al.*, “Are mutants really natural?: A study on how ”naturalness” helps mutant selection,” in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, M. Oivo, D. M. Fernández, and A. Mockus, Eds., ACM, 2018, 3:1–3:10. DOI: 10.1145/3239235.3240500. [Online]. Available: https://doi.org/10.1145/3239235.3240500.

[32] M. Tufano, J. Kimko, S. Wang, *et al.*, “Deepmutation: A neural mutation tool,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 29–32.

[33] J. Patra and M. Pradel, “Semantic bug seeding: A learning-based approach for creating realistic bugs,” ser. ESEC/FSE 2021, Athens, Greece: Association for Computing Machinery, 2021, pp. 906–918, ISBN: 9781450385626. DOI: 10.1145/3468264.3468623. [Online]. Available: https://doi.org/10.1145/3468264.3468623.

[34] Z. Tian, J. Chen, Q. Zhu, J. Yang, and L. Zhang, “Learning to construct better mutation faults,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22, Rochester, MI, USA: Association for Computing Machinery, 2023, ISBN: 9781450394758. DOI: 10.1145/3551349.3556949. [Online]. Available: https://doi.org/10.1145/3551349.3556949.

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

[36] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[37] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the naturalness of software,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE ’12, Zurich, Switzerland: IEEE Press, 2012, pp. 837–847, ISBN: 9781467310673.

[38] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, “On the ”naturalness” of buggy code,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 428–439. DOI: 10.1145/2884781.2884848.

[39] M. Jimenez, C. Maxime, Y. Le Traon, and M. Papadakis, “On the impact of tokenizer and parameters on n-gram based code analysis,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 437–448.

[40] 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 *Proceedings of the 2012 International Conference on Software Engineering (ICSE)*, 2012, pp. 14–24.

[41] 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.

[42] A. Koyuncu, K. Liu, T. F. Bissyandé, *et al.*, “iFixR: Bug report driven program repair,” in *Proceedings of the 13th Joint Meeting on Foundations of Software Engineering (FSE)*, 2019.

[43] A. Khanfir, A. Koyuncu, M. Papadakis, *et al.*, “Ibir: Bug report driven fault injection,” *ACM Trans. Softw. Eng. Methodol.*, May 2022, ISSN: 1049-331X. DOI: 10.1145/3542946. [Online]. Available: https://doi.org/10.1145/3542946.

[44] A. Khanfir, "Effective and scalable fault injection using bug reports and generative language models," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, Singapore, Singapore: Association for Computing Machinery, 2022, pp. 1790–1794, ISBN: 9781450394130. DOI: 10.1145/3540250.3558907. [Online]. Available: https://doi.org/10.1145/3540250.3558907.

[45] R. Degiovanni and M. Papadakis, "μbert: Mutation testing using pre-trained language models," in *15th IEEE International Conference on Software Testing, Verification and Validation Workshops ICST Workshops 2022, Valencia, Spain, April 4-13, 2022*, IEEE, 2022, pp. 160–169. DOI: 10.1109/ICSTW55395.2022.00039. [Online]. Available: https://doi.org/10.1109/ICSTW55395.2022.00039.

[46] A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Efficient mutation testing via pre-trained language models," *CoRR*, vol. abs/2301.03543, 2023. DOI: 10.48550/arXiv.2301.03543. arXiv: 2301.03543. [Online]. Available: https://doi.org/10.48550/arXiv.2301.03543.

[47] A. Khanfir, M. Jimenez, M. Papadakis, and Y. L. Traon, "Codebert-nt: Code naturalness via codebert," *22nd IEEE International Conference on Software Quality, Reliability and Security (QRS'22)*, 2022.

[48] *Codebert*, https://github.com/microsoft/CodeBERT.

[49] 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. DOI: 10.1002/stvr.1529. [Online]. Available: https://doi.org/10.1002/stvr.1529.

[50] 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*, IEEE Com-

puter Society, 2018, pp. 32–39. DOI: 10.1109/ICSTW.2018.00025. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICSTW.2018.00025.

[51] 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*, IEEE Computer Society, 2010, pp. 121–130. DOI: 10.1109/ISSRE.2010.38. [Online]. Available: https://doi.org/10.1109/ISSRE.2010.38.

[52] P. Ammann, “System testing via mutation analysis of model checking specifications,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 25, no. 1, p. 33, 2000. DOI: 10.1145/340855.340862. [Online]. Available: https://doi.org/10.1145/340855.340862.

[53] W. Krenn and B. K. Aichernig, “Test case generation by contract mutation in spec#,” *Electron. Notes Theor. Comput. Sci.*, vol. 253, no. 2, pp. 71–86, 2009. DOI: 10.1016/j.entcs.2009.09.052. [Online]. Available: https://doi.org/10.1016/j.entcs.2009.09.052.

[54] 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*, IEEE Computer Society, 2014, pp. 1–10. DOI: 10.1109/ICST.2014.11. [Online]. Available: https://doi.org/10.1109/ICST.2014.11.

[55] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, and P. Heymans, “Featured model-based mutation analysis,” in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds., ACM,

2016, pp. 655–666. DOI: 10.1145/2884781.2884821. [Online]. Available: https://doi.org/10.1145/2884781.2884821.

[56] B. K. Aichernig, E. Jöbstl, and S. Tiran, “Model-based mutation testing via symbolic refinement checking,” *Sci. Comput. Program.*, vol. 97, pp. 383–404, 2015. DOI: 10.1016/j.scico.2014.05.004. [Online]. Available: https://doi.org/10.1016/j.scico.2014.05.004.

[57] 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*, IEEE Computer Society, 1996, pp. 304–313. DOI: 10.1109/FTCS.1996.534615. [Online]. Available: https://doi.org/10.1109/FTCS.1996.534615.

[58] J. M. Voas, F. Charron, G. McGraw, K. W. Miller, and M. Friedman, “Predicting how badly ”good” software can behave,” *IEEE Softw.*, vol. 14, no. 4, pp. 73–83, 1997. DOI: 10.1109/52.595959. [Online]. Available: https://doi.org/10.1109/52.595959.

[59] 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. DOI: 10.1109/12.238482. [Online]. Available: https://doi.org/10.1109/12.238482.

[60] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[61] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, “Spectrum-based multiple fault localization,” in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2009, pp. 88–99.

[62] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," en, *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, Sep. 1990, ISSN: 1097-4571.

[63] W. B. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms*, 1st ed. Prentice Hall, Jun. 1992, ISBN: 0-13-463837-9.

[64] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*, English, 1 edition. Cambridge, Mass: The MIT Press, Jun. 1999, ISBN: 978-0-262-13360-9.

[65] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, Inc., 1986, ISBN: 978-0-07-054484-0.

[66] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent Dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[67] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 345–355.

[68] S. Wang and D. Lo, "Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, 2014, pp. 53–63.

[69] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 262–273.

[70] K. C. Youm, J. Ahn, J. Kim, and E. Lee, "Bug Localization Based on Code Change Histories and Bug Reports," in *Proceedings of the 2015 Asia-Pacific Software Engineering Conference (ICSE)*, 2015, pp. 190–197.

[71] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019. DOI: 10.1145/3318162. [Online]. Available: https://doi.org/10.1145/3318162.

[72] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 35th ICSE*, IEEE, 2013, pp. 802–811.

[73] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2018.

[74] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 12–23.

[75] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 648–659.

[76] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, "Dynamic patch generation for null pointer exceptions using metaprogramming," in *Proceedings of the 24th SANER*, IEEE, 2017, pp. 349–358.

[77] A. Koyuncu, K. Liu, T. F. Bissyandé, *et al.*, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1980–2024, 2020.

[78] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Proceedings of the 10th SSBSE*, Springer, 2018, pp. 65–86.

[79] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, “Avatar: Fixing semantic bugs with fix patterns of static analysis violations,” in *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 1–12.

[80] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, “TBar: Revisiting template-based automated program repair,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019, pp. 31–42.

[81] *Github copilot*, `https://github.com/features/copilot`.

[82] M. Chen, J. Tworek, H. Jun, *et al.*, “Evaluating large language models trained on code.(2021),” *arXiv preprint arXiv:2107.03374*, 2021.

[83] *Amazon codewhisperer*, `https://aws.amazon.com/codewhisperer/`.

[84] Z. Feng, D. Guo, D. Tang, *et al.*, “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., ser. Findings of ACL, vol. EMNLP 2020, Association for Computational Linguistics, 2020, pp. 1536–1547. DOI: `10.18653/v1/2020.findings-emnlp.139`. [Online]. Available: `https://doi.org/10.18653/v1/2020.findings-emnlp.139`.

[85] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[86] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.

[87] J. M. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs against Errors*. USA: John Wiley & Sons, Inc., 1997, ISBN: 0471183814.

[88] 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., ACM, 2018, pp. 456–467. DOI: 10.1145/3180155.3180191. [Online]. Available: https://doi.org/10.1145/3180155.3180191.

[89] 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. DOI: 10.1002/stvr.308. [Online]. Available: https://doi.org/10.1002/stvr.308.

[90] 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*, 2016, pp. 449–452. DOI: 10.1145/2931037.2948707. [Online]. Available: https://doi.org/10.1145/2931037.2948707.

[91] T. Loise, X. Devroey, G. Perrouin, M. Papadakis, and P. Heymans, “Towards security-aware mutation testing,” in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 97–102. DOI: 10.1109/ICSTW.2017.24.

[92] K. Herzig and A. Zeller, “Untangling changes,” *Unpublished manuscript, September*, vol. 37, pp. 38–40, 2011.

[93] C. Richter and H. Wehrheim, “Learning realistic mutations: Bug creation for neural bug detectors,” in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 162–173. DOI: 10.1109/ICST53961.2022.00027.

[94] J. Andrews, L. Briand, Y. Labiche, and A. Namin, “Using mutation analysis for assessing and comparing testing coverage criteria,” *Software Engineering,*

IEEE Transactions on, vol. 32, no. 8, pp. 608–624, 2006, ISSN: 0098-5589. DOI: 10.1109/TSE.2006.83.

[95] 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. DOI: 10.1007/s10664-019-09778-7. [Online]. Available: https://doi.org/10.1007/s10664-019-09778-7.

[96] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Guided mutation testing for javascript web applications," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 429–444, 2015. DOI: 10.1109/TSE.2014.2371458. [Online]. Available: https://doi.org/10.1109/TSE.2014.2371458.

[97] C. Sun, F. Xue, H. Liu, and X. Zhang, "A path-aware approach to mutant reduction in mutation testing," *Information & Software Technology*, vol. 81, pp. 65–81, 2017. DOI: 10.1016/j.infsof.2016.02.006. [Online]. Available: https://doi.org/10.1016/j.infsof.2016.02.006.

[98] D. Gong, G. Zhang, X. Yao, and F. Meng, "Mutant reduction based on dominance relation for weak mutation testing," *Information & Software Technology*, vol. 81, pp. 82–96, 2017. DOI: 10.1016/j.infsof.2016.05.001. [Online]. Available: https://doi.org/10.1016/j.infsof.2016.05.001.

[99] M. Ojdanic, W. Ma, T. Laurent, T. T. Chekam, A. Ventresque, and M. Papadakis, "On the use of commit-relevant mutants," *Empir. Softw. Eng.*, vol. 27, no. 5, p. 114, 2022. DOI: 10.1007/s10664-022-10138-1. [Online]. Available: https://doi.org/10.1007/s10664-022-10138-1.

[100] 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*,

2016, pp. 571–582. DOI: 10.1145/2950290.2950322. [Online]. Available: https://doi.org/10.1145/2950290.2950322.

[101] A. Garg, M. Ojdanic, R. Degiovanni, T. T. Chekam, M. Papadakis, and Y. L. Traon, “Cerebro: Static subsuming mutant selection,” *IEEE Trans. Software Eng.*, DOI: 10.1109/TSE.2022.3140510.

[102] C. B. Junior, V. H. S. Durelli, R. S. Durelli, S. R. S. Souza, A. M. R. Vincenzi, and M. E. Delamaro, “A preliminary investigation into using machine learning algorithms to identify minimal and equivalent mutants,” in *13th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW*, IEEE, 2020, pp. 304–313. DOI: 10.1109/ICSTW50294.2020.00056. [Online]. Available: https://doi.org/10.1109/ICSTW50294.2020.00056.

[103] R. Gheyi, M. Ribeiro, B. Souza, *et al.*, “Identifying method-level mutation subsumption relations using Z3,” *Inf. Softw. Technol.*, vol. 132, p. 106 496, 2021. DOI: 10.1016/j.infsof.2020.106496. [Online]. Available: https://doi.org/10.1016/j.infsof.2020.106496.

[104] J. M. Zhang, L. Zhang, D. Hao, L. Zhang, and M. Harman, “An empirical comparison of mutant selection assessment metrics,” in *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2019, pp. 90–101. DOI: 10.1109/ICSTW.2019.00037.

[105] S.-W. Kim, Y.-S. Ma, and Y.-R. Kwon, “Combining weak and strong mutation for a noninterpretive java mutation system,” *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 647–668, 2013.

[106] S. Vercammen, M. Ghafari, S. Demeyer, and M. Borg, “Goal-oriented mutation testing with focal methods,” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018, Lake Buena Vista, FL, USA: Association for Computing

Machinery, 2018, pp. 23–30, ISBN: 9781450360531. DOI: 10.1145/3278186.3278190. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3278186.3278190.

[107] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, “Faster mutation analysis via equivalence modulo states,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, Santa Barbara, CA, USA: Association for Computing Machinery, 2017, pp. 295–306, ISBN: 9781450350761. DOI: 10.1145/3092703.3092714. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3092703.3092714.

[108] R. H. Untch, A. J. Offutt, and M. J. Harrold, “Mutation analysis using mutant schemata,” *SIGSOFT Softw. Eng. Notes*, vol. 18, no. 3, pp. 139–148, Jul. 1993, ISSN: 0163-5948. DOI: 10.1145/174146.154265. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/174146.154265.

[109] 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, 2014*, 2014, pp. 654–665. DOI: 10.1145/2635868.2635929. [Online]. Available: https://doi.org/10.1145/2635868.2635929.

[110] 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. DOI: 10.1109/TSE.2006.83. [Online]. Available: https://doi.org/10.1109/TSE.2006.83.

[111] P. G. Frankl, S. N. Weiss, and C. Hu, “All-uses vs mutation testing: An experimental comparison of effectiveness,” *J. Syst. Softw.*, vol. 38, no. 3, pp. 235–253, 1997. DOI: 10.1016/S0164-1212(96)00154-9. [Online]. Available: https://doi.org/10.1016/S0164-1212(96)00154-9.

[112] 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 *Second International Conference on Software Testing Verification and Validation, ICST, 2009, Workshops Proceedings*, IEEE Computer Society, 2009, pp. 220–229. DOI: 10.1109/ICSTW.2009.30. [Online]. Available: https://doi.org/10.1109/ICSTW.2009.30.

[113] C. E. Shannon, “Prediction and entropy of printed english,” *The Bell System Technical Journal*, vol. 30, no. 1, pp. 50–64, 1951. DOI: 10.1002/j.1538-7305.1951.tb01366.x.

[114] C. E. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[115] M. Allamanis and C. Sutton, “Mining source code repositories at massive scale using language modeling,” in *2013 10th working conference on mining software repositories (MSR)*, IEEE, 2013, pp. 207–216.

[116] S. F. Chen and J. Goodman, “An empirical study of smoothing techniques for language modeling,” *Computer Speech & Language*, vol. 13, no. 4, pp. 359–394, 1999, ISSN: 0885-2308. DOI: https://doi.org/10.1006/csla.1999.0128. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0885230899901286.

[117] R. Kneser and H. Ney, “Improved backing-off for m-gram language modeling,” in *1995 International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, 1995, 181–184 vol.1. DOI: 10.1109/ICASSP.1995.479394.

[118] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, “Error models for the representative injection of software defects,” in *Software Engineering & Management 2015, Multikonferenz der GI-Fachbereiche Softwaretechnik (SWT)*

und Wirtschaftsinformatik (WI), FA WI-MAW, 17. März - 20. März 2015, ser. LNI, vol. P-239, GI, 2015, pp. 118–119.

[119] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?” In *Proceedings of the 20th ISSTA*, ACM, 2011, pp. 199–209.

[120] Q. Wang, C. Parnin, and A. Orso, “Evaluating the usefulness of ir-based fault localization techniques,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 1–11.

[121] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, “Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis,” in *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 181–190.

[122] C. Fellbaum, *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[123] 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*, ser. SAC ’04, New York, NY, USA: Association for Computing Machinery, 2004, pp. 1232–1237, ISBN: 1581138121. DOI: 10.1145/967900.968151. [Online]. Available: https://doi.org/10.1145/967900.968151.

[124] M. T. Maybury, “Karen spärck jones and summarization,” in *Charting a New Course: Natural Language Processing and Information Retrieval*, Springer, 2005, pp. 99–103.

[125] A. Koyuncu, T. F. Bissyandé, D. Kim, *et al.*, “D&c: A divide-and-conquer approach to ir-based bug localization,” *arXiv preprint arXiv:1902.02703*, 2019.

[126] A. Khanfir, A. Koyuncu, M. Papadakis, *et al.*, *Ibir*, 2022. [Online]. Available: https://github.com/serval-uni-lu/IBIR.

[127] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, “Context-aware patch generation for better automated program repair,” in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, ACM, 2018, pp. 1–11. DOI: 10.1145/3180155.3180233. [Online]. Available: https://doi.org/10.1145/3180155.3180233.

[128] S. Moon, Y. Kim, M. Kim, and S. Yoo, “Ask the mutants: Mutating faulty programs for fault localization,” in *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014*, IEEE Computer Society, 2014, pp. 153–162. DOI: 10.1109/ICST.2014.28. [Online]. Available: https://doi.org/10.1109/ICST.2014.28.

[129] G. Fraser and A. Arcuri, “Whole test suite generation,” *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013. DOI: 10.1109/TSE.2012.14. [Online]. Available: https://doi.org/10.1109/TSE.2012.14.

[130] 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, pp. 437–440.

[131] M. Fischer, M. Pinzger, and H. C. Gall, “Populating a release history database from version control and bug tracking systems,” in *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 2003*, IEEE Computer Society, 2003, p. 23. DOI: 10.1109/ICSM.2003.1235403. [Online]. Available: https://doi.org/10.1109/ICSM.2003.1235403.

[132] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan, “The impact of classifier configuration and classifier combination on bug localization,” *IEEE Trans. Software Eng.*, vol. 39, no. 10, pp. 1427–1443, 2013. DOI: 10.1109/TSE.2013.27. [Online]. Available: https://doi.org/10.1109/TSE.2013.27.

[133] 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. DOI: 10.3102/10769986025002101.

[134] K. N. King and A. J. Offutt, “A fortran language system for mutation-based software testing,” *Softw., Pract. Exper.*, vol. 21, no. 7, pp. 685–718, 1991.

[135] H. Agrawal, R. A. DeMillo, B. Hathaway, *et al.*, “Design of mutant operators for the c programming language,” Purdue University, West Lafayette, Indiana, Tech. Rep. SERC-TR-41-P, Mar. 1989.

[136] A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon, “The class-level mutants of mujava,” in *Proceedings of the International Workshop on Automation of Software Test (AST’06)*, Shanghai, China, May 2006, pp. 78–84.

[137] A. J. Offutt, G. Rothermel, and C. Zapf, “An experimental evaluation of selective mutation,” in *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993*, 1993, pp. 100–107. [Online]. Available: http://portal.acm.org/citation.cfm?id=257572.257597.

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

[139] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, “Interface mutation: An approach for integration testing,” *IEEE Trans. Software Eng.*, vol. 27, no. 3, pp. 228–247, 2001. DOI: 10.1109/32.910859. [Online]. Available: https://doi.org/10.1109/32.910859.

[140] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: Ten years later," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 305–318.

[141] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider, "An empirical study of the impacts of clones in software maintenance," in *The 19th IEEE International Conference on Program Comprehension, ICPC 2011*, IEEE Computer Society, 2011, pp. 242–245. DOI: `10.1109/ICPC.2011.14`. [Online]. Available: `https://doi.org/10.1109/ICPC.2011.14%5C%5C`.

[142] M. Beller, C. Wong, J. Bader, *et al.*, "What it would take to use mutation testing in industry - A study at facebook," in *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP)*, IEEE, 2021, pp. 268–277. DOI: `10.1109/ICSE-SEIP52600.2021.00036`. [Online]. Available: `https://doi.org/10.1109/ICSE-SEIP52600.2021.00036`.

[143] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 437–440, ISBN: 9781450326452. DOI: `10.1145/2610384.2628055`. [Online]. Available: `https://doi.org/10.1145/2610384.2628055`.

[144] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Mutation testing in the wild: Findings from github," *Empir. Softw. Eng.*, vol. 27, no. 6, p. 132, 2022. DOI: `10.1007/s10664-022-10177-8`. [Online]. Available: `https://doi.org/10.1007/s10664-022-10177-8`.

[145] *Pitest*, `http://pitest.org/`.

[146] *Pitest-rv-plugin*, `https://github.com/pitest/pitest-rv-plugin`.

[147] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, “Spoon: A Library for Implementing Analyses and Transformations of Java Source Code,” *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. DOI: 10.1002/spe.2346. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document.

[148] J. Eclipse, *Eclipse java development tools (jdt)*, 2013.

[149] *Mberta*, https://github.com/Ahmedfir/mBERTa.

[150] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, “Code smell detection by deep direct-learning and transfer-learning,” *Journal of Systems and Software*, vol. 176, p. 110 936, 2021, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2021.110936. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221000339.

[151] B. Lin, C. Nagy, G. Bavota, and M. Lanza, “On the impact of refactoring operations on code naturalness,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 594–598. DOI: 10.1109/SANER.2019.8667992.

[152] D. Posnett, A. Hindle, and P. Devanbu, “Reflections on: A simpler model of software readability,” *SIGSOFT Softw. Eng. Notes*, vol. 46, no. 3, pp. 30–32, Jul. 2021, ISSN: 0163-5948. DOI: 10.1145/3468744.3468754. [Online]. Available: https://doi.org/10.1145/3468744.3468754.

[153] V. J. Hellendoorn, P. T. Devanbu, and A. Bacchelli, “Will they like this? evaluating code contributions with language models,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 157–167. DOI: 10.1109/MSR.2015.22.

[154] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, “Bugram: Bug detection with n-gram language models,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, Singapore, Singapore: Association for Computing Machinery, 2016, pp. 708–719, ISBN: 9781450338455. DOI: 10.1145/2970276.2970341. [Online]. Available: https://doi.org/10.1145/2970276.2970341.

[155] S. H. Alexander Trautsch Fabian Trautsch, *The smartshark repository mining data*, 2021. arXiv: 2102.11540.

[156] J. Kim, J. Jeon, S. Hong, and S. Yoo, “Predictive mutation analysis via natural language channel in source code,” *CoRR*, vol. abs/2104.10865, 2021. [Online]. Available: https://arxiv.org/abs/2104.10865.

[157] S. Kang and S. Yoo, “Language models can prioritize patches for practical program patching,” in *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022*, IEEE, 2022, pp. 8–15. DOI: 10.1145/3524459.3527343. [Online]. Available: https://doi.org/10.1145/3524459.3527343.

[158] Z. Sun, J. M. Zhang, Y. Xiong, M. Harman, M. Papadakis, and L. Zhang, “Improving machine translation systems via isotopic replacement,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1181–1192.

[159] *Pytorch*, https://pytorch.org/.

[160] M. K. Thota, F. H. Shajin, and P. Rajesh, “Survey on software defect prediction techniques,” *International Journal of Applied Science and Engineering*, vol. 17, pp. 331–344, 4 Dec. 2020, ISSN: 1727-7841. DOI: 10.6703/IJASE.202012_17(4).331.

[161] M. Rahman, D. Palani, and P. C. Rigby, "Natural software revisited," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 37–48. DOI: 10.1109/ICSE.2019.00022.

[162] M. Jimenez, C. Maxime, Y. Le Traon, and M. Papadakis, "Tuna: Tuning naturalness-based analysis," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 715–715. DOI: 10.1109/ICSME.2018.00087.

[163] M. Leszak, D. E. Perry, and D. Stoll, "Classification and evaluation of defects in a project retrospective," *Journal of Systems and Software*, vol. 61, no. 3, pp. 173–187, 2002, ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(01)00146-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121201001467.

[164] 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*, ser. ISSTA 2016, Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 449–452, ISBN: 9781450343909. DOI: 10.1145/2931037.2948707. [Online]. Available: https://doi.org/10.1145/2931037.2948707.

[165] M. Ojdanic, A. Khanfir, A. Garg, R. Degiovanni, M. Papadakis, and Y. L. Traon, "On comparing mutation testing tools through learning-based mutant selection," *4th ACM/IEEE International Conference on Automation of Software Test (AST 2023)*, 2023.

[166] W. Bonnaventure, A. Khanfir, A. Bartel, M. Papadakis, and Y. Le Traon, "Confuzzion: A java virtual machine fuzzer for type confusion vulnerabilities," in *IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2021, pp. 586–597. DOI: 10.1109/QRS54544.2021.00069.

[167] B. PETIT, A. Khanfir, E. Soremekun, G. Perrouin, and M. Papadakis, “Intject: Vulnerability intent bug seeding,” in *22nd IEEE International Conference on Software Quality, Reliability, and Security*, 2022.

[168] B. Souani, A. Khanfir, A. Bartel, K. Allix, and Y. Le Traon, “Android malware detection using bert,” in *Applied Cryptography and Network Security Workshops: ACNS 2022 Satellite Workshops, AIBlock, AIHWS, AIoTS, CIMSS, Cloud S&P, SCI, SecMT, SiMLA, Rome, Italy, June 20–23, 2022, Proceedings*, 2022, pp. 575–591.