Analysis of the Effectiveness and Efficiency of a Java Mutation Testing Tool Based on Pre-Trained Language Models

Bachelor Thesis

Supervisor:
Prof. Dr. Gabriele Taentzer

Submitted by:
Khayitboev Khojiakbar

Marburg, 02.05.2025

Contents

List of Tables

Acronyms

AI Artificial Intelligence. 1, 12

API Application Programming Interface. 19–21

BERT Bidirectional Encoder Representations from Transformers. 6

CD Continuous Delivery. 25, 36

CI Continuous Integration. 25, 33, 36

CPU Central Processing Unit. 33

EMF Eclipse Modeling Framework. 21, 24

GPU Graphics Processing Unit. 33, 34, 37

I/O Input/Output. 19

IDE Integrated Development Environment. 25, 37

JUnit Java Unit Testing Framework. 10, 11, 23, 25

JVM Java Virtual Machine. 9, 19, 24

LOC Lines of Code. 7, 32, 33, 36

MMT Model-based Mutation Testing. 1–3, 10, 15, 16, 19–26, 28, 29, 35–37

NLP Natural Language Processing. 6

OCL Object Constraint Language. 21, 24

PITRV PIT Research Version. 11, 21

PLM Pre-trained Language Model. 1, 6, 12, 13, 15, 25, 30, 33–37

RAM Random-Access Memory. 30

RTX Ray Tracing Texel eXtreme. 30

Chapter 1

Introduction

1.1 The Imperative of Software Testing and Quality

The digital revolution has made interconnected computers and software systems an indispensable part of our personal and professional lives, enabling us to fulfill global demands for storing, retrieving, and processing information, knowledge, and wisdom. To meet these needs, these systems must be supported by high-quality software that functions correctly and reliably, is user-friendly, safe, and suitable for use, testing, reuse, and maintenance, and ultimately conforms to stakeholders' requirements. Consequently, software quality is not only one of the most crucial but also a multifaceted characteristic of computer software [26].

Software testing functions as the essential foundation for quality assurance because it serves as the main method to identify defects before software deployment. However, simply executing tests is insufficient. The main difficulty emerges from determining whether a test suite is adequate for detecting software faults. A test suite which runs extensive code sections without testing it in ways that expose hidden defects creates an illusion of safety. The evaluation of test suite quality demands strict methods because of their necessity.

1.2 Mutation Testing: A Rigorous Approach to Test Adequacy

Traditional testing metrics, such as line and branch coverage, have long been the standard for assessing test effectiveness. However, these methods repeatedly fall short in finding subtle errors that can substantially influence software quality. Mutation testing appears as a favorable alternative [8].

Mutation testing process involves systematically introducing small, syntactically plausible changes – known as mutations – into the source code (or bytecode) of the program under test, creating variants called mutants . Each mutant represents a potential fault. The existing test suite is then executed against each mutant. If a test case causes a mutant to produce different output than the original program, the mutant is considered killed. If no test case distinguishes the mutant from the original program, the mutant survives. The ratio of killed mutants to the total number of non-equivalent mutants yields the mutation score, a quantitative measure of the test suite's fault-detection capability. A high mutation score indicates a thorough test suite, sensitive to the types of faults simulated by the mutation operators [16, 22, 31, 32, 35].

1.3 Challenges in Traditional Mutation Testing

Despite its power, the practical adoption of mutation testing has historically faced significant challenges:

1. **Computational Cost** : The primary challenge is the potentially enormous computational expense. Generating mutants and, more significantly, executing the entire test suite against each one can be time-consuming and resource-intensive, especially for large, complex systems and extensive test suites [35, 36, 40, 51].

2. **Equivalent Mutants**: The manual identification of equivalent mutants which do not affect program behavior remains a time-consuming and error-prone process that increases both the cost and complexity of mutation testing [37].
3. **Mutant Realism**: Traditional mutation tools depend on predefined mutation operators which mostly focus on syntactic changes based on common programming errors or empirical studies [29]. The effectiveness of these artificially generated mutants remains uncertain because it is unclear how well they match actual faults that developers introduce during software development.

1.4 The Advent of Pre-trained Language Models in Software Engineering

The transformer architecture-based large-scale PLMs like BERT [17] have recently advanced the field of NLP. The models show impressive abilities to understand and produce human language through their ability to learn deep contextual patterns from large datasets. Researchers have successfully adapted PLMs for software engineering tasks because they recognized that source code contains both formal structure and statistical regularities and contextual dependencies. The field of automated software engineering tools has undergone a fundamental transformation because code-specific PLMs such as CodeBERT [18], GraphCodeBERT [20] and others now perform tasks including code completion, code summarization, vulnerability detection, automated program repair and type inference.

1.5 μBERT: Leveraging PLMs for Mutation Generation

The convergence of realistic mutation testing requirements with proven PLM capabilities drives researchers to develop new methods for mutant generation. The pioneering example of μBERT demonstrates its capabilities. The μBERT system uses CodeBERT as a pre-trained language model to create mutants dynamically through context-based generation instead of traditional fixed operator sets. The system uses token masking of identifiers, operators, literals, and method calls in source code to prompt the PLM for probable replacement predictions based on surrounding context [14].

The central hypothesis is that mutants generated via this data-driven, context-aware approach will be more "natural"—that is, they will more closely resemble the types of errors human developers are prone to introduce—than mutants created by applying predefined, often context-agnostic, syntactic transformation rules. This enhanced realism has the potential to make μBERT a more effective tool for evaluating the true fault-detection capabilities of test suites, especially concerning subtle or context-dependent bugs [14].

1.6 Research Gap and Problem Statement

The introduction of μBERT represents a fundamental change in how mutant generation operates. The new innovation demands extensive empirical research to determine its real-world effects. The attractive potential of generating more realistic mutants through PLMs creates efficiency concerns due to their computational intensity. The effectiveness of these "natural" mutants in challenging test suites and their correlation with actual software defects need systematic validation.

A critical gap exists in the comprehensive, comparative evaluation of μBERT against well-established traditional mutation testing tools. Understanding the trade-offs between mutant realism (effectiveness) and computational cost (efficiency) is essential for determining μBERT's practical applicability and guiding future research in PLM-based testing techniques. This thesis directly addresses this gap by providing a systematic analysis of μBERT's effectiveness and efficiency characteristics in the context of Java mutation testing.

1.7 Research Questions

This thesis seeks to answer the following primary research questions:

1. **RQ1:** What are the qualitative characteristics of the mutation generation process and the types of mutants produced by μBERT compared to traditional Java mutation testing tools?

2. **RQ2:** How effective is μBERT in generating mutants that challenge test suites and align with real software faults, compared quantitatively against traditional tools using established benchmarks?
3. **RQ3:** What is the quantitative efficiency profile of μBERT in terms of mutant generation and compilation time, and how does it correlate with source code metrics like Lines of Code (LOC) and Cyclomatic Complexity?

Chapter 2

Background

This chapter provides foundational background information on mutation testing. It defines the core concepts, outlines the process, discusses key elements like mutation operators and the mutation score, addresses the challenge of equivalent mutants, and briefly introduces the common levels at which mutation is applied and the types of traditional tools used, setting the stage for the detailed comparisons in later chapters.

2.1 Mutation Testing: Principles and Process

As introduced in Chapter 1, Mutation Testing is a fault-based testing technique used to evaluate and improve the quality of a software test suite [16, 22]. Its primary goal is not to find new bugs in the program directly, but rather to assess how effectively an existing set of tests can detect potential (simulated) bugs. The underlying assumption is that a test suite capable of detecting a wide range of small, simple faults (mutants) is also likely to be effective at detecting real, more complex faults.

The fundamental process typically involves the following steps:

1. **Mutant Generation**: Small, specific changes are systematically applied to the original program's code (either source or compiled bytecode) based on predefined rules called mutation operators. Each change results in a new version of the program, called a *mutant*.
2. **Test Execution**: The existing test suite is executed against each generated mutant.
3. **Outcome Determination**: For each mutant, the output is compared to the output of the original program when run with the same test cases.
    - If any test case causes the mutant to produce a different result (e.g., different output, crash, different state) than the original program, the mutant is considered *killed*. This indicates the test suite successfully detected the simulated fault.
    - If all test cases in the suite produce the same result for the mutant as for the original program, the mutant *survives*. This suggests a potential weakness in the test suite i.e. it failed to detect the specific fault represented by that mutant.
4. **Mutation Score Calculation**: The results are aggregated to calculate the mutation score, typically defined as:

    ```
    Mutation Score = (Number of Killed Mutants / Total Number of Mutants) * 100%
    ```

    A higher score suggests a more effective test suite.

2.2 Key Concepts in Mutation Testing

Several key concepts are central to understanding mutation testing:

- **Mutation Operators:** These are the rules used to create mutants. They are designed to mimic common programming errors or introduce simple syntactic changes. Examples include replacing

arithmetic operators (`+` to `-`), relational operators (`<` to `>=`), logical connectors (`&&` to `||`), deleting statements, or modifying literal values (e.g., 0 to 1). Traditional tools employ a predefined set of these operators.

- **Mutants:** These are the modified versions of the original program, each differing by typically one small change introduced by a mutation operator. The most common form is first-order mutation, where each mutant contains only a single fault.
- **Killed Mutant:** A mutant is killed if at least one test case detects the difference between the mutant's behavior/output and the original program's behavior/output.
- **Survived Mutant:** A mutant that is not killed by any test case in the suite. Surviving mutants highlight specific types of faults that the test suite is insensitive to, guiding developers on where to add or improve tests
- **Mutation Score:** The primary metric derived from mutation testing. While a high score is desirable, aiming for 100% can be impractical due to equivalent mutants. It serves as a relative measure of test suite adequacy.
- **Equivalent Mutants:** A significant challenge in mutation testing is the existence of equivalent mutants. These are mutants that, despite having a syntactic difference from the original program, are semantically identical – they compute the exact same function for all possible inputs. Such mutants can never be killed by any test case because they always produce the same output as the original program. They artificially deflate the mutation score if included in the denominator. Identifying equivalent mutants automatically is generally an undecidable problem, although various heuristics and techniques exist to approximate detection [33]. Manual identification is often required, adding to the cost.

2.3 Levels of Mutation Application

Mutation can be applied at different stages of the software representation:

- **Source Code Mutation:** Changes are applied directly to the program's source code (e.g., Java files). This approach makes the mutants easily understandable by developers, as they directly reflect changes in the familiar source language. The main drawback is that each mutant must typically be recompiled before test execution, which can be very time-consuming. μBERT operates at this level.
- **Bytecode Mutation:** Changes are applied to the compiled intermediate representation (e.g., Java bytecode). This approach is often significantly faster because it avoids the recompilation step for each mutant. Tools can directly manipulate the bytecode instructions loaded into the JVM. The trade-off is that the generated mutants may be harder to map back to the original source code for developer understanding, and the available mutations are constrained by the instruction set of the bytecode. Tools like PIT and Jumble primarily operate at this level.

2.4 Mutation Testing Tools

Various tools have been developed to automate the process of mutation testing, especially for popular programming languages like Java. The more commonly used tools include:

- PIT (Pitest): A widely used mutation testing tool for Java, known for its effectiveness in detecting faults. It is often compared with other tools like MuJava and Major for its fault-detection capabilities [24].
- Major: A Java mutation testing tool that is part of comparative studies to assess its strengths and weaknesses relative to other tools [24].
- LittleDarwin: Mutation testing framework specifically designed to handle large and complex Java software systems. [34]
- muPLSQL: A tool specifically designed for mutation testing in PL/SQL programs, facilitating automation in mutant generation and test execution [42]

- MMT: A tool designed to enhance mutation testing by allowing flexible definition of mutations through model transformation [9].
- Ajmutator, Advice Tracer, MuAspectJ, and Proteum/AJ: These tools are used in aspect-oriented programming environments, particularly with AspectJ, to enhance the feasibility of mutation testing in real software development processes [46].
- Jumble: Byte code level mutation testing tool for Java which inter-operates with JUnit [21].

Chapter 3

Related Work

3.1 Comparative Studies of Mutation Testing Tools

While much research has focused on developing novel mutation testing tools, several studies have taken a complementary path—systematically comparing existing tools to assess their relative strengths and weaknesses. These comparative evaluations play a crucial role in understanding which tools best suit specific testing needs, and where innovation is still required.

3.1.1 Traditional Tools: Effectiveness and Limitations

A foundational study by Kintis et al. [25] compares three widely-used Java mutation testing tools—MuJava, MAJOR, and PIT—through a detailed manual analysis of 3324 mutants. The results show significant variation in effectiveness: MuJava achieved the highest detection rate (88%), followed by MAJOR (80%) and PIT (76%). However, no tool was able to completely subsume the others. This finding underscores the diversity in implementation strategies and mutation operator sets. Additionally, the tools differed in terms of equivalent mutant generation and test effort: MuJava produced the most equivalent mutants (11%) and required 138 test cases, while PIT generated fewer equivalents (7%) and required only 80 test cases.

Another study [24] extends this comparison by introducing PITRV, a research version of PIT. PITRV outperformed the original PIT and all other tools in real-fault detection, revealing 6% more faults overall. This study, which analyzed over 6,000 mutants and numerous Defects4J faults, provided practical guidance on improving existing tools and emphasized the importance of disjoint mutants to avoid score inflation from redundant mutations.

Delahaye and du Bousquet [15] present a comparative study of mutation analysis tools available for Java. Their work stands out by introducing usage profiles (teaching, research, industry) as a key factor in selecting the most appropriate tool. They follow a structured methodology, starting with a systematic selection of eight contemporary Java mutation tools (including Bacterio, Javalanche, Jester, Judy, Jumble, MAJOR, MuJava/MuClipse, PIT). A crucial part of their study is an experimental evaluation where they attempt to apply the tools to a benchmark suite of Java programs with varying characteristics (e.g., Java version, size, JUnit version). They conclude that tool choice heavily depends on the context: PIT is highlighted for its usability and integration (suitable for teaching and industry automation), while tools like Judy and MAJOR offer more comprehensive and customizable fault models often preferred in research, despite potential usability hurdles.

Márki and Lindström [28] contribute a more nuanced perspective by comparing the mutation operators across MuJava, MAJOR, and PIT. Their analysis reveal substantial overlap but also significant divergence in operator semantics and implementation. They conduct cross-testing using a consistent test suite and observe that even small differences in mutator sets could lead to varied mutation scores and fault-detection outcomes. Moreover, their study include a usability-focused evaluation, comparing integration, observability, and configurability—factors that affect real-world adoption beyond raw mutation scores.

A qualitative comparison of traditional mutation tools with μBERT based on mutation operator support is presented in Section 4.2. This comparison integrates both findings from the literature [9] and empirical observations made during the evaluation of μBERT.

3.1.2 Modern Tools: AI-Based Approaches

A more recent empirical comparison evaluates the next generation of mutation testing tools that rely on machine learning and pre-trained models. One such study [30] benchmarks μBERT, IBIR, DeepMutation, and PIT across fault-detection effectiveness and cost-efficiency. Each tool represents a different mutation strategy: μBERT uses CodeBERT for masked token prediction, IBIR is based on inverted bug-fix patterns, DeepMutation applies neural translation, and PIT uses grammar-based mutation.

While IBIR had the highest fault detection overall, it was not the most cost-effective. μBERT, though slightly lower in raw detection, emerged as the most cost-efficient tool. Notably, when combined with deep learning-based mutant selection, μBERT improves its fault detection by 12%—surpassing the other tools. This highlights that selection strategy significantly influences the practical utility of mutation tools, and that LLM-based tools can outperform rule-based ones under optimized configurations.

3.2 Strategies for Improving Traditional Mutation Testing

Despite its theoretical strength, mutation testing faces several practical limitations that hinder its widespread adoption, especially in industry. To address issues such as computational cost, equivalent mutant handling, and the realism of generated mutants, numerous techniques have been proposed over the years.

3.2.1 Predictive and Intelligent Techniques

One line of research introduces predictive models to avoid executing all mutants. For example, the authors in [51] propose a classification-based approach that predicts whether a mutant will be killed, enabling speedups of over 150 times while maintaining high accuracy.

Similarly, genetic algorithms and optimization strategies have been employed to improve test data generation. A technique introduced in [12] uses a multi-population genetic algorithm to generate test cases more efficiently, reducing redundancy and execution time.

3.2.2 Clustering and Reduction Techniques

Reducing the number of mutants without sacrificing test effectiveness is another key strategy. A spectral clustering technique [49] identifies and selects representative mutants, lowering the execution cost while maintaining detection strength. Another method, ReMuSSE [6], leverages symbolic execution to eliminate up to 31.4% of redundant mutants, yielding time savings of over 35%.

3.2.3 Optimization and Search-Based Methods

Search-based mutation testing approaches have also shown promise. For example, a method presented in [13] uses multi-task optimization to target hard-to-kill mutants. Other approaches, like Markov chain-based test data generation [50], balance efficiency and fault detection by using statistical modeling.

Together, these efforts demonstrate that improving the scalability of mutation testing often involves either reducing the number of mutants or making test execution more intelligent and selective.

3.3 Pre-trained Language Models in Software Engineering

Pre-trained Language Models (PLM) have become a cornerstone in the field of software engineering, offering significant advancements in tasks such as code summarization, code clone detection, and software testing. These models, initially developed for natural language processing, are now being adapted to understand and process programming languages, providing new opportunities and challenges in software engineering.

PLMs like CodeBERT have been successfully adapted for software engineering tasks through techniques such as adapter-based knowledge transfer. This approach shows improvements in tasks like code clone detection and code summarization, often outperforming traditional fine-tuning methods while being more parameter-efficient [38, 39].

Large Language Models (LLMs) are increasingly used in software testing, particularly in tasks like test case preparation and program repair. These models offer innovative approaches to handle the growing complexity of software systems, although challenges remain in fully leveraging their capabilities [48].

3.3.1 Language Models for Mutation Testing

Mutation testing has traditionally relied on applying a fixed set of syntactic mutation operators—such as replacing arithmetic or logical operators, modifying constants, or altering branch conditions—to evaluate the fault detection capabilities of software test suites. Tools such as PIT and Major exemplify this rule-based approach for Java programs. However, these tools often fail to simulate more complex, real-world bugs and face scalability issues due to the explosion of generated mutants and the presence of equivalent mutants.

To overcome these limitations, recent research has turned to large language models (LLMs), which can capture rich semantic context and generate more natural, human-like code transformations. One of the earliest tools to explore this direction is μBERT, a Java mutation testing tool that utilizes CodeBERT, a transformer-based model pre-trained on source code. μBERT generates mutants by masking individual tokens in code expressions and predicting replacements using CodeBERT. This approach enables the generation of diverse, semantically plausible mutants without predefined mutation rules. In experiments on 40 real bugs from Defects4J, μBERT detected 27 faults—outperforming the traditional PIT baseline, which detected 26. Additionally, μBERT showed twice the cost-efficiency when analyzing the same number of mutants and demonstrated utility beyond fault detection by improving the results of assertion inference tools [14].

LLMs have also shown a higher fault detection rate compared to traditional methods. Studies indicate that LLMs can generate more diverse mutations that are behaviorally closer to real bugs, leading to a 17-19% increase in fault detection compared to existing approaches [11, 47].

Another complementary approach is proposed in LLMorpheus, a mutation testing tool for JavaScript and TypeScript that uses placeholder-based prompts to instruct LLMs (such as CodeLLaMA or GPT-4o) to inject buggy code. Instead of relying on masked tokens, LLMorpheus replaces code fragments with placeholders and uses LLM completions to generate mutations at the expression level. The results show that LLMorpheus produces mutants that more closely resemble real-world faults and outperform traditional tools like StrykerJS in diversity and fault simulation [44]. While μBERT focuses on token-level Java mutations, LLMorpheus explores prompt-based expression-level mutation in JavaScript, showcasing the versatility of LLM-based mutation generation across languages and input strategies.

In addition to mutant generation, recent work has investigated the application of LLMs to test generation, particularly for the purpose of killing mutants. A study combining scientific debugging and LLMs introduced an iterative process where LLMs form hypotheses about how to detect mutants, generate test cases, and refine them based on feedback. This method significantly outperformed both baseline prompting strategies and the search-based tool Pynguin in terms of mutation score, code coverage, and equivalent mutant identification [41]. These findings illustrate the potential of LLMs not only to simulate faults, but also to autonomously evolve high-quality tests that are tailored to specific mutants—an area μBERT could potentially benefit from in future extensions.

A persistent challenge in mutation testing is the detection of equivalent mutants, which waste computational resources and distort mutation scores. A recent empirical study evaluated LLMs on the task of equivalent mutant detection (EMD) using over 3,000 Java mutant pairs. LLM-based techniques, particularly those using fine-tuned embeddings, achieved a 35.7% improvement in F1-score over traditional EMD methods while maintaining lower training and inference overhead [43]. This positions LLMs as not only generators of meaningful mutants but also effective filters for uninformative or redundant ones. Integrating such techniques with tools like μBERT could help further improve precision and performance.

Taken together, these works reveal a growing consensus that LLMs are transforming the mutation testing landscape—from how mutants are generated, to how they are tested, and how their equivalence is assessed. While μBERT represents a pioneering effort in applying pre-trained models to Java mutant

generation, tools like LLMorpheus and techniques such as LLM-assisted test generation and mutant filtering provide promising directions for future work and integration.

Chapter 4

Mutation Generation and Execution (Qualitative Evaluation)

In this chapter, the mutation generation process of conventional mutation testing tools and the new μBERT-based mutation testing is described. The chapter begins with the description of the mutation generation pipeline utilized with μBERT, indicating how it utilizes pre-trained language models for the generation of mutants. The chapter then describes the mutation operators and techniques employed by classical tools like PIT, Jumble, and MMT. Lastly, a qualitative analysis is conducted to compare the characteristics and quality of the mutants generated by these methods.

4.1 Mutation Generation Process using μBERT

Overview

Traditional mutation testing tools often rely on predefined syntactic transformations, which can lead to artificial or semantically implausible code changes. μBERT offers an innovative approach by utilizing PLMs, specifically CodeBERT, to generate mutants that closely resemble real-world coding practices. This method aims to produce natural mutants, increasing the likelihood of representing actual faults and improving the effectiveness of mutation testing [14].

Mutation Generation Pipeline

μBERT implements an automated four-step pipeline to generate mutants:

1. **Parsing and Candidate Selection**

   μBERT begins by parsing the Java source code and identifying candidate expressions for mutation. These candidates include binary and unary expressions, assignment statements, literals, variable names, method calls, field accesses, arrays, and type references. For each candidate, μBERT selects one token to be mutated.

2. **Masking the Token and Invoking CodeBERT**

   Once a token is selected, μBERT masks it by replacing it with a special $\langle mask \rangle$ token. The surrounding code context (up to 512 tokens, CodeBERT's input limit) is fed into CodeBERT. The model predicts the most probable replacements for the masked token based on the surrounding context. An example of masking the token is as follows:

```
1     if (pos != -1) {
2         tokens.add(token.substring(pos + 1));
3     }
4
```

Example 4.1: Original Code

```
    if (pos != -1) {
        tokens.<mask>(token.substring(pos + 1));
    }

```

Example 4.2: Masked Code

For the masked token CodeBERT predictions might include methods such as `remove`, `push`, `delete`, `append` and `add`.

3. **Generating Mutants**

   For each predicted token, μBERT generates a mutant by substituting the masked token with the prediction. This process results in multiple first-order mutants for each candidate expression.

4. **Filtering Mutants**

   After generating the mutants, μBERT filters out all non-compilable and syntactically equivalent mutants.[14]

4.2 Mutation Operators Supported by μBERT

Unlike traditional mutation testing tools that are based on a predefined and bounded set of mutation operators, μBERT generates mutants by leveraging the contextual knowledge in a pre-trained language model. μBERT does not specify mutation operators, but uses CodeBERT to predict probable replacements for a specific token in the source code depending on the context where the tokens are used.

Mutations in μBERT can be categorized into familiar mutation operator types that are readily used in standard mutation testing. The following explains the types of mutation that μBERT is able to successfully implement, demonstrating its ability to emulate and add to the mutation operations that are typically found in tools such as PIT, Jumble, and MMT.

4.2.1 Array Access Mutator

μBERT is capable of mutating array read and write expressions.

```
    private static boolean containsBase64Byte(byte[] arrayOctet) {
        for (int i = 0; i < arrayOctet.length; i++) {
            if (isBase64(arrayOctet[<mask>])) {
                return true;
            }
        }
        return false;
    }

```

Example 4.3: Masked Code

Generated mutants include alterations to the masked token, such as: `0, I, j` and so on.

4.2.2 Assignment Mutator

The Assignment Mutator targets assignment expressions, introducing mutations by altering how values are assigned to variables or fields.

```
        pos <mask>= 0;

```

Example 4.4: Masked Code

CodeBERT predicts alternative assignment operators like `|=,*=, &=, /=, +=`.

4.2.3 Binary Operator Mutator

Binary operators perform operations that involve two operands. These operators include arithmetic operators (e.g. `*, +, -`), relational operators (e.g. `>, <=`), logical operators (e.g. `&&, ||`), and bitwise operators (e.g. `&, |`). The following example shows masking of a relational operator `!=`:

```
    boolean hasData() {
        return this.buffer <mask> null;
    }

```

Example 4.5: Masked Code

Generated tokens may include: `==, =, ===, <, >=`. Mutation of arithmetic and logical operators happens in a similar fashion.

4.2.4 Field Reference Mutator

μBERT, through its use of the CodeBERT, can perform field replacement by predicting alternative field names that fit the context in which they are used. The following example shows masking of a field `encodeTable`:

```
    public boolean isUrlSafe() {
        return this.<mask> == URL_SAFE_ENCODE_TABLE;
    }

```

Example 4.6: Masked Code

The predictions from CodeBert include tokens, like `table`, `enc`, `code`, `type` and `name`.

4.2.5 Identifier Mutators

μBERT supports multiple types of identifier mutators, targeting the replacement or modification of identifiers (such as variable names, literals, `this` access and conditional expressions) within Java source code. These mutators aim to simulate common developer mistakes, such as using incorrect variables, misreferencing `this`, or substituting identifiers in conditional expressions. Below is an overview of the different identifier mutator categories, that μBERT supports, along with descriptions and examples.

1. **Identifier-Mutator-Conditional**

   This mutation replaces conditional expressions.

```
        int avail() {
            return buffer != null ? pos - readPos : 0;
        }

```

Example 4.7: Original Code

```
        int avail() {
            return <mask>;
        }

```

Example 4.8: Masked Code

The predictions of CodeBERT may include tokens like `avail`, `0`, `1`, `available` and `capacity`

2. **Identifier-Mutator-Literal**

   The Identifier-Mutator-Literal mutation alters literal values in the source code. Literals can be numeric constants, boolean constants, string literals, or even null references.

```
        pos = 0;

```

Example 4.9: Original Code

```
        pos = <mask>;

```

Example 4.10: Masked Code

In this example, the numeric literal `0` is masked. CodeBERT might predict alternative literals such as `1,-1, 2` or other constants based on the context of the program.

3. **Identifier-Mutator-ThisAccess**

   The Identifier-Mutator-ThisAccess mutation focuses on explicit `this` references in the java code. This mutation can replace the referenced field or method with another, or substitute the entire object reference. This type of mutation simulates errors related to scope confusion or misuse of instance variables.

```
    public boolean isUrlSafe() {
     return this.encodeTable == URL_SAFE_ENCODE_TABLE;
    }

```

Example 4.11: Original Code

```
    public boolean isUrlSafe() {
     return <mask>.encodeTable == URL_SAFE_ENCODE_TABLE;
    }

```

Example 4.12: Masked Code

In this example, μBERT masks the `this` keyword and CodeBERT predict alternatives such as another object reference in scope. The resulting mutant may reference the wrong object or variable, leading to incorrect behavior.

4. **Identifier-Mutator-Variable**

   This mutation replaces variable identifiers within expressions or assignments. It imitates errors where developers mistakenly reference the wrong variable.

```
     int avail() {
        return buffer != null ? pos - readPos : 0;
    }

```

Example 4.13: Original Code

```
        int avail(){
         return buffer != null ? <mask> - readPos : 0;
        }

```

Example 4.14: Masked Code

Prediction from CodeBert may include following tokens: `buffer`, `position`, `limit` and `available`.

4.2.6 Method Call Mutator

The Method Call Mutator is a mutation type supported by μBERT that focuses on replacing method calls within the source code with contextually plausible alternatives.

```
    int len = Math.min(avail(), bAvail);

```

Example 4.15: Original Code

```
    int len = Math.<mask>(avail(), bAvail);

```

Example 4.16: Masked Code

Predictions include: `max, div, len` and so on.

Unary Operator Mutator

The Unary Operator Mutator targets unary operations in source code, introducing mutations by replacing or altering unary operators. Unary operators perform operations on a single operand and are commonly used for incrementing or decrementing values, logical negation, and bitwise operations.

Below is an example of a mutation of an increment operator:

```
    int increment(int num){
        return num++;
    }

```

Example 4.17: Original Code

```
        int increment(int num){
        return num<mask>;
    }

```

Example 4.18: Masked Code

The CodeBert prediction includes: `--` , `+=` , `++` and `+`.

4.3 Mutation Generation in Traditional Tools

Traditional mutation testing tools, such as PIT and Jumble, are widely used by the Java community as they can generate and execute mutants on a systematic basis. Unlike the approach using pre-trained language models (LLMs), these tools rely on pre-defined mutation operators to imitate faults as well as static code structure analysis. This section investigates PIT Jumble and MMT's mutation generation mechanisms, execution process, as well as their supported mutation operators and gives qualitative baseline to contrast them with LLM-based ones.

4.3.1 Mutation Generation Approach

Traditional mutation tools rely on syntactic transformations of program code to create mutants. Each transformation is defined by a mutation operator, which specifies a rule for altering the program. Mutation operators generally target common programming constructs, such as arithmetic operators, relational conditions, and method calls.

4.3.2 PIT

PIT operates by manipulating Java bytecode rather than source code, making it efficient and scalable for large codebases. This bytecode manipulation reduces overhead by eliminating the need to recompile code after each mutation is introduced. PIT's mutant generation is performed in a two-stage process:

1. **Identification of mutation points**: The tool scans compiled classes to identify potential locations for mutations. It records these as MutationIdentifiers, which include details such as the class, method signature, and the specific instruction to mutate [10].
2. **Dynamic mutant generation and execution**: PIT generates the mutant's bytecode on demand during test execution. It uses the Java instrumentation API to insert the mutated bytecode into a running JVM. This approach minimizes disk I/O and memory usage since only one mutant exists in memory at a time [10].

PIT's efficiency is further enhanced by its targeted test execution strategy, where only tests that exercise the mutated code are executed. By default, PIT runs all mutants of a given class in a shared JVM instance to balance performance and isolation, although stricter isolation levels (e.g., one JVM per mutant) are configurable [10].

4.3.3 Jumble

Jumble is an older tool that, like PIT, manipulates Java bytecode to produce mutants. However, it primarily focuses on method-level mutations, specifically arithmetic and conditional operations, and is

often regarded as more limited in scope compared to PIT. Jumble's approach is optimized for mutation testing at the unit level, and its mutators are considered a subset of those in PIT [28].

4.3.4 MMT

MMT [9] offers a new approach for creating mutants. Unlike traditional tools that apply predefined sets of mutation operators, MMT uses model transformation techniques to define and implement mutations. Key features of MMT include:

- **Model-driven mutation**: MMT translates Java bytecode into a model representation, allowing mutations to be specified and applied at different abstraction levels. This approach facilitates the creation of complex and domain-specific mutation operators that are challenging to implement with traditional methods.
- **Advanced Mutation Operators**: Along with standard set of mutation operators MMT supports a broad spectrum of adavnaced mutation operators, including those that modify object-oriented structures, Java-specific properties, and API method calls. This capability positions MMT as the only mutation testing tool for Java bytecode that supports such mutations.

Mutation generation and execution process in MMT can be described as follows:

1. **Model Extraction**: MMT parses the Java bytecode to create a corresponding model representation.
2. **Mutation Application**:Defined mutation operators are applied to the model, generating a set of mutated models.
3. **Reversing Back To Code**: The mutated models are transformed back into bytecode.
4. **Test Execution**: The test suite is executed against the mutated bytecode to evaluate the effectiveness of the tests in detecting the introduced faults [9].

4.4 Supported Mutation Operators By Tradition Tools

Studies [7, 27] suggest that mutation operators in Java mutation testing are crucial for simulating faults, enhancing test suite robustness, and improving mutation testing efficiency, particularly by addressing object-oriented features, stream-related faults, and computational costs. This section explores various mutation operators that the traditional tools support.

4.4.1 PIT's Mutation Operators

Operator Name	**Description**
Conditionals Boundary	Modifies conditional operators that involve relational boundaries (e.g., replacing `<` with `<=`).
Increments Mutator	Replaces increment and decrement operators with alternative behavior (e.g., `++` becomes `--`).
Invert Negatives Mutator	Inverts negation of integer and floating point variables.
Math Mutator	Alters arithmetic operations by replacing operators such as `+` with `-`, or `*` with `/`.
Negate Conditionals Mutator	Negates relational operators in conditional statements, changing their logic (e.g., `==` becomes `!=`).
Return Values Mutator	Changes the return values of methods based on the return type. This mutator has been superseded by the new returns mutator set: Empty returns, False returns, True returns, Null returns and Primitive returns.

Table 4.1: Mutation Operators Supported by PIT

Furthermore, PIT supports series of optional and experimental mutators, such as: Constructor Call Mutator, Inline Constant Mutator, Experimental Argument Propagation etc. [5].

4.4.2 Jumble's Mutation Operators

Jumble, being more minimalistic, implements a subset of the mutation operators seen in PIT. It provides a smaller, more targeted set of mutants but lacks many of the extended capabilities available in PIT and PITRV [28]. Optionally, Jumble can mutate inline constants, which are literal values directly used

Operator Name	**Description**
Conditionals	Replaces each condition with its negation (e.g., replacing `x > y` with `!(x > y)`).
Binary Arithmetic Conditionals	replace binary arithmetic operations for either integer or floating-point arithmetic with another operation (e.g., `|` becomes `&`).
Increments	Increment operations are mutated to decrements and vice versa.
Negate Conditionals Mutator	Negates relational operators in conditional statements, changing their logic (e.g., `==` becomes `!=`).
Return Values	Changes return values. Primitive non-zero return can be changed to 0 and 0 retruns are changed to 1. Non-null object returns are changed to null returns.
Switch Statements	Swaps each case of switch statement with the default case or another case, or changes the case value

Table 4.2: Mutation Operators Supported by Jumble

in code, by changing their numeric or boolean values. It can also modify class pool constants, replacing primitive constants and string literals to test whether the program's behavior changes are detected by the test suite [2].

MMT's Mutation Operators

MMT currently supports a total of 89 mutation operators, categorized across different levels of abstraction, from low-level instruction manipulations to high-level structural changes. These operators are specified as model transformation rules within the Henshin framework, operating on the Mod-BEAM metamodel representation of Java bytecode. This model-driven approach not only enables the expression of complex mutation scenarios but also ensures syntactic correctness through adherence to Object Constraint Language (OCL) constraints [9].

MMT distinguishes itself by guaranteeing syntactic and semantic correctness in its mutants. By using the Mod-BEAM framework to model Java bytecode and enforcing EMF constraints (defined using OCL), MMT ensures that every mutant passes bytecode verification. This capability reduces the generation of illegal or non-executable mutants, a common problem in non-model-based mutation testing tools [9].

The mutation operators in MMT can be summarized as follows:

Operator Name	**Description**
Data Type	Arithmetic/Relational operator replacements
Replacement	Values, conditions, methods replaced; swaps and removals
Java-Specific	`this` insertion/deletion, accessor/mutator changes, reference/content operations.
Object-Oriented	Class-level mutations: inheritance, polymorphism, type casts, method bodies
API/Library-Specific	API parameter/method replacements and swaps

Table 4.3: Mutation Operators Supported by MMT

In total, 89 mutation operators have been implemented, covering method-level and class-level transformations [9].

4.5 Qualitative Comparison of Traditional Tools and μBERT

In this section, we present a qualitative comparison between μBERT, and traditional mutation testing tools, specifically PIT, Jumble, and MMT. The comparison focuses on their mutation generation strategies, diversity of mutation operators, guarantees of syntactic and semantic correctness, extensibility, as well as their support for test execution and reporting.

4.5.1 Mutation Generation Approaches

Traditional mutation testing tools—PIT, Jumble, and MMT—employ deterministic mutation generation strategies based on explicitly defined mutation operators. Each operator describes a syntactic transformation to systematically introduce faults in the code, typically simulating common programming mistakes. These tools apply such transformations directly to, Java bytecode (PIT, Jumble), or model representations of bytecode (MMT). This approach ensures a predictable, repeatable mutant generation process [9, 10, 21].

By contrast, μBERT uses pre-trained language model (CodeBERT) to predict alternative tokens within the context of Java source code. Rather than relying on manually defined operators, μBERT generates mutants by masking code tokens and requesting CodeBERT to suggest contextually plausible replacements. This allows μBERT to generate diverse and semantically informed mutants, resembling mistakes developers are more likely to make [14].

4.5.2 Comparison of Mutation Operators

Mutation operators determine the diversity and relevance of mutants generated. Following table summarizes the capabilities of each tool regarding mutation operators and their application. The table represents direct extension of the comparison made in [9] for PIT, Jumble and MMT.

Table 4.4: Qualitative Comparison of Mutation Testing Tools

Aspect	**PIT**	**Jumble**	**MMT**	**μBERT**
Operator Specification Language	AST	BCEL	MTL	Spoon
Java-specific	No	No	Yes	Partially. Replacement of `this` and `instanceof` keywords
Object-Oriented Structure Mutations (Polymorphism & Inheritance)	Not Supported	Not Supported	Supported	Limited to method calls and identifier replacements; no class structure modifications
Arithmetic Repl.	Supported	Supported	Partly supported	Supported
Conditional Repl.	Supported	Supported	Not supported	Supported
Logical Repl.	Supported	Supported	Not supported	Supported
Relational Repl.	Supported	Supported	Partly supported	Supported
Shift Repl.	Supported	Supported	Not supported	Supported
Value Repl.	Supported	Supported	Supported	Supported
Call Repl.	Supported	Not Supported	Partly supported	Supported
Increment Repl.	Supported	Supported	Supported	Supported
Return Repl.	Supported	Supported	Supported	Supported

Continued on next page

Table 4.4 Continued from previous page

Aspect	PIT	Jumble	MMT	μBERT
Parameter Exchange	Supported	Not Supported	Supported	Partly Supported

4.5.3 Integration and Usability

Mutation testing tools help in identifying the robustness of test suites and are crucial for improving software quality. However, their integration and usability vary significantly across different tools.

PIT is noted for its robust integration. Along with the command-line interface, PIT is well integrated with Java development tools such as Ant and Maven. Third party components provide integration with Gradle, Eclipse, IntelliJ and other tools making it more suitable for industry use. Moreover, PIT is an open-source tool with a strong community and ongoing support, which can be beneficial for developers looking for a reliable and up-to-date tool [5].

Along with MMT, Jumble can be used as Eclipse plugin, however, it also offers command-line-interface to execute the mutations [2, 9].

4.5.4 Test Execution and Report Generation

All traditional mutation testing tools, that are being discussed, not only have the capability to generate mutants, but also provide built-in mechanisms to execute the available test suites against the mutated versions of the program and automatically assess whether the test cases successfully detect (kill) the mutants or not. For instance, PIT offers advanced features such as incremental analysis, parallel execution of mutants, and support for different test engines (JUnit, TestNG), making it easy to incorporate into continuous integration pipelines [5].

In contrast, μBERT stops at the mutant generation phase. Although it produces compilable and semantically valid mutants, it does not provide any functionality to run test cases against these mutants. Users of μBERT are expected to manage the execution of mutants manually or develop additional tooling to automate this step. This lack of integration creates an additional overhead for users who wish to use μBERT in a real-world setting [3].

Beyond test execution PIT, Jumble, and MMT also generate comprehensive reports that summarize:

- Mutation coverage statistics.
- Survived and killed mutants.
- Code locations and types of applied mutation operators.

These reports are essential for evaluating the quality of test suites and providing actionable feedback to developers. They also enable teams to track improvements in test effectiveness over time [2, 4, 5].

μBERT, on the other hand, does not include built-in reporting capabilities. It does not offer a mechanism to automatically aggregate or visualize the results of mutation testing campaigns. This can limit its usability in practice, particularly in large projects where mutation analysis at scale is required [3].

4.5.5 Optimization Techniques

Mutation testing is computationally expensive due to the large number of mutants generated, which require significant resources to execute against test suites [35, 36, 40, 51]. This high cost is a major barrier to its adoption in large-scale environments like Google's extensive codebase [35]. To address this issue, the mutation testing tools offer some optimization techniques, that we are going to explore in the following section.

Optimization techniques deployed by PIT

For handling the large number of mutants and test executions, PIT implements several optimization techniques:

- **Bytecode Execution**: PIT operates directly on Java bytecode, which allows it to execute mutants more quickly than source-level mutation testing tools. This approach reduces the overhead associated with compiling and running tests, thereby speeding up the mutation testing process [10].
- **Memoization**: The MeMu approach, implemented on top of PIT, optimizes execution by memoizing expensive methods. This technique avoids redundant computations by storing results of expensive method calls and reusing them when the same inputs are encountered again, leading to significant reductions in execution time [19].
- **Test Selection**: PIT uses coverage based technique for selecting test suites. Tests are selected by measuring their line, block or instruction coverage. The test is only run if it exercises the code that was affected by the mutant. Although it may produce some overhead to measure coverage, this selective approach facilitates faster test execution [1].
- **Cost amortization**: PIT is the first generally available incremental mutation testing system, with the option to amortize the cost of analysis by storing a history of results [1].
- **Parallel Execution**: To leverage multi-core processors, PIT supports parallel execution of tests against mutants, distributing the workload and reducing overall analysis time [1].

Optimization techniques deployed by Jumble

Similar to PIT, Jumble also operates at bytecode level. Additionally, Jumble uses three key heuristics to minimize the number of tests that need to be executed for each mutation:

1. **Timing-Based Test Order**: Before mutating, Jumble runs all tests and sorts their execution time from fastest to slowest. During mutation, Jumble then runs the shortest tests first to avoid running long and expensive tests unnecessarily.
2. **Remembering the Test Case for Each Method**: Once Jumble finds a test case that fails for a mutation in a method, it remembers the test for future mutations within the same method, reducing redundant test executions.
3. **Remembering the Last Failing Test**: If a mutation is re-tested in the future, Jumble first tries the test that failed previously, reducing test execution time.

Lastly, Jumble extends the BCEL class loader to dynamically modify and reload classes on demand, so that only the mutated version of a class is modified and loaded, allowing it to reuse the test environment without the need for a full JVM restart for each mutation [21].

Optimization techniques deployed by MMT

In order to enhance performance, MMT utilizes static and dynamic filtering techniques. It uses static analysis to analyze cyclomatic complexity and method dependencies before they are modified in order to avoid unnecessary mutants. Additionally, dynamic analysis only changes code covered by current tests to minimize further mutant execution and enhance efficiency [9].

Contrary to PIT and Jumble that introduce mutations in raw bytecode, MMT follows a rigorous validation procedure to ensure syntactic validity. By utilizing the EMF and OCL, MMT prevents the generation of invalid or non-executable mutants. The validation procedure reduces wasted computation and ensures that only meaningful and well-formed mutants are produced [9].

Absence of Explicit Optimization Strategies in μBERT

In contrast to traditional mutation testing tools, μBERT does not particularly have means to speed up mutation generation and test running. It uses a pre-trained language model (CodeBERT) to generate mutations, so it takes more computational resources than rule-based tools. Each mutation needs a new analysis of the language model, so it takes much more resources than bytecode or model-based mutation testing tools. In addition, μBERT does not include test execution feature for running tests and reporting test outcomes, so it needs other tools to run and check the mutants. Although it generates good and context-aware mutations, its lack of performance optimization reduces its effectiveness in large mutation testing unless you have other tools.

4.6 Conclusion

Mutation testing remains a powerful methodology for assessing test suite adequacy by introducing controlled faults into a software system. Traditional mutation testing tools such as PIT, Jumble, and MMT rely on predefined, systematic mutation operators to generate mutants, ensuring repeatability, structured fault simulation, and efficient test execution mechanisms. These tools, while differing in their approaches—whether bytecode-level mutation (PIT, Jumble) or model-driven transformation (MMT)—share a common focus on efficiency and structured fault injection. Through syntactic and, in some cases, semantic validation (MMT), they ensure that generated mutants maintain logical consistency within the program.

By contrast, μBERT represents a fundamentally different approach, leveraging (PLMs) to generate mutants dynamically. Instead of relying on fixed mutation operators, μBERT masks tokens within source code and utilizes CodeBERT to predict plausible replacements. This context-aware generation technique allows μBERT to create more naturalistic and realistic mutants, aligning closer to real-world programming errors than traditional syntactic transformations. However, this innovation comes at a cost. μBERT lacks formal syntactic and semantic correctness validation, potentially leading to unintended or irrelevant mutants.

From a usability and integration perspective, PIT stands out as the most industry-ready tool, offering seamless integration with Java testing frameworks (JUnit, TestNG), IDE support, and continuous integration pipelines. Jumble, while functional, is limited in scope, and MMT's model-driven approach, enables generating unique class-level mutants and could potentially be used also industry settings as it already offers plugin (Eclipse) for development environment integration. μBERT, in contrast, is currently research-focused and lacks built-in test execution or mutation report generation. The absence of automated test execution and integration capabilities means that μBERT requires external orchestration for practical use in software testing workflows.

Regarding performance optimization, traditional tools implement various techniques to solve the issue of the high computational cost of mutation testing. PIT employs incremental analysis, selective mutation, parallel execution, and memoization strategies, making it the most efficient of the traditional tools. Jumble utilizes heuristic-driven test execution order to minimize redundant test runs, while MMT optimizes mutant generation through static and dynamic filtering. In contrast, μBERT does not employ explicit optimization strategies, leading to significant computational overhead. Since each mutation requires a separate inference pass through CodeBERT, μBERT is inherently slower than traditional approaches, making large-scale mutation testing impractical without further optimizations.

In conclusion, while traditional mutation testing tools emphasize efficiency, structured mutant generation, and execution scalability, μBERT introduces a novel, approach that enhances mutation diversity and realism. However, its lack of performance optimizations, execution automation, and integration support currently limits its practical application. Future advancements in performance optimization, test execution automation, and CI/CD integration would be necessary for μBERT to transition from a research prototype to a viable industry tool. The trade-off between mutation realism and execution efficiency remains a key consideration in choosing the most suitable approach for mutation testing in software engineering.

Chapter 5

Quantitative Experiments on Effectiveness

5.1 Quantitative Evaluation of Effectiveness

Evaluating the effectiveness of mutation testing tools is a crucial aspect to understand their practical use in modern software development and testing workflows.

This chapter presents a quantitative evaluation of the effectiveness of μBERT, a mutation testing tool that employs a pre-trained language model to generate mutant classes.

The focus of this evaluation is twofold. First, the real-world fault detection capability of the tool is examined by assessing how well the generated mutants correlate with actual bugs in open-source projects. Second, the adequacy of test suites is assessed when confronted with the tool's mutants. By comparing the effectiveness of the tool across different projects and test configurations, this study aims to quantify the power of the generated mutants and evaluate whether they provide a stronger or weaker challenge to existing test suites compared to traditional mutants.

The data repository containing the experimental findings is available at `https://github.com/hojiakbar9/mBERT-analysis-thesis/`

5.2 Experimental Setup

5.2.1 Defects4J

For evaluating the effectiveness of μBERT Defects4J projects are employed. Defects4J provides a collection of real bugs, which allows researchers to conduct reproducible studies in software testing. By using real bugs instead of artificial ones, the framework ensures that research findings are more representative of real-world scenarios, enhancing the comparability of empirical studies across different research efforts [23].

Defects4J also offers numerous features for executing different types of operations on the projects. For the purpose of the evaluation, testing framework of the Defects4J is used to access whether or not the generated mutant by μBERT is detected by the corresponding test suite. Additionally, three projects (Cli, Codec, Jsoup) with different versions are used to conduct the experiments. The versions of the projects correspond to those that are used to evaluate MMT, that was discussed in [9].

5.2.2 μBERT Configuration

μBERT offers several configuration options to generate mutants [3]. Those options include:

- Source file name
- Path, where mutants should be located

- Maximum number of mutants
- Method name for mutation with the option for specifying the method definition line number
- Specific line number in source file to mutate

Specification of the source file and mutants directory are required. By default, μBERT mutates all methods in the given source class.

For evaluation purposes, default configurations are used.

5.2.3 Workflow

Below is the illustration of the workflow of how the experiments were conducted in the example of mutating the 22nd patched version of the Defects4J project Cli:

1. Checking out the fixed version of Defects4J project using Defects4J's framework:

```
defects4j checkout -p Cli -v 22f -w /temp/cli_22_fixed
```

2. Configure the μBERT to mutate the patched class by passing the source file to mutate and the path for saving the generated mutants as arguments.

```
./mBERT.sh -in=/temp/cli_22_fixed/     \
src/java/org/apache/commons/cli/PosixParser.java  \
-out=/temp/cli_22_fixed/generated-mutants
```

3. Compile the generated mutants.

```
./compile-mutants.sh /temp/cli_22_fixed/generated-mutants
```

4. Execute the test suites using script that retrieves all compilable mutants, replaces original source file with mutant and runs the test suite using Defects4J test command.
5. Save all relevant reports

Following the experiments, an analysis was conducted to determine whether the introduced modifications in the mutant overlapped with the original defect and whether the bug-detecting test cases were triggered.

5.3 Evaluation of Effectiveness

In this section, the efficacy of μBERT was evaluated using results from controlled experiments described in previous sections. Specifically, we examine whether the mutants generated by μBERT are detected by existing test suites, and aligned with real bug locations.

Table 5.1 summarizes the experimental outcomes from applying μBERT to 24 real fixed version of three Java projects: Cli, Codec, and Jsoup. Each entry indicates whether the generated mutants were killed by the corresponding test suite and whether the mutant location overlapped with the actual bug location.

Table 5.1: Overview of Mutation Results

Project	**Patched class**	**Killed?**	**Patch?**	**Comment**
Cli 22	PosixParser	Yes	Yes	
Cli 27	OptionGroup	Yes	Yes	
Cli 33	HelpFormatter	Yes	Yes	
Cli 38	DefaultParser	Yes	Yes	
Codec 1	Caverphone	Yes	Yes	
Codec 1	Metaphone	Yes	Yes	
Codec 1	SoundexUtils	Yes	Yes	

Codec 4	Base64	Yes	No	Patch is located in the constructor
Codec 7	Base64	Yes	Yes	
Codec 8	Base64	Yes	No	Patch only consists of deleted code
Codec 9	Base64	Yes	Yes	
Codec 10	Caverphone	Yes	Yes	
Codec 13	DoubleMetahone	Yes	Yes	
Codec 13	StringUtils	Yes	Yes	
Codec 14	Lang	Yes	No	Patch is located in static code block
Codec 14	PhonetcEngine	Yes	Yes	
Jsoup 3	Element	Yes	Yes	
Jsoup 3	Parser	Yes	Yes	
Jsoup 4	Entities	Yes	Yes	
Jsoup 6	Entities	Yes	Yes	
Jsoup 8	Node	Yes	Yes	
Jsoup 22	Element	Yes	Yes	
Jsoup 22	Elements	Partly (2 out of 3)	No	Patch is located in the constructor
Jsoup 22	Node	Yes	Yes	
Total		23	20	

5.4 Analysis & Discussion

The experimental results presented in Section 5.3 demonstrate that the mutants generated by μBERT are effectively detect by test suites in the majority of cases. Specifically, in all 24 evaluated scenarios, at least some of the generated mutants are eliminated by existing test cases, indicating that these mutants are indeed meaningful and capable of challenging test suites.

A key observation from the results is that μBERT's mutants are well-aligned with real-world defects. In most cases, the generated mutants are located in the same region of code as the actual patches applied in the fixed versions of the Defects4J projects. This suggests that μBERT is capable of introducing mutations that resemble real bugs, supporting its validity as an effective mutation testing tool.

However, there were instances where μBERT-generated mutants does not overlap with the actual patched defects. For example, in Codec 14 and Jsoup 22, the mutation is located outside the actual patched region, such as within a constructor or static code block. This indicates that while μBERT can produce meaningful mutations, it does not explicitly target known defect locations, which may limit its effectiveness in some contexts. Enhancing μBERT to incorporate defect localization techniques could improve its precision in generating relevant mutants.

5.4.1 Comparison with MMT, PIT, and Jumble

To further contextualize the effectiveness of μBERT as a mutation testing tool, we compare its performance against MMT, PIT, and Jumble. The paper *Mutation Testing of Java Bytecode: A Model-Driven Approach* [9] provides a detailed evaluation of these tools using Defects4J, which aligns well with our experimental setup.

Effectiveness in Generating Realistic Mutants

When comparing the results from [9] MMT, Jumble, and PIT, μBERT demonstrated the ability to generate mutants that were slightly more effective. Specifically, 23 out of 24 bug-detecting test cases were triggered, and in 20 out of 24 cases, the generated mutants were found to overlap with the original bug, though they were marginally outperformed by PIT.

Despite these strengths, μBERT also presented notable challenges:

- Unlike PIT, MMT and Jumble which ensure syntactic correctness through bytecode validation, μBERT relies solely on compilation filtering. This means some generated mutants may be less meaningful or compilable but not semantically valid.
- Traditional tools offer built-in test execution and detailed mutation analysis reports. μBERT, on the other hand, requires external automation for running tests and collecting results.
- Generating mutants with a pre-trained language model requires a higher computational cost compared to bytecode-based tools. More in-depth discussions on performance and efficiency metrics will follow in subsequent sections.

5.4.2 Implications for Future Work

The results highlight the potential of μBERT as a viable mutation testing approach, but several areas for improvement remain:

1. **Targeted Mutation Generation** : Improving defect-localization approaches, like support for static code blocks or constructor mutations, could further enhance the probability of generating more relevant mutants.
2. **Automated Execution and Reporting** : Integrating μBERT with existing testing frameworks to automate mutant execution and report generation would make it more practical for industry adoption.
3. **Optimization Strategies**: Applying performance optimizations, such as caching CodeBERT predictions or reducing redundant inferences, could improve scalability.

5.5 Conclusion

Overall, μBERT shows a notable capability for generating realistic mutants, which closely align with real-world software bugs. However, its limited support for automated execution, reporting features, and performance optimization hinder its usability in comparison to conventional mutation testing tools. Nonetheless, future advancements in these areas could elevate μBERT as a viable and competitive alternative in mutation testing.

Chapter 6

Quantitative Experiments on Efficiency

6.1 Introduction to Quantitative Experiments on Efficency

Mutation testing is a powerful technique used to evaluate the fault-detection ability of software test suites. However, its practical adoption is often hindered by its inefficiency due to potentially high computational costs. The process involves generating, compiling, and executing tests against a potentially large number of mutants, which collectively demand significant computational resources. This can render the approach time-consuming and resource-intensive, particularly for large-scale industrial software systems[12, 45, 51].

Given that μBERT introduces a novel approach based on computationally intensive PLMs like CodeBERT for mutant generation, analyzing its efficiency is paramount. Unlike traditional tools that often rely on predefined syntactic rules or bytecode manipulation, μBERT leverages deep learning inference, which inherently carries a different performance profile. This chapter quantitatively evaluates the efficiency of μBERT, focusing primarily on the time required to generate and prepare mutants for execution. The primary metric used is execution time, segmented into mutation generation time and mutant compilation time, allowing for a nuanced understanding of the performance characteristics and bottlenecks of the μBERTs approach. Understanding this efficiency profile is crucial for assessing its practical viability and identifying potential areas for future optimization.

6.2 Experimental Setup for Efficiency

The experiments designed to evaluate μBERT's efficiency utilized the same foundational setup described in Chapter 5 for the effectiveness evaluation, ensuring consistency and comparability.

6.2.1 Defects4J and μBERT Configuration

The Defects4J benchmark [23] served as the source for real-world Java projects and their associated bugs. The specific projects (Cli, Codec, Jsoup) and versions selected correspond to those used in the effectiveness study and those evaluated in related work [9], facilitating cross-study comparisons.

6.2.2 Medium for Execution of the Experiments

All efficiency experiments were conducted on a single, consistent hardware platform to ensure reproducibility and fair measurement. The specifications of the machine were as follows:

- Processor: AMD Ryzen 7 5800H (8 cores, 16 threads)
- Graphics Card: NVIDIA GeForce 3070 RTX
- RAM: 16Gb
- Vendor: Lenovo

- Model: Legion 5 Pro

6.3 Evaluation of Efficiency

The efficiency of μBERT was measured by recording the time taken for two key phases:

1. **Mutation Generation Time**: The time elapsed from invoking the μBERT script until it completed generating all candidate mutant source files for a given target class. This includes parsing, token masking, CodeBERT inference, and mutant source code generation.
2. **Compilation Time**: The time required to attempt compilation of all generated mutants and filter out those that failed to compile. This step is essential as μBERT's output is source code, and only compilable mutants are useful for subsequent test execution.

An "Efficiency Score" was calculated to provide a normalized measure of performance, defined as:

Efficiency Score = Number of Compilable Mutants / (Mutation Generation Time [sec] + Compilation Time [sec])

This score represents the rate at which usable (compilable) mutants are produced by the entire μBERT generation and filtering pipeline. Table 6.1 presents the detailed results for each experimental run across the selected Defects4J project versions. Table 6.2 provides summary statistics averaged across all runs.

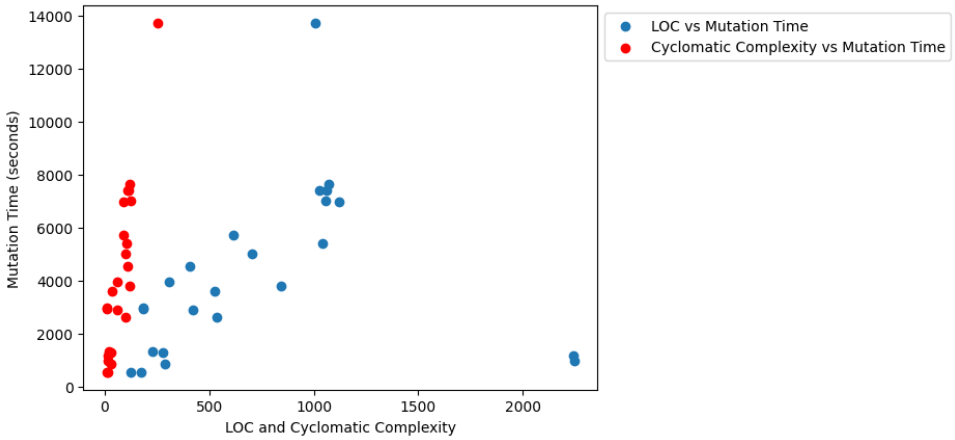
Table 6.1: Efficiency Overview per Project

Project	**Number of generated mutants**	**Number of compilable mutants**	**Mutation Generation Time**	**Compilation Time**	**Efficiency Score**
Cli 22	864	129	13 min 26 sec	7 min 59 sec	0.10038
Cli 27	359	41	5 min 59 sec	3 min 10 sec	0.07468
Cli 33	3118	669	57 min 25 sec	33 min 1 sec	0.12329
Cli 38	2926	506	54 min 27 sec	29 min 25 sec	0.10055
Codec 1, C	1866	329	29 min 16 sec	19 min 43 sec	0.11194
Codec 1, M	2474	656	47 min 22 sec	28 min 21 sec	0.14439
Codec 1, S	312	80	6 min 46 sec	2 min 36 sec	0.14234
Codec 4	4405	1705	1 h 13 min	44 min 15 sec	0.24236
Codec 7	4462	1727	1 h 14 min	53 min 15 sec	0.22619
Codec 8	4218	1636	1 h 19 min	44 min 33 sec	0.22069
Codec 9	4248	1636	1 h 19 min	44 min 37 sec	0.22057
Codec 10	1865	326	32 min 01 sec	17 min 46 sec	0.10914
Codec 13, D	8786	2917	1 h 53 min	1 h 55 min	0.21245
Codec 13, S	407	59	7 min 52 sec	6 min 21 sec	0.06916
Codec 14, L	613	97	11 min 26 sec	10 min 41 sec	0.07309
Codec 14, P	1956	269	33 min 51 sec	26 min 25 sec	0.07439
Jsoup 3, E	2002	334	35 min 00 sec	28 min 09 sec	0.08815
Jsoup 3, P	2213	276	40 min 23 sec	25 min 53 sec	0.06941
Jsoup 4	513	100	10 min 39 sec	5 min 44 sec	0.10172
Jsoup 6	525	100	10 min 44 sec	9 min 05 sec	0.08481
Jsoup 8	1599	303	28 min 29 sec	19 min 48 sec	0.10459
Jsoup 22, E	2659	403	1h 02 min	54 min 8 sec	0.05783
Jsoup 22, Es	1254	207	23 min 12 sec	20 min 18 sec	0.07931
Jsoup 22, N	2780	513	43 min 08 sec	52 min 21 sec	0.08954

Table 6.2: Average Efficiency Metrics of μBERT-based Mutation Testing

Metric	Average Value
Number of Generated Mutants	2,351
Number of Compilable Mutants	625.75
Mutation Generation Time	40 minutes 28 seconds
Compilation Time	29 minutes 16 seconds
Efficiency Score	0.1218 mutants/sec

Figure 6.1: Relationship between source code metrics (Lines of Code - LOC, and Cyclomatic Complexity) and Mutation Generation Time (in seconds)



Scatter plot illustrating the relationship between source code metrics (Lines of Code - LOC, and Cyclomatic Complexity) and Mutation Generation Time (in seconds) for μBERT across the evaluated Defects4J scenarios. The plot visually depicts the correlations calculated: LOC vs. Mutation Time (Correlation ≈ 0.22) and Cyclomatic Complexity vs. Mutation Time (Correlation ≈ 0.91)

6.4 Analysis & Discussion

The quantitative results presented in Tables 6.1 and 6.2, along with the correlations visualized in Figure 6.1, provide significant insights into the efficiency characteristics of μBERT.

6.4.1 Overall Performance

The average mutation generation time across the experiments was approximately 40 minutes and 28 seconds, with an additional average of 29 minutes and 16 seconds required for compiling the generated mutants. These times are substantial, particularly when considering that these experiments targeted single Java classes within the Defects4J projects. Extrapolating this performance to larger codebases or entire projects suggests that μBERT, in its current form, would require considerable time for comprehensive mutation analysis. The average efficiency score of approximately 0.12 compilable mutants generated per second (or roughly 7 compilable mutants per minute) further underscores the computational effort involved. This rate is significantly lower than what is typically achievable with highly optimized traditional tools like PIT, which operate on bytecode and employ numerous optimization strategies.

6.4.2 Variation in Performance

There is considerable variation in performance across different target classes. Mutation generation times ranged from under 6 minutes (Cli 27) to nearly 2 hours (Codec 13, D). This variance highlights that μBERT's efficiency is highly dependent on the specific characteristics of the code being mutated.

6.4.3 Correlation with Code Metrics

Figure 6.1 and the calculated correlation coefficients reveal critical factors influencing μBERT's performance:

- **Weak Correlation with LOC ($\approx$ 0.22):** The mutation generation time shows only a weak positive correlation with the Lines of Code (LOC) of the target class. This finding is somewhat counter-intuitive, as one might expect time to scale directly with code size. It suggests that the sheer volume of code is not the primary driver of μ BERT's execution time. Factors like the density of potential mutation points (expressions, identifiers, literals) or the fixed overhead associated with parsing and model invocation might play a more significant role, especially for smaller classes.
- **Strong Correlation with Cyclomatic Complexity ($\approx$ 0.91):** In contrast, there is a very strong positive correlation between the cyclomatic complexity of the target class and the mutation generation time. This is a key finding. Cyclomatic complexity measures the number of linearly independent paths through the code, essentially quantifying its structural complexity (e.g., number of loops, conditional branches). The strong correlation suggests that μBERT's performance degrades significantly as the logical complexity of the code increases. This is likely because more complex code offers:
  - A higher number of decision points and execution paths.
  - More complex contextual information that the CodeBERT model needs to process during inference when predicting token replacements.
  - Potentially more candidate locations identified by the initial parsing and masking steps within complex methods. This implies that the cognitive complexity of the code, rather than just its length, is the dominant factor determining μBERT's generation time.

6.4.4 Bottlenecks

The primary bottleneck appears to be the mutation generation phase itself, driven by the iterative process of masking tokens and invoking the CodeBERT PLM for predictions. Each potential mutation site requires interaction with the language model, and the complexity of the surrounding context heavily influences this process. The subsequent compilation phase also consumes significant time, reflecting the overhead of invoking the Java compiler repeatedly for potentially hundreds or thousands of generated mutant source files. Unlike tools operating at the bytecode level (e.g., PIT), μBERT necessitates this explicit, time-consuming source code compilation step for filtering.

6.4.5 Practical Implications

The observed efficiency profile presents challenges for the practical application of μBERT, especially in contexts requiring rapid feedback, such as Continuous Integration (CI) pipelines or large-scale industrial projects. The lengthy execution times, particularly for complex code segments, could make its routine use prohibitive without substantial optimizations or significant computational resources (e.g., powerful multi-core CPUs, potential GPU acceleration for inference). The lack of built-in optimization techniques comparable to those found in mature tools like PIT (e.g., test selection, bytecode manipulation, memoization, parallel execution support beyond basic scripting) further limits its scalability.

6.5 Conclusion

This chapter quantitatively assessed the efficiency of the μBERT mutation testing tool, focusing on the time required for mutant generation and compilation. The experiments, conducted on Defects4J projects, revealed that μBERT exhibits substantial execution times, averaging over an hour to generate and compile mutants for a single Java class.

The analysis identified a critical performance characteristic: μBERT's mutation generation time is strongly correlated with the cyclomatic complexity of the target code, indicating that logical intricacy, rather than simple code length (LOC), is the primary determinant of its efficiency. This dependency on complexity, coupled with the inherent computational cost of invoking a large pre-trained language model

(CodeBERT) for each mutation opportunity and the necessity of source code compilation, results in a relatively low throughput of usable mutants (average 0.12 compilable mutants/sec).

While Chapter 3 demonstrated μBERT's potential effectiveness in generating realistic and relevant mutants, the findings in this chapter highlight that this effectiveness comes at a significant efficiency cost. The current performance profile poses a substantial barrier to its practical adoption in large-scale or time-sensitive software development workflows. μBERT's current state positions it more as a research prototype demonstrating the potential of PLMs in mutation testing, rather than a readily deployable industrial tool.

Future work must prioritize addressing these efficiency limitations. Potential avenues include exploring PLM-specific optimizations such as model quantization or pruning, implementing caching mechanisms for CodeBERT predictions, investigating strategies for batching model inferences, exploring selective mutation techniques guided by model uncertainty, leveraging hardware acceleration (GPUs), and potentially integrating μBERT's generation capabilities with more optimized execution frameworks derived from traditional tools. Overcoming these efficiency challenges is crucial for realizing the practical potential of PLM-based mutation testing.

Chapter 7

Conclusion & Future Work

7.1 Summary of the Research

Mutation testing is a powerful technique for assessing the quality of software test suites, but its practical adoption has been hindered by challenges related to computational cost, equivalent mutants, and the realism of generated mutants. The advent of large-scale, PLMs in software engineering has opened new avenues for addressing some of these challenges, particularly mutant realism. This thesis focused on μBERT, a pioneering tool that leverages PLMs (specifically CodeBERT) to dynamically generate mutants for Java code based on contextual predictions, departing from the traditional reliance on predefined mutation operators.

The main purpose of this research was to perform a detailed evaluation of μBERT in relation to standard Java mutation testing tools (PIT, Jumble, MMT). We wanted to grasp the inherent trade-offs of this PLM-based approach. To accomplish this, the research employed a mixed-methods approach, including:

1. A qualitative evaluation of the mutation generation process, supported operators (implicit vs. explicit), usability, integration, and optimization strategies of μBERT versus traditional tools.
2. Quantitative experiments on effectiveness, using the Defects4J benchmark to assess μBERT's ability to generate mutants that are killed by test suites and align with real software faults.
3. Quantitative experiments on efficiency, measuring μBERT's mutant generation and compilation times on Defects4J projects and analyzing the correlation between its performance and source code metrics like LOC and Cyclomatic Complexity.

7.2 Answering the Research Questions

This study systematically addressed the research questions posed in Chapter 1:

- **RQ1: What are the qualitative characteristics of the mutation generation process and the types of mutants produced by μBERT compared to traditional Java mutation testing tools?**

  Our qualitative analysis (Chapter 4) revealed fundamental differences. Traditional tools like PIT, Jumble, and MMT use predefined, explicit sets of mutation operators, often based on common syntactic errors, applied systematically at the bytecode or model level. They typically offer integrated test execution, reporting, and various optimization techniques.

  In contrast, μBERT employs a dynamic, context-aware approach at the source code level. It masks tokens and uses CodeBERT to predict plausible replacements, implicitly defining mutation operators based on the model's learned understanding of code. This allows for potentially more "natural" and diverse mutants, particularly excelling at identifier and method call replacements that mimic developer mistakes. However, μBERT lacks the formal syntactic/semantic validation guarantees of tools like MMT, does not currently offer built-in test execution or reporting, and lacks explicit performance optimization strategies beyond basic filtering.

- **RQ2: How effective is μBERT in generating mutants that challenge test suites and align with real software faults, compared quantitatively against traditional tools using established benchmarks like Defects4j?**

  The quantitative experiments on effectiveness (Chapter 5) demonstrated that μBERT is indeed effective. Mutants generated by μBERT for Defects4J projects were successfully killed by the projects' test suites in the vast majority of cases (23 out of 24 scenarios had killed mutants). Furthermore, in most instances (20 out of 24), the locations of the μBERT-generated mutants overlapped with the actual patched code regions associated with the real bugs. This suggests μBERT generates meaningful, challenging mutants that correlate well with real-world faults. When compared against results reported for MMT, PIT, and Jumble on the same benchmarks [9], μBERT showed comparable, and in some aspects slightly superior, effectiveness in terms of triggering bug-detecting tests and overlapping with original bug locations, although PIT marginally outperformed it in patch overlap.

- **RQ3: What is the quantitative efficiency profile of μBERT in terms of mutant generation and compilation time, and how does it correlate with source code metrics like LOC and Cyclomatic Complexity?**

  The quantitative efficiency analysis (Chapter 6) revealed that μBERT's effectiveness comes at a significant computational cost. Average execution times to generate and compile mutants for single Java classes in Defects4J often exceeded one hour. The efficiency analysis identified a critical performance characteristic: μBERT's execution time exhibits a very strong positive correlation (≈ 0.91) with the Cyclomatic Complexity of the target class, while showing only a weak correlation (≈ 0.22) with LOC. This indicates that the logical complexity of the code, rather than sheer size, is the dominant factor driving µBERT's performance degradation. The primary bottlenecks identified were the computationally intensive PLM inference step required for each potential mutation and the subsequent need for source code compilation for filtering. The overall throughput of usable (compilable) mutants was relatively low (average ≈ 0.12 mutants/sec).

7.3 Implications of the Findings

- **The findings of this thesis have several implications for researchers and practitioners:** μBERT demonstrates that PLMs can generate mutants that align well with real faults, potentially offering a more realistic assessment of test suite quality compared to purely syntactic traditional operators. This encourages further research into PLM-driven mutation.
- **Efficiency is a Major Hurdle:** The significant computational cost, strongly linked to code complexity, currently limits μBERT's practical applicability, especially in time-sensitive contexts like CI/CD or for large-scale systems. Traditional tools, particularly bytecode-based ones like PIT, remain far more efficient.
- **Complexity Matters More Than Size:** The strong correlation with Cyclomatic Complexity suggests that applying μBERT (and potentially other PLM-based analysis tools) will be particularly challenging for complex code units, even if they are not exceptionally large in terms of LOC.
- **Integration Gap:** The lack of built-in test execution and reporting in μBERT necessitates external tooling, adding an overhead for practical use compared to integrated traditional tools.
- **Trade-off:** There is a clear trade-off between the potential realism offered by μBERT's PLM-based approach and the efficiency/integration maturity of traditional tools. The choice depends heavily on the specific context, available resources, and desired balance between mutant quality and analysis speed.

7.4 Limitations of the Study

This study presents a comprehensive analysis of μBERT, yet it is subject to several limitations. The investigation primarily centers on μBERT, which employs CodeBERT for generating mutants in Java. As a result, the findings may not be directly applicable to other mutation tools based on pre-trained language models, such as LLMorpheus, or those that utilize different models like GPT or LLaMA variants.

Additionally, the scope is limited to the Java programming language, which may affect the generalizability of the results to other languages.

The experimental evaluation draws heavily from the Defects4J benchmark. While Defects4J is a well-established and realistic benchmark due to its inclusion of real-world bugs, the results may vary when applied to more diverse industrial codebases or projects with distinct characteristics.

In terms of comparative analysis, the study's evaluation of μBERT's effectiveness in relation to tools like MMT, PIT, and Jumble is based on data reported in a previous study [9]. Due to practical constraints, it was not feasible to run all tools within the same experimental environment. Efficiency comparisons were also constrained, relying primarily on μBERT's recorded performance metrics alongside existing documentation of the performance characteristics and optimizations of traditional mutation testing tools.

Finally, the static analysis presented is based on the μBERT implementation available at the time of the study, which may not reflect future changes or improvements to the tool.

7.5 Future Work

Based on the findings and limitations, several avenues for future work emerge:

- **Efficiency Optimization:** This is critical for practical viability. Research should explore:
  - Inference optimization: Caching predictions, batching inference requests, leveraging GPU acceleration.
  - Reducing Redundancy: Investigating whether fewer PLM predictions are needed, perhaps guided by static analysis or code change information.
  - Hybrid Approaches: Combining μBERT's generation with more efficient execution/filtering mechanisms from traditional tools (e.g., generating mutants with μBERT's but executing/-analyzing with a PIT-like framework).
- **Integration and Usability:** Develop wrappers, plugins (e.g., for Maven, Gradle, IDEs), or integrate μBERT's core generation logic into existing testing frameworks to provide automated test execution and reporting, similar to traditional tools.
- **Broader Empirical Studies:**
  - Evaluate μBERT and other PLM-based tools on a wider range of projects, including industrial codebases.
  - Conduct direct comparative studies against state-of-the-art traditional tools in the same controlled environment, focusing on both effectiveness and efficiency.
  - Analyze the "naturalness" of μBERT-generated mutants by comparing them qualitatively and quantitatively against real developer-introduced bugs.

7.6 Concluding Remarks

This thesis provided a detailed analysis of μBERT, an innovative mutation testing tool leveraging pre-trained language models. Our findings indicate that μBERT holds significant potential for generating effective, realistic mutants that align well with real software faults, offering advantages over the purely syntactic operators of many traditional tools. However, this potential is currently counterbalanced by substantial efficiency challenges, primarily driven by the computational cost of PLM inference and its strong correlation with code complexity, along with a lack of practical integration features.

While traditional tools like PIT remain more practical for general-purpose, large-scale mutation testing due to their maturity and efficiency, PLM-based approaches like μBERT represent a vital and promising direction for research. Overcoming the identified efficiency and integration hurdles through targeted optimizations and framework development is crucial for unlocking the full potential of using PLMs to enhance the realism and effectiveness of mutation testing in the future.

Bibliography

[1] java mutation testing systems. `https://pitest.org/java_mutation_testing_systems`. Accessed: 20-03-2025.

[2] Jumble documentation. `https://jumble.sourceforge.net/mutations.html`. Accessed: 18-03-2025.

[3] mbert github repository. `https://github.com/rdegiovanni/mBERT`. Accessed: 20-03-2025.

[4] Mmt repository. `https://gitlab.uni-marburg.de/fb12/plt/modbeam-mt/mmt`. Accessed: 17-03-2025.

[5] Pitest documentation. `https://pitest.org/quickstart/mutators/`. Accessed: 17-03-2025.

[6] Chang ai Sun, An Fu, Xin Guo, and T. Chen. Remusse: A redundant mutant identification technique based on selective symbolic execution. *IEEE Transactions on Reliability*, 71:415–428, 2022.

[7] Manoel Aranda, Elvys Soares, Márcio Ribeiro, F. Ferrari, Rohit Gheyi, and Arthur Lima. Mutation operators for java streams. *Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing*, 2022.

[8] Muhammad Awais. Analyzing the effectiveness of mutation testing in real-world software development. 2025.

[9] Christoph Bockisch, Deniz Eren, Sascha Lehmann, Daniel Neufeld, and Gabriele Taentzer. Mutation testing of java bytecode: A model-driven approach. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, MODELS '24, page 237–248, New York, NY, USA, 2024. Association for Computing Machinery.

[10] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 449–452, New York, NY, USA, 2016. Association for Computing Machinery.

[11] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and M. Desmarais. Effective test generation using pre-trained large language models and mutation testing. *ArXiv*, abs/2308.16557, 2023.

[12] Xiangying Dang, D. Gong, Xiangjuan Yao, Tian Tian, and Huai Liu. Enhancement of mutation testing via fuzzy clustering and multi-population genetic algorithm. *IEEE Transactions on Software Engineering*, 48:2141–2156, 2021.

[13] Xiangying Dang, Xiangjuan Yao, D. Gong, Tian Tian, and Baicai Sun. Multi-task optimization-based test data generation for mutation testing via relevance of mutant branch and input variable. *IEEE Access*, 8:144401–144412, 2020.

[14] Renzo Degiovanni and Mike Papadakis. μbert: Mutation testing using pre-trained language models, 2022.

[15] Mickaël Delahaye and Lydie du Bousquet. A comparison of mutation analysis tools for java. In *2013 13th International Conference on Quality Software*, pages 187–195, 2013.

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

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[19] Ali Ghanbari and Andrian Marcus. Toward speeding up mutation analysis by memoizing expensive methods. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 71–75. IEEE, 2021.

[20] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[21] Sean A. Irvine, Tin Pavlinic, Leonard Trigg, John G. Cleary, Stuart Inglis, and Mark Utting. Jumble java byte code to measure the effectiveness of unit tests. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 169–175, 2007.

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

[23] René Just, D. Jalali, and Michael D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. pages 437–440, 2014.

[24] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, N. Malevris, and Yves Le Traon. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, 23:2426 – 2463, 2017.

[25] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, and Nicos Malevris. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 147–156, 2016.

[26] Peter Kokol. Software quality: How much does it matter? *Electronics*, 11(16):2485, 2022.

[27] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.*, pages 352–363, 2002.

[28] András Márki and Birgitta Lindström. Mutation tools for java. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 1364–1415, New York, NY, USA, 2017. Association for Computing Machinery.

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

[30] 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 *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 35–46, 2023.

[31] Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Mutation testing in evolving systems: Studying the relevance of mutants to code evolution. *ACM Trans. Softw. Eng. Methodol.*, 32(1), February 2023.

[32] Mike Papadakis and René Just. Special issue on mutation testing. *Inf. Softw. Technol.*, 81:1–2, 2017.

[33] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in computers*, volume 112, pages 275–378. Elsevier, 2019.

[34] Ali Parsai, Alessandro Murgia, and S. Demeyer. Littledarwin: A feature-rich and extensible mutation testing framework for large and complex java systems. pages 148–163, 2017.

[35] Goran Petrović, M. Ivankovic, G. Fraser, and René Just. Practical mutation testing at scale: A view from google. *IEEE Transactions on Software Engineering*, 48:3900–3912, 2021.

[36] Goran Petrović, M. Ivankovic, Bob Kurtz, P. Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 47–53, 2018.

[37] A. V. Pizzoleto, F. Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *J. Syst. Softw.*, 157, 2019.

[38] Iman Saberi and F. H. Fard. Model-agnostic syntactical information for pre-trained programming language models. *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 183–193, 2023.

[39] Iman Saberi, Fatemeh H. Fard, and Fuxiang Chen. Utilization of pre-trained language model for adapter-based knowledge transfer in software engineering. *Empir. Softw. Eng.*, 29:94, 2023.

[40] N. Shomali and Bahman Arasteh. Mutation reduction in software mutation testing using firefly optimization algorithm. *Data Technol. Appl.*, 54:461–480, 2020.

[41] Philipp Straubinger, Marvin Kreis, Stephan Lukasczyk, and Gordon Fraser. Mutation testing via iterative large language model-driven scientific debugging. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 358–367, 2025.

[42] Arzu Behiye Tarimci and Hasan Sözer. Mutation testing of pl/sql programs. *J. Syst. Softw.*, 192:111399, 2022.

[43] Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. Large language models for equivalent mutant detection: How far are we? In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1733–1745, New York, NY, USA, 2024. Association for Computing Machinery.

[44] Frank Tip, Jonathan Bell, and Max Schäfer. Llmorpheus: Mutation testing using large language models. *arXiv preprint arXiv:2404.09952*, 2024.

[45] Kevin J. Valle-Gómez, A. García-Domínguez, Pedro Delgado-Pérez, and I. Medina-Bulo. Mutation-inspired symbolic execution for software testing. *IET Softw.*, 16:478–492, 2022.

[46] Vineeta, Preeti Lochab, and Abhishek Singhal. Analysis of mutation testing tools in aspect oriented software engineering. *International Journal of Computer Applications*, 61:24–33, 2013.

[47] Bo Wang, Mingda Chen, Youfang Lin, Mike Papadakis, and Jie Zhang. An exploratory study on using large language models for mutation testing. *ArXiv*, abs/2406.09843, 2024.

[48] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50:911–936, 2023.

[49] Changqing Wei, Xiangjuan Yao, D. Gong, and Huai Liu. Spectral clustering based mutant reduction for mutation testing. *Inf. Softw. Technol.*, 132:106502, 2021.

[50] Changqing Wei, Xiangjuan Yao, Dunwei Gong, and Huai Liu. Test data generation for mutation testing based on markov chain usage model and estimation of distribution algorithm. *IEEE Transactions on Software Engineering*, 50:551–573, 2024.

[51] Jie M. Zhang, Lingming Zhang, M. Harman, Dan Hao, Yue Jia, and Lu Zhang. Predictive mutation testing. *IEEE Transactions on Software Engineering*, 45:898–918, 2016.